Translate Toolkit Documentation

Release 3.1.1

Translate.org.za

Sep 23, 2020

Contents

Welcome to Translate Toolkit's documentation. This documentation covers both user's and programmer's perspective.

CHAPTER 1

User's Guide

This part has the user's documentation for the tools included in the Translate Toolkit.

1.1 Features

- Work with **ONE localisation format**. You'll no longer be editing DTD files in one tool, .properties in another, OpenOffice GSI in a third. Simply do all your localisation in a PO or XLIFF editor
- **Converters** for a number of *formats*
  - OpenOffice.org SDF/GSI
  - Mozilla: .properties, DTD, XHTML, .inc, .ini, etc
  - Others: Comma Separated Value, TMX, XLIFF, TBX, PHP, WordFast TXT, Qt .ts, txt, .ini, Windows .rc, ical, subtitles, Mac OS X strings
- **File access to localization files** through the format API in all the above formats, as well as .qph, .qm, .mo
- Output **valid target file** types. We make sure that your output files (e.g. .properties) contain all comments from the original file and preserves the layout of the original as far as possible. If your PO entry is marked as fuzzy we use the English text, not your half complete translation. The converters for OpenOffice.org and Mozilla formats will also perform simple checks and corrections to make sure you have none of those hard to find localisation bugs.
- Our checker has over *42 checks* to find errors such as: missing or translated variables, missing accelerator keys, bad escaping, start capitalisation, missing sentences, bad XML and much more.
- Language awareness, taking language conventions for capitalisation, quotes and other punctuation into account
- **Find conflicting translations** easily, cases where you have translated a source word differently or used a target word for 2 very different English concepts
- **Extract messages** using simple text or a regular expression allowing you to quickly find and extract words that you need to fix due to glossary changes.
- **Merge snippets** of PO files into your existing translations.

- Create word, string and file **counts** of your files. Making it much easier to budget time as string counts do not give you a good indication of expected work.
- Create a set of PO files with **debugging** entries to allow you to easily locate the source of translations. Very useful in OpenOffice.org which provides scant clues as to where the running application has sourced the message.

The Translate Toolkit is also a **powerful API** for writing translation and localisation tools, already used by our own and several other projects. See the *base class* section for more information.

1.2 Installation

This is a guide to installing the Translate Toolkit on your system. If the Translate Toolkit is already packaged for your system, this is probably the easiest way to install it. For several Linux distributions, the package might be available through your package manager. On Windows, we recommend using a virtual environment.

If your system already has the toolkit prepackaged, then please let us know what steps are required to install it.

1.2.1 Building

For build instructions, see the *Building* page.

1.2.2 Download

Download a stable released version. Or if you have a python environment, run *pip install translate-toolkit*. For those who need problems fixed, or who want to work on the bleeding edge, get the latest source from *Git*.

If you install through your distribution's package manager, you should automatically have all the dependencies you need. If you are installing a version from Version Control, or from a source release, you should check the README file for information on the dependencies that are needed. Some of the dependencies are optional. The README file documents this.

1.2.3 Installing packaged versions

Get the package for your system:

RPM	If you want to install easily on an RPM based system
.tar.gz	for source based installing on Linux
.deb	for Debian GNU/Linux (etch version)

The RPM package can be installed by using the following command:

```
$ rpm -Uvh translate-toolkit-1.0.1.rpm
```

To install a tar.bz2:

```
$ tar xvjf translate-toolkit-1.1.0.tar.bz2
$ cd translate-toolkit-1.1.0
$ su
$ ./setup.py install
```

On Debian (if you are on etch), just type the following command:

```
$ aptitude install translate-toolkit
```

If you are using an old Debian stable system, you might want to install the .tar.bz2 version. Be sure to install python and python development first with:

```
$ apt-get install python python-dev
```

Alternatively newer packages might be in testing.

1.2.4 Installing on Windows

On Windows we recommend that you install Translate Toolkit using a virtual environment. This makes installation clean and isolated.

Use the latest Python 3.8. Install virtualenvwrapper-win to simplify handling of virtualenvs.

1. Install latest Python 3.8
2. Open cmd.exe or similar
3. *pip install virtualenvwrapper-win*
4. *mkvirtualenv ttk* where "ttk" is the name for the new virtualenv
5. *pip install translate-toolkit[recommended]* to install latest stable or *pip install –pre translate-toolkit[recommended]* to try a pre-release
6. *po2prop –version* to double check you have the right version

Next times you need to use Translate Toolkit just remember to:

1. Open cmd.exe or similar
2. *workon ttk* to enable the virtualenv again
3. Run the Translate Toolkit commands you want

1.2.5 Installing from Git

If you want to try the bleeding edge, or just want to have the latest fixes from a stabilising branch then you need to use Git to get your sources:

```
$ git clone https://github.com/translate/translate.git
```

This will retrieve the `master` branch of the Toolkit. Further Git instructions are also available.

Once you have the sources you have two options, a full install:

```
$ su
$ ./setup.py install
```

or, running the tools from the source directory:

```
$ su
$ pip install -e .
```

1.2.6 Verify installed version

To verify which version of the toolkit you have installed run:

```
$ prop2po --version
prop2po 3.1.1
```

1.2.7 Cleaning up existing installation

To remove old versions of the toolkit which you might have installed without a virtual environment or without your package manager.

The following advice only applies to manual installation from a tarball.

1. Find location of your python packages:

```
$ python -c "from distutils.sysconfig import get_python_lib; print(get_python_
↪lib())"
```

2. Delete toolkit package from your Python site-packages directory e.g.:

```
$ rm -R /usr/local/lib/python3.8/dist-packages/translate
```

1.3 Converters

1.3.1 General Usage

The tools follow a general usage convention which is helpful to understand.

Input & Output

The last two arguments of your command are the input and output files/directories:

```
moz2po <input> <output>
```

You can of course still use the `-i` and `-o` options which allows you to reorder commands

```
moz2po -o <output> -i <input>
```

Error Reporting

All tools accept the option `--errorlevel`. If you find a bug, add this option and send the traceback to the developers.

```
moz2po <other-options> --errorlevel=traceback
```

Templates

If you are working with any file format and you wish to preserve comments and layout then use your source file as a template.

```
po2dtd -t <source-file> <input> <output>
```

This will use the files in <source-file> as a template, merge the PO files in <input>, and create new DTD files in <output>

If you ran this without the templates you would get valid DTD files but they would not preserve the layout or all the comments from the source DTD file

The same concept of templates is also used when you merge files.

```
pomerge -t <old> <fixes> <new>
```

This would take the <old> files merge in the <fixes> and output new PO files, preserving formatting, into <new>. You can use the same directory for <old> and <new> if you want the merges to overwrite files in <old>.

source2target

The converters all follow this convention:

- source = the format from which you are converting e.g. in *oo2po* we are converting from OpenOffice.org SDF/GSI
- target = the format into which you are converting e.g. in *oo2po* we are converting to Gettext PO

Getting Help

The --help option will always list the available commands for the tool.

```
moz2po --help
```

1.3.2 moz2po

moz2po converts Mozilla files to PO files. It wraps converters that handle .properties, .dtd and some strange Mozilla files. The tool can work with files from Mozilla's Mercurial repository. The tools thus provides a complete roundtrip for Mozilla localisation using PO files and PO editors.

Note: This page should only be used as a reference to the command-line options for moz2po and po2moz. For more about using the Translate Toolkit and PO files for translating Mozilla products, please see the page on *Mozilla L10n Scripts*.

Usage

```
moz2po [options] <dir> <po>
po2moz [options] <po> <dir>
```

Where:

<dir>	is a directory containing valid Mozilla files
<po>	is a directory of PO or POT files

Options (moz2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in inc, it, *, dtd, properties formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in it.po, it.pot, manifest, xhtml.po, xhtml.pot, ini.po, ini.pot, rdf, js, *, html.po, html.pot, inc.po, inc.pot, dtd.po, dtd.pot, properties.po, properties.pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in it, *, properties, dtd, inc formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2moz):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in dtd.po, dtd.pot, ini.po, ini.pot, inc.po, inc.pot, manifest, it.po, it.pot, *, html.po, html.pot, js, rdf, properties.po, properties.pot, xhtml.po, xhtml.pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in dtd, *, inc, it, properties formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in dtd, *, inc, it, properties formats

-S, --timestamp skip conversion if the output file has newer timestamp

-l LOCALE, --locale=LOCALE set output locale (required as this sets the directory names)

--removeuntranslated remove untranslated strings from output

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

Creating POT files

See also:

Creating Mozilla POT files.

After extracting the en-US l10n files, you can run the following command:

```
moz2po -P l10n/en-US pot
```

This creates a set of POT (`-P`) files in the `pot` directory from the Mozilla files in `l10n/en-US` for use as PO Templates.

If you want to create a set of POT files with another base language try the following:

```
moz2po -P l10n/fr-FR fr-pot
```

This will create a set of POT files in `fr-pot` that have French as your source language.

Creating PO files from existing non-PO translations

If you have existing translations (Mozilla related or other Babelzilla files) and you wish to convert them to PO for future translation then the following generic instructions will work:

```
moz2po -t en-US af-ZA af-ZA_pofiles
```

This will combine the untranslated template en-US files from `en-US` combine them with your existing translations in `af-ZA` and output PO files to `af-ZA_pofiles`.

```
moz2po -t l10n/fr l10n/xh po/xh
```

For those who are not English fluent you can do the same with another languages. In this case `msgid` will contain the French text from `l10n/fr`. This is useful for translating where the translators other languages is not English but French, Spanish or Portuguese. Please make sure that the source languages i.e. the `msgid` language is fully translated as against en-US.

Creating Mercurial ready translations

```
po2moz -t l10n/en-US po/xh l10n/xh
```

Create Mozilla files using the templates files in `l10n/en-US` (see above for how to create them) with PO translations in `po/xh` and output them to `l10n/xh`. The files now in `l10n/xh` are ready for submission to Mozilla and can be used to build a language pack or translated version of Mozilla.

Issues

You can perform the bulk of your work (99%) with moz2po.

Localisation of XHTML is not yet perfect, you might want to work with the files directly.

Issue 203 tracks the outstanding features which would allow complete localisation of Mozilla including; all help, start pages, rdf files, etc. It also tracks some bugs.

Accesskeys don't yet work in .properties files and in several cases where the Mozilla .dtd files don't follow the normal conventions, for example in `security/manager/chrome/pippki/pref-ssl.dtd.po`. You might also want to check the files mentioned in this Mozilla bug 329444 where mistakes in the DTD-definitions cause problems in the matching of accelerators with the text.

You might want to give special attention to the following files since it contains customisations that are not really translations.

- mail/chrome/messenger/downloadheaders.dtd.po
- toolkit/chrome/global/intl.properties.po

Also, all width, height and size specifications need to be edited with feedback from testing the translated interfaces.

There are some constructed strings in the Mozilla code which we can't do much about. Take good care to read the localisation notes. For an example, see `mail/chrome/messenger/downloadheaders.dtd.po`. In that specific file, the localisation note from the DTD file is lost, so take good care of those.

The file extension of the original Mozilla file is required to tell the Toolkit how to do the conversion. Therefore, a file like foo.dtd must be named foo.dtd.po in order to *po2moz* to recognise it as a DTD file.

1.3.3 oo2po

Convert between OpenOffice.org GSI/SDF files and the PO format. This tool provides a complete roundtrip; it preserves the structure of the GSI file and creates completely valid PO files.

oo2xliff will convert the SDF files to XLIFF format.

Usage

```
oo2po [options] <sdf> <output>
po2oo [options] [-t <en-US.sdf>] -l <targetlang> <input> <sdf|output>
```

or for XLIFF files:

```
oo2xliff [options] -l <targetlang> <sdf> <output>
xliff2oo [options] [-t <en-US.sdf>] -l <targetlang> <input> <sdf|output>
```

Where:

<sdf>	is a valid OpenOffice.org GSI or SDF files
<output>	is a directory for the resultant PO/POT/XLIFF files
<input>	is a directory of translated PO/XLIFF files
<targetlang>	is the ISO 639 language code used in the sdf file, e.g. af

Options (oo2po and oo2xliff):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in oo, sdf formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot, xlf formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po) (only available in oo2po

-l LANG, --language=LANG set target language to extract from oo file (e.g. af-ZA) (required for oo2xliff)

--source-language=LANG set source language code (default en-US)

--nonrecursiveinput don't treat the input oo as a recursive store

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

--multifile=MULTIFILESTYLE how to split po/pot files (*single, toplevel or onefile*)

Options (po2oo and xliff2oo):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot, xlf formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in oo, sdf formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in oo, sdf formats

-S, --timestamp skip conversion if the output file has newer timestamp

-l LANG, --language=LANG set target language code (e.g. af-ZA) [required]

--source-language=LANG set source language code (default en-US)

-T, --keeptimestamp don't change the timestamps of the strings

--nonrecursiveoutput don't treat the output oo as a recursive store

--nonrecursivetemplate don't treat the template oo as a recursive store

--skipsource don't output the source language, but fallback to it where needed

--filteraction=ACTION action on pofilter failure: *none (default), warn, exclude-serious, exclude-all*

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

--multifile=MULTIFILESTYLE how to split po/pot files (*single, toplevel or onefile*)

Examples

These examples demonstrate most of the useful invocations of oo2po:

Creating POT files

```
oo2po -P en-US.sdf pot
```

Extract messages from *en-US.sdf* and place them in a directory called *pot*. The `-P` option ensures that we create POT files instead of PO files.

```
oo2po -P --source-language=fr fr-FR.sdf french-pot
```

Instead of creating English POT files we are now creating POT files that contain French in the msgid. This is useful for translators who are not English literate. You will need to have a fully translated sdf in the source language.

Creating PO files from existing work

```
oo2po --duplicates=merge -l zu zu-ZA.sdf zulu
```

Extract all existing Zulu (*zu*) messages from *zu-ZA.sdf* and place them in a directory called *zulu*. If you find duplicate messages in a file then merge them into a single message (This is the default behaviour for traditional PO files). You might want to use *pomigrate2* to ensure that your PO files match the latest POT files.:

```
cat GSI_af.sdf GSI_xh.sdf > GSI_af-xh.sdf
oo2po --source-language=af -l xh GSI_af-xh.sdf af-xh-po
```

Here we are creating PO files with your existing translations but a different source language. Firstly we combine the two SDF files. Then oo2po creates a set of PO files in *af-xh-po* using Afrikaans (*af*) as the source language and Xhosa (*xh*) as the target language from the combined SDF file *GSI_af-xh.sdf*

Creating a new GSI/SDF file

```
po2oo -l zu zulu zu_ZA.sdf
```

Using PO files found in *zulu* create an SDF files called *zu_ZA.sdf* for language *zu*:

```
po2oo -l af -t en-US.sdf --nofuzzy --keeptimestamp --filteraction=exclude-serious␣
↪afrikaans af_ZA.sdf
```

Create an Afrikaans (*af*) SDF file called *af_ZA.sdf* using *en-US.sdf* as a template and preserving the timestamps within the SDF file while also eliminating any serious errors in translation. Using templates ensures that the resultant SDF file has exactly the same format as the template SDF file. In an SDF file each translated string can have a timestamp attached. This creates a large amount of unuseful traffic when comparing version of the SDF file, by preserving the timestamp we ensure that this does not change and can therefore see the translation changes clearly. We have included the *nofuzzy* option (on by default) that prevent fuzzy PO messages from getting into the SDF file. Lastly the *filteraction* option is set to exclude serious errors: variables failures and translated XML will be excluded from the final SDF.

helpcontent2

The escaping of `helpcontent2` from SDF files was very confusing, issue 295 implemented a fix that appeared in version 1.1.0 (All known issues were fixed in 1.1.1). Translators are now able to translate helpcontent2 with clean escaping.

1.3.4 odf2xliff and xliff2odf

Convert OpenDocument (ODF) files to XLIFF localization files. Create translated ODF files by combining the original ODF files with XLIFF files containing translations of strings in the original document.

XLIFF is the XML Localization Interchange File Format developed by OASIS (The Organization for the Advancement of Structured Information Standards) to allow translation work to be standardised no matter what the source format and to allow the work to be freely moved from tool to tool.

If you are more used to software translation or l10n, you might want to read a bit about *Document translation*. This should help you to get the most out of translating ODF with XLIFF.

Usage

```
odf2xliff [options] <original_odf> <xliff>
xliff2odf [options] -t <original_odf> <xliff> <translated_odf>
```

Where:

<original_odf>	is an ODF document whose strings have to be translated
<xliff>	is an XLIFF file
<translated_odf>	is an ODF file to generate by replacing the strings in
	<original_odf> with the translated strings in <xliff>

Options (odf2xliff):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in ODF format

-o OUTPUT, --output=OUTPUT write to OUTPUT in XLIFF format

-S, --timestamp skip conversion if the output file has newer timestamp

Options (xliff2odf):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in XLIFF formats

-o OUTPUT, --output=OUTPUT write to OUTPUT in ODF format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ODF format

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

```
odf2xliff english.odt english_français.xlf
```

Create an XLIFF file from an ODT file (the source ODF file could also be any of the other ODF files, including ODS, ODG, etc.).

```
xliff2odf -t english.odt english_français.xlf français.odt
```

Using english.odt as the template document, and english_français.xlf as the file of translations, create a translated file français.odt.

Bugs

This filter is not yet extensively used – we appreciate your feedback. For more information on conformance to standards, see the *XLIFF* or *OpenDocument Format* pages.

1.3.5 prop2po

Convert between Java property files (.properties) and Gettext PO format.

Note: this tool completely eliminates the need for *native2ascii* as po2prop does the correct escaping to the Latin1 encoding that is needed by Java.

The following other formats are also supported via the *–personality* parameter:

- Adobe Flex
- Skype .lang
- Mac OS X .strings
- Mozilla .properties

Usage

```
prop2po [options] <property> <po>
po2prop [options] -t <template> <po> <property>
```

Where:

<property>	is a directory containing property files or an individual property file
<po>	is a directory containing PO files and an individual property file
<template>	is a directory of template property files or a single template property file

Options (prop2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in properties format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in properties format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--personality=TYPE override the input file format: *flex, java, mozilla, java-utf8, skype, gaia, strings* (for .properties files, default: java)

--encoding=ENCODING override the encoding set by the personality

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2prop):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in properties format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in properties format

-S, --timestamp skip conversion if the output file has newer timestamp

--personality=TYPE override the input file format: *flex, java, mozilla, java-utf8, skype, gaia, strings* (for .properties files, default: java)

--encoding=ENCODING override the encoding set by the personality (since 1.8.0)

--removeuntranslated remove untranslated strings from output

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

These examples demonstrate most of the useful invocations of prop2po:

Creating POT files

```
prop2po -P properties pot
```

Extract messages from *properties* directory and place them in a directory called *pot*. The `-P` option ensures that we create POT files instead of PO files.:

```
prop2po -P file.properties file.pot
```

Extract messages from *file.properties* and place them in *file.pot*.

Creating PO files from existing work

```
prop2po --duplicates=msgctxt -t reference zu zu-po
```

Extract all existing Zulu messages from *zu* directory and place the resultant PO files in a directory called *zu-po*. If you find duplicate messages in a file then use Gettext's mgsctxt to disambiguate them. During the merge we use the .properties files in *reference* as templates and as the source of the English text for the msgid. Once you have your PO files you might want to use *pomigrate2* to ensure that your PO files match the latest POT files.

Creating .properties files from your translations

```
po2prop -t reference zu-po zu
```

Using our translations found in *zu-po* and the templates found in *reference* we create a new set of property files in *zu*. These new property files will look exactly like those found in the templates, but with the text changed to the translation. Any fuzzy entry in our PO files will be ignored and any untranslated item will be placed in *zu* in English. The .properties file created will be based on the Java specification and will thus use escaped Unicode. Where:

```

```

Will appear in the files as:

```
\u1E7D\u1E01\u1E3D\u1E7B\u1E1D
```

To get output as used by Mozilla localisation do the following:

```
po2prop --personality=mozilla -t reference zu-po zu
```

This will do exactly the same as above except that the output will now appear as real Unicode characters in UTF-8 encoding.

Doing away with native2ascii

The native2ascii command is the traditional tool of property file localisers. With prop2po there is no need to use this command or to ever work directly with the escaped Unicode.

If you are working mostly with Gettext PO files then this is a double benefit as you can now use your favourite PO editor to translate Java applications. Your process would now look like this:

```
prop2po some.properties some.po
```

Firstly create a PO file that you can translate. Now translate it in your favourite PO editor.:

```
po2prop -t some.properties some.po some-other.properties
```

Using the original properties file as a template we preserve all layout and comments, combined with your PO translation we create a new translate properties file. During this whole process we have not needed to understand or process any escaping prop2po and po2prop handle that all automatically.

If you have existing translations you can recover them as follows:

```
prop2po -t some.properties translations.properties translations.po
```

This takes the default English properties file and combines it with your translate properties file and created a PO file. You now continue translating using your PO file.

1.3.6 php2po

Converts PHP localisable string arrays to Gettext PO format.

Usage

```
php2po [options] <php> <po>
po2php [options] <po> <php>
```

Where:

<php>	is a valid PHP localisable file or directory of those files
<po>	is a directory of PO or POT files

Options (php2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in php format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in php format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2php):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in php format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in php format

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy	use translations marked fuzzy
--nofuzzy	don't use translations marked fuzzy (default)

Formats Supported

Check *PHP format* document to see to which extent the PHP format is supported.

Examples

This example looks at roundtrip of PHP translations as well as recovery of existing translations.

First we need to create a set of POT files.:

```
php2po -P lang/en pot/
```

All .php files found in the `lang/en` directory are converted to Gettext POT files and placed in the `pot` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following:

```
php2po -t lang/en lang/zu po-zu/
```

Using the English PHP files found in `lang/en` and your existing Zulu translation in `lang/zu` we create a set of PO files in `po-zu`. These will now have your translations. Please be aware that in order for that to work 100% you need to have both English and Zulu at the same revision, if they are not you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2php -t lang/en po-zu/ lang/zu
```

Your translations found in the Zulu PO directory, `po-zu`, will be converted to PHP using the files in `lang/en` as templates and placing your new translations in `lang/zu`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

1.3.7 sub2po

sub2po allows you to use the same principles of PO files with *Subtitles*. In PO only items that change are marked fuzzy and only new items need to be translated, unchanged items remain unchanged for the translation.

Usage

```
sub2po [options] <foo.srt> <foo.po>
po2sub [options] [-t <foo.srt>] <XX.po> <foo-XX.srt>
```

Where:

foo.srt	is the input subtitle file
foo.po	is an empty PO file that may be translated
XX.po	is a PO file translated into the XX language
foo-XX.srt	is the foo.srt file translated into language XX

Options (sub2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in .srt format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ass, srt, ssa, sub formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2sub):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in srt format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in txt format

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

To create the POT files is simple:

```
sub2po -P SUBTITLE_FILE subtitles.pot
```

A translator would copy the POT file to their own PO file and then create translations of the entries. If you wish to create a PO file and not a POT file then leave off the `-P` option.

To convert back:

```
po2sub -t SUBTITLE_FILE   subtitles-XX.po  subtitles-XX.srt
```

Translating

Translate as normal. However, see the issues mentioned at *Subtitles*.

Bugs

There might be some issues with encodings, since the srt files don't specify them. We assume files to be encoded in UTF-8, so a conversion should solve this easily. Note that most of the handling of the srt files come from gaupol.

1.3.8 txt2po

txt2po allows you to use the same principles of PO files with normal text files. In PO only items that change are marked fuzzy and only new items need to be translated, unchanged items remain unchanged for the translation.

Usage

```
txt2po [options] <foo.txt> <foo.po>
po2txt [options] [-t <foo.txt>] <XX.po> <foo-XX.txt>
```

Where:

foo.txt	is the input plain text file
foo.po	is an empty PO file that may be translated
XX.po	is a PO file translated into the XX language
foo-XX.txt	is the foo.txt file translated into language XX

Options (txt2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in *, txt formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--encoding=ENCODING The encoding of the input file (default: UTF-8)

--flavour=FLAVOUR The flavour of text file: plain (default), dokuwiki, mediawiki

--no-segmentation Don't segment the file, treat it like a single message

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2txt):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in txt format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in txt format

-S, --timestamp skip conversion if the output file has newer timestamp

--encoding=ENCODING The encoding of the template file (default: UTF-8)

-w WRAP, --wrap=WRAP set number of columns to wrap text at

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

A roundtrip example

Preparing input files

With **txt2po** a text file is broken down into sections. Each section is separated by a line of whitespace. Each section will appear as a msgid in the PO file. Because of this simple method of breaking up the input file it might be necessary to alter the layout of your input file. For instance you might want to separate a heading from a paragraph by using whitespace.

For steps in a process you would want to leave a blank line between each step so that each step can be translated independently.

For a list of items you might want to group them together so that a translator could for example place them in alphabetic order for their translation.

Once the input file is prepared you can proceed to the next step.

Creating the POT files

This is simple:

```
txt2po -P TEXT_FILE text_file.pot
```

A translator would copy the POT file to their own PO file and then create translations of the entries. If you wish to create a PO file and not a POT file then leave off the `-P` option.

You might want to manually edit the POT file to remove items that should not be translated. For instance if part of the document is a license you might want to remove those if you do not want the license translated for legal reasons.

Translating

Translate as normal. However translators should be aware that writers of the text file may have used spaces, dashes, equals, underscores and other aids to indicate things such as:

```
* Headings and sub-headings
* Code examples, command lines examples
* Various lists
* etc
```

They will need to adapt these to work in their language being aware of how they will appear once they are merged with the original text document.

Creating a translated text file

With the translations complete you can create a translated text file like this:

```
po2txt -w 75 -t TEXT_FILE translated.po TEXT_FILE.translated
```

This uses the original text file as a template and creates a new translated text file using the translations found in the PO file.

The `-w` command allows you to reflow the translated text to `N` number of characters, otherwise the text will appear as one long line.

Help with Wiki syntax

dokuwiki

To retrieve the raw syntax for your dokuwiki page add '?do=export_raw' to you URL. The following would retrieve the DokuWiki home page in raw dokuwiki format https://www.dokuwiki.org/dokuwiki?do=export_raw

```
wget https://www.dokuwiki.org/dokuwiki?do=export_raw -O txt2po.txt
txt2po --flavour=dokuwiki -P txt2po.txt txt2po.pot
# edit txt2po.pot
po2txt -t txt2po.txt fr.po fr.txt
```

First we retrieve the file in raw dokuwiki format, then we create a POT file for editing. We created a French translation and using po2txt plus the original file as a template we output fr.txt which is a French version of the original txt2po.txt. This file can now be uploaded to the wiki server.

MediaWiki

To retrieve the raw media wiki syntax add '?action=raw' to you wiki URL. The following retrieves the Translate Toolkit page from Wikipedia in raw MediaWiki format http://en.wikipedia.org/wiki/Translate_Toolkit?action=raw or http://en.wikipedia.org/w/index.php?title=Pootle&action=raw.

To process follow the instructions above but substituting the MediaWiki retrieval method.

1.3.9 po2wordfast

Convert Gettext PO files to a *Wordfast Translation Memory* translation memory file.

Wordfast is a popular Windows based computer-assisted translation tool.

Usage

```
po2wordfast [options] --language <target> <po> <wordfast>
```

Where:

<po>	a PO file or directory
<wordfast>	a Wordfast translation memory file

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in tmx format

-S, --timestamp skip conversion if the output file has newer timestamp

-l LANG, --language=LANG set target language code (e.g. af-ZA) [required]

--source-language=LANG set source language code (default: en)

Examples

```
po2wordfast -l xh-ZA browser.po browser.txt
```

Use the Xhosa (*xh-ZA*) translations in the PO file *browser.po* to create a Wordfast translation memory file called *browser.txt*

1.3.10 po2tmx

Convert *Gettext PO* files to a *TMX* translation memory file. TMX is the Translation Memory eXchange format developed by OSCAR.

If you are interested in po2tmx, you might also be interested in *posegment* that can be used to perform some automated segmentation on sentence level.

Usage

```
po2tmx [options] --language <target> <po> <tmx>
```

Where:

<po>	is a PO file
<tmx>	is a TMX file

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in tmx format

-S, --timestamp skip conversion if the output file has newer timestamp

-l LANG, --language=LANG set target language code (e.g. af-ZA) [required]

--source-language=LANG set source language code (default: en)

--comments=COMMENT set default comment import: none, source, type or others (default: none)

Examples

```
po2tmx -l xh browser.po browser.tmx
```

Use the Xhosa (*xh*) translations in the PO file *browser.po* to create a TMX file called *browser.tmx*

Bugs and issues

Markup stripping

po2tmx conforms to TMX v1.4 without stripping markup. See the *TMX* conformance page for more details.

It has not been widely tested so your mileage may vary.

TMX and PO in OmegaT

In some tools, like OmegaT, PO files are parsed without expanding escaped sequences, even though such tools use TMX for translation memory. Keep this in mind when using po2tmx, because po2tmx converts `\n` and `\t` to newlines and tabs in the TMX file. If such a TMX file is used while translating PO files in OmegaT, matching will be less than 100%.

In other tools, such as Swordfish, the PO comment "no-wrap" is interpreted in the same way as the equivalent function in XML, which may also lead to mismatches if TMXes from po2tmx are used.

There is nothing wrong with po2tmx, but if used in conjunction with tools that handle PO files differently, it may lead to less than perfect matching.

Tips

TMX with only unique segments

To create a TMX with no duplicates (in other words, only unique strings), use msgcat to first create a large PO file with non-uniques removed.

1.3.11 pot2po

Convert a Gettext PO Template file to a PO file and merge in existing translations if they are present. A translation memory (compendium) can also be used for fuzzy matching. This corresponds to a large extent with the program "msgmerge" from the gettext package.

Note: This tool also works with translation formats other than Gettext PO, for example `XLIFF`.

Usage

```
pot2po [options] <pot> <po>
```

Where:

<pot>	is a PO Template (POT) file or directory of POT files
<po>	is a PO file or a directory of PO files

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in catkeys, lang, pot, ts, xlf, xliff formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in catkeys, lang, po, pot, ts, xlf, xliff formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in catkeys, lang, po, pot, ts, xlf, xliff formats (old translations)

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--tm=TM The file to use as translation memory when fuzzy matching

-s MIN_SIMILARITY, --similarity=MIN_SIMILARITY The minimum similarity for inclusion (default: 75%)

--nofuzzymatching Disable all fuzzy matching

Examples

```
pot2po -t zu-1.0.1 pot-2.0.2 zu-2.0.2
```

Here we are initialising the PO files in *zu-2.0.2* based on the POT files in *pot-2.0.2*. We are using the old translations in *zu-1.0.1* as templates so that we can reuse our existing translations in the new files.

pot2po can also be used to update against newer templates an existing translation file in a format different than Gettext PO, for example XLIFF:

```
pot2po -t af.xlf -i templates.xlf -o updated-af.xlf
```

If the POT files have undergone major reshuffling then you may want to use *pomigrate2* which can now use pot2po as its merging backend. pomigrate2 will do its best to migrate your files to the correct locations before merging. It will also make use of a compendium if requested.

```
pot2po --tm=compendium.po --similarity=60 -t xh-old pot xh-new
```

With this update we are using *compendium.po* as a translations memory (you can make use of other files such as TMX, etc). We will accept any match that scores above *60%*.

Merging

It helps to understand when and how pot2po will merge. The default is to follow msgmerge's behaviour but we add some extra features with fuzzy matching:

- If everything matches we carry that across
- We can resurrect obsolete messages for reuse
- Messages no longer used are made obsolete
- If we cannot find a match we will first look through the current and obsolete messages and then through any global translation memory
- Fuzzy matching makes use of the *Levenshtein distance* algorithm to detect the best matches

Performance

Fuzzy matches are usually of good quality. Installation of the python-Levenshtein package will speed up fuzzy matching. Without this a Python based matcher is used which is considerably slower.

Bugs

- *pomerge* and pot2po should probably become one.

1.3.12 csv2po

Convert between CSV (Comma Separated Value) files and the PO format. This is useful for those translators who can only use a Spreadsheet, a modern spreadsheet can open CSV files for editing. It is also useful if you have other data such as translation memory in CSV format and you wish to use it with your PO translations.

If you are starting out with your own CSV files (not created by po2csv), take note of the assumptions of the column layout explained below.

Usage

```
csv2po [options] <csv> <po>
po2csv [options] <po> <csv>
```

Where:

<csv>	is a file or directory containing CSV files
<po>	is a file or directory containing PO files

Options (csv2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in csv format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in po, pot, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--charset=CHARSET set charset to decode from csv files

--columnorder=COLUMNORDER specify the order and position of columns (location,source,target)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2csv):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in csv format

-S, --timestamp skip conversion if the output file has newer timestamp

--columnorder=COLUMNORDER specify the order and position of columns (location,source,target)

CSV file layout

The resultant CSV file has the following layout

Column	Data	Description
A	Location	All the PO #: location comments. These are needed to reconstruct or merge the CSV back into the PO file
B	Source Language	The msgid or source string
C	Target Language	The msgstr or target language

Examples

These examples demonstrate the use of csv2po:

```
po2csv -P pot csv
```

We use the `-P` option to recognise POT files found in *pot* and convert them to CSV files placed in *csv*:

```
csv2po csv po
```

Convert CSV files in *csv* to PO files placed in *po*:

```
csv2po --charset=windows-1250 -t pot csv po
```

User working on Windows will often return files encoded in everything but Unicode. In this case we convert CSV files found in *csv* from *windows-1250* to UTF-8 and place the correctly encoded files in *po*. We use the templates found in *pot* to ensure that we preserve formatting and other data. Note that UTF-8 is the only available destination encoding.

```
csv2po --columnorder=location,target,source fr.csv fr.po
```

In case the CSV file has the columns in a different order you may use `--columnorder`.

Bugs

- Translation comments #[space] and KDE comments _: are not available in CSV mode which effects the translators effectiveness
- Locations #: that are not conformant to PO (i.e. have spaces) will get messed up by PO tools.

1.3.13 csv2tbx

Convert between CSV (Comma Separated Value) files and the *TBX* format for terminology exchange.

Usage

```
csv2tbx [--charset=CHARSET] [--columnorder=COLUMNORDER] <csv> <tbx>
```

Where:

<csv>	is a CSV file
<tbx>	is the target TBX file

Options (csv2tbx):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in csv format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in tbx format

-S, --timestamp skip conversion if the output file has newer timestamp

--charset=CHARSET set charset to decode from csv files

--columnorder=COLUMNORDER specify the order and position of columns (comment,source,target)

CSV file layout

The CSV file is expected to have three columns (separated by commas, not other characters like semicolons), of which the default layout is

Column	Data	Description
A	Comment	All the PO #: location comments. These are not used in the TBX files, and can be left empty, but could be generated by *po2csv*
B	Source Language	The msgid or source string
C	Target Language	The msgstr or target language

Examples

These examples demonstrate the use of csv2tbx:

```
csv2tbx terms.csv terms.tbx
```

to simply convert *terms.csv* to *terms.tbx*.

To convert a directory recursively to another directory with the same structure of files:

```
csv2tbx csv-dir tbx-target-dir
```

This will convert CSV files in *csv-dir* to TBX files placed in *tbx-target-dir*.:

```
csv2tbx --charset=windows-1250 csv tbx
```

Users working on Windows will often return files in encoding other the Unicode based encodings. In this case we convert CSV files found in *csv* from *windows-1250* to UTF-8 and place the correctly encoded files in *tbx*. Note that UTF-8 is the only available destination encoding.

Two column CSV

```
csv2tbx --columnorder=source,target foo.csv foo.tbx
```

Notes

For conformance to the standards and to see which features are implemented, see *CSV* and *TBX*.

1.3.14 tbx2po

Convert between *TermBase eXchange (.tbx) glossary* format and *Gettext PO* format.

Usage

```
tbx2po <tbx> <po>
```

Where:

<tbx>	is a TBX file
<po>	is the target PO file

Options (tbx2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in csv format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in tbx format

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

These examples demonstrate the use of tbx2po:

```
tbx2po terms.tbx terms.po
```

to simply convert *terms.tbx* to *terms.po*.

To convert a directory recursively to another directory with the same structure of files:

```
tbx2po tbx-dir po-target-dir
```

This will convert TBX files in *tbx-dir* to PO files placed in *po-target-dir*.

Notes

For conformance to the standards and to see which features are implemented, see *PO Files* and *TBX*.

1.3.15 html2po

Convert translatable items in HTML to the PO format.

Usage

```
html2po [options] <html> <po>
po2html [options] <po> <html>
```

Where:

<html>	is an HTML file or a directory of HTML files
<po>	is a PO file or directory of PO files

Options (html2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in htm, html, xhtml formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

-u, --untagged include untagged sections

--keepcomments preserve html comments as translation notes in the output

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2html):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in htm, html, xhtml formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in htm, html, xhtml formats

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

```
html2po -P site pot
```

This will find all HTML files (.htm, .html, .xhtml) in *site*, convert them to POT files and place them in *pot*.

You can create and update PO files for different languages using the *pot2po* command.

```
po2html -t site -i xh -o site-xh
```

All the PO translations in *xh* will be converted to HTML using HTML files in *site* as templates and outputting new translated HTML files in *site-xh*.

Notes

The *HTML format description* gives more details on the format of the localisable HTML content and the capabilities of this converter.

Bugs

Some items end up in the msgid's that should not be translated

1.3.16 flatxml2po

Converts flat XML (.xml) files to Gettext PO format, a simple monolingual and single-level XML.

Usage

```
flatxml2po [options] <xml> <po>
po2flatxml [options] <po> <xml> [-t <base-xml>]
```

Where:

<xml>	is a valid .xml file or directory of those files
<po>	is a directory of PO or POT files
<base-xml>	is a template or the original file before translation. required for roundtrips preserving extraneous data.

Options (flatxml2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in xml format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-r ROOT, --root=ROOT name of the XML root element (default: "root")

-v VALUE, --value=VALUE name of the XML value element (default: "str")

-k KEY, --key=KEY name of the XML key attribute (default: "key")

-n NS, --namespace=NS XML namespace uri (default: None)

Options (po2flatxml):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in xml format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in xml format

-S, --timestamp skip conversion if the output file has newer timestamp

-r ROOT, --root=ROOT name of the XML root element (default: "root")

-v VALUE, --value=VALUE name of the XML value element (default: "str")

-k KEY, --key=KEY name of the XML key attribute (default: "key")

-n NS, --namespace=NS XML namespace uri (default: None)

-w INDENT, --indent=INDENT indent width in spaces, 0 for no indent (default: 2)

Formats Supported

Check *flat XML format* document to see to which extent the XML format is supported.

Examples

This example looks at roundtrip of flat XML translations as well as recovery of existing translations.

First we need to create a set of POT files.:

```
flatxml2po -P lang/en pot/
```

All .xml files found in the `lang/en` directory are converted to Gettext POT files and placed in the `pot` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following:

```
flatxml2po -t lang/en lang/zu po-zu/
```

Using the English XML files found in `lang/en` and your existing Zulu translation in `lang/zu` we create a set of PO files in `po-zu`. These will now have your translations. Please be aware that in order for that to work 100% you need to have both English and Zulu at the same revision, if they are not you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2flatxml -t lang/en po-zu/ lang/zu
```

Your translations found in the Zulu PO directory, `po-zu`, will be converted to XML using the files in `lang/en` as templates and placing your new translations in `lang/zu`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

Limitations

Indentation only supports spaces (specified with `--indent` greater than zero) or flattened (no indentation, everything on a single line; specified with `--indent` set to zero). Tabs are not supported using po2flatxml.

1.3.17 ical2po

New in version 1.2.

Converts iCalendar (*.ics) files to Gettext PO format.

Usage

```
ical2po [options] <ical> <po>
po2ical [options] -t <ical> <po> <ical>
```

Where:

<ical>	is a valid .ics file or directory of those files
<po>	is a directory of PO or POT files

Options (ical2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in ics format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ics format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2ical):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in ics format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ics format

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

This example looks at roundtrip of iCalendar translations. While you can do recovery of translations, its unlikely that you will ever need to do that.

First we need to create a set of POT files.

```
ical2po -P ical.ics ical.pot
```

The ical.ics file is converted to Gettext POT files called ical.pot. Directories of iCalendar files can also be processed.

Begin translating the ical.pot file by first copying it to make a PO file.

```
cp ical.pot ical-af.po
```

You are now in a position to translate the file ical-af.po in your favourite translation tool.

Once translated you can convert back as follows:

```
po2ical -t ical.ics ical-af.po ical-af.ics
```

Your translations found in the Afrikaans PO file, `ical-af.po`, will be converted to .ics using the file `ical.ics` as a template and creating your newly translated .ics file `ical-af.ics`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

Notes

The converter will only process events in the calendar file, the file itself can contain many other things that could be localisable. Please raise a bug if you want to extract additional items.

The converter does not make use of the LANGUAGE attribute which is permitted in the format. The LANGUAGE attribute does not aid multilingualism in this context so is ignored.

The converter could conceivably also process vCard files, but this has not been implemented for lack of a clear need. Please raise a bug with an example if you have such a file that could benefit from localisation.

1.3.18 ini2po

Converts .ini files to Gettext PO format.

Usage

```
ini2po [options] <ini> <po>
po2ini [options] -t <ini> <po> <ini>
```

Where:

<ini>	is a valid .ini file or directory of those files
<po>	is a directory of PO or POT files

Options (ini2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in ini, isl, iss formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ini, isl, iss formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2ini):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in ini, isl formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ini, isl formats

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Formats Supported

INI files need to be organized into separate languages per file and in the following format:

```
[Section]
  ; a comment
  a = a string
```

Comment marked with the hash symbol (#) are also allowed, and the colon (:) is also accepted as key-value delimiter:

```
[Section]
  # another comment
  b : a string
```

This variants in comment marks and key-value delimiters can be mixed in one single INI file:

```
[Section]
; a comment
a = a string
# another comment
b : a string
c:'other example with apostrophes'
d:"example with double quotes"
```

The spacing between the key-value delimiter and the key, and the between the value and the key-value delimiter is not important since the converter automatically strips the blank spaces.

Note: A section must be present at the file beginning in order to get ini2po working properly. You may add it by hand at the file beginning.

Note: Strings marked with double quotes and/or apostrophes will carry these quotation marks to the generated .po file, so they will appear like:

```
#: [Section]c
msgid "'other example with apostrophes'"
msgstr ""

#: [Section]d
msgid "\"example with double quotes\""
msgstr ""
```

Examples

This example looks at roundtrip of .ini translations as well as recovery of existing translations.

First we need to create a set of POT files.

```
ini2po -P ini/ pot/
```

All .ini files found in the `ini/` directory are converted to Gettext POT files and placed in the `pot/` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following:

```
ini2po -t lang/ zu/ po-zu/
```

Using the English .ini files found in `lang/` and your existing Zulu translation in `zu/` we create a set of PO files in `po-zu/`. These will now have your translations. Please be aware that in order for the to work 100% you need to have both English and Zulu at the same revision. If they are not, you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2ini -t lang/ po-zu/ zu/
```

Your translations found in the Zulu PO directory, `po-zu/`, will be converted to .ini using the files in `lang/` as templates and placing your newly translated .ini files in `zu/`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

Issues

We do not extract comments from .ini files. These are sometimes needed as developers provide guidance to translators in these comments.

1.3.19 json2po

Converts .json files to Gettext PO format.

Usage

```
json2po [options] <json> <po>
po2json [options] -t <json> <po> <json>
```

Where:

<json>	is a valid .json file or directory of those files
<po>	is a directory of PO or POT files

Options (json2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in JSON format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in JSON format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--filter=FILTER leaves to extract e.g. 'name,desc': (default: extract everything)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2json):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in JSON format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in JSON format

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

--removeuntranslated remove untranslated strings from output

Examples

This example looks at roundtrip of .json translations as well as recovery of existing translations.

First we need to create a set of POT files.

```
json2po -P json/ pot/
```

All .json files found in the `json/` directory are converted to Gettext POT files and placed in the `pot/` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following:

```
json2po -t lang/ zu/ po-zu/
```

Using the English .json files found in `lang/` and your existing Zulu translation in `zu/` we create a set of PO files in `po-zu/`. These will now have your translations. Please be aware that in order for the to work 100% you need to have both English and Zulu at the same revision. If they are not, you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2json -t lang/ po-zu/ zu/
```

Your translations found in the Zulu PO directory, `po-zu/`, will be converted to .json using the files in `lang/` as templates and placing your newly translated .json files in `zu/`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

1.3.20 web2py2po

Converts web2py translation files to PO files and vice versa.

Web2py, formerly known as Gluon) is an open-source, Python-based web application framework by Massimo Di Pierro (inspired by Django and Rails).

Web2py uses an internal localization engine based on Python dictionaries, which is applied with the T() lookup function. Web2py provides a built-in translation interface for the T()-engine, which is excellent for rapid application development.

On the other hand, for collaboration and workflow control in a wider community you might probably rather want to use Pootle, Launchpad or similar facilities for translation, thus need to transform the web2py dictionaries into PO files and vice versa. And exactly that is what the web2py2po converters are good for.

Usage

```
web2py2po [options] <web2py> <po>
po2web2py [options] <po> <web2py>
```

Where:

<web2py>	is a valid web2py translation file
<po>	is a PO or POT file or a directory of PO or POT files

Options (web2py2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in php format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2web2py):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in php format

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Notes

Handling of blanks/untranslated messages:

Untranslated messages in the web2py translation files are usually marked with a leading %%"*** "%%, so:

- All target strings from the web2py sources with a leading %%"*** "%% are inserted as blank msgstr's into the PO result (web2py2po)
- Blank msgstr's from the PO file will get the msgid string with a leading %%"*** "%% as target string in the web2py result (po2web2py)

1.3.21 rc2po

Converts Windows Resource .rc files to Gettext PO format.

Usage

```
rc2po [options] <rc> <po>
po2rc [options] -t <rc> <po> <rc>
```

Where:

<rc>	is a valid Windows Resource file or directory of those files
<po>	is a directory of PO or POT files

Options (rc2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in rc format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in rc format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--charset=CHARSET charset to use to decode the RC files (default: cp1252)

-l LANG, --lang=LANG LANG entry (default: LANG_ENGLISH)

--sublang=SUBLANG SUBLANG entry (default: SUBLANG_DEFAULT)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2rc):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in rc format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in rc format

-S, --timestamp skip conversion if the output file has newer timestamp

--charset=CHARSET charset to use to decode the RC files (default: utf-8)

-l LANG, --lang=LANG LANG entry

--sublang=SUBLANG SUBLANG entry (default: SUBLANG_DEFAULT)

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Formats Supported

Note: This implementation is based mostly on observing WINE .rc files, these should mimic other non-WINE .rc files.

Examples

This example looks at roundtrip of Windows Resource translations as well as recovery of existing translations.

First we need to create a set of POT files.

```
rc2po -P lang/ pot/
```

All .rc files found in the `lang/` directory are converted to Gettext POT files and placed in the `pot/` directory.

If you are translating for the first time then you can skip the next step. If you need to recovery your existing translations then we do the following:

```
rc2po -t lang zu po-zu/
```

Using the English .rc files found in `lang` and your existing Zulu translation in `zu` we create a set of PO files in `po-zu`. These will now have your translations. Please be aware that in order for the to work 100% you need to have both English and Zulu at the same revision, if they are not you will have to review all translations. Also the .rc files may be in different encoding, we cannot at the moment process files of different encodings and assume both are in the same encoding supplied.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2rc -t lang/ po-zu/ zu/
```

Your translations found in the Zulu PO directory, `po-zu`, will be converted to .rc using the files in `lang/` as templates and placing your new translations in `zu/`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

Issues

If you are recovering translation using `rc2po -t en.rc xx.rc xx.po` then both en.rc and xx.rc need to be in the same encoding.

There might be problems with MENUs that are deaply nested.

1.3.22 resx2po

Converts .Net Resource (.resx) files to Gettext PO format, a monolingual file format used in Microsoft .Net Applications.

Usage

```
resx2po [options] <resx> <po>
po2resx [options] <po> <resx> -t <resx>
```

Where:

<resx>	is a valid .resx file or directory of those files
<po>	is a directory of PO or POT files

Options (resx2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in RESX format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in RESX format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--filter=FILTER leaves to extract e.g. 'name,desc': (default: extract everything)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): merge, msgctxt (default: 'msgctxt')

Options (po2resx):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in RESX format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in RESX format

-S, --timestamp skip conversion if the output file has newer timestamp

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

This example looks at roundtrip of .resx translations as well as recovery of existing translations.

First we need to create a set of POT files

```
resx2po -P resx/ pot/
```

All .resx files found in the `resx/` directory are converted to Gettext POT files and placed in the `pot/` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following

```
resx2po zu/ po-zu/ -t lang/
```

Using the English .resx files found in `lang/` and your existing Zulu translation in `zu/` we create a set of PO files in `po-zu/`. These will now have your translations. Please be aware that in order for the to work 100% you need to have both English and Zulu at the same revision. If they are not, you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2resx po-zu/ zu/ -t lang/
```

Your translations found in the Zulu PO directory, `po-zu/`, will be converted to .resx using the files in `lang/` as templates and placing your newly translated .resx files in `zu/`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

1.3.23 symb2po

New in version 1.3.

Converts Symbian-style translation files to PO files and vice versa. The Symbian translation files currently have a strong Buddycloud flavour, but the tools will be made more general as the need arises.

Usage

```
symb2po [options] [-t <target_lang_symb>] <source_lang_symb> <po>
po2symb [options] -t <target_lang_symb> <po> <target_lang_symb>
```

Where:

<target_lang_symb>	is a valid Symbian translation file or directory of those files
<source_lang_symb>	is a valid Symbian translation file or directory of those files
<po>	is a PO or POT file or a directory of PO or POT files

Options (symb2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in php format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in the Symbian translation format

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2symb):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in php format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in the Symbian translation format

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

symb2po

The most common use of symb2po, is to generate a POT (PO template) file from the English translation (note that the tool currently expects the Symbian translation file to end with the extension .r01, which is the code for English translation files). This file then serves as the source document from which all translations will be derived.

To create a POT file called `my_project.pot` from the source Symbian translation file `my_project.r01`, the following is executed:

```
symb2po my_project.r01 my_project.pot
```

In order to re-use existing translations in the Symbian translation format, symb2po can merge that translation into the source Symbian translation to produce a translated PO file. The existing Symbian translation file is specified with the `-t` flag.

To create a file called `my_project-en-fr.po` (this is not the recommended PO naming convention) from the source Symbian translation file `my_project.r01` and its French translation `my_project.r02`, execute:

```
symb2po -t my_project.r02 my_project.r01 my_project-en-fr.po
```

Note: Ensure that the English and French files are well aligned, in other words, no changes to the source text should have happened since the translation was done.

---

po2symb

The po2symb tool is used to extract the translations in a PO into a template Symbian translation file. The template Symbian translation file supplies the "shape" of the generated file (formatting and comments).

In order to produce a French Symbian translation file using the English Symbian translation file `my_project.r01` as a template and the PO file `my_project-en-fr.po` (this is not the recommended PO naming convention) as the source document, execute:

```
po2symb -t my_project.r01 my_project-en-fr.po my_project.r02
```

Notes

The tools won't touch anything appearing between lines marked as:

```
// DO NOT TRANSLATE
```

The string `r_string_languagegroup_name` is used to set the `Language-Team` PO header field.

The Symbian translation header field `Author` is used to set the `Last-Translator` PO header field.

Issues

The file format is heavily tilted towards the Buddycould implementation

The tools do nothing with the `Name` and `Description` Symbian header fields. This means that `po2symb` will just copy the values in the supplied template. So you might see something such as:

```
Description : Localisation File : English
```

in a generated French translation file.

Bugs

Probably many, since this software hasn't been tested much yet.

1.3.24 tiki2po

Converts TikiWiki language.php files to Gettext PO format.

Usage

```
tiki2po [options] <tiki> <po>
po2tiki [options] <po> <tiki>
```

Where:

<tiki>	is a valid language.php file for TikiWiki
<po>	is a PO file

Options (tiki2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in php format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

--include-unused When converting, include strings in the "unused" section?

Options (po2tiki):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in php format

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

These examples demonstrate the use of tiki2po:

```
tiki2po language.php language.po
```

Convert the tiki language.php file to .po:

```
po2tiki language.po language.php
```

Convert a .po file to a tiki language.php file

Notes

- Templates are not currently supported.

1.3.25 ts2po

Convert Qt .ts localization files to Gettext .po format files using ts2po and convert the translated *PO Files* files back to *Qt .ts* using po2ts.

The Qt toolkit comes with a localization application, Qt Linguist, however you might wish to standardise on one localization tool. ts2po allows you to standardise on the PO format and PO related tools.

Note: Virtaal and Pootle can edit .ts files directly without the need for any conversion.

Warning: po2ts uses our older .ts support. Thus many of the newer features in .ts are not supported. To support those features rather edit directly in Virtaal or Pootle.

Usage

```
ts2po [options] <ts> <po>
po2ts [options] <po> <ts>
```

Where:

<ts>	is a Qt .ts file or directory that contains .ts files
<po>	is a PO file or a directory of PO files

Options (ts2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in ts format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2ts):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in ts format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in ts format

-S, --timestamp skip conversion if the output file has newer timestamp

-c CONTEXT, --context=CONTEXT use supplied context instead of the one in the .po file comment

Examples

```
ts2po -P psi.ts psi.pot
```

This will create a POT file called *psi.pot* from the Qt .ts file called *psi.ts*.

```
po2ts af.po psi_af.ts
```

Now take your translated PO files *af.po* and convert it into a translated Qt .ts file, *psi_af.ts*.

Note: You need to use the tools from the Qt toolkit to create the compiled .qm language files for the application.

Bugs

There are probably still some bugs related to migrating the various attributes across for the different formats. The converters don't support all the newer features of the TS format, whereas the native support of Virtaal and Pootle is much better.

1.3.26 xliff2po

Converts XLIFF localization files to Gettext PO files. XLIFF is the XML Localization Interchange File Format developed by OASIS (Organization for the Advancement of Structured Information Standards) to allow translation work to be standardised no matter what the source format and to allow the work to be freely moved from tool to tool.

Usage

```
po2xliff [options] <po> <xliff>
xliff2po [options] <xliff> <po>
```

Where:

<po>	is a PO file or directory of PO files
<xliff>	is an XLIFF file or directory of XLIFF files

Options (xliff2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in xliff format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2xliff):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in xliff format

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in xliff format

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

```
xliff2po -P xliff pot
```

Create POT files from the XLIFF files found in directory *xliff* and output them to the directory *pot*

```
po2xliff xh xh-xlf
```

Convert the Xhosa PO files in *xh* to XLIFF and place them in *xh-xlf*

Bugs

This filter is not yet extensively used... expect bugs. See *XLIFF* to see how well our implementation conforms to the standard.

The PO plural implementation is still very new and needs active testing.

1.3.27 yaml2po

New in version 2.2.6.

Converts YAML localization files to Gettext PO format.

Usage

```
yaml2po [options] <yml> <po>
po2yaml [options] <po> <yml>
```

Where:

<yml>	is a valid YAML localisable file or directory of those files
<po>	is a directory of PO or POT files

Options (yaml2po):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in yaml, yml formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in yaml, yml formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

--duplicates=DUPLICATESTYLE what to do with duplicate strings (identical source text): *merge, msgctxt* (default: 'msgctxt')

Options (po2yaml):

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in yaml, yml formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in yaml, yml formats

-S, --timestamp skip conversion if the output file has newer timestamp

--threshold=PERCENT only convert files where the translation completion is above PERCENT

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Formats Supported

Check *YAML format* document to see to which extent the YAML format is supported.

Examples

This example looks at roundtrip of YAML translations as well as recovery of existing translations.

First we need to create a set of POT files:

```
yaml2po -P lang/en pot/
```

All .yml files found in the `lang/en` directory are converted to Gettext POT files and placed in the `pot` directory.

If you are translating for the first time then you can skip the next step. If you need to recover your existing translations then we do the following:

```
yaml2po -t lang/en lang/zu po-zu/
```

Using the English YAML files found in `lang/en` and your existing Zulu translation in `lang/zu` we create a set of PO files in `po-zu`. These will now have your translations. Please be aware that in order for that to work 100% you need to have both English and Zulu at the same revision, if they are not you will have to review all translations.

You are now in a position to translate your recovered translations or your new POT files.

Once translated you can convert back as follows:

```
po2yaml -t lang/en po-zu/ lang/zu
```

Your translations found in the Zulu PO directory, `po-zu`, will be converted to YAML using the files in `lang/en` as templates and placing your new translations in `lang/zu`.

To update your translations simply redo the POT creation step and make use of *pot2po* to bring your translation up-to-date.

1.3.28 –errorlevel=ERRORLEVEL

This is a parameter that can be passed to most of the programs in the translate toolkit in order to choose the level of feedback that you need when errors occur. It is mostly useful for debugging. Please report your errors to the developers with `--errorlevel=traceback`.

none

Display no error messages

message

Display on the error message

```
An error occurred processing PO file
```

exception

Give the error message and name and Python exception

```
ValueError: An error occurred processing PO file
```

traceback

Provide a full traceback for debugging purposes

```
csv2po: warning: Error processing: nso/readlicense_oo/docs/readme.csv: Traceback␣
↪(most recent call last):

  File "/usr/lib/python2.4/site-packages/translate/misc/optrecurse.py", line 415, in␣
↪recursiveprocess
    success = self.processfile(fileprocessor, options, fullinputpath, fulloutputpath,␣
↪fulltemplatepath)

  File "/usr/lib/python2.4/site-packages/translate/misc/optrecurse.py", line 468, in␣
↪processfile
    if fileprocessor(inputfile, outputfile, templatefile, **passthroughoptions):

  File "/usr/lib/python2.4/site-packages/translate/convert/csv2po.py", line 183, in␣
↪convertcsv
    outputpo = convertor.convertfile(inputcsv)

  File "/usr/lib/python2.4/site-packages/translate/convert/csv2po.py", line 159, in␣
↪convertfile
    raise ValueError("An error occurred processing PO file")
```

(continues on next page)

(continued from previous page)

```
ValueError: An error occurred processing PO file
```

1.3.29 –duplicates=DUPLICATESTYLE

Gettext PO files only allow one message with a common msgid (source string). Many other formats allow duplicate entries. To create a valid PO file you need to merge these duplicate entries into one PO message. However, this often negatively affects the roundtrip or is not what is expected by the user. Thus we have a number of methods of handling duplicates which we call *duplicate styles*.

Also affected are conversions in which the source format is empty (allowing possible translation). As the header in a PO file is identified by an empty source string, your message will appear to be a duplicate of the header. In this case duplicate removal is critical.

Previously the tools used msgid_comment (KDE style comments) to disambiguate text. However, with the release of Gettext 0.15, the new msgctxt disambiguation is now recommended, especially if you wish to use your files with other Gettext the tools. Many other pieces of software now also support this feature, and will probably become the best choice for almost all circumstances. It is the default in our converters.

merge

This is the traditional Gettext approach. All messages with the same source string or English string are merged into one PO message.

```
#: file1.dtd:instruction_manual
#: file1.dtd:manual_process
msgid "Manual"
msgstr ""
```

If however the source text is blank (these are often configuration options in Mozilla) then the *merge* style will use KDE comments as used in the *msgid_comment* style in order to create unambiguous entries that can still be used for configuration.

```
#: file1.dtd:translators_name
msgid "_: file1.dtd:translators_name\n"
msgstr ""

#: file1.dtd:translators_email
msgid "_: file1.dtd:translators_email\n"
msgstr ""
```

msgctxt (default)

This uses the msgctxt feature of Gettext that was introduced with Gettext 0.15. Some tools might not support it 100%. This option is the default in recent releases of the Translate Toolkit.

```
#: file1.dtd:instruction_manual
msgctxt "instruction_manual"
msgstr "Manual"
msgstr ""

#: file1.dtd:manual_process
```

(continues on next page)

(continued from previous page)

```
msgctxt "manual_process"
msgid "Manual"
msgstr ""
```

1.3.30 –progress=PROGRESS

All of the programs can give visual feedback. This options allows you to select the style of that feedback.

In the examples we are converting and OpenOffice.org 2.0 sdf/gsi file into POT files using *oo2po*.

none

No visual feedback, this is useful if you want to use any of the scripts as part of another script and don't want feedback to interfere with the operation.

```
$ oo2po -P --progress=none en-US.sdf pot
$
```

dots

Use visual dots to represent progress. Each dot represent a file that has been processed.

```
$ oo2po -P --progress=dots en-US.sdf pot
.....................................................................................
 ↪.......
.....................................................................................
 ↪.......
.....................................
$
```

bar (default)

Use a progress bar consisting of hashes (#) to show progress.

```
$ oo2po -P --progress=bar en-US.sdf pot
processing 227 files...
[##############################             ]  69%
```

This is the default mode of operation, therefore this command would create the same output.

```
$ oo2po -P en-US.sdf pot
```

verbose

Combine the hash (#) progress bar form the *bar* option with the actual names of files that have been processed.

```
$ oo2po -P --progress=verbose en-US.sdf pot
processing 227 files...
so3/src.oo
```

(continues on next page)

(continued from previous page)

```
dbaccess/source/ui/uno.oo
helpcontent2/source/text/shared.oo
wizards/source/formwizard.oo
sch/source/ui/dlg.oo
helpcontent2/source/text/sbasic/shared/01.oo
dbaccess/source/core/resource.oo
svtools/source/sbx.oo
dbaccess/source/ui/relationdesign.oo
scp2/source/writer.oo
filter/source/xsltdialog.oo
[##                                          ]   5%
```

names

Prints out only the filenames without any other progress indicator. This is a good option when outputting to a log file rather than a terminal.

```
$ oo2po -P --progress=names en-US.sdf pot
so3/src.oo
dbaccess/source/ui/uno.oo
helpcontent2/source/text/shared.oo
wizards/source/formwizard.oo
sch/source/ui/dlg.oo
helpcontent2/source/text/sbasic/shared/01.oo
dbaccess/source/core/resource.oo
svtools/source/sbx.oo
dbaccess/source/ui/relationdesign.oo
scp2/source/writer.oo
filter/source/xsltdialog.oo
```

1.3.31 –filteraction=ACTION

none (default)

Take no action. Messages from failing test will appear in the output file

warn

Print a warning but otherwise include the message in the output file.

exclude-serious

Only exclude errors that are listed as serious by the convertor. All other are included.

exclude-all

Exclude any message that fails a test.

1.3.32 –multifile=MULTIFILESTYLE

This options determines how the POT/PO files are spli from the source files. In many cases you have source files that generate either too many small files or one large files which you would rather see split up into smaller files.

single

Output individual files.

toplevel

Split the source files at the top level. Ie you see a number of top level files.

onefiles

One large file instead of many smaller files.

1.3.33 –personality=TYPE

java (default)

Create output strictly according to the specification for .properties files. This will use escaped Unicode for any non-ASCII characters. Thus the following string found in a PO file:

```
```

Will appear as follows in the output .properties file:

```
\u1E7D\u1E01\u1E3D\u1E7B\u1E1D
```

mozilla

Mozilla has made slight adjustments to the Java .properties spec. Mozilla will accept UTF-8 encoded strings in the property file and thus does not need escaped Unicode. Thus the above string – – will not be escaped. Mozilla property files are thus more useful for non-Latin languages in that they are actually readable.

Of course this style of file is only used by Mozilla and should not be used for other projects that follow the Java spec more strictly.

skype

Skype .lang files are .properties files in UTF-16. The & is used as an accelerator (marked in the PO header).

flex

Flex follows the Mozilla approach, a UTF-8 encoded file with no escaped unicode. We include it as its own dialect for ease of use.

strings

Much Mac OS X and iPhone software is translated using .strings files. These are quite different from properties files and we treat them here as key value files.

The files are in UTF-16 with a few minor escaping conventions.

1.3.34 –accelerator=ACCELERATOR

Accelerator Marker	Used by
&	KDE Desktop and Mozilla (when using *moz2po*)
_	GNOME Desktop and other GTK+ based applications
~	LibreOffice and Apache OpenOffice

Converters change many different formats to PO and back again. Sometimes only one direction is supported, or conversion is done using non-PO formats. The converters follow a *general pattern of usage*, understanding that will make the converters much easier to use and understand.

- *moz2po* – Mozilla .properties and .dtd converter. Works with Firefox and Thunderbird
- *oo2po* – OpenOffice.org SDF converter (Also works as `oo2xliff`).
- *odf2xliff* – Convert OpenDocument (ODF) documents to XLIFF and vice-versa.
- *prop2po* – Java property file (.properties) converter
- *php2po* – PHP localisable string arrays converter.
- *sub2po* – Converter for various subtitle files
- *txt2po* – Plain text to PO converter
- *po2wordfast* – Wordfast Translation Memory converter
- *po2tmx* – TMX (Translation Memory Exchange) converter
- *pot2po* – initialise PO Template files for translation
- *csv2po* – Comma Separated Value (CSV) converter. Useful for doing translations using a spreadsheet.
- *csv2tbx* – Create TBX (TermBase eXchange) files from Comma Separated Value (CSV) files
- *html2po* – HTML converter
- *flatxml2po* – Flat XML converter
- *ical2po* – iCalendar file converter
- *ini2po* – Windows INI file converter
- *json2po* – JSON file converter
- *web2py2po* – web2py translation to PO converter
- *rc2po* – Windows Resource .rc (C++ Resource Compiler) converter
- *resx2po* – .Net Resource (.resx) file converter
- *symb2po* – Symbian-style translation to PO converter
- *tiki2po* – TikiWiki language.php converter
- *ts2po* – Qt Linguist .ts converter

- *xliff2po* – XLIFF (XML Localisation Interchange File Format) converter
- *yaml2po* – YAML (Yet Another Markup Language) converter

1.4 Tools

The PO tools allow you to manipulate and work with PO files

1.4.1 Quality Assurance

poconflicts

poconflicts takes a PO file and creates an set of output PO files that contain messages that conflict. During any translation project that involves a large amount of work or a number of translators you will see message conflicts. A conflict is where the same English message has been translated differently (in some languages this may have been intentional). Conflicts occur due to different translation style or a shift in translations as the translators or project mature.

poconflicts allows you to quickly identify these problem messages, investigate and correct them. To merge the files back, they have to be restructured into the correct directory structure using *porestructure* in order to enable merging using *pomerge*.

Usage

```
poconflicts [options] <po> <conflicts>
```

Where:

<po>	is a directory of existing PO files or an individual PO file
<conflicts>	is a directory containing one PO file for each conflict

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po format

-I, --ignore-case ignore case distinctions

-v, --invert invert the conflicts thus extracting conflicting destination words

--accelerator=ACCELERATORS ignores the given *accelerator characters* when matching

Examples

Here are some examples that demonstrate the usefulness of poconflict

```
poconflicts --accelerator=~ -I xhosa conflicts
```

This extracts messages from the PO files in the *xhosa* directory and places a new PO file for each identified conflict in *conflicts*. We are working with OpenOffice files and we therefore use the tilde (~) as the accelerator marker (with this set *F~ile* is considered the same as *~File*). We are also ignoring the case of the message using `-I` (thus *File* is considered the same as *file* or *FILE*)

Another useful option is to look at the inverted conflicts. This will detect target words that have been used to translate different source words.

```
poconflicts --accelerator=~ -I -v xhosa conflicts
```

Now in the *conflicts* directory we will find PO files based on the Xhosa word. We can now check where a Xhosa word has been used for different source or English words. Often there is no problem but you might find cases where the same Xhosa word was used for Delete and Cancel – clearly a usability issue.

The translator makes the needed corrections to the files and then we can proceed to merge the results back into the PO files. Unchanged entries can be removed.

Now restructure the files to resemble the original directory structure using *porestructure*:

```
porestructure -i conflicts -o conflicts_tree
```

Now merge the changes back using pomerge:

```
pomerge -t xhosa -i conflicts_tree -o xhosa
```

This takes the corrected files from *conflicts_tree* and merge them into the files in *xhosa* using the same files as templates.

pofilter

Pofilter allows you to run a *number of checks* against your PO, XLIFF or TMX files. These checks are designed to pick up problems with capitalisation, accelerators, variables, etc. Those messages that fail any of the checks are output and marked so that you can correct them.

Use `pofilter -l` to get a list of available checks.

Once you have corrected the errors in your PO files you can merge the corrections into your existing translated PO files using *pomerge*.

Usage

```
pofilter [options] <in> <out>
```

Where:

<in>	the input file or directory which contains PO or XLIFF files
<out>	the output file or directory that contains PO or XLIFF files that fail the various tests

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in pot, po, xlf, tmx formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot, xlf, tmx formats

-l, --listfilters list filters available

--review include elements marked for review (default)

--noreview exclude elements marked for review

--fuzzy include elements marked fuzzy (default)

--nofuzzy exclude elements marked fuzzy

--nonotes don't add notes about the errors (since version 1.3)

--autocorrect output automatic corrections where possible rather than describing issues

--language=LANG set target language code (e.g. af-ZA) [required for spell check]. This will help to make pofilter aware of the conventions of your language

--openoffice use the standard checks for OpenOffice translations

--libreoffice use the standard checks for LibreOffice translations

--mozilla use the standard checks for Mozilla translations

--drupal use the standard checks for Drupal translations

--gnome use the standard checks for Gnome translations

--kde use the standard checks for KDE translations

--wx use the standard checks for wxWidgets translations – identical to –kde

--excludefilter=FILTER don't use FILTER when filtering

-t FILTER, --test=FILTER only use test FILTERs specified with this option when filtering

--notranslatefile=FILE read list of untranslatable words from FILE (must not be translated)

--musttranslatefile=FILE read list of translatable words from FILE (must be translated)

--validcharsfile=FILE read list of all valid characters from FILE (must be in UTF-8)

Example

Here are some examples to demonstrate how to use pofilter:

```
pofilter --openoffice af af-check
```

Use the default settings (accelerator and variables) for OpenOffice.org. Check all PO files in *af* and output any messages that fail the check in *af-check* (create the directory if it does not already exist).

```
pofilter -t isfuzzy -t untranslated zu zu-check
```

Only run the *isfuzzy* and *untranslated* checks, this will extract all messages that are either fuzzy or untranslated.

```
pofilter --excludefilter=simplecaps --nofuzzy nso nso-check
```

Run all filters except *simplecaps*. You might want to do this if your language does not make use of capitalisation or if the test is creating too many false positives. Also only run the checks against messages that are not marked fuzzy. This is useful if you have already marked problem strings as fuzzy or you know that the fuzzy strings are bad, with this option you don't have to see the obviously wrong messages.

```
pofilter --language=fr dir dir-check
```

Tell pofilter that you are checking French translations so that it can take the conventions of the language into account (for things like punctuation, spacing, quoting, etc.) It will also disable some tests that are not meaningful for your language, like capitalisation checks for languages that don't have capital letters.

```
pofilter --excludefilter=untranslated
```

Tell pofilter not to complain about your untranslated units.

```
pofilter -l
```

List all the available checks.

Bugs

There are minor bugs in the filters. Most relate to false positives, corner cases or minor changes for better fault description.

Descriptions of all pofilter tests

The following are descriptions of the tests available in *pofilter*, Pootle and Virtaal with some details about what type of errors they are useful to test for and the limitations of each test.

Keep in mind that the software might point to errors which are not necessarily wrong (false positives).

Currently there are 48 tests. You can always get a list of the currently available tests by running:

```
pofilter -l
```

To see test specific to a specific targeted application or group of applications run:

```
pofilter --gnome -l
```

Adding new tests and new language adaptations

If you have an idea for a new test or want to add target language adaptations for your language then please help us with information about your test idea and the specifics of your language.

Test Classification

Some tests are more important than others so we have classified them to help you determine which to run first.

- Critical – can break a program
  - *dialogsizes*, *escapes*, *newlines*, *nplurals*, *printf*, *pythonbraceformat*, *tabs*, *variables*, *xmltags*
- Functional – may confuse the user
  - *accelerators*, *acronyms*, *blank*, *emails*, *filepaths*, *functions*, *gconf*, *kdecomments*, *long*, *musttranslatewords*, *notranslatewords*, *numbers*, *options*, *purepunc*, *sentencecount*, *short*, *spellcheck*, *urls*, *unchanged*
- Cosmetic – make it look better
  - *brackets*, *doublequoting*, *doublespacing*, *doublewords*, *endpunc*, *endwhitespace*, *puncspacing*, *simplecaps*, *simpleplurals*, *startcaps*, *singlequoting*, *startpunc*, *startwhitespace*, *validchars*
- Extraction – useful mainly for extracting certain types of string
  - *compendiumconflicts*, *credits*, *hassuggestion*, *isfuzzy*, *isreview*, *untranslated*

Test Description

accelerators

Checks whether accelerators are consistent between the two strings.

Make sure you use the `--mozilla`, `--kde`, etc options so that pofilter knows which type of accelerator it is looking for. The test will pick up accelerators that are missing and ones that shouldn't be there.

This check alters its default behavior in Mozilla checker for some languages so it instead checks that accelerators are not present in translation. The purpose of this is to ensure that for languages where the accelerators shouldn't be used the accelerators are not present in the translations. This is common for Indic languages.

acronyms

Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank

Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets

Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

compendiumconflicts

Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert #-#-#-#-# into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits

Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

dialogsizes

Checks that dialog sizes are not translated.

This is a Mozilla specific test. Mozilla uses a language called XUL to define dialogues and screens. This can make use of CSS to specify properties of the dialogue. These properties include things such as the width and height of the box. The size might need to be changed if the dialogue size changes due to longer translations. Thus translators can change these settings. But you are only meant to change the number not translate the words 'width' or 'height'. This check capture instances where these are translated. It will also catch other types of errors in these units.

doublequoting

Checks whether doublequoting is consistent between the two strings.

Checks on double quotes " to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing

Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords

Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off using the `--excludefilters` option.

emails

Checks to see that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. info@example.com are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc

Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as ? ! are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devenagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace

Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes

Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n` `\uNNNN` to ensure that if they exist in the original string you also have them in the translation.

filepaths

Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file-path, unless it is being used as an example, e.g. [your_user_name/path/to/filename.conf].

functions

Checks to see that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

gconf

Checks if we have any gconf config settings translated.

Gconf settings should not be translated so this check checks that gconf settings such as "name" or "modification_date" are not translated in the translation. It allows you to change the surrounding quotes but will ensure that the setting values remain untranslated.

hassuggestion

Checks if there is at least one suggested translation for this unit.

If a message has a suggestion (an alternate translation stored in alt-trans units in XLIFF and .pending files in PO) then these will be extracted. This is used by Pootle and is probably only useful in pofilter when using XLIFF files.

isfuzzy

Checks if the po element has been marked fuzzy.

If a message is marked fuzzy in the PO file then it is extracted. Note this is different from `--fuzzy` and `--nofuzzy` options which specify whether tests should be performed against messages marked fuzzy.

isreview

Checks if the po element has been marked for review.

If you have made use of the 'review' flags in your translations:

```
# (review) reason for review
# (pofilter) testname: explanation for translator
```

Then if a message is marked for review in the PO file it will be extracted. Note this is different from `--review` and `--noreview` options which specify whether tests should be performed against messages already marked as under review.

kdecomments

Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long

Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords

Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines

Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

nplurals

Checks for the correct number of noun forms for plural translations.

This uses the plural information in the language module of the toolkit. This is the same as the Gettext nplural value. It will check that the number of plurals required is the same as the number supplied in your translation.

notranslatewords

Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers

Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

Some languages don't use latin numbers but instead use different numbers. This check will take that into account.

options

Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not

translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf

Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are place holders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also:

pythonbraceformat

See also:

printf Format String

puncspacing

Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc

Checks that strings that are purely punctuation are not changed.

This extracts strings like "+" or "-" as these usually should not be changed.

pythonbraceformat

Checks whether Python brace format strings match.

Python supports both a variant of the *printf* formatting system, and its own formatting language which uses placeholders enclosed in braces. The placeholders can be named, numbered, or anonymous; the former two are filled in from positional args, the latter from keyword arguments. Example:

```
'the {} {0} hungry {insect}'.format('very', insect='caterpiller')
# --> 'the very very hungry caterpiller'
```

The `pythonbraceformat` filter checks for the following problems:

- named placeholders that are present in the original, but missing in the translation, and vice versa.
- originals and translations that require different numbers of positional args.

When the translation has variables not in the original, this can lead to program crashes. The translation not using all variables the original uses is safe. Nonetheless, this filter triggers in both cases.

See also:

PEP 3101 – Advanced String Formatting

sentencecount

Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

short

Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps

Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals

Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting

Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck

Checks for words that don't pass a spell-check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps

Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc

Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace

Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs

Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged

Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated

Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls

Checks to see that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars

Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables

Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. Make sure you use the `--kde`, `--openoffice`, etc flags as these define what variables will be searched for. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags

Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. "<Error>" but will produce false positives for things like "An <Error> occurred" as here "Error" should be translated. It also will allow translation of the alt attribute in e.g. <img src=bob.png alt="Image description"> or similar translatable attributes in OpenOffice.org help files.

pogrep

The pogrep tool extracts messages that match a regular expression into a new set of PO files that can be examined, edited and corrected. These corrections can then be merged using *pomerge*.

Usage

```
pogrep [options] <in> <out>
```

Where:

<in>/<out>	*In* and *out* are either directories or files. *Out* will contain PO/XLIFF files with only those messages that match the regular expression that was you searched for.

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in gmo, mo, po, pot, tmx, xlf, xliff, xliff formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in gmo, mo, po, pot, tmx, xlf, xliff, xliff formats

--search=SEARCHPARTS searches the given parts (source, target, notes, locations)

-I, --ignore-case ignore case distinctions

-e, --regexp use regular expression matching

-v, --invert-match select non-matching lines

--accelerator=ACCELERATORS ignores the given *accelerator characters* when matching

-k, --keep-translations always extract units with translations

Example

```
pogrep --accelerator="_" --search msgid -I -e "software|hardware" only-zu only-zu-
↪check
```

Search for the words "software" or "hardware" in the msgid field. Ignore case (`-I`) and treat the underscore (_) character as an accelerator key. Search through all PO files in the directory "only-zu" and place any matches in PO files in the directory "only-zu-check". This would be useful to run if you know that the word for software and hardware has been changed during the course of translation and you want to check and correct all these instances.

```
pogrep --search=msgid -e '^\w+(\s+\w+){0,3}$' -i templates -o short-words
```

Find all messages in the *templates* directory that have between 1 and 4 words and place them in *short-words*. Use this if you want to see quick results by translating messages that are most likely menu entries or dialogue labels.

```
pogrep --search=msgstr -I -e "Ifayile" zu zu-check
```

Search all translations for the occurrence of *Ifayile*. You would use this to check if words have been used correctly. Useful if you find problematic use of the same word for different concepts. You can use *pocompendium* to find these conflicts.

Notes

Unicode normalization

pogrep will normalize Unicode strings. This allows you to search for strings that contain the same character but that are using precomposed Unicode characters or which are composed using another composition recipe. While an individual user will in all likelihood only compose characters in one way, normalization ensures that data created in a team setting can be shared.

Further reading

Here is a blog post explaining how pogrep can be used to do more targeted localisation of GNOME: http://translate.org.za/blogs/friedel/en/content/better-lies-about-gnome-localisation

pomerge

Pomerge will merge corrected PO, XLIFF, or TMX files (or snippets) into your existing PO, XLIFF, TMX files. Usually you would extract errors using *pofilter*, make corrections to these PO (or XLIFF, TMX) snippets then merge them back using pomerge. You could also use *pogrep* to extract a number of messages matching a certain string, make corrections then merge the correction back using pomerge.

It is probably best to run pomerge against files stored in some kind of version control system so that you can monitor what changes were made.

Pomerge will also attempt to make as small a change as possible to the text, making it easier to see the changes using your version control system.

Usage

```
pomerge [options] [-t <template>] -i <input> -o <output>
```

Where:

<template>	is a set of reference PO, XLIFF, TMX files, either the originals or a set of POT files
<input>	contains the corrected files that are to override content in <output>
<output>	contains the files whose content will be overridden by <input>. This can be the same directory as <template>

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot, xlf formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot, xlf formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in po, pot, xlf formats

-S, --timestamp skip conversion if the output file has newer timestamp

--mergeblanks=MERGEBLANKS whether to overwrite existing translations with blank translations (yes/no). Default is yes.

--mergefuzzy=MERGEFUZZY whether to overwrite existing translations with fuzzy translations (yes/no). Default is yes.

--mergecomments=MERGECOMMENTS whether to merge comments as well as translations (yes/no). Default is yes.

Examples

These examples show pomerge in action.

```
pomerge -t af -i af-check -o af
```

Take corrections from *af-check* merge them with the templates in *af* and output into *af*. Thus merge af-check and override entries found in *af*. Do this only if you are using a version control system so that you can check what changes pomerge made or if you have complete and utter confidence in this tool.

```
pomerge --mergeblanks=yes -t af -i af-check -o af-new
```

Merge the corrections from *af-check* with templates in *af* and output to *af-new*. If an entry is blank in *af-check* then make it blank in the output in *af-new*.

Issues

- Seems to have trouble merging KDE style comments back. (Probably not relevant with newest versions any more.)
- Only files found in the input directory will be copied to the output. The template directory is not searched for extra files to copy to the output. Therefore it is always best to have your input directory in version control, and use the same directory as output. This will allow you to use the diff function of the version control system to double check changes made, with all the files of the input still present.

porestructure

porestructure takes the PO files output by *poconflicts* (a flat structure), and recreates the directory structure according to the poconflict location comments found in each PO message. After being restructured, the messages in the resulting directory structure can be merged back using *pomerge*.

Since poconflicts adds conflicting messages, from many different PO files, into a single PO file, the original structure of the files and directories are lost and the new PO files are output to a single directory. The original structure information is left in "(pofilter)" comments for each PO element.

Usage

```
porestructure [options] <conflicts> <po>
```

Where:

<conflicts>	is a directory containing one the corrected output from poconflict
<po>	is an output directory to write the restructured files to

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po format

Examples

The documentation for poconflicts has *Examples* for the complete process using poconflict, porestructure, and pomerge.

junitmsgfmt

New in version 1.7.

Run msgfmt and provide JUnit type output for use in continuous integration systems like Hudson and Jenkins.

Usage

```
junitmsgfmt po/*.po > msgfmt_junit.xml
```

These tools are especially useful for measuring and improving translation quality.

- *poconflicts* – extract messages that have conflicting translation
- *pofilter* – filter PO files to find common errors using a *number of tests*
- *pogrep* – find strings in your PO files
- *pomerge* – merge file extracted using pofilter back into the original files

- *porestructure* – restructures PO files according to poconflict directives
- *junitmsgfmt* – run msgfmt and provide JUnit type output for use in continuous integration systems like Hudson and Jenkins

1.4.2 Other tools

tmserver

tmserver is a Translation Memory service that can be queried via HTTP using a simple REST like URL/http and data is exchanged between server and client encoded in JSON.

Note: If you are searching for an enterprise Translation Memory server then rather use amaGama.

Usage

```
tmserver.py --bind=HOSTNAME --port=PORT [--tmdb=TMDBFILE] [--import-translation-
↪file=TMFILE [--import-source-lang=SOURCE_LANG] [--import-target-lang=TARGET_LANG]]
```

Where:

TMDB-FILE	is the SQLite database file containing translation memory data, if not specified a new temporary database is created
TMFILE	is a translation file (po, xliff, etc.) that should be imported into the database (mostly useful when no tmdb file is specified).

Options:

-h, --help show this help message and exit

-d TMDBFILE, --tmdb=TMDBFILE translation memory database file

-f TMFILES, --import-translation-file=TMFILES translation file to import into the database

-t TARGET_LANG, --import-target-lang=TARGET_LANG target language of translation files

-s SOURCE_LANG, --import-source-lang=SOURCE_LANG source language of translation files

-b BIND, --bind=BIND address to bind server to (default: localhost)

-p PORT, --port=PORT port to listen on (default: 8888)

--max-candidates=MAX_CANDIDATES Maximum number of candidates

--min-similarity=MIN_SIMILARITY minimum similarity

--max-length=MAX_LENGTH Maxmimum string length

--debug enable debugging features

Testing

easiest way to run the server for testing is to pass it a large translation file (maybe generated by *pocompendium*) to create a tmdb database on the fly.

```
tmserver -b localhost -p 8080 -f compendium.po -s en_US -t ar
```

The server can be queried using a webbrowser. the url would be:

```
http://HOST:PORT/tmserver/SOURCE_LANG/TARGET_LANG/unit/STRING
```

So to see suggestions for "open file" try the url http://localhost:8080/tmserver/en_US/ar/unit/open+file

poterminology

poterminology takes Gettext PO/POT files and extracts potential terminology.

This is useful as a first step before translating a new project (or an existing project into a new target language) as it allows you to define key terminology for consistency in translations. The resulting terminology PO files can be used by Pootle to provide suggestions while translating.

Generally, all the input files should have the same source language, and either be POT files (with no translations) or PO files with translations to the same target language.

The more separate PO files you use to generate terminology, the better your results will be, but poterminology can be used with just a single input file.

Read more about terminology extraction

Usage

```
poterminology [options] <input> <terminology>
```

Where:

<input>	translations to be examined for terminology
<terminology>	extracted potential terminology

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in pot, po formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-u UPDATEFILE, --update=UPDATEFILE update terminology in UPDATEFILE

-S STOPFILE, --stopword-list=STOPFILE read stopword (term exclusion) list from STOPFILE (default site-packages/translate/share/stoplist-en)

-F, --fold-titlecase fold "Title Case" to lowercase (default)

-C, --preserve-case preserve all uppercase/lowercase

-I, --ignore-case make all terms lowercase

--accelerator=ACCELERATORS ignore the given accelerator characters when matching (accelerator characters probably require quoting)

-t LENGTH, --term-words=LENGTH generate terms of up to LENGTH words (default 3)

--nonstop-needed=MIN omit terms with less than MIN nonstop words (default 1)

--inputs-needed=MIN omit terms appearing in less than MIN input files (default 2, or 1 if only one input file)

--fullmsg-needed=MIN omit full message terms appearing in less than MIN different messages (default 1)

--substr-needed=MIN omit substring-only terms appearing in less than MIN different messages (default 2)

--locs-needed=MIN omit terms appearing in less than MIN different original program locations (default 2)

--sort=ORDER output sort order(s): frequency, dictionary, length (default is all orders in the above priority)

--source-language=LANG the source language code (default 'en')

-v, --invert invert the source and target languages for terminology

Examples

You want to generate a terminology file for Pootle that will be used to provide suggestions for translating Pootle itself:

```
poterminology Pootle/po/pootle/templates/*.pot .
```

This results in a `./pootle-terminology.pot` output file with 23 terms (from "file" to "does not exist") – without any translations.

The default output file can be added to a Pootle project to provide terminology matching suggestions for that project; alternately a special Terminology project can be used and it will provide terminology suggestions for all projects that do not have a pootle-terminology.po file.

Generating a terminology file containing automatically extracted translations is possible as well, by using PO files with translations for the input files:

```
poterminology Pootle/po/pootle/fi/*.po --output fi/pootle-terminology.po --sort␣
↪dictionary
```

Using PO files with Finnish translations, you get an output file that contains the same 23 terms, with translations of eight terms – one ("login") is fuzzy due to slightly different translations in jToolkit and Pootle. The file is sorted in alphabetical order (by source term, not translated term), which can be useful when comparing different terminology files.

Even though there is no translation of Pootle into Kinyarwanda, you can use the Gnome UI terminology PO file as a source for translations; in order to extract only the terms common to jToolkit and Pootle this command includes the POT output from the first step above (which is redundant) and require terms to appear in three different input sources:

```
poterminology Pootle/po/pootle/templates/*.pot pootle-terminology.pot \
  Pootle/po/terminology/rw/gnome/rw.po --inputs-needed=3 -o terminology/rw.po
```

Of the 23 terms, 16 have Kinyarwanda translations extracted from the Gnome UI terminology.

For a language like Spanish, with both Pootle translations and Gnome terminology available, 18 translations (2 fuzzy) are generated by the following command, which initializes the terminology file from the POT output from the first step, and then uses `--update` to specify that the pootle-es.po file is to be used both for input and output:

```
cp pootle-terminology.pot glossary-es.po
poterminology --inputs=3 --update glossary-es.po \
  Pootle/po/pootle/es/*.po Pootle/po/terminology/es/gnome/es.po
```

Reduced terminology glossaries

If you want to generate a terminology file containing only single words, not phrases, you can use `-t/--term-words` to control this. If your input files are very large and/or you have a lot of input files, and you are finding that poterminology is taking too much time and memory to run, reducing the phrase size from the default value of 3 can be helpful.

For example, running poterminology on the subversion trunk with the default phrase size can take quite some time and may not even complete on a small-memory system, but with `--term-words=1` the initial number of terms is reduced by half, and the thresholding process can complete:

```
poterminology --progress=none -t 1 translate

1297 terms from 64039 units in 216 files
254 terms after thresholding
254 terms after subphrase reduction
```

The first line of output indicates the number of input files and translation units (messages), with the number of unique terms present after removing C and Python format specifiers (e.g. %d), XML/HTML <elements> and &entities; and performing stoplist elimination.

The second line gives the number of terms remaining after applying threshold filtering (discussed in more detail below) to eliminate terms that are not sufficiently "common" in the input files.

The third line gives the number of terms remaining after eliminating subphrases that did not occur independently. In this case, since the term-words limit is 1, there are no subphrases and so the number is the same as on the second line.

However, in the first example above (generating terminology for Pootle itself), the term "not exist" passes the stoplist and threshold filters, but all occurrences of this term also contained the term "does not exist" which also passes the stoplist and threshold filters. Given this duplication, the shorter phrase is eliminated in favor of the longer one, resulting in 23 terms (out of 25 that pass the threshold filters).

Reducing output terminology with thresholding options

Depending on the size and number of the source files, and the desired scope of the output terminology file, there are several thresholding filters that can be adjusted to allow fewer or more terms in the output file. We have seen above how one (`--inputs-needed`) can be used to require that terms be present in multiple input files, but there are also other thresholds that can be adjusted to control the size of the output terminology file.

–inputs-needed

This is the most flexible and powerful thresholding control. The default value is 2, unless only one input file (not counting an `--update argument`) is provided, in which case the threshold is 1 to avoid filtering out all terms and generating an empty output terminology file.

By copying input files and providing them multiple times as inputs, you can even achieve "weighted" thresholding, so that for example, all terms in one original input file will pass thresholding, while other files may be filtered. A simple version of this technique was used above to incorporate translations from the Gnome terminology PO files without having it affect the terms that passed the threshold filters.

–locs-needed

Rather than requiring that a term appear in multiple input PO or POT files, this requires that it have been present in multiple source code files, as evidenced by location comments in the PO/POT sources.

This threshold can be helpful in eliminating over-specialized terminology that you don't want when multiple PO/POT files are generated from the same sources (via included header or library files).

Note that some PO/POT files have function names rather than source file names in the location comments; in this case the threshold will be on multiple functions, which may need to be set higher to be effective.

Not all PO/POT files contain proper location comments. If your input files don't have (good) location comments and the output terminology file is reduced to zero or very few entries by thresholding, you may need to override the default value for this threshold and set it to 0, which disables this check.

The setting of the `--locs-needed` comment has another effect, which is that location comments in the output terminology file will be limited to twice that number; a location comment indicating the number of additional locations not specified will be added instead of the omitted locations.

–fullmsg-needed & –substr-needed

These two thresholds specify the number of different translation units (messages) in which a term must appear; they both work in the same way, but the first one applies to terms which appear as complete translation units in one or more of the source files (full message terms), and the second one to all other terms (substring terms). Note that translations are extracted only for full message terms; poterminology cannot identify the corresponding substring in a translation.

If you are working with a single input file without useful location comments, increasing these thresholds may be the only way to effectively reduce the output terminology. Generally, you should increase the `--substr-needed` threshold first, as the full message terms are more likely to be useful terminology.

Stop word files

Much of the power of poterminology in generating useful terminology files is due to the default stop word file that it uses. This file contains words and regular expressions that poterminology will ignore when generating terms, so that the output terminology doesn't have tons of useless entries like "the 16" or "Z".

In most cases, the default stop word list will work well, but you may want to replace it with your own version, or possibly just supplement or override certain entries. The default *poterminology stopword file* contains comments that describe the syntax and operation of these files.

If you want to completely replace the stopword list (for example, if your source language is French rather than English) you could do it with a command like this:

```
poterminology --stopword-list=stoplist-fr logiciel/ -o glossaire.po
```

If you merely want to modify the standard stopword list with your own additions and overrides, you must explicitly specify the default list first:

```
poterminology -S /usr/lib/python2.5/site-packages/translate/share/stoplist-en \
  -S my-stoplist po/ -o terminology.po
```

You can use poterminology `--help` to see the default stopword list pathname, which may differ from the one shown above.

Note that if you are using multiple stopword list files, as in the above, they will all be subject to the same case mapping (fold "Title Case" to lower case by default) – if you specify a different case mapping in the second file it will override the mapping for all the stopword list files.

Issues

When using poterminology on Windows systems, file globbing for input is not supported (unless you have a version of Python built with cygwin, which is not common). On Windows, a command like `poterminology -o test.po podir/\*.po` will fail with an error "No such file or directory: 'podir*.po'" instead of expanding the podir/*.po glob expression. (This problem affects all Translate Toolkit command-line tools, not just poterminology.) You can work around this problem by making sure that the directory does not contain any files (or subdirectories) that you do not want to use for input, and just giving the directory name as the argument, e.g. `poterminology -o test.po podir` for the case above.

When using terminology files generated by poterminology as input, a plethora of translator comments marked with (poterminology) may be generated, with the number of these increasing on each iteration. You may wish to run *pocommentclean* (or a slightly modified version of it which only removes (poterminology) comments) on the input and/or output files, especially since translator comments are displayed as tooltips by Pootle (thankfully, they are truncated at a few dozen characters).

Default threshold settings may eliminate all output terms; in this case, poterminology should suggest threshold option settings that would allow output to be generated (this enhancement is tracked as issue 582).

While poterminology ignores XML/HTML entities and elements and %-style format strings (for C and Python), it does not ignore all types of "variables" that may occur, particularly in OpenOffice.org, Mozilla, or Gnome localization files. These other types should be ignored as well (this enhancement is tracked as issue 598).

Terms containing only words that are ignored individually, but not excluded from phrases (e.g. "you are you") may be generated by poterminology, but aren't generally useful. Adding a new threshold option `--nonstop-needed` could allow these to be suppressed (this enhancement is tracked as issue 1102).

Pootle ignores parenthetical comments in source text when performing terminology matching; this allows for terms like "scan (verb)" and "scan (noun)" to both be provided as suggestions for a message containing "scan." poterminology does not provide any special handling for these, but it could use them to provide better handling of different translations for a single term. This would be an improvement over the current approach, which marks the term fuzzy and includes all variants, with location information in { } braces in the automatically extracted translation.

Currently, message context information (PO msgctxt) is not used in any way; this could provide an additional source of information for distinguishing variants of the same term.

A single execution of poterminology can only perform automatic translation extraction for a single target language – having the ability to handle all target languages in one run would allow a single command to generate all terminology for an entire project. Additionally, this could provide even more information for identifying variant terms by comparing the number of target languages that have variant translations.

On single files

If poterminology yields 0 terms from single files, try the following:

```
poterminology --locs-needed=0 --inputs-needed=0 --substr-needed=5 -i yourfile.po -o␣
↪yourfile_term.po
```

... where "substr-needed" is the number of times a term should occur to be considered.

Stopword file format

New in version 1.2.

The default stopword file for *poterminology* describes the syntax of these files and provides a good default for most applications using English source text. You can find the location of the default stopword file by looking at the output of poterminology `--help`, or using the following command:

```
poterminology --manpage | sed -n '/STOPFILE/s/.*(\(.*\)).*/\1/p'
```

Overview

The basic syntax of this file is line-oriented, with the first character of each line determining its function. The order of the lines is generally not significant (with one exception noted below), and the selection of function characters was made so that an ASCII sort of the file would leave it in a generally logical order (except for comment lines).

Apart from comment lines (which begin with '#') and empty lines (which are also ignored), there are three general types of lines, which may appear in any order:

- case mapping specifiers
- stoplist regular expressions
- stoplist words

Case mapping specifiers

A line beginning with a '**!**' specifies upper-/lower-case mapping for words or phrases before comparison with this stoplist (no mapping is applied to the words or regular expressions in this file, only to the source messages). The second character on this line must be one of the following:

- **C** no uppercase / lowercase mapping is performed
- **F** 'Title Case" words / terms are folded to lower case (default)
- **I** all words are mapped to lowercase

These correspond to the equivalent `--preserve-case` / `--fold-titlecase` / `--ignore-case` options to poterminology, but are completely independent and only apply to stoplist matching. You can run poterminology with `-I` to map all terms to lowercase, and if the case mapping specifier in the stopword file is '**!C**' a stoplist with "pootle" in it will not prevent a term containing "Pootle" from passing the stoplist (and then being mapped to "pootle").

There should only be one case mapping specifier in a stoplist file; if more than one are present, the last one will take precedence over the others, and its mapping will apply to all entries. If multiple stoplist files are used, the last case mapping specifier processed will apply to all entries **in all files**.

Stoplist regular expressions

Lines beginning with a '**/**' are regular expression patterns – any word that matches will be ignored by itself, and any phrase containing it will be excluded as well. The regular expression consists of all characters on the line following the initial '/' – these are extended regular expressions, so grouping, alternation, and such are available.

Regular expression patterns are only checked if the word itself does not appear in the stoplist file as a word entry. The regular expression patterns are always applied to individual words, not phrases, and must match the entire word (i.e. they are anchored both at the start and end).

Use regular expressions sparingly, as evaluating them for every word in the source files can be expensive. In addition to stoplist regular expressions, poterminology has precompiled patterns for C and Python format specifiers (e.g. %d) and XML/HTML <elements> and &entities; – these are removed before stoplist processing and it is not possible to override this.

Stoplist words

All other lines should begin with one of the following characters, which indicate whether the word should be **ignored** (as a word alone), **disregarded** in a phrase (i.e. a phrase containing it is allowed, and the word does not count against the `--term-words` length limit), or any phrase containing it should be **excluded**.

- **+** allow word alone, allow phrases containing it
- **:** allow word alone, disregarded (for `--term-word-length`) inside phrase
- **<** allow word alone, but exclude any phrase containing it
- **=** ignore word alone, but allow phrases containing it
- **>** ignore word alone, disregarded (for `--term-word-length`) inside phrase
- **@** ignore word alone, and exclude any phrase containing it

Generally '+' is only needed for exceptions to regular expression patterns, but it may also be used to override an entry in a previous stoplist if you are using multiple stoplists.

Note that if a word appears multiple times in a stoplist file with different function characters preceding it, the *last entry will take precedence* over the others. This is the only exception to the general rule that order is not important in stopword files.

Default file example

```
# apply title-case folding to words before comparing with this stoplist
!F
```

The fold-titlecase setting is the default, even if it were not explicitly specified. This allows capitalized words at the start of a sentence (e.g. "Who") to match a stopword "who" but allows acronyms like WHO (World Health Organization) to be included in the terminology. If you are using poterminology with source files that contain large amounts of ALL UPPERCASE TEXT you may find the ignore-case setting to be preferable.

```
# override regex match below for phrases with 'no'
+no
```

The regular expression /..? below would normally match the word 'no' and both ignore it as a term and exclude any phrases containing it. The above will allow it to appear as a term and in phrases.

```
# ignore all one or two-character words (unless =word appears below)
/..?
# ignore words with parenthesis, typically function() calls and the like
/.*\(.*
# ignore numbers, both cardinal (e.g. 1,234.0) and ordinal (e.g. 1st, 22nd)
/[0-9,.]+(st|nd|rd|th)?
```

These regular expressions ignore a lot of uninteresting terms that are typically code or other things that shouldn't be translated anyhow. There are many exceptions to the one or two-character word pattern in the default stoplist file, not only with = like '=in' but also '+no' and ':on' and '<ok' and '>of'.

```
# allow these words by themselves and don't count against length for phrases
:off
:on
```

These prepositions are common as button text and thus useful to have as terms; they also form an important part of phrases so are disregarded for term word count to allow for slightly longer phrases including them.

```
# allow these words by themselves, but ignore any phrases containing them
<first
<hello
<last
```

These are words that are worth including in a terminology, as they are common in applications, but which aren't generally part of idiomatic phrases.

```
# ignore these words by themselves, but allow phrases containing them
=able
=about
=actually
=ad
=as
=at
```

This is the largest category of stoplist words, and these are all just rather common words. The purpose of a terminology list is to provide specific translation suggestions for the harder words or phrases, not provide a general dictionary, so these words are not of interest by themselves, but may well be part of an interesting phrase.

```
# ignore these words by themselves, but allow phrases containing them,   and
# don't count against length for phrases
#
# (possible additions to this list for multi-lingual text: >di >el >le)
#
>a
>an
>and
```

These very common words aren't of interest by themselves, but often form an important part of phrases so are disregarded for term word count to allow for slightly longer phrases including them.

```
# ignore these words and any phrases containing them
@ain't
@aint
@al
@are
```

These are "junk" words that are not only uninteresting by themselves, they generally do not contribute anything to the phrases containing them.

pocount

pocount will count the number of strings and words in translatable files.

Supported formates include: PO and XLIFF. Almost all bilingual file formats supported by the Translate Toolkit will work with pocount, including: *TMX*, *TBX*, *Gettext .mo*, *Qt .qm*, *Wordfast .txt TM*.

A number of other *formats* should be countable as the toolkit develops. Note that only multilingual formats based the storage *base class* are supported, but that includes almost all storage formats.

Usage

```
pocount [options] <directory|file(s)>
```

Where:

directory	will recurse and count all files in the specified directory
file(s)	will count all files specified

Options:

-h, --help show this help message and exit

--incomplete skip 100% translated files

Output format:

--full (default) statistics in full, verbose format

--csv statistics in CSV format

--short same as –short-strings

--short-strings statistics of strings in short format – one line per file

--short-words statistics of words in short format – one line per file

Examples

pocount makes it easy to count the current state of a body of translations. The most interesting options are those that adjust the output style and decide what to count.

Easy counting

To count how much work is to be done in you project:

```
pocount project/
```

This will count all translatable files found in the directory *project/* and output the results in `--full` format.

You might want to be more specific and only count certain files:

```
pocount *.po
```

This will count all PO files in the current directory but will ignore any other files that 'pocount' can count.

You can have full control of the files to count by using some of the abilities of the Unix commandline, these may work on Mac OS X but are unlikely to work on Windows.:

```
pocount $(find . -name "*.properties.po")
```

This will first find all files that match `*.properties.po` and then count them. That would make it easy to count the state of your Mozilla translations of .properties files.

Incomplete work

To count what still needs to be done, ignoring what is 100% complete you can use the `--incomplete` option.:

```
pocount --incomplete --short *.xlf
```

We are now counting all XLIFF files by using the `*.xlf` expansion. We are only counting files that are not 100% complete and we're outputting string counts using the `--short` option.

Output formats

The output options provide the following types of output

–full

This is the normal, or default, mode. It produces the most comprehensive and easy to read data, although the amount of data may overwhelm the user. It produces the following output:

```
avmedia/source/viewer.po
type              strings      words (source)    words (translation)
translated:   73465 ( 99%)     538598 ( 99%)          513296
fuzzy:           13 (  0%)        141 (  0%)             n/a
untranslated:    53 (  0%)        602 (  0%)             n/a
Total:        73531             539341                 513296
```

A grand total and file count is provided if the number of files is greater than one.

–csv

This format is useful if you want to reuse the data in a spreadsheet. In CSV mode the following output is shown:

```
Filename, Translated Messages, Translated Source Words, Translated Target Words,␣
↪Fuzzy Messages, Fuzzy Source Words, Untranslated Messages, Untranslated Source␣
↪Words, Review Messages, Review Source Words
avmedia/source/viewer.po,  1, 3, 3, 0, 0, 4, 22, 1, 3
```

Totals are not provided in CSV mode.

–short-strings (alias –short)

The focus is on easily accessible data in a compact form. This will only count strings and uses a short syntax to make it easy for an experienced localiser to read.:

```
test-po/fuzzy.po strings: total: 1    | 0t    1f      0u      | 0%t   100%f   0%u
```

The filename is followed by a word indicating the type of count, here we are counting strings. The total give the total string count. While the letters t, f and u represent 'translated', 'fuzzy' and 'untranslated' and here indicate the string counts for each of those categories. The counts are followed by a percentage representation of the same categories.

–short-words

The output is very similar to `--short-strings` above:

```
test-po/fuzzy.po source words: total: 3        | 0t     3f       0u       | 0%t    100%f
↪0%u
```

But instead of counting string we are now counting words as indicated by the term 'source words'

Bugs

- There are some miscounts related to word breaks.
- When using the short output formats the columns may not be exactly aligned. This is because the number of digits in different columns is unknown before all input files are processed. The chosen tradeoff here was instanteous output (after each processed file) instead of waiting for the last file to be processed.

podebug

Insert pseudo translations or debug markers into target text in XLIFF, Gettex PO and other localization files.

The pseudo translation or debug markers make it easy to reference and locate strings when your translated application is running.

Use it to:

- *Target your translations*: see what files are being referenced for string appearing in your programs.
- *Debug translations*: if you know in what file the message occurs then you can quickly find it and fix it.
- *Check that everything is translatable*: any English only text needs to be analysed so that it can be localised.
- *Check for Unicode compliance*: by inserting Unicode text outside of the Latin range it allows you to check that your program can handle non-Latin correctly.

Usage

```
podebug [options] <in> <out>
```

Where:

<in>	is an input directory or localisation file
<out>	is an output directory or localisation file, if missing output will be to standard out.

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

-f FORMAT, --format=FORMAT specify format string

--rewrite=STYLE the translation rewrite style: *xxx, en, blank, chef (v1.2), unicode (v1.2)*

--ignore=APPLICATION apply tagging ignore rules for the given application: kde, gtk, openoffice, libreoffice, mozilla

Formats

A format string can have these various options:

f	full filename including directory
F	as %f but with .po file extension
b	base of filename
B	base of filename with .po file extension
d	directory name
s	preset OpenOffice.org modifier
c	use only consonants
h	hash value (since version 1.4 – see notes below)
N	a set number of characters

A format string may look like this:

- `%cf` – the full filename without vowels
- `[%10cb]` – the first ten character after compressing the base of the filename and place it in square brackets with a space before the real message
- `[%5cd - %cB]` – the first 5 consonants of the directory, followed by a dash then the consonants of the filename with a .po extension. All surrounded by square brackets with a space before the translations.
- `%4h.` – insert a hash value of length 4

Complex format strings may make it too difficult to actually read the translation, so you are probably best served using as short a string as possible.

Rewriting (style)

The rewriting options are designed to change the target text in various ways (c.f. the various *rewriting styles* available). This is mostly valuable for debugging English text. The 'xxx' rewriter is useful in that it allows you to identify text that has not localisable as that text will lack the xxx characters.

The 'en' rewriter can be used to prepare English hashed (see below) files for quickly finding strings that have spelling or other errors. It can also be used to create a translated English file which can then be used for other purposes such as British English translation.

Ignoring messages

In some applications their are translations that should not be translated (usually these are configuration options). If you do translate them then the application will fail to compile or run.

The `--ignore` option allows you to specify the application for which you are producing PO debug files. In this case it will then not mark certain of the PO entries with debug messages.

In Mozilla we do not mark lone `.accesskey`, `.width`, `.height`, etc since these can really be thought of as configuration options.

Hashing

Sometimes you find an error in a string. But it is difficult to search for the occurrence of the error. In order to make it easy to find a string in your files we can produce a hash on the strings location and other data. This produces unique alphanumeric sequences which are prepended to the target text. Thus now in your application you have your translated text and an alphanumeric value. Its is then easy to search for that value and find your problem string.

Usings podebug

Here are some more examples in a series of blog posts.

–rewrite=STYLE

podebug allows you to rewrite the output text in a number of ways.

xxx

The target text is surrounded by `xxx` as follows

```
msgid "English"
msgstr "xxxEnglishxxx"
```

This is useful when you want to identify which text is localisable. There might be text in your application which you cannot localise this will allow you to quickly identify that text.

en

The source text is copied to the target

```
msgid "English"
msgstr "English"
```

In this way you can create translations that contain only the source text. Useful if you are preparing a roundtrip test or want to start an English derived translation such as British English. It produces the same results as msgen but with the advantage that you can add debug markers.

blank

This simply empties your current translations

```
msgid "English"
msgstr ""
```

When you have a set of translation files but no template this allows you to essentially convert a PO into a POT file. This mimics the `--empty` functionality of msghack.

bracket

New in version 1.4.

Places brackets around the translated text.

```
msgid "English"
msgstr "[English]"
```

This can be used in the same way as `xxx` to check for translatability. It is also useful with very long strings as it allows you to check that the full string in rendered and has not been cutoff by the application.

chef

New in version 1.2.

Rewrites the source text using mock Swedish as popularised by the Swedish Chef.

```
msgid "English"
msgstr "Ingleesh"
```

This is probably only useful for some fun. It's not guaranteed that every string will be rewritten as the mock Swedish rules might not apply thus its not ideal for identifying untranslatable strings.

flipped

New in version 1.4.

Change the text into a version that uses equivalent Latin characters that are upside down.

```
msgid "English"
msgstr "uɪs"
```

`flipped` can give an output that simulates RTL languages. It inserts RTL characters to try to achieve RTL-like results. Its not perfect but will give you some sense of whether your application can do RTL. Or just use it for fun!

For really testing right-to-left GUIs, you want to make sure that the whole application is shown in RTL, not just the strings. Test your pseudo-translated file as a translation of an RTL language like Arabic or Hebrew. In case the application relies on other files coming from libraries (like GTK+), you might need to repeat the process for them, or at least ensure that you have the Arabic/Hebrew .mo files for them installed.

unicode

New in version 1.2.

Rewrites the source text with Unicode characters that looks like the Latin characters that they are replacing.

```
msgid "English"
msgstr "ĪŞ"
```

This allows a translator or programmer to test a programs ability to use Unicode message strings. By using characters in the Unicode range but that are related to the plain Latin characters that they replace we ensure that the messages are still readable.

Note: Before version 1.4, the rewrite rule will also rewrite variables and XML tags, which would cause problems in some situations. Run *pofilter* as a quick method to fix up incorrect changes, or upgrade to version 1.4.

---

posegment

posegment takes a Gettext PO or XLIFF file and segments the entries, generating a new file with revised and smaller translation units.

This is useful for the creation of a file that can be used as a Translation Memory as you should get better matching after you have exposed translated sentences that might occur elsewhere in your work.

Posegment won't do very advanced sentence boundary detection and alignment, but has customisations for the punctuation rules of several languages (Amharic, Afrikaans, Arabic, Armenian, Chinese, Greek, Japanese, Khmer, Oriya, Persian). For the purpose of increasing your TM (as described below), it is already very useful. Give it a try and help us to improve it even more for your language.

Usage

```
posegment [options] <input> <segmented>
```

Where:

<input>	translations to be segmented
<segmented>	translations segmented at the sentence level

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in po, pot, tmx, xlf formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot, tmx, xlf formats

-S, --timestamp skip conversion if the output file has newer timestamp

-P, --pot output PO Templates (.pot) rather than PO files (.po)

-l LANG, --language=LANG the target language code

--source-language=LANG the source language code (default 'en')

--keepspaces Disable automatic stripping of whitespace

--only-aligned Removes units where sentence number does not correspond

Examples

You want to reuse all of your Pidgin translations in another Instant Messenger:

```
posegment pidgin-af.po pidgin-af-segmented.po
```

Now all of our Pidgin translation are available, segmented at a sentence level, to be used as a Translation Memory for our other translation work.

You can do the same at a project level. Here we want to segment all of our OpenOffice.org translation work, a few hundred files:

```
posegment af/ af-segmented/
```

We start with all our files in `af` which are now duplicated in `af-segmented` except files are now fully segmented.

Issues

- If the toolkit doesn't have segmentation rules for your language then it will default to English which might be incorrect.
- Segmentation does not guarantee reuse as your TM software needs to know how to segment when matching. If you use software that doesn't do segmentation, you can consider joining the original and the segmented files together with msgcat, to get the best of both worlds.
- You cannot (yet) use the tool to break a file into segments, translate, and then recreate as the segmented file does not know which parts should be joined together to recreate a file.

pocompile

Compile PO or XLIFF files into MO (Machine Object) files. MO files are installed on your computer and allow a Gettext enabled computer to provide the translations for the application.

Usage

```
pocompile <po> <mo>
```

Where:

<po/xliff>	is a standard PO file, XLIFF file or directory
<mo>	is the output MO file or directory of MO files

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in xlf, po, pot formats

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in mo format

-S, --timestamp skip conversion if the output file has newer timestamp

--fuzzy use translations marked fuzzy

--nofuzzy don't use translations marked fuzzy (default)

Examples

```
pocompile --fuzzy file.po file.mo
```

Creates a new MO file called *file.mo* based on the translation in the PO file *file.po*. By using the `--fuzzy` option we use all translations including those marked fuzzy.

```
pocompile file.xlf file.mo
```

Create an MO file from an XLIFF file called *file.xlf* (available from version 1.1 of the toolkit).

poswap

This tool builds a new translation file with the target text (translation) of the input file(s) as source language of the output file it creates.

This makes it possible to have French as the source file for translation, rather than English. Note that this requires no change in the software project and is only a manipulation of the strings in the existing files. The only requirement for this tool is a French translation.

It can also be used to convert translatable files that use logical IDs instead of source text into a format usable by human localisers.

Usage

```
poswap [options] <newsource> [-t current] <new>
```

Where:

<newsource>	is the translations (preferably 100% translated) of the preferred source language (like French)
<current>	is the (optional) current English based translation in your intended target language
<new>	is the intended output file / directory

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in pot format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read from TEMPLATE in po, pot formats

--reverse Do the inverse operation (converting back to a normal English based file). See the examples.

Examples

Ensure that the two po files / directories correspond 100% to the same pot file before using this.

To start a fresh Afrikaans (af) translation from Dutch (nl):

```
poswap nl.po nl-af.po
```

This initialises a new, empty file nl-af.po with Dutch as the source language.

To change the nl-af.po file back to the expected English based af.po:

```
poswap --reverse nl.po -t nl-af.po af.po
```

To translate Kurdish (ku) through French (fr):

```
poswap -i fr/ -t ku/ -o fr-ku/
```

This will take the existing (English based) Kurdish translation in ku/ and produce files in fr-ku with French as the source language and Kurdish as the target language.

To convert the fr-ku files back to en-ku:

```
poswap --reverse -i fr/ -t fr-ku/ -o en-ku/
```

This recreates the English based Kurdish translation from the French based files previously created in fr-ku/.

Issues

- Behaviour is undetermined if the two files don't match 100%. If PO files are based in the same template, there should be no problem.
- We should probably be doing fuzzy matching in future to ease the migration over the lifetime of a changing French translation.

poclean

This is a rudimentary tool to produce a clean file from an unclean file (Trados/Wordfast) by stripping out the tw4win indicators.

Usage

```
poclean <input> <output>
```

Where:

<input>	is the text versions of the unclean RTF files
<output>	is the intended output file / directory

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in pot format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-S, --timestamp skip conversion if the output file has newer timestamp

Examples

To create a text version of the unclean RTF file, you need UnRTF, available here: project site or here (windows).

```
unrtf translation.rtf  --text > translation.po
```

You might need to convert the encoding of the file, with iconv, for example:

```
iconv -f latin1 -t utf-8 translation.po > new_translation.po
```

Now you can clean the file with poclean

```
poclean new_translation.po clean_translation.po
```

pretranslate

Merge existing translations from an old translation file to a new one as well as fill any missing translations from translation memory via fuzzy matching.

This functionality used to be part of pot2po and corresponds to "msgmerge" from the gettext package.

pretranslate works on PO and XLIFF files.

Usage

```
pretranslate [options] <input> <output>
```

Where:

<input>	is the translation file or directory to be pretranslated
<output>	is the translation file or a directory where the pretranslated version will be stored

Options:

--version show program's version number and exit

-h, --help show this help message and exit

--manpage output a manpage based on the help

--progress=PROGRESS show progress as: *dots, none, bar, names, verbose*

--errorlevel=ERRORLEVEL show errorlevel as: *none, message, exception, traceback*

-i INPUT, --input=INPUT read from INPUT in pot format

-x EXCLUDE, --exclude=EXCLUDE exclude names matching EXCLUDE from input paths

-o OUTPUT, --output=OUTPUT write to OUTPUT in po, pot formats

-t TEMPLATE, --template=TEMPLATE read old translations from TEMPLATE

-S, --timestamp skip conversion if the output file has newer timestamp

--tm=TM The file to use as translation memory when fuzzy matching

-s MIN_SIMILARITY, --similarity=MIN_SIMILARITY The minimum similarity for inclusion (default: 75%)

--nofuzzymatching Disable all fuzzy matching

Examples

```
pretranslate -t zu-1.0.1 -tm zu_tm.po zu-2.0.2 zu-2.0.2-translated
```

Here we are pretranslating the PO or XLIFF files in *zu-2.0.2* using the old translations in *zu-1.0.1* and fuzzy matches from the zu_tm.po compendium. the result is stored in *zu-2.0.2-translate*

Unlike pot2po pretranslate will not change anything in the input file except merge translations, no reordering or changes to headers.

Merging

It helps to understand when and how pretranslate will merge. The default is to follow msgmerge's behaviour but we add some extra features with fuzzy matching:

- If everything matches we carry that across
- We can resurrect obsolete messages for reuse
- If we cannot find a match we will first look through the current and obsolete messages and then through any global translation memory
- Fuzzy matching makes use of the *Levenshtein distance* algorithm to detect the best matches

Performance

Fuzzy matches are usually of good quality. Installation of the python-Levenshtein package will speed up fuzzy matching. Without this a Python based matcher is used which is considerably slower.

Levenshtein distance

The levenshtein distance is used for measuring the *"distance"* or similarity of two character strings. Other similarity algorithms can be supplied to the code that does the matching.

This code is used in *pot2po*, *tmserver* and Virtaal. It is implemented in the toolkit, but can optionally use the fast C implementation provided by python-Levenshtein if it is installed. It is strongly recommended to have **python-levenshtein** installed.

To exercise the code the classfile *"Levenshtein.py"* can be executed directly with:

```
$ python Levenshtein.py "The first string." "The second string"
```

Note: Remember to quote the two parameters.

The following things should be noted:

- Only the first `MAX_LEN` characters are considered. Long strings differing at the end will therefore seem to match better than they should. A penalty is awarded if strings are shortened.
- The calculation can stop prematurely as soon as it realise that the supplied minimum required similarity cannot be reached. Strings with widely different lengths give the opportunity for this shortcut. This is by definition of the Levenshtein distance: the distance will be at least as much as the difference in string length. Similarities lower than your supplied minimum (or the default) should therefore not be considered authoritative.

Shortcommings

The following shortcommings have been identified:

- **Cases sensitivity:** *'E'* and *'e'* are considered different characters and according differ as much as *'z'* and *'e'*. This is not ideal, as case differences should be considered less of a difference.
- **Diacritics:** *'ê'* and *'e'* are considered different characters and according differ as much as *'z'* and *'e'*. This is not ideal, as missing diacritics could be due to small input errors, or even input data that simply do not have the correct diacritics.
- **Similar but different words:** Words that have similar characters, but are different, could increase the similarity beyond what is wanted. The sentences *"It is though."* and *"It is dough."* differ markedly semantically, but score similarity of almost 85%. A possible solution is to do an additional calculation based on words, instead of characters.
- **Whitespace:** Differences in tabs, newlines, and space usage should perhaps be considered as a special case.
- *tmserver* – a Translation Memory server, can be queried over HTTP using JSON
- *poterminology* – extracts potential terminology from your translation files
- *pocount* – Count words and strings in PO, XLIFF and other types of translatable files
- *podebug* – Add debug strings to messages
- *posegment* – Break a PO or XLIFF files into sentence segments, useful for creating a segmented translation memory
- *pocompile* – create an MO (Machine Object) file from a PO or XLIFF file
- *poswap* – uses a translation of another language that you would rather use than English as source language
- *poclean* – produces a clean file from an unclean file (Trados/Wordfast) by stripping out the tw4win indicators

- *pretranslate* – fill any missing translations from translation memory via fuzzy matching.
- *Levenshtein distance* – edit distance algorithms for translation memory matching

1.5 Scripts

1.5.1 Mozilla L10n Scripts

Introduction

This page describes the purpose and usage of scripts available in the Translate Toolkit specifically for making the translation of Mozilla products easier.

Mozilla's move from CVS to Mercurial made a lot of these scripts necessary. For more information about Mozilla l10n from CVS, see the *moz-l10n-builder* page.

All of these scripts are available on Subversion from here.

We are currently generating POT files for most major betas, RCs and releases of Firefox and Thunderbird. They are available here: http://l10n.mozilla.org/pootle/pot/

As a start you might want to just use these POT files and gradually learn more about the processes described below. Contact us for more help on using these.

Requirements

- The *Translate Toolkit* (>=1.3)
- All scripts in the `tools/mozilla` directory (from the project sources) should be executable and in your `PATH`.

build_ff3.1_langs.sh

Description

This is a simple bash script that embodies most of the Mozilla l10n process and does the following:

1. Update Mozilla sources
2. Update language files from Mozilla's L10n Mercurial repository.
3. Replace old l10n en-US files with a fresh copy from the updated source tree.
4. *Create new POT files* from the *en-US* l10n files.
5. Create archives of the POT files.
6. For each language:
    1. Update existing PO files if the checked out from a CVS, Subversion or Mercurial repository.
    2. *Migrate* PO files to new POT files.
    3. *Create Mozilla l10n files* for the language based on the migrated PO files.
    4. Create archives of the PO files.
    5. *Build langpack* for the language.

This script is used on the l10n.mozilla.org server to create most (if not all) of the files available from http://l10n.mozilla.org/pootle/. It was originally written as a stable way to provide these files and as such making it as general as possible was not the biggest requirement. This is evident in the script's very narrow focus.

Usage

This script takes no command-line parameters and is only configurable via the variables at the top and, failing that, custom hacking of the script.

The variables are used in the following ways:

BUILD_DIR	The base build directory from where building is done.
MOZCENTRAL_DIR	The directory containing a checkout of the Mozilla source tree http://hg.mozilla.org/mozilla-central/
HG_LANGS	A space-separated list of language codes to build for.
L10N_DIR	The directory where Mozilla l10n files (from l10n-central) should be collected.
PO_DIR	The directory containing the externally-hosted or previously available source PO files (e.g. PO files managed in another VCS repository). It contains a sub-directory for each language.
POPACK_DIR	The output directory for PO archives.
PORECOVER_DIR	The directory to put recovered PO files in. It contains a sub-directory for each language.
POT_INCLUDES	A space-separated list of files to be included in POT archives.
POTPACK_DIR	The output directory for POT archives.
POUPDATED_DIR	The directory to use for updated PO files. It contains a sub-directory for each language.
LANGPACK_DIR	The directory to put langpacks (XPIs) in.
FF_VERSION	The version of Firefox that is being built for. This is used in the file names of archives.

Note: It is **strongly** recommended that you mirror the directory structure specified by the default values of the `*_DIR` variables. For example the default value for `L10N_DIR` is `${BUILD_DIR}/l10n`, then you should put your l10n-central check-outs in the `l10n` directory under your main build directory (`BUILD_DIR`).

Basically, you should have an ideally separate build directory containing the following sub-directories: `l10n`, `mozilla-central`, `po`, `popacks`, `potpacks`, `po-updated` and `xpi` (if used). This way the only variable that need to be changed is `BUILD_DIR`.

build_tb3_langs.sh

This is the script that the `build_ff3.1_langs.sh` script above was actually adapted from. It is 90% similar with the obvious exception that it is aimed at building Thunderbird 3.0 packages in stead of Firefox 3.1. Also note that this script uses the comm-central repository in stead of mozilla-central.

buildxpi.py

Description

Creates XPI language packs from Mozilla sources and translated l10n files. This script has only been tested with Firefox 3.1 beta sources.

It is basically the scripted version of the process described on Mozilla's "Creating a language pack" page.

This script is used by `build_ff3.1_langs.sh` to build language packs in its final step.

Note: This script uses the `.mozconfig` file in your home directory. Any existing `.mozconfig` is renamed to `.mozconfig.bak` during operation and copied back afterwards.

Usage

```
buildxpi.py [<options>] <lang> [<lang2> ...]
```

Example:

```
buildxpi.py -L /path/to/l10n -s /path/to/mozilla-central -o /path/to/xpi_output af ar
```

Options:

-h, --help show this help message and exit

-L L10NBASE, --l10n-base=L10NBASE The directory containing the <lang> subdirectory.

-o OUTPUTDIR, --output-dir=OUTPUTDIR The directory to copy the built XPI to (default: current directory).

-p MOZPRODUCT, --mozproduct=MOZPRODUCT The Mozilla product name (default: "browser").

-s SRCDIR, --src=SRCDIR The directory containing the Mozilla l10n sources.

-d, --delete-dest Delete output XPI if it already exists.

-v, --verbose Be more noisy

get_moz_enUS.py

Description

A simple script to collect the en-US l10n files from a Mozilla source tree ('`comm-central`' or '`mozilla-central`') by traversing the product's `l10n.ini` file.

Usage

```
get_moz_enUS.py [options]
```

Options:

-h, --help show this help message and exit

-s SRCDIR, --src=SRCDIR The directory containing the Mozilla l10n sources.

-d DESTDIR, --dest=DESTDIR The destination directory to copy the en-US locale files to.

-p MOZPRODUCT, --mozproduct=MOZPRODUCT The Mozilla product name.

--delete-dest Delete the destination directory (if it exists).

-v, --verbose Be more noisy

moz-l10n-builder

This is the pre-Mercurial build script originally written by Dwayne Bailey. This is the script that all the others on this page replaces for post-CVS Mozilla l10n.

Note: This script is **not** applicable to the l10n process of any Mozilla products after the move to Mercurial.

For more information about this script see its *dedicated page*.

moz_l10n_builder.py

This script was intended to be a simple and direct port of the `moz-l10n-builder` script from above. It has pro's and cons in comparison to the original, but is very similar for the most part. So for more information about this script, see the original script's *page*.

1.5.2 moz-l10n-builder

Take a set of Mozilla (Firefox, Thunderbird, SeaMonkey, etc.) localisation and migrate them to the latest Mozilla source, building XPIs and repackaging hte Windows .exe file as needed.

Please also check the page on creating a language pack on the Mozilla wiki, to stay abreast of the latest Mozilla way of doing things.

Note: This page is only applicable to Mozilla products with its source hosted in CVS. This includes Firefox versions before 3.1 and Thunderbird versions before 3.0.

For information about working with the new source trees in Mercurial, see the *Mozilla L10n Scripts* page.

Prerequisites

- Translation update component and building XPIs
  - *Translate Toolkit*
  - Existing Mozilla translations in PO format
  - A checkout of Mozilla sources updated to the correct BRANCH or RELEASE
- Building Windows executables
  - Firefox or Thunderbird en-US .exe file e.g. Firefox 2.0 en-US
  - upx for executable compression
  - Nullsoft installer to package the installer.
  - 7zip for various compression
  - Linux: WINE to run the Nullsoft installer
- Directory structure under the directory you want to run moz-l10n-builder in:

l10n/	Contains Mozilla l10n files for available/needed language(s)
mozilla/	The Mozilla source tree
po/	Contains your PO files (output from moz2po)
potpacks/	Where POT-archives go

Note these instructions are for building on Linux, they may work on Windows. All software should be available through your distribution. You will need to use Wine to install the Nullsoft installer and may need to sort out some path issues to get it to run correctly.

Latest Version

moz-l10n-builer is not currently distributed as part of the toolkit. You can get the latest version from Git and you will also need this minor patch to the mozilla source code.

Usage

```
moz-l10n-builder [language-code|ALL]
```

Where:

language-code	build only the supplied languages, or build ALL if specified or if no option is supplied

Your translations will not be modified.

Operation

moz-l10n-builder does the following:

- Updates the mozilla/ directory
- Creates POT files
- Migrates your translations to this new POT file
- Converts the migrated POT files to .dtd and .properties files
- Builds XPI and .exe files
- Performs various hacks to cater for the anomalies of file formats
- Outputs a diff of you migrated PO files and your newly generated Mozilla l10n/ files

Bugs

Currently it is too Translate.org.za specific and not easily configurable without editing. It is also not intelligent enough to work our that you want Firefox vs Thunderbird generation. A lot of this functionality should be in the Mozilla source code itself. We hope over time that this might happen.

1.5.3 phase

phase is a script that allows you to perform a number of tasks on a set of PO files that have been broken into phases. You can create a ZIP file for a phase, run checks against a phase, review a phase, edit files in a phase, etc. All the tasks that would be involved in sending work to various translators, receiving work, checking it and committing to CVS.

Prerequisites

- An environment that will run bash
- diff
- cvs

Latest Version

phase is not currently distributed as part of the toolkit. You can get the latest version from Git

Usage

```
phase <command> [options]
```

Mostly the usage follows the format of:

```
phase <command> <language-dir> <phaselist> <phase-name>
phase <command> <language-dir> <phase-name>
```

A full list of commands and options can be seen by running:

```
phase --help
```

Commands

These are the commands that you can use:

- makephaselist <new-phase-list-name> – creates a phase list
- listphases <phase-list> – lists the different phases that appear in the phase-list file
- listfiles <phase-list> <phase-name> – list all files for the given phase in the phase-list file
- checkphaselist <language-dir> <phase-list> – checks to see which files are not included in the phaselist
- countpo <language-dir> <phase-list> <phase-name> – counts PO file in the given phase
- countpot <template-dir> <phase-list> <phase-name> – counts POT file in the given phase
- missingpo <language-dir> <phase-list> <phase-name> – lists files that have not been returned for a phase
- packpot <template-dir> <phase-list> <phase-name> – packs all POT files for a given phase into a ZIP file
- packpo <language-dir> <phase-list> <phase-name> – packs all PO files for a given phase into a ZIP file
- packall <template-dir> <phase-list> – packs all phases found in the phase list
- packallpo <language-dir> <phase-list> – packs all phases found in the phase list for the given language
- countmismatch <language-dir> <template-dir> <phase-list> <phase-name> – compares the source word count between PO and POT to determine if there are any file errors.
- editpo <language-dir> <phase-list> <phase-name> – edit the PO files in a phase
- editpochecks <language> <phase-name> – edit the PO checks output by checkpo
- editconflicts <language-dir> <phase-list> <phase-name> – edit the extracted conflict items

- checkpo <language-dir> <phase-list> <phase-name> [pofilter options] – run pofilter checks against the given phase
- mergepo <language> <phase-name> – merge the checks back into the main language directory
- conflictpo <language-dir> <phase-list> <phase-name> [poconflict options] – run poconflict checks against the given phase
- diffpo <language-dir> <phase-list> <phase-name> – perform a cvs diff for the phase
- cvslog <language-dir> <phase-list> <phase-name> – perform a cvs log against files in the phase
- lastlog <language-dir> <phase-list> <phase-name> – retrieves the last cvs log entry for each file in a phase
- cvsadd <languages-dir> <phase-list> <phase-name> – CVS adds files and directories that are not already in CVS
- diffpo <language-dir> <phase-list> <phase-name> – perform a cvs diff for the phase
- reviewpo <language-dir> <phase-list> <phase-name> [pofilter options] – extract items marked for review for the given phase
- editreviews <language-dir> <phase-list> <phase-name> – edit the extracted review items
- countreviews <language-dir> <phase-list> <phase-name> – count the number of strings and words under review
- checkinpo <language-dir> <phase-list> <phase-name> – cvs checkin the files in the given phase
- creategsi <language-dir> <en-US.gsi> <traget-language> – creates a BZ2 GSI/SDF file for the language against the en-US GSI file
- reviewsinout <language> <phase-name> – counts the number of review files returned vs sent and shows which are missing
- reviewsdiff <language> <phase-name> – create a diff between what was sent for review and what was returned

Bugs

There are probably lots mostly the bug is that the command line options are pretty inconsistent

1.5.4 pocompendium

Takes a directory of translated PO files and creates a single PO files called a PO compendium. This compendium can be used to review word choice conflicts or as input during a merge using *pomigrate2*.

Prerequisites

GNU Gettext:

- msgattrib
- msgcat
- msghack (may not be present on your installation of Gettext, but is only required for the invert command)
- msgfilter

Usage

```
pocompendium [options] output.po <-d po-directory(ies)|po-file(s)>
```

Where:

output.po	the name of the output PO compendium
po-directory(ies)	one or more directories to use as input for the compendium
po-file(s)	one or more PO files to use as input for the compendium

Options:

-v, --invert swap the msgid and msgstr in the input PO files

-e, --errors only return those msg blocks that have conflicts

-i, --ignore-case drops all msgstr's to lowercase

-st, -tilde, --strip-accel-amp remove all & style accelerator markers

-sa, -amp, --strip-accel-tilde remove all ~ style accelerator markers

-su, --strip-accel-under remove all _ style accelerator markers

Examples

- *Compendium creation* — create a compendium with all your translations to use as input during a message merge either when migrating an existing project or starting a new one.
- *Conflicting translations* — use `--errors` to find where you have translated an English string differently. Many times this is OK but often it will pick up subtle spelling mistakes or help you to migrate older translations to a newer choice of words
- *Conflicting word choice* — use `--invert` and `--errors` to get a compendium file that show how you have used a translated word for different English words. You might have chosen a word that is valid for both of the English expressions but that in the context of computers would cause confusion for the user. You can now easily identify these words and make changes in the underlying translations.

Narrowing Results

PO files treat slight changes in capitalisation, accelerator, punctuation and whitespace as different translations. In cases 2) and 3) above it is sometimes useful to remove the inconsistencies so that you can focus on the errors in translation not on shifts in capitals. To this end you can use the following:

`--ignore-case`, `--strip-accel-amp`, `--strip-accel-tilde`, `--strip-accel-under`

Operation

pocompendium makes use of the Gettext tool msgcat to perform its task. It traverses the PO directories and cat's all found PO files into the single compendium output file. It then uses msgattrib to extract only certain messages, msghack to invert messages and msgfilter to convert messages to lowercase.

Bugs

There are some absolute/relative path name issues

1.5.5 pocommentclean

pocommentclean will remove all translator comments from a directory of PO files.

Prerequisites

- sed

Usage

```
pocommentclean [--backup] <po>
```

Where:

po	is a directory of existing PO files that you want to clean

Options:

--backup Create a backup file for each PO file converted, .po.bak

Operation

Using sed pocommentclean will delete all lines starting with # but which are not standard Gettext PO format lines. So it won't delete developer comments (#.), obsolete messages (#~), flags (#,) or locations (#:).

Bugs

pocommentclean cannot clean individual PO files, it only cleans directories

1.5.6 pomigrate2

pomigrate2 aims to move an existing translation to a new version based on updated PO Template files automatically without user intervention. Therefore it is ideal for when you are migrating many languages or migrating from related but divergent products e.g. Mozilla to Firefox.

Prerequisites

GNU Gettext:

- msginit
- msgcat
- msgmerge

Usage

```
pomigrate [options] <from> <to> <new templates>
```

Where:

from	is a directory of existing PO files
to	is the directory where the migrated PO files will be stored
new templates	this is the directory that contains the PO Template files

Options:

-F, --use-fuzzy-matching use fuzzy algorithms when merging to attempt to match strings

-C, --use-compendium create and use a compendium built from the migrating files

-C, --use-compendium=COMPENDIUM use an external compendium during the migration

--no-wrap do not wrap long lines

--locale set locale for newly born files

-q, --quiet suppress most output

-p, --pot2po use pot2po instead of msgmerge to migrate

Operation

pomigrate2 makes use of the Gettext tools msgmerge or Translate Toolkit's *pot2po* to perform its merging tasks.

It firstly finds all files with the same name and location in the <from> directory as in the <template> directory and copies these to the <to> directory. If there is no file in the <from> directory to match one needed by the <template> directory then it will msgcat all files in the <from> directory with the same name and copy them to the correct destination in the <to> directory. If all of that fails then msginit is used to initialise any missing PO files.

Lastly all the files in <to> are merged using msgmerge or pot2po. This process updates the files to match the layout and messages in <templates>. Optionally, by using `--use-compendium`, a compendium of all the translations in <from> can be created to be used in the final merge process.

1.5.7 popuretext

Extracts all the source text from a directory of POT files or the target text from a directory of PO files, removing PO headers and optionally the accelerator keys.

If you want to use other tools to analyse the text within a translation project, then this is the tool for you. For example, you can use it to calculate word frequencies to create an initial glossary based on the pure source text.

Prerequisites

- GNU Gettext
- sed

Usage

```
popuretext <-P pot-dir|po-dir> <file.txt> [accelerator]
```

Where:

pot-dir	a directory containing POT files
po-dir	a directory containing PO files
file.txt	file that contains the output text
accelerator	optional: accelerator marker to be removed from the text

Examples

```
popuretext -P pot pot.txt '&'
```

Extract all the source text from the *pot* directory and place it in the *pot.txt* file removing all occurrences of the `&` accelerator.

```
popuretext af af.txt
```

Extract all target text from the Afrikaans files in the *af* directory, placing the extracted text in *af.txt*. In this case we are not filtering any accelerator characters.

1.5.8 poreencode

Takes a directory of existing PO files and converts them to a given encoding.

Prerequisites

GNU Gettext

Usage

```
poreencode <encoding> <PO directory>
```

Where:

encoding	is the encoding you would like to convert to e.g. UTF-8
PO directory	is a directory of existing PO files

It is best to backup files before the conversion or to perform it against CVS which prevents a potential loss of data.

Operation

poreencode makes use of the Gettext tool msgconv to perform its task. It traverses the PO directory and finds all PO file. It uses msgconv to convert the PO file from its existing encoding to the new encoding.

Bugs

Like most Gettext tools they do a little bit more than documented, msgconv will decide which strings are in fact fuzzy and delete fuzzy marking – not a lot but you do need to diff (this probably related to #, fuzzy entries that are not placed in the place Gettext expects them).

1.5.9 posplit

Takes an existing PO file and splits it into three components: translated, untranslated and fuzzy. This is useful for reviewing translations or for extracting good translations from a compendium file.

Note that the input file is removed by the script (until version 1.9.1). The generated output files can be combined again with msgcat.

Prerequisites

GNU Gettext

Usage

```
posplit ./file.po
```

Where:

file.po	is an existing PO file or PO compendium

Bugs

- Some relative path bugs thus the need for ./ before file.po.
- Until version 1.9.1, the original input file was removed, issue 2006.

The scripts are for working with and manipulating PO files. Unlike the `tools` which are written in Python, the scripts are written in `bash`. Some of them are packaged since version 1.0 of the Toolkit, but you might need to download them from version control and do a manual installation .

- *moz-l10n-builder* – Create Mozilla XPIs and rebuild Windows installers from existing translations
- *Mozilla L10n Scripts* – Build Mozilla products Firefox and Thunderbird
- *phase* – Helps manage a project divided into phases of work, including sending, checking, etc
- *pocompendium* – Creates various types of PO compendium (i.e. combines many PO files into a single PO file)
- *pocommentclean* – Remove all translator comments from a PO file
- *pomigrate2* – Migrate older PO files to new POT files
- *popuretext* – Extracts all the source text from a directory of POT files
- *poreencode* – Converts PO files to a new character encoding
- *posplit* – Split a PO file into translate, untranslated and fuzzy files

1.6 Use Cases

1.6.1 Migrating your translations

You very often need to migrate older translations to newer template or POT files. There are a number of Gettext tools that can manage this but they do not handle the situation where files have been renamed and moved. The *pomigrate2* script allows us to migrate between versions where there has been considerable change.

This migration HOWTO takes you through the steps in a generic fashion so that you can apply it to any of your projects. We use OpenOffice.org as an example for clarity. Our task in the examples is to migrate old translation for OpenOffice.org 1.1.3 to OpenOffice.org 2.0.

Requirements

You will need:

- *pomigrate2*
- *pocompendium*
- A text editor
- A PO editing tool

Preparing the new POT files

We need the new POT files. Either download these from the project or generate them using *moz2po*, *oo2po* or the other tools of the Translate Toolkit. The POT files are templates for the destination files that we will be creating.

```
oo2po -P en-US.sdf ooo-20-pot
```

This will create new POT files in *ooo-20-pot*.

Checking your old PO files for errors

We will be migrating your old PO files into the new POT files. This is a good opportunity to check for encoding errors and inconsistencies.

We use *pocompendium* to check for encoding errors:

```
pocompendium check.po -d ooo-113-old
```

This will create a compendium PO files, *check.po*, from all the PO files in the directory *ooo-113-old*, where *ooo-113-old* contains all your old translations. pocompendium is a wrapper around various Gettext tools, encoding errors will appear as errors from those tools.

Use your text editor to find and correct these errors. If you do not correct these now they will migrate to your new version. Once encoding errors are fixed they're usually gone for good, so it is time well spent.

Optional: Checking your old PO files for consistency

Note: Note this step is optional, a more detailed explanation is given in *Checking for inconsistencies in your translations*.

We now look at consistency within the translations. The first check extracts situations were the same English string was translated in two different ways:

```
pocompendium --ignore-case --accel-amp --errors check.po -d ooo-113-old
```

In *check.po* you will find all situations where the same English text was translated differently. We use `--accel-amp` to remove accelerator markers (you'll change this depending on the one used by the project – we can do & _ or ~). Now view *check.po* in a PO editor or text editor. You will need to correct each inconsistency in the source PO files, using *check.po* as the guide. Many of the errors are usually spelling mistakes. You can regenerate *check.po* from time to time until all inconsistencies are justified or removed.

Then we check for words in your language that are used for more than one English concept. You don't for instance want the same word for *Cancel* and *Delete*. For this we invert the compendium:

```
pocompendium --invert --ignore-case --accel-amp --errors check.po -d ooo-113-old
```

We now have a file similar to the previous one except your language appears in the msgid and the English appears in the msgstr. Look for inconsistencies that would cause problems for the user and correct them in the source files.

Migrate

You are now ready to migrate using *pomigrate2*. You have created your destination POT files and all your PO files are clean and ready to migrate.

```
pomigrate2 ooo-113-old ooo-20-new ooo-20-pot
```

This will take all translations from *ooo-113-old* and migrate them to *ooo-20-new* using *ooo-20-pot* as templates. By default pomigrate2 migrates without any fancy text matching, there are options to allow for fuzzy matching and the use of a compendium. Read the *pomigrate2* help page to find out more about these options.

Techie: what does pomigrate2 do to your file?

This section is for those insanely curious about what pomigrate will do to their files. You don't need to understand this section :-)

- Init stage
  - If a file has not changed location between old and new then it is simply copied across
  - If it has moved then we try to find a file by the same name and move ours there. If there are multiple files by the same name, then we join them together and copy them
  - If a file does not exist then we initialise it
- Update stage
  - We now update our translations using msgmerge or pot2po
  - If you asked for a compendium, we will build one from the existing files and update using it and optionally other external compendiums

That's it. At the end you should have every file that needs translation updated to the latest template files. Files that moved should still be preserved and not lost. Files that where renamed will still be translated if you used a compendium otherwise they will be untranslated.

How well did you do

Congratulations! Your files are now migrated.

You might want to see how much of your old work was reusable in the new version:

```
pocount ooo-20-new
```

This will use *pocount* to count the words in your new files and you can compare the number of translate and untranslated messages from your old version.

Conclusion

Your files have now been migrated and are ready for updating. If files have been moved or renamed, and you used a compendium, then most likely you have most of that work translated.

1.6.2 Checking your files with PO filter

pofilter allows you to check your PO or XLIFF files for certain common errors. This quick-start guide takes you through the process of using this tool, making corrections and merging your correction back into your translations.

The toolkit also other tools that can assist with *quality assurance*.

Quickstart

Use any preferred text editor wherever `vim` *is used.*

1. Select filter(s): `pofilter -l`
2. Run filter(s): `pofilter -i existing_files/ -o errors/ [-t specific tests] [--excludefilter don't perform specific tests]`
3. Delete items you don't want changed, set fuzzy if needed, delete if not needed: `vim errors/*.po`
4. Merge changes back: `pomerge -i errors/ -o existing_files/ -t existing_files/` (will overwrite existing files)
5. Create a patch for the changes: `cvs diff -u existing_files/ > x.diff`
6. Check to see that the updates are what you want: `vim x.diff`
7. Commit changes: `cvs ci existing_files/`

Detailed Description

pofilter runs a number of checks against your translation files. Any messages that fail are output to a set of new files (in the same structure as the source/input files). You then edit these new/output files to correct any errors. Once you are satisfied with your corrections these corrected files are then merged back into the original files using *pomerge*.

Extracting Errors

pofilter will run all tests unless you use the `-t` or `--excludefilter` options. There are over *38 tests* and pofilter can itself provide you with a current list of all the available checks:

```
pofilter -l
```

We want to run the: accelerators, escapes, variables and xmltags tests as these are the ones most likely to break programs at runtime. We are also working with OpenOffice.org PO files created using *oo2po* so we want to ensure that we set the accelerator key marker and variables definitions correctly:

```
pofilter -t accelerators -t escapes -t variables -t xmltags --openoffice existing_
↪files errors
```

Any messages that fail one of the 4 checks will be placed in files in *errors*. We also used the `--openoffice` option to ensure that the tool is aware of the OpenOffice.org accelerator marker (~) and the OpenOffice.org variable styles (OpenOffice.org has over 10 variable styles). You can also specify other styles of project including GNOME, KDE or Mozilla.

You can also specify whether you want fuzzy entries included and checked, by specifying the `--fuzzy` parameter. By default this is off because fuzzy strings are usually known to be broken and will be reviewed by translators anyway.

Similarly you can include items marked for review by specifying `--review` or `--ingnorereview`. By default review items are included. This is not part of the standard Gettext format. We have allowed entries like this when we want to communicate to someone what error we have picked up:

```
# (review) - wrong word for gallery chosen
```

You can run pofilter without the `-t` option. This runs all the checks. This can be confusing if you have a lot of errors as you easily lose focus. One strategy is to run each test individually. This allows you to focus on one problem at a time across a number of files. It is much easier to correct end punctuation on its own then to correct many different types of errors. For a small file it is probably best to run all of the test together.

By using the `--autocorrect` option you can automatically correct some very common errors. Use with caution though. This option assumes you use the same punctuation style as the source text.

Edit the files

Once the errors have been marked you can edit them with any text editor or PO editor e.g. Virtaal. You will be editing the files in the *errors* directory. Only messages that failed one of the tests will be present. If no messages failed then there will be no error PO file for the source PO file. Only critical errors are marked fuzzy – all others are simply marked with the pofilter marker. Critical errors are marked fuzzy as this allows you to simply merge them back into you PO files and then rely on the fact that all po2* tools will ignore a message marked fuzzy. This allows you to quickly eliminate messages that can break builds.

To edit run:

```
vi `find errors -name "*.po"`
virtaal `find errors -name "*.po"`
```

or similar command.

The pofilter marker helps you determine what error was discovered:

```
# (pofilter) <test> - <explanation of test error>
```

Use the test description to help you determine what is wrong with the message. Remember that all your changes will be ported back into the PO files. So if you leave a string fuzzy in the error files, it will become fuzzy in the main files when you merge the corrected file back into the main file. Therefore delete anything you do not want to migrate back when you merge the files. Delete the test comments and fuzzy markings as needed. Leave them in if you want another translator to see them.

The computer can get it wrong, so an error that pofilter finds may in fact not be an error. We'd like to hear about these false positives so that we can improve the checks. Also if you have some checks that you have added or ideas for better checks, then let us know.

Merging your corrections back into the originals

After correcting the errors in the PO files its time to merge these corrections back into the originals using *pomerge*.

```
pomerge -t existing_files -i errors -o files_without_errors
```

If `-t` and `-o` are the same directory, the corrections will be merged into the existing files. Do this only if you are using some kind of version control system so that you can check the changes made by *pomerge*.

Checking the corrections

We have done this against CVS but you could run a normal diff between a good copy and your modifications. Thus we assume in the last step that we merged the corrections into the existing translations:

```
pomerge -t existing_files -i errors -o existing_files
```

Now we check the changes using *cvs diff*:

```
cvs diff -u existing_files > x.diff
```

This creates a unified diff (one with + and - lines so you can see what was added and what was removed) in the file x.diff:

```
vim x.diff
```

Check the diff file in any editor, here we use vim. You should check to see that the changes you requested are going in and that something major did not go wrong. Also look to see if you haven't left any lines with "# (pofilter): test description" which should have been deleted from the error checking PO files. Also check for stray fuzzy markers that shouldn't have been added. You will have to make corrections in the files in *existing_files* not in *errors*.

When you are happy that the changes are correct run:

```
cvs ci existing_files
```

Congratulations you have helped eliminate a number of errors that could give problems when running the application. Now you might want to look at running some of the other tests that check for style and uniformity in translation.

1.6.3 Using csv2po

csv2po allows you to create CSV files from PO files. This allows you to send translation work to translators who do not or cannot use PO Editors but who can use a Spreadsheet.

Quickstart

1. `pofilter --fuzzy --review -t untranslated <po-dir> <po-filtered-dir>` (this step is optional)
2. divide into sections

3. `po2csv <po-dir|po-filtered-dir> <csv-out>`
4. edit in Excel or OpenOffice.org Calc
5. `csv2po --charset=windows-1250 -t templates <csv-in> <po-in>` (you must work against a template directory, the charset option corrects problems with characters sets)
6. `/commands/phase` – to do basic checks sort out encoding issues
7. `pomerge --mergeblank=no -t <po-dir> <po-in> <po-dir>`
8. `git diff` — check the changes
9. `git add` & `git commit` — commit changes

Detailed Description

po2csv allows you to send CSV files, which can be edited in any spreadsheet, to a translator. This document outlines the process to follow from the raw po files -> CSV files -> back to PO. We also look at a case where you may have submitted a subset of the PO files for translation and you need to integrate these.

Creating a subset

This step is optional.

To send a translator only those messages that are untranslated, fuzzy or need review run:

```
pofilter --isfuzzy --isreview -t untranslated <po-dir> <po-filtered-dir>
```

Divide into sections

You might want to divide the work into sections if you are apportioning it to different translators. In that case create new directories:

```
e.g. po-filtered-dir-1 po-filtered-dir-2
or  po-filtered-dir-bob po-filtered-dir-mary
```

Copy files from *po-filtered-dir* to *po-filtered-dir-N* in a way that balance the work or apportions the amounts you want for each translator. Try to keep sections together and not break them up to much e.g. Give one translator all the OpenOffice.org Calc work don't split it between two people – this is just a simple measure to ensure constancy.

Now continue as normal and convert to CSV and perform word counts for each separate directory.

Creating the CSV files

```
po2csv <po-dir|po-filtered-dir> <csv-out>
```

This will create a set of CSV files in *csv-out* which you can compress using zip.

Creating a word count

Professional translators work on source word counts. So we create a word count to go with the file:

```
pocount `find po-dir|po-filtered-dir -name "*.po"`
```

We work on source words regardless of whether the string is fuzzy or not. You might want to get a lower rate for work on fuzzy strings.

Place the word count file in both the PO and CSV directory to avoid the problem of finding it later. Check the number to make sure you haven't inadvertently including something that you didn't want in.

Package the CSV files

```
zip -r9 work.zip <csv-out>
```

Translating

Translators can use most Spreadsheets. Excel works well. However there are a few problems with spreadsheets:

- Encoding – you can sort that out later
- Strings that start with ' – most spreadsheets treat cells starting with ' as text and gobble up the '. A work around is to escape those like this '. po2csv should do this for you.
- Autocorrect – Excel changes ... to a single character and does other odd things. pofilter will help catch these later.
- Sentences with + – or +- will create errors and the translators will have to escape them as + - +-
- Sentences that only contain numbers can get broken: "1." will be converted to "1"

Converting Excel spreadsheets to CSV file

You can, and should, keep your files as CSV files. However, many translators are not the best wizzes at using their spreadsheet. In this case many files will have been changed to XLS files. To convert them by hand is tedious and error prone. Rather make use of xlHtml which can do all the work for you.

```
xlhtml -xp:0 -csv file.xls > file.csv
```

Converting CSV back to PO

Extract the CSV files here we assume they are in *csv-in*:

```
csv2po --charset=windows-1250 -t <templates> <csv-in> <po-in>
```

This will create new PO files in *po-in* based on the CSV files in the *csv-in* and the template PO files in *templates*. You shouldn't run the csv2po command without templates as this allows you to preserve the original file layout. Only run it without `-t` if you are dealing with a partial part of the PO that you will merge back using a *pomerge*.

Note: Running csv2po using the input PO files as templates give spurious results. It should probably be made to work but doesn't

---

Note: You might have encoding problems with the returned files. Use the `--charset` option to convert the file from another encoding (all PO files are created using UTF-8). Usually Windows user will be using something like WINDOWS-1250. Check the file after conversion to see that characters are in fact correct if not try another encoding.

Checking the new PO files

Use *pofilter* to run checks against your new files. Read *Checking your files with PO filter* to get a good idea of how to use the tool.

Removing fuzzies

When you merge work back that you know is good you want to make sure that it overrides the fuzzy status of the existing translations, in order to do that you need to remove the "#, fuzzy" markers.

This is best performed against CVS otherwise who knows what changed.

```
po-in-dir=your-incoming-po-files
po-dir=your-existing-po-files

for pofile in `cd $po-in-dir; find . -name "\*.po"`
do
       egrep -v "^#, fuzzy" < $po-dir/$pofile > $po-dir/${pofile}.unfuzzy && \
       mv $po-dir/${pofile}.unfuzzy $po-dir/$pofile
done
```

Merging PO files into the main PO files

This step would not be necessary if the CSV contained the complete PO file. It is only needed when the translator has been editing a subset of the whole PO file.

```
pomerge --mergeblank=no -t po-dir -i po-in -o po-dir
```

This will take PO files from *po-in* merge them with those in *po-dir* using *po-dir* as the template – i.e. overwriting files in *po-dir*. It will also ignore entries that have blank msgstr's i.e. it will not merge untranslated items. The default behaviour of pomerge is to take all changes from *po-in* and apply them to *po-dir* by overriding this we can ignore all untranslated items.

There is no option to override the status of the destination PO files with that of the input PO. Therefore all your entries that were fuzzy in the destination will still be fuzzy even thought the input was corrected. If you are confident that all your input is correct then relook at the previous section on removing fuzzies.

1.6.4 Creating OpenOffice.org POT files

This quick start guide shows you how to create the PO Template files for your OpenOffice.org translation.

Quick Start

1. Download the latest POT and GSI files
2. `oo2po -P <gsi> <new-pots>`

Detailed Description

Download the latest POT and GSI files

The POT files produced by Pavel Janik contain the associated en-US.sdf file that you need to create your own languages SDF file. This is the same file that produces the POT files. So to begin translating you don't need to go further than this.

- Download the latest POT and GSI files

However, you will need this file if you need to use some of the other features of *oo2po* such as changing the source language from English.

Produce the POT files using oo2po

```
oo2po -P <gsi> <new-pots>
oo2po -P en-US.gsi pot
```

This takes the *en-US.gsi* file and creates POT files in the *pot* directory. The `-P` option ensures that .pot files are created instead of .po file.

If you want to create one large .pot file instead of a lot of small ones, you should use the:

```
oo2po -P --multifile=onefile en-US.gsi pot
```

option as described in *oo2po*.

Produce a POT files with French source text

You will need to have access to a French GSI file. The following commands will create a set of POT files with French as the source language:

```
oo2po -P --source-language=fr fr.gsi pot-fr
```

This will take translations from *fr.gsi* and create a set of POT files in *pot-fr*. These POT files will have French as the source language. You need to make sure that fr.gsi is in fact up to date.

1.6.5 Checking for inconsistencies in your translations

Over time language changes, hopefully not very quickly. However, if your language is new to computers the change might be rapid. So now your older translations have different text to your new translations. In this use case we look at how you can bring alignment back to your translations.

Other cases in which you can expect inconsistencies:

- Multiple translators are involved
- Translations are very old
- You prepared this set of translations with translations from multiple sources
- You changed terminology at some stage in the translation
- You did not do a formal glossary development stage

What we won't be able to achieve

We cannot find grammatical errors and we won't be able to find all cases of words, etc

Scenario

You are translating Mozilla Firefox into Afrikaans. The files are stored in *af*. You have the following issues:

- Your current translator is good but took over from a team of three
- Terminology is well defined but not well used by the old translators

We'll look at the translations first from the English, or source text, point of view. Then we will look at it from the Afrikaans point of view. The first will pick up where we have translated the same English word differently in Afrikaans i.e. an inconsistency. While the second will determine if we use the same English word for different English words, possibly this will confuse a user.

Step 1: Extracting conflicting target text translations

```
poconflicts -I --accelerator="&" af af-conflicts
```

From our existing translation in *af* we extract conflicts and place them in *af-conflicts*. We are ignoring case with `-I` so that `Save as` is considered the same as `Save As`. The `--accelerator` options allows us to ignore accelerators so that `File` is the sane as `&File` which is also the same as `Fi&le`

If we browse into *af-conflicts* we will see a flat structure of words with conflicts.

```
$ cd af-conflicts
$ ls
change.po          disc.po          functionality.po  letter.po          overwrite.po  ␣
↪    restored.po
changes.po         document.po      gb.po             library.po         page.po       ␣
↪    restore.po
character.po       dots.po          graphic.po        light.po           pager.po      ␣
↪    retry.po
chart.po           double.po        grayscale.po      limit.po           percent.po    ␣
↪    return.po
check.po           down.po          grid.po           line.po            pies.po       ␣
↪    right.po
circle.po          drawing.po       group.po
etc...
```

These are normal PO files which you can edit in any PO editor or text editor. If we look at the first file `change.po` we can see that the source text *Change* was translated as *Verander* and *Wysig*. The translators job is now to correct these PO files, ignoring instances where the difference is in fact correct.

Once all fixes have been made we can merge our changes back into the original files.

Step 2: Merging our corrections back into the original files

Our files in *af-conflicts* are in a flat structure. We need to structure them into the hierarchy of the existing PO files.

```
porestructure af-conflicts af-restructured
```

The entries that where in the files in *af-conflicts* have been placed in *af-restructured*, they now appear in the correct place in the directory structure and also appear in the correct file. We are now ready to merge.

```
pomerge -t af -i af-restructure -o af
```

Using the existing files in *af* we merge the corrected and restructured file from *af-restructure* and place them back into *af*. Note: use a different output directory if you do not want to overwrite your existing files. All your conflict corrections are now in the correct PO file in *af*.

You might want to run **Step 1** again to make sure you didn't miss anything or introduce yet another problem.

Next we look at the inverted conflict problem.

Step 3: Extracting conflicts of meaning

If you have used the same Afrikaans word for two different English words then you could have created a conflict of meaning. For instance in our Xhosa translations the word `Cima` was used for both `Delete` and `Cancel`. Clearly this is a serious issue. This step will allow us to find those errors and take action.

```
poconflicts -v -I --accelerator="&" af af-conflicts-invert
```

We use the same command line as in **Step 1** but add `-v` to allow us to invert the match. We are also now outputting to *af-conflicts-invert* to make things clear.

This time the PO files that are created have Afrikaans names

```
$ cd af-conflicts-invert
$ ls
dataveld.po              grys.po             lisensieooreenkoms.po  paragraaf.po      ␣
↪  sny.po
datumgekoop.po           hallo.po            lysinhoud.po           pasmaak.po        ␣
↪  soek.po
datum.po                 hiperboliese.po     maateenheid.po         persentasie.po    ␣
↪  sorteer.po
deaktiveer.po            hoekbeheer.po       maatskappynaam.po      posadres.po       ␣
↪  sorteervolgorde.po
etc...
```

We edit these as usual. You need to remember that you will see a normal PO file but that you are looking at how the translation might be confusing to a user. If you see the same Afrikaans translation for two different English terms but there is no conflict of meaning or no alternative then leave it as is. You will find a lot of these instances so the results are less dramatic then the results from a normal conflict analysis.

Lastly follow **Step 2** to restructure and merge these conflicts back into your translations

Conclusion

You've now gone a long way to improving the quality of your translations. Congratulations! You might want to take some of what you've learnt here to start building a terminology list that can help prevent some of the issues you have seen.

1.6.6 Creating a terminology list from your existing translations

If you did not create a terminology list when you started your translation project or if you have inherited some old translations you probably now want to create a terminology list.

A terminology list or glossary is a list of words and phrases with their expected translation. They are useful for ensuring that your translations are consistent across your project.

With existing translations you have embedded a list of valid translation. This example will help you to extract the terms. It is only the first step you will need to review the terms and must not regard this as a complete list. And of course you would want to take your corrections and feed them back into the original translations.

Quick Overview

This describes a multi-stage process for extracting terminology from translation files. It is provided for historical interest and completeness, but you will probably find that using *poterminology* is easier and will give better results than following this process.

- Filter our phrases of more than N words
- Remove obviously erroneous phrases such as numbers and punctuation
- Create a single PO compendium
- Extract and review items that are fuzzy and drop untranslated items
- Create a new PO files and process into CSV and TMX format

Get short phrases from the current translations

We will not be able to identify terminology within bodies of text, we are only going to extract short bit of text i.e. ones that are between 1 and 3 words long.

```
pogrep --header --search=msgid -e '^\w+(\s+\w+){0,2}$' zulu zulu-short
```

We use `--header` to ensure that the PO files have a header entry (which is important for encoding). We are searching only in the msgid and the regular expression we use is looking for a string with between 1 and 3 words in it. We are searching through the folder *zulu* and outputting the result in *zulu-short*

Remove any translations with issues

You can for instance remove all entries with only a single letter. Useful for eliminating all those spurious accelerator keys.

```
pogrep --header --search=msgid -v -e "^.$" zulu-short zulu-short-clean
```

We use the `-v` option to invert the search. Our *cleaner* potential glossary words are now in *zulu-short-clean*. What you can eliminate is only limited by your ability to build regular expressions but yu could eliminate:

- Entries with only numbers
- Entries that only contain punctuation

Create a compendium

Now that we have our words we want to create a single files of all terminology. Thus we create a PO compendium:

```
~/path/to/pocompendium -i -su zulu-gnome-glossary.po -d zulu-short-clean
```

You can use various methods but our bash script is quite good. Here we ignore case, `-i`, and ignore the underscore (_) accelerator key, `-su`, outputting the results in.

We now have a single file containing all glossary terms and the clean up and review can begin.

Split the file

We want to split the file into translated, untranslated and fuzzy entries:

```
~/path/to/posplit ./zulu-gnome-glossary.po
```

This will create three files:

- zulu-gnome-glossary-translated.po – all fully translated entries
- zulu-gnome-glossary-untranslated.po – messages with no translation
- zulu-gnome-glossary-fuzzy.po – words that need investigation

```
rm zulu-gnome-glossary-untranslated.po
```

We discard `zulu-gnome-glossary-untranslated.po` since they are of no use to us.

Dealing with the fuzzies

The fuzzies come in two kinds. Those that are simply wrong or needed updating and those where there was more then one translation for a given term. So if someone had translated 'File' differently across the translations we'd have an entry that was marked fuzzy with the two options displayed.

```
pofilter -t compendiumconflicts zulu-gnome-glossary-fuzzy.po zulu-gnome-glossary-
↪conflicts.po
```

These compendium conflicts are what we are interested in so we use pofilter to filter them from the other fuzzies.

```
rm zulu-gnome-glossary-fuzzy.po
```

We discard the other fuzzies as they where probably wrong in the first place. You could review these but it is not recommended.

Now edit `zulu-gnome-glossary-conflicts.po` to resolve the conflicts. You can edit them however you like but we usually follow the format:

```
option1, option2, option3
```

You can get them into that layout by doing the following:

```
sed '/#, fuzzy/d; /\"#-#-#-#-# /d; /# (pofilter) compendiumconflicts:/d; s/\\n"$/, "/
↪' zulu-gnome-glossary-conflicts.po > tmp.po
msgcat tmp.po > zulu-gnome-glossary-conflicts.po
```

Of course if a word is clearly wrong, misspelled etc. then you can eliminate it. Often you will find the "problem" relates to the part of speech of the source word and that indeed there are two options depending on the context.

You now have a cleaned fuzzy file and we are ready to proceed.

Put it back together again

```
msgcat zulu-gnome-glossary-translated.po zulu-gnome-glossary-conflicts.po > zulu-
↪gnome-glossary.po
```

We now have a single file `zulu-gnome-glossary.po` which contains our glossary texts.

Create other formats

It is probably good to make your terminology available in other formats. You can create CSV and TMX files from your PO.

```
po2csv zulu-gnome-glossary.po zulu-gnome-glossary.csv
po2tmx -l zu zulu-gnome-glossary.po zulu-gnome-glossary.tmx
```

For the terminology to be usable by Trados or Wordfast translators they need to be in the following formats:

- Trados – comma delimited file `source,target`
- Wordfast – tab delimited file `source[tab]target`

In that format they are now available to almost all localisers in the world.

FIXME need scripts to generate these formats.

1.6.7 The work has only just begun

The lists you have just created are useful in their own right. But you most likely want to keep growing them, cleaning and improving them.

You should as a first step review what you have created and fix spelling and other errors or disambiguate terms as needed.

But congratulations a Terminology list or Glossary is one of your most important assets for creating good and consistent translations and it acts as a valuable resource for both new and experienced translators when they need prompting as to how to translate a term.

1.6.8 Running the tools on Microsoft Windows

Since the toolkit is written in Python, it should work perfectly on Windows.

Add the toolkit to your path

Windows 95/98

You might need to add the installation directory of the translate toolkit to your path

```
path "C:\Program Files\translate-toolkit\"
```

This will work for one session, but will be lost when you reboot again. Therefore you might want to add it to the autoexec.bat file.

Windows 2000/XP

You can add to the path permanently. Check this useful guide. You should add the following to your path:

```
C:\Programs Files\translate-toolkit\
```

If you have the Gettext tools installed, add it to your path as well:

```
C:\Program Files\GnuWin32\bin\
```

Change Windows file to Unix file

Some programs in Windows will add CRLFs to the file which is considered rather poor practice for l10ns that require Unix files. To fix a text file, drag and drop it to the dos2unix.exe utility from http://www.bastet.com/

1.6.9 Cleanup translator comments

Translate Toolkit 1.1 saw source comments being converted to developer comments instead of translator comments.

This use case shows you how to get rid of the old translator comments.

The Change

We used to put all source comments into translator comments.

```
# Some Comment
```

But now place them in developer comments.

```
#. Some Comment
```

This ensures that these source comments are updated to the newest versions from the source files, which is a good thing. Translator comments survive these updates, just like you want, while developer comments are discarded.

If you don't clean up your PO files you will now end up with:

```
# Some Comment
#. Some Comment
```

Thus a duplicated comment. Fortunately you only need to clean your PO files once.

Removing old translator comments

Note: This will remove all your translator comments. So if you have some that you actually want to keep then you will need to manual editing

Removal is simple using *pocommentclean*:

```
pocommentclean my-po-dir
```

Which will clean all your PO files in `my-po-dir`

`pocommentclean` is simply a nice wrapper for this sed command:

```
sed -i "/^#$/d;/^#[^\:\~,\.]/d" $(find po -name "*.po")
```

This will delete all lines starting with # that are not used by PO for locations (#:), automatic/developer comments (#.), state (#,) and obsolete (#~).

You can now safely commit your changes and begin your migrations using *pot2po* of *pomigrate2*

1.6.10 Creating Mozilla POT files

You can do this using Mozilla source from CVS or Mercurial

Using Mercurial

Since Firefox 3.1 and Thunderbird 3.0, Mozilla has switched to using Mercurial for version control. See the Mozilla's L10n on Mercurial page for instructions on how to checkout and update your Mozilla sources and l10n files.

You can use *get_moz_enUS.py* to extract an en-US directory from the source tree:

```
get_moz_enUS.py -s mozilla-central/ -d l10n/ -p browser
```

This will move the correct en-US files to `l10n/en-US`. You can now create POT files as follows:

```
moz2po -P l10n/en-US l10n/pot
```

This will create the POT files in `l10n/pot` using the American English files from `en-US`. You now have a set of POT files that you can use for translation or updating your existing PO files.

There are also *other scripts* that can help with creating and updating POT and PO files for Mozilla localisation.

Using CVS

Firefox versions before 3.1 and Thunderbird versions before 3.0 still has its source in CVS. Check out files from the Mozilla repository. If you don't want to checkout all files do:

```
make -f client.mk l10n-checkout
```

The English files are in the `mozilla/` module, while the translated files all reside in the `l10n/` module. They have different structure but not enough to kill you.

Once you have checked out `mozilla/` you will need to get the correct files for en-US. To do this we will create en-US as a pseudo language.

```
make -f tools/l10n/l10n.mk create-en-US
```

This will move the correct en-US files to `l10n/en-US`. You can now create POT files as follows:

```
moz2po -P l10n/en-US l10n/pot
```

This will create the POT files in `l10n/pot` using the American English files from `en-US`. You now have a set of POT files that you can use for translation or updating your existing PO files.

1.6.11 Document translation

Translating documents can be quite different from translating software interfaces. Many issues specific to software localisation might not be relevant in documents, such as accelerators, translation length, constructed phrases, etc. However, document translation has several other issues that is good to be aware of.

Preparing for translation

Ideally a document should be prepared for translation. A good source document will make translation easier. Possibilities:

- Proofread the document (spelling, grammar, clarity)
- Use consistent terminology
- Read "writing for translation"
- For structured documents, use proper structure like headings and subheadings instead of using style only.

Translation

A lot can be said about translation in general, but this is only meant to give you some tips.

Be to be aware of issues arising out of translation memory. You could possibly have exact matches (identical string translated before), or In Context Exact (ICE) matches, where some translation tools will specifically indicate that the translation is identical, but also that the surrounding text from the paragraph is the same. It could also indicate agreement with regards to domain, file, date, etc.

Post-processing

After generating the translated document, you very likely need to do some post processing. Things to consider:

- Ensuring correct translation in cases where context might not have been obvious during translation
- Document layout, page layout
- Fonts or other styling changes
- Style of generated content, such as numbers
- Generated sections, such as Table of contents, list of figures, index, variables
- *Migrating an older version of your translations to the latest templates*
- *Checking for technical errors in your translations*
- *Translating using only a spreadsheet* (a look at the whole roundtrip from PO to CSV and back)
- *Creating OpenOffice.org POT files*
- *Checking for inconsistencies in your translations*
- *Creating a terminology list from your existing translations*
- *Running the tools on Microsoft Windows*
- Using `phase` for the complete translation roundtrip
- *Cleanup translator comments*
- *Creating Mozilla POT files*

- *Document translation*

1.7 Translation Related File Formats

These are the different storage formats for translations and files associated with translations that are supported by the toolkit. See also *Standards conformance* for standards conformance.

The Translate Toolkit implements a set of *classes* for handling translation files which allows for a uniform API which covers other issues such as *quoting and escaping* of text.

1.7.1 Primary translation formats

XLIFF

XLIFF[*] is the OASIS standard for translation.

References

- XLIFF Standard
- OASIS XLIFF Technical Committee website

Flavours

XLIFF also has documents that specify the conversion from various standard source documents and localisation formats.

- PO – For conformance to the po2xliff spec, see *xliff2po*.
  - Draft XLIFF 1.2 Representation Guide for Gettext PO
- HTML – not implemented
  - Draft XLIFF 1.2 Representation Guide for HTML
- Java (includes .properties and Java resource bundles) – not implemented
  - Draft XLIFF 1.2 Representation Guide for Java Resource Bundles
- ICU Resource Bundles – not officially being developed by XLIFF – Proposed representation guide

Standard conformance

Done

- File creation and parsing
- API can create multiple files in one XLIFF (some tools only read the first file)
- source-language attribute
- **trans-unit with**
  - note: addnote() and getnotes()

    - **state**
        - fuzzy: isfuzzy() and markfuzzy()
        - translated: marktranslated()
        - approved
        - needs-review-translation: isreview(), markreviewneeded()
    - id: setid()
    - context-group: createcontextgroup()
- context groups
- alt-trans

XLIFF and other tools

Here is a small report on XLIFF support by Windows programs.

PO Files

PO files use the file format of the Gettext tools.

See also:

Gettext manual

Supported Features

- Headers
- Language header (since gettext version 0.17)
- Plural forms and plural form handling
- Message context

```
msgctxt "noun"
msgid "View"
msgstr ""
```

- Normal comments

```
# this is another comment
```

- Automatic comments

```
#. comment extracted from the source code
```

- Source location comments

```
#: sourcefile.xxx:35
```

- Typecomments

```
#, fuzzy
```

- Msgidcomments, also known as KDE style comments as they are used by KDE for message disambiguation and comments to translators.

  Note: Support for this is being phased out in favor of `msgctxt`.

```
msgid "_: comment\n"
"translation"
```

- Obsolete messages

```
#~ msgid "Blah"
#~ msgstr "Bleeh"
```

- Previous msgid

```
#| msgid "previous message"
```

- Previous msgctxt

```
#| msgctxt "previous context"
```

1.7.2 Other translation formats

CSV

CSV (Comma Separated Values) is a simple file format for general data interchange. It can be used in the toolkit for simple data interchange, and can be edited with most spreadsheet programs. There is no formal specification for the CSV file format, but more information can be acquired from Comma-Separated Values

Conformance

CSV files were initially used to convert from and to *po files*, and therefore contained three columns as follows:

Column	Description
location	A column with the location of the original msgid (in other words, a line in a programming source file, as indicated in the #: comments of PO files).
source	The source text (or msgid)
target	The target text (or msgstr)

Tabs and newlines are maintained, although it is not clear how easy it is to edit these things in a spreadsheet.

Quoting is a problem, because the different spreadsheet programs handle these things differently. Notably, Microsoft's excel handles single quotes slightly differently. In future, it might be worthwhile to handle excel CSV as a different format from other CSV files. An entry like 'mono' is ambiguous as it is not sure whether this refers simply to the word *mono* or to the entry *'mono'* quoted with single quotes. (Example from Audacity pot file)

INI Files

Also known as initialisation files. These are in some cases used to store translations.

Conformance

The toolkit uses iniparse, an INI file parser that preserves layout and follows the .ini format as supported by the Python language.

Dialects

The format supports two dialects:

- default: standard iniparse handling of INI files
- inno: follows Inno escaping conventions

References

Further information is available on .ini files:

- Wikipedia INI file format article
- Unofficial specification

Mozilla and Java properties files

The Translate Toolkit can manage Java .properties files with the *prop2po* and po2prop tool. As part of the Mozilla localisation process, the *moz2po* tool handles the properties files along with the other files. The tools can also handle Skype .lang files. Some related formats with their own documentation:

- *Mac OSX strings*
- *Adobe Flex* properties files.

Features

- Fully manage Java escaping (Mozilla non-escaped form is also handled)
- Preserves the layout of the original source file in the translated version

New in version 1.12.0.

- Mozilla accelerators – if a unit has an associated access key entry then these are combined into a single unit

Not implemented

- We don't allow filtering of unchanged values. In Java you can inherit translations, if the key is missing from a file then Java will look to other files in the hierarchy to determine the translation.

Examples

```
editmenu.label = "Edit"
saveas.label = "Save As"
```

References

- Java Properties Class's load() describes the properties format.
- http://www.oracle.com/webfolder/technetwork/jsc/dtd/properties.dtd – alternate XML based property representation

Mozilla DTD format

Mozilla makes use of a .dtd file to store many of its translatable elements, the *moz2po* converter can handle these.

References

- XML specification

Features

- Comments – these are handled correctly and integrated with the unit
- Accelerators – if a unit has an associated access key entry then these are combined into a single unit
- Translator directive – all LOCALIZATION NOTE items such as DONT_TRANSLATE are handled and such items are discarded
- Entities – some entities such as `&` or `&quot;` are expanded when reading DTD files and escaped when writing them, so that translator see and type `&` and `"` directly

Issues

- We don't expand some character entities like `<`, `&#38;` – this doesn't break anything but it would be nicer to see © rather than `&copy;`

OpenOffice.org GSI/SDF format

OpenOffice.org uses an internal format called SDF to manage localisation text. The toolkit can successfully manage all features of this format converting it to XLIFF or PO format with the *oo2po* and oo2xliff tools.

Features

- Handles all translatable text from the SDF
- Can also use 'x-comments' 'language' found in the SDF to provide translator comments

PHP

Many PHP programs make use of a localisable string array. The toolkit supports the full localisation of such files with *php2po* and po2php.

Conformance

Our format support allows:

- Single and double quoted strings (both for keys and values)

```
<?php
$variable = 'string';
$messages["language"] = 'Language';
define('item', "another string");
```

- PHP simple variable syntax

```
<?php
$variable = 'string';
$another_variable = "another string";
```

- PHP square bracket array syntax

```
<?php
$messages['language'] = 'Language';
$messages['file'] = "File";
$messages["window"] = 'Window';
$messages["firewall"] = "Firewall";
```

- PHP array syntax

  New in version 1.7.0.

```
<?php
// Can be 'array', 'Array' or 'ARRAY'.
$lang = array(
    'name' => 'value',
    'name2' => "value2",
    "key1" => 'value3',
    "key2" => "value4",
);
```

- PHP define syntax

  New in version 1.10.0.

```
<?php
define('item', 'string');
define('another_item', "another string");
define("key", 'and another string');
define("another_key", "yet another string");
```

- PHP short array syntax

  New in version 2.3.0.

```
<?php
$variable = [
    "foo" => "bar",
    "bar" => "foo",
];
```

- Heredoc

  New in version 2.3.0.

```
<?php
$variable = <<<EOT
bar
EOT;
```

- Nowdoc

  New in version 2.3.0.

```
<?php
$variable = <<<'EOD'
Example of string
spanning multiple lines
using nowdoc syntax.
EOD;
```

- Escape sequences (both for single and double quoted strings)

```
<?php
$variable = 'He said: "I\'ll be back"';
$another_variable = "First line \n second line";
$key = "\tIndented string";
```

- Multiline entries

```
<?php
$lang = array(
   'name' => 'value',
   'info' => 'Some hosts disable automated mail sending
          on their servers. In this case the following features
          cannot be implemented.',
   'name2' => 'value2',
);
```

- Various layouts of the id

```
<?php
$string['name'] = 'string';
$string[name] = 'string';
$string[ 'name' ] = 'string';
```

- Comments

  Changed in version 1.10.0.

```
<?php
# Hash one-line comment
$messages['language'] = 'Language';
```

(continues on next page)

(continued from previous page)

```
// Double slash one-line comment
$messages['file'] = 'File';

/*
   Multi-line
   comment
*/
$messages['help'] = 'Help';
```

- Whitespace before end delimiter

  New in version 1.10.0.

```
<?php
$variable = 'string'     ;

$string['name'] = 'string'     ;

$lang = array(
   'name' => 'value'           ,
);

define('item', 'string'    );
```

- Nested arrays with any number of nesting levels

  New in version 1.11.0.

```
<?php
$lang = array(
   'name' => 'value',
   'datetime' => array(
      'TODAY' => 'Today',
      'YESTERDAY' => 'Yesterday',
      'AGO' => array(
          0 => 'less than a minute ago',
          2 => '%d minutes ago',
          60 => '1 hour ago',
      ),
      'Converted' => 'Converted',
      'LAST' => 'last',
   ),
);
```

- Whitespace in the array declaration

  New in version 1.11.0.

```
<?php
$variable = array    (
   "one" => "this",
   "two" => "that",
);
```

- Blank array declaration, then square bracket syntax to fill that array

  New in version 1.12.0.

```
<?php
global $messages;
$messages = array();

$messages['language'] = 'Language';
$messages['file'] = 'File';
```

- Unnamed arrays:

  New in version 2.2.0.

```
<?php
return array(
    "one" => "this",
);
```

- Array entries without ending comma:

  New in version 2.3.0.

```
<?php
$variable = array(
    "one" => "this",
    "two" => "that"
);
```

- Array entries with space before comma:

  New in version 2.3.0.

```
<?php
$variable = array(
    "one" => "this",
    "two" => "that"    ,
);
```

- Nested arrays declared on the next line:

  New in version 2.3.0.

```
<?php
$variable = array(
    "one" =>
        array(
            "two" => "dous",
        ),
);
```

- Nested arrays with blank entries:

  New in version 2.3.0.

```
<?php
$variable = array(
    "one" => array(
            "" => "",
            "two" => "dous",
        ),
);
```

- Strings with slash asterisk on them:

  New in version 2.3.0.

```
<?php
$variable = array(
    'foo' => 'Other value /* continued',
 );
```

- Array entries with value on next line:

  New in version 2.3.0.

```
<?php
$variable = array(
    'foo' =>
        'bar',
 );
```

- Array defined in a single line:

  New in version 2.3.0.

```
<?php
$variable = array( 'item1' => 'value1', 'item2' => 'value2', 'item3' => 'value3'␣
↪);
```

- Keyless arrays:

  New in version 2.3.0.

```
<?php
$days = array('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
↪'Saturday');
```

- Nested arrays without key for a nested array:

  New in version 2.3.0.

```
<?php
$lang = array(array("key" => "value"));
```

- Concatenation of strings and variables:

  New in version 2.3.0.

```
<?php
$messages['welcome'] = 'Welcome ' . $name . '!';
$messages['greeting'] = 'Hi ' . $name;
```

- Assignment in the same line a multiline comment ends:

  New in version 2.3.0.

```
<?php
/*
   Multi-line
   comment
*/ $messages['help'] = 'Help';
```

- Keyless arrays assigned to another array:

```
<?php
$messages['days_short'] = array('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat');
```

- Laravel plurals are supported in the `LaravelPHPFile` class:

```
<?php
return [
    'apples' => 'There is one apple|There are many apples',
];
```

Non-Conformance

The following are not yet supported:

- There are currently no known limitations.

Qt .ts

The Qt toolkit uses a .ts file format to store translations which are traditionally edited using Qt Linguist.

References

The format is XML and seems to only have been documented properly since Qt 4.3

- Current DTD Specification for Qt 5, older versions; Qt 4.3
- http://svn.ez.no/svn/ezcomponents/trunk/Translation/docs/linguist-format.txt

Complete

Note that *ts2po* uses and older version and does not support all of these features. Virtaal, Pootle and other users of the new ts class support the following:

- Context
- Message: status (unfinished, finished, obsolete), source, translation, location
- Notes: comment, extracomment, translatorcomment (last two since Toolkit 1.6.0)
- Plurals: numerusform

TODO

Note: A new parser has been added to the toolkit in v1.2. This allows Virtaal, *pocount* and other users to work with v1.1 of the .ts format. This corrects almost all of the issues listed below. The converter *ts2po* continues to use the older storage class and thus continue to experience some of these problems.

- Compliance with above DTD
- byte: within various text areas

- translation: obsolete (currently handled with comments in conversion to PO. But should be able to convert Obsolete PO back into obsolete TS. This might mean moving this format properly onto the base class).
- lengthvariants
- *comment: various new comment fields
- old*: ability to store previous source and comments

Validate

These might work but need validation

- Encoding handling for non-UTF-8 file encodings

Windows RC files

New in version 1.2.

Windows .rc files, or resource files, are used to store translatable text, dialogs, menu, etc. for Windows applications. The format can be handled by the Translate Toolkit *rc2po* and po2rc.

Conformance

The actual specification of .rc files is hard to come by. The parser was built using WINE .rc files as a reference. This was done as WINE is a good target for .rc translations. We are confident though that the extraction will prove robust for all .rc files.

Useful resource

- RC converter
- ReactOS translation instructions

Supported elements

- DIALOG, DIALOGEX: All translatables
- MENU: POPUP, MENUITEM
- STRINGTABLE
- LANGUAGE: We only parse the first language tag, further LANGUAGE section are ignored

Bugs

- There may be problems with very deeply nested MENU's
- LANGUAGE elements cannot yet be updated in *po2rc* (Issue 360)

Mac OSX strings

New in version 1.8.

Mac OSX .strings files are used for some Cocoa / Carbon application localization, such as for the iPhone, iPod, and OSX. They are somewhat similar to Java properties, and therefore *prop2po* and po2prop are used for conversion.

References

- Localising string resources
- Manual creation of .strings files
- String format specifiers

Adobe Flex properties files

New in version 1.8.

Adobe Flex applications use *Java properties* encoded in UTF-8. The *prop2po* and po2prop commands are used for conversion.

References

- Description for Adobe Flex properties files

Haiku catkeys

New in version 1.8.

Localisation for the Haiku operating system is done with a file format called catkeys. It is a bilingual file format.

The is a tab separated value (TSV) file, where each line represents a translatable unit. A line consists of four elements:

Column	Description
source	The source text (in English)
context	The context of where the source text is used.
remarks	An additional remark by the developer, that gives a hint to the translator. Within the context of this toolkit, this is stored as the note of the unit.
target	The target text

The first line of the file is the header file, with four tab separated values:

- The version (currently: 1)
- The name of the language in lower case (for example: catalan)
- The signature (for example: x-vnd.Haiku-StyledEdit)
- A checksum (32 bit unsigned integer)

The checksum is calculated by an algorithm that hashes the source, context and remark values of all units. The target text is not relevant for the checksum algorithm.

Links

- Some notes about the format
- Some example files

Android string resources

Android programs make use of localisable string resources.

Note: The toolkit supports this format, but still doesn't provide any converter.

References

- Android Resource files reference
- Android String resources reference
- Localizing Android Applications tutorial
- Reference for translatable attribute

.NET Resource files (.resx)

.Net Resource (.resx) files are a monolingual file format used in Microsoft .Net Applications. The .resx resource file format consists of XML entries, which specify objects and strings inside XML tags. It contains a standard set of header information, which describes the format of the resource entries and specifies the versioning information for the XML used to parse the data. Following the header information, each entry is described as a name/value pair.

Comments can be added per string using the optional `<comment>` field. As only one comment field is available, both translator and developer comments are stored in the same place. Translator comments are automatically wrapped with brackets and prefixed with 'Translator Comment:' during the po2resx process to make it easy to distinguish comment origin inside the .resx files.

Example:

```
<data name="key">
    <value>hello world</value>
    <comment>Optional developer comment about the string [Translator Comment:␣
↪Optional translator comment]</comment>
</data>
```

resx2po and po2resx are used for conversion.

References

- Resources in .Resx File Format
- ASP.NET Web Page Resources Overview

Mozilla .lang files

Mozilla's custom .lang format is used for some of their websites.

References

- .lang specification
- www.mozilla.org repository of translations
- *CSV*
- *INI Files* (including Inno Setup .isl dialect)
- Java *Mozilla and Java properties files* (also Mozilla derived properties files)
- Mozilla *Mozilla DTD format*
- OpenOffice.org *OpenOffice.org GSI/SDF format* (Also called SDF)
- *PHP* translation arrays
- Qt Linguist *Qt .ts* (both 1.0 and 1.1 supported, 1.0 has a converter)
- Symbian localization files
- Windows *Windows RC files* files
- Mac OSX *Mac OSX strings* files (also used on the iPhone) (from version 1.8)
- Adobe *Adobe Flex properties files* files (from version 1.8)
- Haiku *Haiku catkeys* (from version 1.8)
- *Android string resources* (supports storage, not conversion)
- *.NET Resource files (.resx)* .NET Resource files (.resx)
- Mozilla *.lang* files

1.7.3 Translation Memory formats

TMX

TMX is the LISA OSCAR standard for translation memories.

Standard conformance

Summary: TMX version 1.4 conformance to Level 1, except that no markup is stripped.

- All required header fields are supplied.
- The `adminlang` field in the header is always English.
- None of the optional header fields are supplied.
- We assume that only two languages are used (source and single target language).
- No special consideration for segmentation.
- Currently text is treated as plain text, in other words no markup like HTML inside messages are stripped or interpreted as it should be for complete Level 1 conformance.

Wordfast Translation Memory

The Wordfast format, as used by the Wordfast translation tool, is a simple tab delimited file.

The storage format can read and write Wordfast TM files.

Conformance

- Escaping – The format correctly handles Wordfast `&'XX;` escaping and will unescape and escape seamlessly.
- Soft-breaks – these are not managed and are left as escaped
- Replaceables – these are not managed
- Header – Only basic updating and reading of the header is implemented
- Tab-separated value (TSV) – the format correctly handles the TSV format used by Wordfast. There is no quoting, Windows newlines are used and the `\t` is used as a delimiter (see issue 472)
- *TMX*
- *Wordfast Translation Memory*: TM
- Trados: .txt TM (from v1.9.0 – read only)

1.7.4 Glossary formats

OmegaT glossary

New in version 1.5.1.

OmegaT allows a translator to create a terminology list of glossary files. It uses this file to provide its glossary matches to the OmegaT users.

Format specifications

The glossary files is a tab delimited files with three columns:

1. source
2. target
3. comment

The files is stored in the system locale if the files extension is `.txt` or in UTF-8 if the file extension is `.utf8`.

Conformance

The implementation can load files in UTF-8 or the system encoding.

Issues

- There has not been extensive testing on system encoded files and there are likely to be issues in these files for encodings that fall outside of common ASCII characters.
- Files with additional columns are read correctly but cannot be written.

Qt Phrase Book (.qph)

New in version 1.2.

Qt Linguist allows a translator to collect common phrases into a phrase book. This plays a role of glossary lookup as opposed to translation memory.

Conformance

There is no formal definition of the format, although it follows a simple structure

```
<!DOCTYPE QPH><QPH>
  <phrase>
    <source>Source</source>
    <target>Target</target>
    <definition>Optional definition</definition>
  </phrase>
</QPH>
```

Missing features

There are no missing features in our support in the toolkit. The only slight difference are:

- We don't focus on adding and removing items, just updating and reading
- Comments are not properly escaped on reading, they might be on writing
- An XML header is output on writing while it seems that no files in the wild contain an XML header.
- The `<definition>` is aimed at users, the toolkits addnote feature focuses on programmer, translators, etc comments while there is really only one source of comments in a .qph. This causes duplication on the offline editor.

TBX

TBX is the LISA OSCAR standard for terminology and term exchange.

For information on more file formats, see *Standards conformance*.

References

- Standard home page
- Specification
- ISO 30042 – TBX is an approved ISO standard
- Additional TBX resources

You might also be interested in reading about TBX-Basic – a simpler, reduced version of TBX with most of the useful features included.

Additionally notes and examples about TBX are available in Terminator TBX conformance notes which might help understanding this format.

Also you might want to use TBXChecker in order to check that TBX files are valid. Check the TBXChecker explanation.

Conformance

Translate Toolkit TBX format support allows:

- Basic TBX file creation
- Creating a bilingual TBX from CSV using *csv2tbx*
- Using `<tig>` tags only
- Simple extraction of Parts of Speech and definitions

Non-Conformance

The following are not yet supported:

- `id` attribute for `<termEntry>` tags
- Multiple languages
- Multiple translations in the same language
- Cross references
- Context
- Abbreviations
- Synonyms
- `<ntig>` tag, read and write

Other features can be picked from the Terminator TBX conformance notes which also include examples and notes about the TBX format.

Universal Terminology eXchange (UTX)

New in version 1.9.

UTX is implemented by the Asia-Pacific Association for Machine Translation

Resources

- UTX site
- Current Specification (implementation is based on UTX 1.0 which is no longer available)

Conformance

The Translate Toolkit implementation of UTX can correctly:

- Handle the header. Although we don't generate the header at the moment
- Read any of the standard columns and optional columns. Although we can access these extra columns we don't do much with them.

Adjustments and not implemented features where the spec is not clear:

- We do not implement the "#." comment as we need clarity on this

- The "<space>" override for no part of speech is not implemented
- The spec calls for 2 header lines, while examples in the field have 2-3 lines. We can read as many as supplied but assume the last header line is the column titles
- We remove # from all field line entries, some examples in the field have `#tgt` as a column name
- *OmegaT glossary* (from v1.5.1)
- *Qt Phrase Book (.qph)*
- *TBX*
- *Universal Terminology eXchange (UTX)* (from v1.9.0)

1.7.5 Formats of translatable documents

HTML

The Translate Toolkit is able to process HTML files using the *html2po* converter.

Conformance

- Can identify almost all HTML elements and attributes that are localisable.
- The localisable and localised text in the PO/POT files is fragments of HTML. Therefore, reserved characters must be represented by HTML entities:
  - Content from HTML elements uses the HTML entities `&` (&), `<` (<), and `>` (>).
  - Content from HTML attributes uses the HTML entities `&quot;` (") or `&apos;` (').
- Leading and trailing tags are removed from the localisable text, but only in matching pairs.
- Can cope with embedded PHP, as long as the documents remain valid HTML. If you place PHP code inside HTML attributes, you need to make sure that the PHP doesn't contain special characters that interfere with the HTML.

References

- Reserved characters
- Using character entities

Flat XML

The Translate Toolkit is able to process flat XML files using the *flatxml2po* converter.

Flat XML (eXtensible Markup Language) is a simple monolingual file format similar to a very basic form of the *Android string resources* format. Flat in this context means a single level of elements wrapped in the root-element with no other structuring.

Conformance

- Single-level XML with attributes identifying a resource:

```
<root>
  <str key="hello_world">Hello World!</str>
  <str key="resource_key">Translated value.</str>
</root>
```

- Customizable element- and attribute-names (including namespaces):

```
<dictionary xmlns="urn:translate-toolkit:flat-xml-dictionary">
  <entry name="hello_world">Hello World!</entry>
  <entry name="resource_key">Translated value.</entry>
</dictionary>
```

- Value whitespace is assumed to be significant (equivalent to setting `xml:space="preserve"`):

```
<root>
  <str key="multiline">The format assumes xml:space="preserve".
There is no need to specify it explicitly.

This assumption only applies to the value element; not the root element.</str>
</root>
```

- Non-resource elements and attributes are preserved (assuming the same file is also used when converting back to XML):

```
<root>
  <str key="translate_me">This needs to be translated</str>
  <const name="the_answer" hint="this isn't translated">42</const>
  <str key="important" priority="100">Some important string</str>
</root>
```

- Indentation can be customized to match an existing and consistent style:

```
<root>
        <str key="indent">This file uses 8 spaces for indent</str>
        <str key="tab_works">Tabs can also be used; but this is limited to the␣
↪Python API at this point</str>
        <str key="linerized">No indent (all in one line) is also supported</str>
        <str key="note_on_eof">End-of-file *always* has a LF to satisfy VCS</str>
</root>
```

Note: To avoid potential issues and extraneous changes in diffs, this format always forces an ending linefeed by default for compatibility with various Version control systems (such as Git).

Non-Conformance

While the format is flexible, not all features are supported:

- Mixed element/attribute names (as well as different namespaces for root- and value-element) and nested structures additional child elements. This format intentionally focuses on a simple structure that can be used by other languages (such as XSLT).

- Comments are preserved on roundtrips, but are not carried over into the resulting *PO Files*.
- XML Fragments and non-wellformed XML.

References

- XML specification

iCalendar

Support for iCalendar (*.ics) files. This allows calendars to be localised.

The format extracts certain properties from VEVENT objects. The properties are limited to textual entries that would need to be localised, it does not include entries such as dates and durations that would indeed change for various locales.

Resources

- rfc2445 – Internet Calendaring and Scheduling Core Object Specification (iCalendar)
- iCal spec in a simple adaptation of the rfc that makes it easy to refer to all sections, items and attributes.
- VObject – the python library used to read the iCal file.
- iCalender validator
- iCalendar
- Components and their properties

Conformance

We are not creating iCal files, simply extracting localisable information and rebuilding the file. We rely on VObject to ensure correctness.

The following data is extracted:

- VEVENT:
  - SUMMARY
  - DESCRIPTION
  - LOCATION
  - COMMENTS

No other sections are extracted.

Notes

LANGUAGE: not a multilingual solution

It is possible to set the language attribute on an entry e.g.:

```
SUMMARY:LANGUAGE=af;New Year's Day
```

However since only one SUMMARY entry is allowed this does not allow you to specify multiple entries which would allow a single multilingual file. With that in mind it is not clear why the LANGUAGE attribute is allowed, the examples they give are for LOCATION entries but that is still not clearly useful.

Development Notes

If we use LANGUAGE at all it will be to ensure that we specify that an entry is in a given language.

JSON

New in version 1.9.0.

JSON is a common format for web data interchange.

Example:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

Following JSON dialects are supported

- Plain JSON files.
- i18next
- Web Extension i18n
- go-i18n
- ARB

YAML

New in version 2.0.0.

YAML is a common format for web data interchange. The two variants of YAML files are supported:

- Plain YAML files.
- Ruby YAML localization files with root node as language. This variant supports plurals as well.

Non-Conformance

The following are not yet supported (in most cases these are properly parsed, but not saved in round trip):

- Booleans:

```
foo: True
```

OpenDocument Format

This page summarises the support for the OpenDocument format (ODF) in the Translate Toolkit. This currently involves only the *odf2xliff* and xliff2odf converters.

The Translate Toolkit aims to support version 1.1 of the ODF standard, although it should work reasonably well with older or newer files to the extent that they are similar.

Our support is implemented to classify tags as not containing translatable text, or as being inline tags inside translatable tags. This approach means that new fields added in future versions will automatically be seen as translatable and should still be extracted successfully, even if the currently released versions of the Translate Toolkit are not aware of their existence.

- Currently used and classified tags

More complex tag uses are still needed to extract 100% correctly in some complex cases. For more information, see the list of issues from testing.

Simple Text Documents

The Translate Toolkit can process simple Text files. This is very useful for translating installation files and READMEs. The processing of these files is performed by the *txt2po* converter.

In some cases you will need to adjust the source text for the conversion management to work well. This is because the text file format support considered units to be space separated blocks of text.

Example

```
Heading
=======

Paragraph One

Paragraph Two:
* First bullet
* Second bullet
```

This example will result in three units. The first will include the underline in the header. The third will include all the bullet points in one paragraph together with the paragraph lead in.

Wiki Syntax

The Translate Toolkit can manage Wiki syntax pages. This is implemented as part of the *text* format and the conversion is supported in *txt2po*.

Those who edit wikis will appreciate that wiki text is simply a normal text document edited using a form of wiki syntax. Whether the final storage is a database or a flat file the part that a user edits is a simple text file.

The format does not support all features of the wiki syntax and will simply dump the full form if it doesn't understand the text. But structures such as headers and lists are understood and the filter can remove these are correctly add them.

Supported Wiki Formats

The following is a list of the wikis supported together with a list of the items that we can process:

- dokuwiki – heading, bullet, numbered list
- MediaWiki – heading, bullet, numbered list

Possible uses

As part of a localisation process for a wiki this format and the filters could be used to provide a good localisation of existing wiki content.

With further enhancement the tool could probably be capable of converting from one wiki syntax to another, but that is of course not its main aim

Additional notes on MediaWiki

Media wiki can also export in XML format, see http://en.wikipedia.org/wiki/Special:Export and http://www.mediawiki.org/wiki/Manual:Parameters_to_Special:Export this however exports in XML so not directly usable by txt2po.

For importing please see http://en.wikipedia.org/wiki/Help:Import this is disabled on most wikis so not directly usable currently.

Subtitles

New in version 1.4.

The translation of subtitles are supported in the toolkit with the commands *sub2po* and po2sub.

The following formats are supported for subtitles:

- MicroDVD
- MPL2
- MPsub
- SubRip (.srt)
- SubViewer 2.0 (.sub)
- TMPlayer
- Sub Station Alpha

- Advanced Sub Station Alpha

YouTube supports a number of formats

Implementation details

Format support is provided by Gaupol a subtitling tool. Further enhancement of format support in Gaupol will directly benefit our conversion ability.

Usage

It must be noted that our tools provide the ability to localise the subtitles. This in itself is useful and allows a translator to use their existing localisation tools. But this is pure localisation and users should be aware that they might still need to post edit their work to account for timing, limited text space, limits in the ability of viewers to keep up with the text.

For most cases simply localising will be good enough. But in some cases the translated work might need to be reviewed to fix any such issues. You can use Gaupol to perform those reviews.

- *HTML*
- *Flat XML* (single-level XML)
- *iCalendar*
- *JSON*
- *YAML*
- OpenDocument – all ODF file types
- *Text* – plain text with blocks separated by whitespace
- *Wiki* – DokuWiki and MediaWiki supported
- *Subtitles* – various formats (v1.4)

1.7.6 Machine readable formats

Gettext .mo

The Gettext .mo (Machine Object) file is a compiled *Gettext PO* file. In execution Gettext enabled programs retrieve translations from the .mo file. The file contains arrays for the English and the translations, an optional hash can speed up the access to the translations.

Conformance

The toolkit can create .mo files from PO or XLIFF files, handling plurals and msgctxt. It can also read .mo files, allowing counting, etc and also allowing the .mo files to act as a translation memory.

Changed in version 1.2: The hash table is also generated (the Gettext .mo files works fine without it). Due to slight differences in the construction of the hashing, the generated files are not identical to those generated by msgfmt, but they should be functionally equivalent and 100% usable. Issue 326 tracked the implementation of the hashing. The hash is platform dependent.

Qt .qm

A .qm file is a compiled *Qt linguist* file. In many ways it is similar to Gettext, in that it uses a hashing table to lookup the translated text. In older version they store only the hash and the translation which doesn't make the format useful for recovering translated text.

Conformance

The toolkit can read .qm files correctly. There are some unimplemented aspects of the message block, but these seem to be legacy features and require examples to be able to implement the feature.

The .qm implementation cannot write a .qm file, thus you are only able to use this format in a read-only context: counting messages (*pocount*), reading in messages for a TM or using it as a source format for a converter e.g. a possible qm2xliff converter.

TODO

- Writing
  - Hash algorithm
- Gettext *Gettext .mo*
- Qt *Qt .qm* (read-only)

1.7.7 In development

1.7.8 Unsupported formats

Formats that we would like to support but don't currently support:

Wireless Markup Language

This page documents the support for WML and is used for planning our work on it.

This is implemented as a generic XML document type that is handled similarly to the way the developers/projects/odf project handles ODF documents.

- Wordfast:
  - Glossary tab-delimited "source,target,comment" i.e. like OmegaT but unsure if any extension is required.
- Apple:
  - AppleGlot
  - .plist – see issue 633 and plistlib for Python
- Adobe:
  - FrameMaker's Maker Interchange Format – MIF (See also python-gendoc, and Perl MIF module)
  - FrameMaker's Maker Markup Language (MML)
- Microsoft
  - Word, Excel, etc (probably through usage of OpenOffice.org)

  - OOXML (at least at the text level we don't have to deal with much of the mess inside OOXML). See also: Open XML SDK v1
  - Rich Text Format (RTF) see also pyrtf-ng
  - Open XML Paper Specification
- XML related
  - Generic XML
  - DocBook (can be handled by KDE's xml2pot)
  - SVG
- DITA
- PDF see spec, PDFedit
- LaTeX – see plasTeX, a Python framework for processing LaTeX documents
- unoconv – Python bindings to OpenOffice.org UNO which could allow manipulation of all formats understood by OpenOffice.org.
- Trados:
  - TTX (Reverse Engineered DTD, other discussion)
  - Multiterm XML TSV to MiltiTerm conversion script or XLST
  - .tmw
  - .txt (You can interchange using TMX) Format explanation with some examples.
- Tcl: .msg files. Good documentation
- Installers:
  - NSIS installer: Existing C++ implementation
  - WiX – MSI (Microsoft Installer) creator. Localization instructions, more notes on localisation. This is a custom XML format, another one!
- catgets/gencat: precedes gettext, looking in man packages is the best information I could find. Also LSB requires it. There is some info about the source (msgfile) format on GNU website
- *Wireless Markup Language*
- GlossML
- Deja Vu External View: Instructions sent to a translator, Description of external view options and process

1.7.9 Unlikely to be supported

These formats are either: too difficult to implement, undocumented, can be processed using some intermediate format or used by too few people to justify the effort. Or some combination or these issues.

Standards conformance

This page links to pages documenting standard conformance for different standards or file *formats*.

LISA and OASIS standards

- *TMX*
- *XLIFF*
- *TBX*

Other formats

- *Gettext PO*
- *Gettext .mo*
- *CSV*
- *Qt Linguist*
- Qt .qph and .qm files
- *Wordfast translation memory*
- OmegaT glossary

Searching and matching

- *Levenshtein distance*

Base classes

NOTE: This page is mostly useful for *developers* as it describes some programming detail of the *toolkit*.

For the implementation of the different storage classes that the toolkit supports, we want to define a set of base classes to form a common API for all formats. This will simplify implementation of new storage formats, and enable easy integration into external tools, such as Pootle. It will also mean less duplication of code in similar storage formats.

These ideas explained here should be seen as drafts only.

Requirements

The base classes should be rich enough in functionality to enable users of the base classes to have access to all or most of the features that are available in the formats. In particular, the following are considered requirements:

- Seamless and hidden handling of escaping, quoting and character sets
- Parsing a file when given a file name or file contents (whole file in a string)
- Writing a file to disk
- Getting and setting source and target languages
- Accessing units, and determining if they are translatable, translated, a unique identifier for the unit in the file, etc.
- Support for plural units that can vary between different languages (as the PO format allows with msgid_plural, etc.)

Other possibilities:

- Support for variable number of languages in the format. Examples: .txt and .properties support one language, PO supports two, *TMX* supports many.
- Support for "multifiles", in other words a file that contain other entities that corresponds to files in other formats. Examples: ZIP and *XLIFF*. In reality this is only used by some of the converters. This isn't present in the base class yet.

All these do not mean that all formats must support al these features, but in the formats that do support these features, it must be accessible through the base class, and it must be possible to interrogate the storage format through the base class to know which features it supports.

The classes

A file contains a number of translation units, and possibly a header. Each translation unit contains one or more strings corresponding to each of the languages represented in that unit.

Message/string (multistring)

This class represents a single conceptual string in a single language. It must know its own requirements for escaping and implement it internally. Escaped versions are only used for internal representation and only exposed for file creation and testing (unit tests, for example).

Note that when storing different plural forms of the same string, they should be stored in this class. The main object is the singular string, and all of the string forms can be accessed in a list at x.strings. Most of the time the object can be dealt with as a single string, only when it is necessary to deal with plural forms do the extra strings have to be taken into account.

Any string from a plural unit must be a multistring.

Translation unit

This class represents a unit of one or several related messages/strings. In most formats the contained strings will be translations of some original message/string. It must associate a language value with each message/string. It must know how to join all contained messages/strings to compile a valid representation. For formats that support at least two languages, the first two languages will serve as "source" and "target" languages for the common case of translating from one language into another language.

Some future ideas:

As the number of languages can be seen as one "dimension" of the translation unit, plurality can be seen as a second dimension. A format can thus be classified according to the dimensionality that it supports, as follows:

- .properties files supports one language and no concept of plurals. This include most document types, such as .txt, HTML and OpenDocument formats.
- Old style PO files supported two languages and no plurals.
- New style PO files support two languages and any number of plurals as required by the target language. The plural forms are stored in the original or target strings, as extra forms of the string (See message/string class above).
- TMX files support any number of languages, but has no concept of plurality.

Comments/notes are supported in this class. Quality or status information (fuzzy, last-changed-by) should be stored. TODO: see if this should be on unit level or language level.

Store

This class represents a whole collection of translation units, usually stored in a single file. It supports the concept of a header, and possibly comments at file level. A file will not necessarily be contained alone in single file on disc. See "multifile" below.

Multifile

This abstraction is only used by a few converters.

This class represents a storage format that contains other files or file like objects. Examples include ZIP, XLIFF, and OpenOffice SDF files. It must give access to the contained files, and possibly give access to the translation units contained in those files, as if they are contained natively.

Additional Notes

Dwayne and I (Andreas) discussed cleaning up the storage base class. A lot of what we discussed is related to the above. A quick summary:

- Implement a new base class.
  - Flesh out the API, clean and clear definitions.
  - Document the API.
- We need to discuss the class hierarchy, e.g.:

```
base
     -- po
     -- text
     -- xml
          -- xhtml
          -- lisa
                -- xliff
                -- tmx
                -- tbx
```

- Clean up converters.
  - Parsing of file content needs to happen only in the storage implementation of each filetype/storage type. Currently parsing happens all over the place.
  - Currently there are separate conversion programs for each type and direction to convert to, e.g. po2xliff and xliff2po (24 commands with lots of duplicate code in them). Ideally conversion should be as simple as:

```
>>> po_store = POStore(filecontent)
>>> print(bytes(po_store))
msgid "bleep"
msgstr "blorp"

>>> xliff_store = XliffStore(po_store)
>>> print(bytes(xliff_store))
<xliff>
  <file>
    <trans-unit>
      <source>bleep</source>
```

(continues on next page)

(continued from previous page)

```
      <target>blorp</target>
    </trans-unit>
  </file>
</xliff>
```

Note that the xliffstore is being instantiated using the postore object. This works because all the data in any translation store object is accessible via the same well-defined base API. A concept class implementing the above code snippet has already been written.

- Move certain options into their respective storage classes.
  - e.g. the `--duplicates` option can move into po.py
- Store the meta data for a storage object.
  - Can be implemented as separate sqlite file that accompanies the real file.
  - Features not directly supported by a file format can be stored in the metadata file.
- A storage object should know all information pertaining to itself.
  - e.g. "am I monolingual?"
- We should discuss how to make an object aware that it is monolingual, bilingual or multilingual.
  - Maybe through mixin-classes?
  - How will the behaviour of a monolingual store differ from a bilingual store?

Quoting and Escaping

Different translation *formats* handle quoting and escaping strings differently. This is meant to be a common page which outlines the differences

PO format

Strings are quoted using double quotes. For long strings multiline quotes are done by opening and closing the quotes on each line. Usually in this case the first line is left blank. The splitting of strings over lines is transparent i.e. it does not imply line breaks in the translated strings.

Escaping is done with a backslash. An escaped double quote (`\"`) corresponds to a double quote in the original string. `\n` for newline, `\t` for tabs etc are used. Backslashes can be escaped to give a native backslash.

See also escaping in the translation guide.

Example:

```
msgid ""
"This is a long string with a \n newline, a \" double quote, and a \\ backslash."
"There is no space between the . at the end of the last sentence "
"and the T at the beginning of this one."
```

DTD format

Strings are quoted using either double or single quotes. The quoting character may not occur within the string. There is no provision for escaping. XML entities can be used e.g. `&apos;` can be used to denote a single quote within the single-quoted string.

Some DTD files seem to have backslash-escapes, but these are anomalies: see discussion thread on Mozilla l10n-dev

Mozilla properties format

Note that this section does not describe the Java properties files, even though they are quite similar.

It seems that the literal string `\n` (a backslash followed by the character 'n') and `\t` and `\r` cannot be encoded in properties files. This is the assumption of the toolkit.

CHAPTER 2

Developer's Guide

If you are a developer interested in using the Translate Toolkit for building new tools, make sure to read through this part.

2.1 Translate Styleguide

The Translate styleguide is the styleguide for all Translate projects, including Translate Toolkit, Pootle, Virtaal and others. Patches are required to follow these guidelines.

This Styleguide follows **PEP 8** with some clarifications. It is based almost verbatim on the Flask Styleguide.

2.1.1 Python

These are the Translate conventions for Python coding style.

General

Indentation

4 real spaces, no tabs. Exceptions: modules that have been copied into the source that don't follow this guideline.

Maximum line length

79 characters with a soft limit for 84 if absolutely necessary. Try to avoid too nested code by cleverly placing *break*, *continue* and *return* statements.

Continuing long statements

To continue a statement you can use backslashes (preceded by a space) in which case you should align the next line with the last dot or equal sign, or indent four spaces:

```
MyModel.query.filter(MyModel.scalar > 120) \
             .order_by(MyModel.name.desc()) \
             .limit(10)

my_long_assignment = MyModel.query.filter(MyModel.scalar > 120) \
                     .order_by(MyModel.name.desc()) \
                     .limit(10)

this_is_a_very_long(function_call, 'with many parameters') \
    .that_returns_an_object_with_an_attribute
```

If you break in a statement with parentheses or braces, align to the braces:

```
this_is_a_very_long(function_call, 'with many parameters',
                    23, 42, 'and even more')
```

If you need to break long strings, on function calls or when assigning to variables, try to use implicit string continuation:

```
this_holds_a_very_long_string("Very long string with a lot of characters "
                              "and words on it, so many that it is "
                              "necessary to break it in several lines to "
                              "improve readability.")
long_string_var = ("Very long string with a lot of characters and words on "
                   "it, so many that it is necessary to break it in "
                   "several lines to improve readability.")
```

For lists or tuples with many items, break immediately after the opening brace:

```
items = [
    'this is the first', 'set of items', 'with more items',
    'to come in this line', 'like this'
]
```

Blank lines

Top level functions and classes are separated by two lines, everything else by one. Do not use too many blank lines to separate logical segments in code. Example:

```
def hello(name):
    print('Hello %s!' % name)


def goodbye(name):
    print('See you %s.' % name)


class MyClass:
    """This is a simple docstring"""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, name):
        self.name = name

    @property
    def annoying_name(self):
        return self.name.upper() + '!!!!111'
```

Strings

- Double quotes are suggested over single quotes, but always try to respect the surrounding coding style. This is overruled by escaping which you should always try to avoid.

```
# Good.
str1 = "Sauron's eye"
str2 = 'Its name is "Virtaal".'


# Bad.
str3 = 'Sauron\'s eye'
str4 = "Its name is \"Virtaal\"."
```

String formatting

While str.format() is more powerful than %-formatting, the latter has been the canonical way of formatting strings in Python for a long time and the Python core team has shown no desire to settle on one syntax over the other. For simple, serial positional cases (non-translatable strings), the old "%s" way of formatting is preferred. For anything more complex, including translatable strings, str.format is preferred as it is significantly more powerful and often cleaner.

```
# Good
print("Hello, {thing}".format(thing="world"))
print("Hello, {}".format("world"))
print("%s=%r" % ("hello", "world"))  # non-translatable strings

# Bad
print("%s, %s" % ("Hello", "world"))  # Translatable string.
print("Hello, %(thing)s" % {"thing": "world"})  # Use {thing}.
```

Imports

Like in **PEP 8**, but:

- Imports should be grouped in the following order:
  1) __future__ library imports
  2) Python standard library imports
  3) Third party libraries imports
  4) Translate Toolkit imports
  5) Current package imports, using explicit relative imports (See PEP 328)

- A blank line must be present between each group of imports (like in PEP8).
- Imports on each group must be arranged alphabetically by module name:
  - Shortest module names must be before longer ones: `from django.db import ...` before `from django.db.models import ...`.
- `import ...` calls must precede `from ... import` ones on each group:
  - On each of these subgroups the entries should be alphabetically arranged.
  - No blank lines between subgroups.
- On `from ... import`
  - Use a `CONSTANT`, `Class`, `function` order, where the constants, classes and functions are in alphabetical order inside of its respective groups.
  - If the import line exceeds the 80 chars, then split it using parentheses to continue the import on the next line (aligning the imported items with the opening parenthesis).

```
from __future__ import absolute_import

import re
import sys.path as sys_path
import time
from datetime import timedelta
from os import path

from lxml.html import fromstring

from translate.filters import checks
from translate.storage.aresource import (EOF, WHITESPACE, AndroidFile,
                                         AndroidUnit, android_decode,
                                         android_encode)

from . import php2po
```

Properties

- Never use `lambda` functions:

```
# Good.
@property
def stores(self):
  return self.child.stores


# Bad.
stores = property(lambda self: self.child.stores)
```

- Try to use `@property` instead of `get_*` or `is_*` methods that don't require passing any parameter:

```
# Good.
@property
def terminology(self):
  ...

@property
```

(continues on next page)

(continued from previous page)

```
def is_monolingual(self):
  ...


# Also good.
def get_stores_for_language(self, language):
  ...


# Bad.
def get_terminology(self):
  ...

def is_monolingual(self):
  ...
```

- Always use `@property` instead of `property(...)`, even for properties that also have a setter or a deleter:

```
# Good.
@property
def units(self):
  ...


# Also good.
@property
def x(self):
  """I'm the 'x' property."""
  return self._x

@x.setter
def x(self, value):  # Note: Method must be named 'x' too.
  self._x = value

@x.deleter
def x(self):  # Note: Method must be named 'x' too.
  del self._x


# Bad.
def _get_units(self):
  ...
units = property(_get_units)


# Also bad.
def getx(self):
  return self._x
def setx(self, value):
  self._x = value
def delx(self):
  del self._x
x = property(getx, setx, delx, "I'm the 'x' property.")
```

Expressions and Statements

General whitespace rules

- No whitespace for unary operators that are not words (e.g.: -, ~ etc.) as well on the inner side of parentheses.
- Whitespace is placed between binary operators.

```
# Good.
exp = -1.05
value = (item_value / item_count) * offset / exp
value = my_list[index]
value = my_dict['key']


# Bad.
exp = - 1.05
value = ( item_value / item_count ) * offset / exp
value = (item_value/item_count)*offset/exp
value=( item_value/item_count ) * offset/exp
value = my_list[ index ]
value = my_dict ['key']
```

Slice notation

While **PEP 8** calls for spaces around operators `a = b + c` this results in flags when you use `a[b+1:c-1]` but would allow the rather unreadable `a[b + 1:c - 1]` to pass. **PEP 8** is rather quiet on slice notation.

- Don't use spaces with simple variables or numbers
- Use brackets for expressions with spaces between binary operators

```
# Good.
a[1:2]
a[start:end]
a[(start - 1):(end + var + 2)]  # Brackets help group things and don't hide the␣
↪slice
a[-1:(end + 1)]


# Bad.
a[start: end]  # No spaces around :
a[start-1:end+var+2]  # Insanely hard to read, especially when your expressions␣
↪are more complex
a[start - 1:end + 2]  # You lose sight of the fact that it is a slice
a[- 1:end]  # -1 is unary, no space
```

Note: String slice formatting is still under discussion.

Comparisons

- Against arbitrary types: `==` and `!=`
- Against singletons with `is` and `is not` (e.g.: `foo is not None`)
- Never compare something with *True* or *False* (for example never do `foo == False`, do `not foo` instead)

Negated containment checks

- Use `foo not in bar` instead of `not foo in bar`

Instance checks

- `isinstance(a, C)` instead of `type(A) is C`, but try to avoid instance checks in general. Check for features.

If statements

- Use `()` brackets around complex if statements to allow easy wrapping, don't use backslash to wrap an if statement.
- Wrap between `and`, `or`, etc.
- Keep `not` with the expression
- Use `()` alignment between expressions
- Use extra `()` to eliminate ambiguity, don't rely on an understanding of Python operator precedence rules.

```
# Good.
if length >= (upper + 2):
    ...

if (length >= 25 and
    string != "Something" and
    not careful):
    do_something()


# Bad.
if length >= upper + 2:
    ...

if (length...
    and string !=...
```

Naming Conventions

Note: This has not been implemented or discussed. The Translate code is not at all consistent with these conventions.

- Class names: `CamelCase`, with acronyms kept uppercase (`HTTPWriter` and not `HttpWriter`)
- Variable names: `lowercase_with_underscores`
- Method and function names: `lowercase_with_underscores`
- Constants: `UPPERCASE_WITH_UNDERSCORES`
- precompiled regular expressions: `name_re`

Protected members are prefixed with a single underscore. Double underscores are reserved for mixin classes.

To prevent name clashes with keywords, one trailing underscore may be appended. Clashes with builtins are allowed and **must not** be resolved by appending an underline to the name. If your code needs to access a shadowed builtin, rebind the builtin to a different name instead. Consider using a different name to avoid having to deal with either type of name clash, but don't complicate names with prefixes or suffixes.

Function and method arguments

- Class methods: `cls` as first parameter
- Instance methods: `self` as first parameter

2.2 Documentation

We use Sphinx to generate our API and user documentation. Read the reStructuredText primer and Sphinx documentation as needed.

2.2.1 Special roles

We introduce a number of special roles for documentation:

- `:issue:` – links to a toolkit issue Github.
  - `:issue:`234`` gives: issue 234
  - `:issue:`broken <234>`` gives: broken
- `:opt:` – mark command options and command values.
  - `:opt:`-P`` gives `-P`
  - `:opt:`--progress=dots`` gives `--progress=dots`
  - `:opt:`dots`` gives `dots`
- `:man:` – link to a Linux man page.
  - `:man:`msgfmt`` gives msgfmt

2.2.2 Code and command line highlighting

All code examples and format snippets should be highlighted to make them easier to read. By default Sphinx uses Python highlighting of code snippets (but it doesn't always work). You will want to change that in these situations:

- The examples are not Python e.g. talking about INI file parsing. In which case set the file level highlighting using:

```
.. highlight:: ini
```

- There are multiple different code examples in the document, then use:

```
.. code-block:: ruby
```

  before each code block.

- Python code highlighting isn't working, then force Python highlighting using:

```
.. code-block:: python
```

Note: Generally we prefer explicit markup as this makes it easier for those following you to know what you intended. So use `.. code-block:: python` even though in some cases this is not required.

With *command line examples*, to improve readability use:

```
.. code-block:: console
```

Add `$` command prompt markers and `#` comments as required, as shown in this example:

```
$ cd docs
$ make html  # Build all Sphinx documentation
$ make linkcheck  # Report broken links
```

2.2.3 User documentation

This is documentation found in `docs/` and that is published on Read the Docs. The target is the end user so our primary objective is to make accessible, readable and beautiful documents for them.

2.2.4 Docstrings

Docstring conventions: All docstrings are formatted with reStructuredText as understood by Sphinx. Depending on the number of lines in the docstring, they are laid out differently. If it's just one line, the closing triple quote is on the same line as the opening, otherwise the text is on the same line as the opening quote and the triple quote that closes the string on its own line:

```
def foo():
    """This is a simple docstring."""


def bar():
    """This is a longer docstring with so much information in there
    that it spans three lines.  In this case the closing triple quote
    is on its own line.
    """
```

Please read **PEP 257** (Docstring Conventions) for a general overview, the important parts though are:

- A docstring should have a brief one-line summary, ending with a period. Use `Do this`, `Return that` rather than `Does ...`, `Returns ...`.
- If there are more details there should be a blank line between the one-line summary and the rest of the text. Use paragraphs and formatting as needed.
- Use reST field lists to describe the input parameters and/or return types as the last part of the docstring.
- Use proper capitalisation and punctuation.
- Don't restate things that would appear in parameter descriptions.

```
def addunit(self, unit):
    """Append the given unit to the object's list of units.
```

(continues on next page)

(continued from previous page)

```
    This method should always be used rather than trying to modify the
    list manually.

    :param Unit unit: Any object that inherits from :class:`Unit`.
    """
    self.units.append(unit)
```

Parameter documentation: Document parameters using reST field lists as follows:

```
def foo(bar):
    """Simple docstring.

    :param SomeType bar: Something
    :return: Returns something
    :rtype: Return type
    """
```

Cross referencing code: When talking about other objects, methods, functions and variables it is good practice to cross-reference them with Sphinx's Python cross-referencing.

Other directives: Use paragraph-level markup when needed.

Note: We still need to gather the useful ones that we want you to use and how to use them. E.g. how to talk about a parameter in the docstring. How to reference classes in the module. How to reference other modules, etc.

Module header: The module header consists of a utf-8 encoding declaration, copyright attribution, license block and a standard docstring:

```
#
... LICENSE BLOCK...

"""A brief description"""
```

Deprecation: Document the deprecation and version when deprecating features:

```
from translate.misc.deprecation import deprecated


@deprecated("Use util.run_fast() instead.")
def run_slow():
    """Run fast

    .. deprecated:: 1.5
       Use :func:`run_fast` instead.
    """
    run_fast()
```

2.2.5 Comments

General:

- The # symbol (pound or hash) is used to start comments.
- A space must follow the # between any written text.

- Line length must be observed.
- Inline comments are preceded by two spaces.
- Write sentences correctly: proper capitalisation and punctuation.

```
# Good comment with space before and full sentence.
statement  # Good comment with two spaces


#Bad comment no space before
statement # Bad comment, needs two spaces
```

Docstring comments: Rules for comments are similar to docstrings. Both are formatted with reStructuredText. If a comment is used to document an attribute, put a colon after the opening pound sign (#):

```
class User:
    #: the name of the user as unicode string
    name = Column(String)
    #: the sha1 hash of the password + inline salt
    pw_hash = Column(String)
```

2.3 Building

2.3.1 UNIX

2.3.2 Windows

Requirements

- Innosetup
- py2exe

Consult the README in the source distribution for the build dependencies.

Building Python packages with C extensions under Windows

In order to build modules which have C extensions, you will need either the Visual Studio C++ compiler or MinGW.

Make sure that your Visual Studio C++ or MinGW program path is part of your system's program path, since the Python build system requires this.

To build and install a package with MinGW, you need to execute:

```
python setup.py build -c mingw32 install
```

from the command line.

To build a Windows installer when using MinGW, execute:

```
python setup.py build -c mingw32 bdist_wininst
```

Building

Simply execute:

```
python setup.py innosetup
```

The generated file can be found under `translate-toolkit-<version>\Output` (where `<version>` is the software version).

2.4 Testing

Our aim is that all new functionality is adequately tested. Adding tests for existing functionality is highly recommended before any major reimplementation (refactoring, etcetera).

We use py.test for (unit) testing. You need at least pytest >= 2.2.

To run tests in the current directory and its subdirectories:

```
$ py.test  # runs all tests
$ py.test storage/test_dtd.py  # runs just a single test module
```

We use several py.test features to simplify testing, and to suppress errors in circumstances where the tests cannot possibly succeed (limitations of tests and missing dependencies).

2.4.1 Skipping tests

Pytest allows tests, test classes, and modules to be skipped or marked as "expected to fail" (xfail). Generally you should *skip* only if the test cannot run at all (throws uncaught exception); otherwise *xfail* is preferred as it provides more test coverage.

importorskip

Use the builtin `_pytest.runner.importorskip` function to skip a test module if a dependency cannot be imported:

```
from pytest import importorskip
importorskip("vobject")
```

If *vobject* can be imported, it will be; otherwise it raises an exception that causes pytest to skip the entire module rather than failing.

skipif

Use the `skipif` decorator to mark tests to be skipped unless certain criteria are met. The following skips a test if the version of *mymodule* is too old:

```
import mymodule

@pytest.mark.skipif("mymodule.__version__ < '1.2'")
def test_function():
    ...
```

You can apply this decorator to classes as well as functions and methods.

It is also possible to skip an entire test module by creating a `pytestmark` static variable in the module:

```
# mark entire module as skipped for py.test if no indexer available
pytestmark = pytest.mark.skipif("noindexer")
```

xfail

Use the `xfail` decorator to mark tests as expected to fail. This allows you to do the following:

- Build tests for functionality that we haven't implemented yet
- Mark tests that will fail on certain platforms or Python versions
- Mark tests that we should fix but haven't got round to fixing yet

The simplest form is the following:

```
from pytest import pytest.mark

@mark.xfail
def test_function():
    ...
```

You can also pass parameters to the decorator to mark expected failure only under some condition (like *skipif*), to document the reason failure is expected, or to actually skip the test:

```
@mark.xfail("sys.version_info >= (3,0)")  # only expect failure for Python 3
@mark.xfail(..., reason="Not implemented")  # provide a reason for the xfail
@mark.xfail(..., run=False)  # skip the test but still regard it as xfailed
```

2.4.2 Testing for Warnings

deprecated_call

The builtin `deprecated_call()` function checks that a function that we run raises a DeprecationWarning:

```
from pytest import deprecated_call

def test_something():
    deprecated_call(function_to_run, arguments_for_function)
```

recwarn

The *recwarn plugin* allows us to test for other warnings. Note that `recwarn` is a funcargs plugin, which means that you need it in your test function parameters:

```
def test_example(recwarn):
    # do something
    w = recwarn.pop()
    # w.{message,category,filename,lineno}
    assert 'something' in str(w.message)
```

2.5 Command Line Functional Testing

Functional tests allow us to validate the operation of the tools on the command line. The execution by a user is simulated using reference data files and the results are captured for comparison.

The tests are simple to craft and use some naming magic to make it easy to refer to test files, stdout and stderr.

2.5.1 File name magic

We use a special naming convention to make writing tests quick and easy. Thus in the case of testing the following command:

```
$ moz2po -t template.dtd translations.po translated.dtd
```

Our test would be written like this:

```
$ moz2po -t $one $two $out
```

Where `$one` and `$two` are the input files and `$out` is the result file that the test framework will validate.

The files would be called:

File	Function	Variable	File naming conventions
test_moz2po_help.sh	Test script	•	test_${command}_${description}.sh
test_moz2po_help/one.dtd	Input	$one	${testname}/${variable}.${extension}
test_moz2po_help/two.po	Input	$two	${testname}/${variable}.${extension}
test_moz2po_help/out.dtd	Output	$out	${testname}/${variable}.${extension}
test_moz2po_help/stdout.txt	Output	$stdout	${testname}/${variable}.${extension}
test_moz2po_help/stderr.txt	Output	$stderr	${testname}/${variable}.${extension}

Note: A test filename must start with `test_` and end in `.sh`. The rest of the name may only use ASCII alphanumeric characters and underscore _.

The test file is placed in the `tests/` directory while data files are placed in the `tests/data/${testname}` directory.

There are three standard output files:

1. `$out` - the output from the command
2. `$stdout` - any output given to the user
3. `$stderr` - any error output

The output files are available for checking at the end of the test execution and a test will fail if there are differences between the reference output and that achieved in the test run.

You do not need to define reference output for all three, if one is missing then checks will be against `/dev/null`.

There can be any number of input files. They need to be named using only ASCII characters without any punctuation. While you can give them any name we recommend using numbered positions such as one, two, three. These are converted into variables in the test framework so ensure that none of your choices clash with existing bash commands and variables.

Your test script can access variables for all of your files so e.g. `moz2po_conversion/one.dtd` will be referenced as `$one` and output `moz2po_conversion/out.dtd` as `$out`.

2.5.2 Writing

The tests are normal bash scripts so they can be executed on their own. A template for a test is as follows:

```
#!/bin/bash

# Import the test framework
source $(basename $0)/test.inc.sh

# You can put any extra preperation here

# Your actual command line to test No need for redirecting to /dev/stdout as
# the test framework will do that automatically
myprogram $one $two -o $out

# Check that the results of the test match your reference resulst
check_results  # does start_check and diff_all

# OR do the following
# start_checks - begin checking
# has_stdout|has_stderr|has $file - checks that the file exists we don't care for␣
↪content
# startswith $file|startswith_stderr|startswith_stdout - the output starts with some␣
↪expression
# startswithi $file|startswithi_stderr|startswithi_stdout - case insensitive␣
↪startswith
# end_checks
```

For simple tests, where we diff output and do the correct checking of output files, simply use `check_results`. More complex tests need to wrap tests in `start_checks` and `end_checks`.

```
start_checks
has $out
containsi_stdout "Parsed:"
end_checks
```

You can make use of the following commands in the `start_checks` scenario:

Command	Description
has $file	$file was output and it not empty
has_stdout	stdout is not empty
has_stderr	stderr is not empty
startswith $file “String”	$file starts with “String”
startswithi $file “String”	$file starts with “String” ignoring case
startswith_stdout “String”	stdout starts with “String”
startswithi_stdout “String”	stdout starts with “String” ignoring case
startswith_stderr “String”	stderr starts with “String”
startswithi_stderr “String”	stderr starts with “String” ignoring case
contains $file “String”	$file contains “String”
containsi $file “String”	$file contains “String” ignoring case
contains_stdout “String”	stdout contains “String”
containsi_stdout “String”	stdout contains “String” ignoring case
contains_stderr “String”	stderr contains “String”
containsi_stderr “String”	stderr contains “String” ignoring case
endswith $file “String”	$file ends with “String”
endswithi $file “String”	$file ends with “String” ignoring case
endswith_stdout “String”	stdout ends with “String”
endswithi_stdout “String”	stdout ends with “String” ignoring case
endswith_stderr “String”	stderr ends with “String”
endswithi_stderr “String”	stderr ends with “String” ignoring case

–prep

If you use the –prep options on any test then the test will change behavior. It won’t validate the results against your reference data but will instead create your reference data. This makes it easy to generate your expected result files when you are setting up your test.

2.6 Contributing

We could use your help. If you are interesting in contributing then please join us on our Gitter development channel.

Here are some idea of how you can contribute

- *Test* – help us test new candidate releases before they are released
- *Debug* – check bug reports, create tests to highlight problems
- *Develop* – add your Python developer skills to the mix
- *Document* – help make our docs readable, useful and complete

Below we give you more detail on these:

2.6.1 Testing

Before we release new versions of the Toolkit we need people to check that they still work correctly. If you are a frequent user you might want to start using the release candidate on your current work and report any errors before we release them.

Compile and install the software to see if we have any platform issues:

```
./setup.py install
```

Check for any files that are missing, tools that were not installed, etc.

Run unit tests to see if there are any issues. Please report any failures.

Finally, simply work with the software. Checking all your current usage patterns and report problems.

2.6.2 Debugging

- Make sure your familiar with the bug reporting guidelines.
- Create a login for yourself at https://github.com
- Then choose an issue

Now you need to try and validate the bug. Your aim is to confirm that the bug is either fixed, is invalid or still exists.

If its fixed please close the bug and give details of how when it was fixed or what version you used to validate it as corrected.

If you find that the bug reporter has made the incorrect assumptions or their suggestion cannot work. Then mark the bug as invalid and give reasons why.

The last case, an existing bug is the most interesting. Check through the bug and do the following:

- Fix up the summary to make it clear what the bug is
- Create new bugs for separate issues
- Set severity level and classifications correctly
- Add examples to reproduce the bug, or make the supplied files simpler
- If you can identify the bug but not fix it then explain what needs fixing
- Move on to the next bug

2.6.3 Developing

Don't ignore this area if you feel like you are not a hotshot coder!

You will need some Python skills, this is a great way to learn.

Here are some ideas to get you going:

- Write a test to expose some bug
- Try to fix the actual code to fix your bug
- Add a small piece of functionality that helps you
- Document the methods in especially the base class and derived classes
- Add a *format* type and converters
- Add more features to help our formats *conform to the standards*

You will definitely need to be on the Development channel

Now is the time to familiarise yourself with the *developers guide*.

2.6.4 Documenting

This is the easy one. Login to the wiki and start!

The key areas that need to be looked at are:

- Do the guides to each tool cover all command line options
- Are the examples clear for the general cases
- Is the tools use clear
- In the Use cases, can we add more, do they need updating. Has upstream changed its approach

After that and always:

- Grammar
- Spelling
- Layout

2.7 Translate Toolkit Developers Guide

The goal of the translate toolkit is to simplify and unify the process of translation.

2.7.1 History

The initial toolkit was designed to convert Mozilla .dtd and .properties files into Gettext PO format. The logic was not that PO was in any way superior but that by simplifying the translations process i.e. allowing a translator to use one format and one tool that we could get more people involved and more translators.

The tools have now evolved to include other formats such as OpenOffice.org and the goal is still to migrate various formats to a common format, PO and in the future XLIFF as more tools become available for that format.

These tools we group as converters. Along the way we developed other tools that allowed us to manipulate PO files and check them for consistency. As we evolved the converter tools we have also improved and abstracted the classes that read the various file types. In the future we hope to define these better so that we have a more or less stable API for converters.

2.7.2 Resources

Git access

Translate Toolkit uses Git as a Version Control System. You can directly clone the translate repository or fork it at GitHub.

```
git clone https://github.com/translate/translate.git
```

Issues

- https://github.com/translate/translate/issues

Communication

- Development - no support related questions
- Help

2.7.3 Working with Bugzilla

When you close bugs ensure that you give a description and git hash for the fix. This ensures that the reporter or code reviewer can see your work and has an easy method of finding your fix. This is made easier by GitHub's Bugzilla integration.

Automated Bugzilla update from commits

Github will post comments on Bugzilla bugs when the commit messages make references to the bug by its bug number.

- Bugs are recognised by the following format (which are case-insensitive):

```
Bug 123
```

- Multiple bugs can be specified by separating them with a comma, ampersand, plus or "and":

```
Bug 123, 124 and 125
```

- Commits to all branches will be processed.
- If there is a "fix", "close", or "address" before the bug then that bug is closed.

```
Fix bug 123
```

2.7.4 Source code map

The source code for the tools is hosted on GitHub. This rough map will allow you to navigate the source code tree:

- convert – convert between different formats and PO format
- filters – *pofilter* and its helper functions (badly named, it is really a checking tool)
- storage – all base file formats: XLIFF, .properties, OpenOffice.org, TMX, etc.
- misc – various helper functions
- tools – all PO manipulation programs: *pocount*, *pogrep*, etc
- lang – modules with data / tweaks for various languages
- search – translation memory, terminology matching, and indexing / searching
- share – data files

2.7.5 Setup

The toolkit is installed by running:

```
./setup.py install
```

As root

The various setup options are yours to explore

2.7.6 General overview of the programs

Each tool in the toolkit has both a core program and a command line wrapper. For example the oo2po converter:

- oo2po – the command line tool
- oo2po.py – the core program

This is done so that the tools can be used from within the Pootle server thus reusing the toolkit easily.

Command line options

Getting lost with the command line options? Well you might want to volunteer to move some of them into configuration files. But in terms of programming you might be confused as to where they are located. Many of the command line options are implemented in each tool. Things such as `--progress` and `--errorlevel` are used in each program. Thus these are abstracted in **misc/optrecurse.py**. While each tools unique command line options are implemented in **xxx.py**.

2.7.7 Converters

The converters each have a class that handles the conversion from one format to another. This class has one important method **convertfile** which handles the actual conversion.

A function **convertXXX** manages the conversion for the command line equivalent and essentially has at least 3 parameters: inputfile, outputfile and templatefile. It itself will call the conversion class to handle conversion of individual files. Recursing through multiple files is handled by the optrecurse.py logic.

The converters **main** function handles any unique command line options.

Where we are headed is to get to a level where the storage formats themselves are more aware of themselves and their abilities. Thus the converter could end up as one program that accepts storage format plugins to convert from anything to almost anything else. Although our target localisation formats are PO and XLIFF only.

If you want to create a new converter it is best to look at a simple instance such as *csv2tbx* or *txt2po* and their associated storage classes. The *storage base class documentation* will give you the information you need for the storage class implementation.

2.7.8 Tools

The tools in some way copy the logic of the converters. We have a class so that we can reuse a lot of the functionality in Pootle. We have a core function that take: input, output and templates. And we have a **main** function to handle the command line version.

pocount should be converted to this but does not follow this conventions. In fact pocount should move the counting to the storage formats to allow any format to return its own word count.

2.7.9 Checks

There's really only one, *pofilter*. But there are lots of helper functions for pofilter. pofilters main task is to check for errors in PO or XLIFF files. Here are the helper file and their descriptions.

- autocorrect.py – when using `--autocorrect` it will attempt some basic corrections found in this file
- checks.py – the heart. This contains: the actual checks and their error reports, and defined variables and accelerators for e.g, `--mozilla`
- decorations.py – various helper functions to identify accelerators, variables and markers
- helpers.py – functions used by the tests
- prefilters.py – functions to e.g. remove variables and accelerators before applying tests to the PO message

pofilter is now relatively mature. The best areas for contributions are:

- more tests
- language specific configuration files
- tests for the tests – so we don't break our good tests
- defining a config files scheme to do cool stuff off of the command line. Globally enable or disable tests based on language, etc
- some approach to retesting that would remove '# (pofilter)' failure markings if the test now passes.
- ability to mark false positives

The *API documentation* is a good start if you want to add a new tests. To add a new language have a look at a language you understand amongst those already implemented.

2.7.10 Storage

These are the heart of the converters. Each destination storage format is implemented in its own file. Up until toolkit version 0.8, there was no formally defined API (the tools have been evolving and only recently stabilised), but they generally followed this structure. These classes are defined:

- XXelement – handles the low level individual elements of the file format. e.g. PO message, CSV records, DTD elements
- XXfile – handles the document or file level of the format. Eg a PO file, a CSV file a DTD file
  - fromlines – read in a file and initialise the various elements
  - tolines – convert the elements stored in XXelements and portions in XXfile to a raw file in that format

In the XML based formats e.g. TMX, XLIFF and HTML there is usually just an extended parser to manage the file creation.

Within each storage format there are many helper functions for escaping and managing the unique features of the actual format.

You can help by:

- abstracting more of the functions and documenting that so that we can get a better API
- adding other formats and converters e.g. .DOC, .ODF and others
- helping us move to a position where any format should convert to the base format: PO and in the future XLIFF without having to create a specific converter wrapper.
- Ensuring that our formats *conform to the standards*

Base Classes

From toolkit 0.9 onwards, we are moving towards basing all storage formats on a set of *base classes*, in the move to a universal API. We're also fixing things so that escaping is much more sane and handled within the class itself not by the converters.

In base classes we have different terminology

- XXXunit = XXXelement
- XXXstore = XXXfile

We have also tried to unify terminology but this has been filtered into the old classes as far as possible.

2.8 Making a Translate Toolkit Release

This page is divided in three sections. The first one lists the tasks that must be performed to get a valid package. The second section includes a list of tasks to get the package published and the release announced. The third one lists and suggests some possible cleanup tasks to be done after releasing.

Note: Please note that this is not a complete list of tasks. Please feel free to improve it.

2.8.1 Create the package

The first steps are to create and validate a package for the next release.

Get a clean checkout

We work from a clean checkout to ensure that everything you are adding to the build is what is in the repository and doesn't contain any of your uncommitted changes. It also ensures that someone else could replicate your process.

```
$ git clone git@github.com:translate/translate.git translate-release
$ cd translate-release
$ git submodule update --init
```

Check copyright dates

Update any copyright dates in `docs/conf.py:copyright` and anywhere else that needs fixing.

```
$ git grep 2013  # Should pick up anything that should be examined
```

Create release notes

We create our release notes in reStructured Text, since we use that elsewhere and since it can be rendered well in some of our key sites.

First we need to create a log of changes in the Translate Toolkit, which is done generically like this:

```
$ git log $previous_version..HEAD > docs/releases/$version.rst
```

Or a more specific example:

```
$ git log 1.10.0..HEAD > docs/releases/1.11.0-rc1.rst
```

Edit this file. You can use the commits as a guide to build up the release notes. You should remove all log messages before the release.

Note: Since the release notes will be used in places that allow linking we use links within the notes. These should link back to products websites (Virtaal, Pootle, etc), references to Translate and possibly bug numbers, etc.

Read for grammar and spelling errors.

Note: When writing the notes please remember:

1. The voice is active. 'Translate has released a new version of the Translate Toolkit', not 'A new version of the Translate Toolkit was released by Translate'.
2. The connection to the users is human not distant.
3. We speak in familiar terms e.g. "I know you've been waiting for this release" instead of formal.

We create a list of contributors using this command:

```
$ git log 1.10.0..HEAD --format='%aN, ' | awk '{arr[$0]++} END{for (i in arr){print␣
↪arr[i], i;}}' | sort -rn | cut -d\  -f2-
```

Up version numbers

Update the version number in:

- `translate/__version__.py`
- `docs/conf.py`
- `tests/cli/data/test_pofilter_manpage/stdout.txt`

In `translate/__version__.py`, bump the build number if anybody used the Translate Toolkit with the previous number, and there have been any changes to code touching stats or quality checks. An increased build number will force a Translate Toolkit user, like Pootle, to regenerate the stats and checks.

For `docs/conf.py` change `version` and `release`.

Todo: FIXME - We might want to consolidate the version and release info so that we can update it in one place.

The version string should follow the pattern:

```
$MAJOR-$MINOR-$MICRO[-$EXTRA]
```

E.g.

```
1.10.0
0.9.1-rc1
```

`$EXTRA` is optional but all the three others are required. The first release of a `$MINOR` version will always have a `$MICRO` of `.0`. So `1.10.0` and never just `1.10`.

Note: You probably will have to adjust the output of some of the functional tests, specifically the manpage ones, to use the right new version.

Build the package

Building is the first step to testing that things work. From your clean checkout run:

```
$ mkvirtualenv build-ttk-release
(build-ttk-release)$ pip install --upgrade setuptools pip
(build-ttk-release)$ pip install -r requirements/dev.txt
(build-ttk-release)$ make build
(build-ttk-release)$ deactivate
```

This will create a tarball in `dist/` which you can use for further testing.

Note: We use a clean checkout just to make sure that no inadvertent changes make it into the release.

Test install and other tests

The easiest way to test is in a virtualenv. You can test the installation of the new release using:

```
$ mkvirtualenv test-ttk-release
(test-ttk-release)$ pip install --upgrade setuptools pip
(test-ttk-release)$ pip install dist/translate-toolkit-$version.tar.gz
```

You can then proceed with other tests such as checking:

1. Documentation is available in the package
2. Converters and scripts are installed and run correctly:

   ```
   (test-ttk-release)$ moz2po --help
   (test-ttk-release)$ php2po --version
   (test-ttk-release)$ deactivate
   $ rmvirtualenv test-ttk-release
   ```

3. Meta information about the package is correct. This is stored in `setup.py`, to see some options to display meta-data use:

   ```
   $ ./setup.py --help
   ```

   Now you can try some options like:

   ```
   $ ./setup.py --name
   $ ./setup.py --version
   $ ./setup.py --author
   $ ./setup.py --author-email
   $ ./setup.py --url
   $ ./setup.py --license
   ```

(continues on next page)

(continued from previous page)

```
$ ./setup.py --description
$ ./setup.py --long-description
$ ./setup.py --classifiers
```

The actual descriptions are taken from `translate/__init__.py`.

2.8.2 Publish the new release

Once we have a valid package it is necessary to publish it and announce the release.

Tag and branch the release

You should only tag once you are happy with your release as there are some things that we can't undo. You can safely branch for a `stable/` branch before you tag.

```
$ git checkout -b stable/2.2.x
$ git push origin stable/2.2.x
$ git tag -a 2.2.5 -m "Tag version 2.2.5"
$ git push --tags
```

Release documentation

We need a tagged release before we can do this. The docs are published on Read The Docs.

- https://readthedocs.org/projects/translate-toolkit/versions/

Use the admin pages to flag a version that should be published.

Note: The branches like `stable/2.2.x` are automatically enabled on Read the Docs using *Automation Rules*, so there might be nothing to do here.

Publish on PyPI

Note: You need a username and password on Python Package Index (PyPI) and have rights to the project before you can proceed with this step.

These can be stored in `$HOME/.pypirc` and will contain your username and password. Check Create a PyPI account for more details.

Run the following to publish the package on PyPI:

```
$ workon build-ttk-release
(build-ttk-release)$ pip install --upgrade pyopenssl ndg-httpsclient pyasn1 twine
(build-ttk-release)$ twine upload dist/translate-toolkit-*
(build-ttk-release)$ deactivate
$ rmvirtualenv build-ttk-release
```

Create a release on Github

- https://github.com/translate/translate/releases/new

You will need:

- Tarball of the release
- Release notes in Markdown

Do the following to create the release:

1. Draft a new release with the corresponding tag version
2. Convert the major changes (no more than five) in the release notes to Markdown with Pandoc. Bugfix releases can replace the major changes with *This is a bugfix release for the X.X.X branch.*
3. Add the converted major changes to the release description
4. Include at the bottom of the release description a link to the full release notes at Read the Docs
5. Attach the tarball to the release
6. Mark it as pre-release if it's a release candidate

Update Translate Toolkit website

We use github pages for the website. First we need to checkout the pages:

```
$ git checkout gh-pages
```

1. In `_posts/` add a new release posting. This is in Markdown format (for now), so we need to change the release notes .rst to .md, which mostly means changing URL links from `` `xxx <link>`_ `` to `[xxx](link)`.
2. Change `$version` as needed. See `_config.yml` and **`git grep $old_release`**.
3. **`git commit`** and **`git push`** – changes are quite quick, so easy to review.

Announce to the world

Let people know that there is a new version:

1. Tweet about the release.
2. Post link to release Tweet to the Translate gitter channel.
3. Update Translate Toolkit's Wikipedia page

2.8.3 Post-Releasing Tasks

These are tasks not directly related to the releasing, but that are nevertheless completely necessary.

Bump version to N+1-alpha1

If this new release is a stable one, bump the version in `master` to `{N+1}-alpha1`. The places to be changed are the same ones listed in *Up version numbers*. This prevents anyone using `master` being confused with a stable release and we can easily check if they are using `master` or `stable`.

Note: You probably will have to adjust the output of some of the functional tests, specifically the manpage ones, to use the right new version.

Add release notes for dev

After updating the release notes for the about to be released version, it is necessary to add new release notes for the next release, tagged as `dev`.

Other possible steps

Some possible cleanup tasks:

- Remove your `translate-release` checkout.
- Update and fix these releasing notes:
  - Make sure these releasing notes are updated on `master`.
  - Discuss any changes that should be made or new things that could be added.
  - Add automation if you can.

We also need to check and document these if needed:

- Change URLs to point to the correct docs: do we want to change URLs to point to the `$version` docs rather then `latest`?
- Building on Windows, building for other Linux distros.
- Communicating to upstream packagers.

2.9 Deprecation of Features

From time to time we need to deprecate functionality, this is a guide as to how we implement deprecation.

2.9.1 Types of deprecation

1. Misspelled function
2. Renamed function
3. Deprecated feature

2.9.2 Period of maintenance

Toolkit retains deprecated features for a period of two releases. Thus features deprecated in 1.7.0 are removed in 1.9.0.

2.9.3 Documentation

Use the `@deprecated` decorator with a comment and change the docstring to use the Sphinx deprecation syntax.

```
@deprecated("Use util.run_fast() instead.")
def run_slow():
    """Run slowly

    .. deprecated:: 1.9.0
       Use :func:`run_fast` instead.
    """
    run_fast()  # Call new function if possible
```

2.9.4 Implementation

Deprecated features should call the new functionality if possible. This may not always be possible, such as the cases of drastic changes. But it is the preferred approach to reduce maintenance of the old code.

2.9.5 Announcements

Note: This applies only to feature deprecation and renamed functions. Announcements for corrections are at the coders discretion.

---

1. On **first release with deprecation** highlight that the feature is deprecated in this release and explain reasons and alternate approaches.
2. On **second relase** warn that the feature will be removed in the next release.
3. On **third release** remove the feature and announce removal in the release announcements.

Thus by examples:

Translate Toolkit 1.9.0: The `run_slow` function has been deprecated and replaced by the faster and more correct `run_fast`. Users of `run_slow` are advised to migrate their code.

Translate Toolkit 1.10.0: The `run_slow` function has been deprecated and replaced by `run_fast` and will be removed in the next version. Users of `run_slow` are advised to migrate their code.

Translate Toolkit 1.11.0: The `run_slow` function has been removed, use `run_fast` instead.

CHAPTER 3

Additional Notes

Changelog and legal information are included here.

3.1 Release Notes

The following are release notes for the Translate Toolkit releases.

These are the changes that have happened in the Translate Toolkit and which may impact you. If you use Pootle, Virtaal or any other application that makes use of the Translate Toolkit you may want to familiarize yourself with these changes.

3.1.1 Final releases

Translate Toolkit 3.1.1

Released on 23 September 2020

This release contains bug fixes.

Changes

General

- Fixed comparing units
- Fixed removal of units from a storage

Contributors

This release was made possible by the following people:

Michal Čihař

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 3.1.0

Released on 21 September 2020

This release contains improvements and bug fixes.

Changes

Formats and Converters

- YAML
  - Fix generating blank Ruby storage
  - Fix changing type of an unit
  - Preserve structured IDs on round trip
  - Several serialization fixes
- JSON
  - Preserve structured IDs on round trip
  - Several serialization fixes
- Android
  - Fix round trip of strings with newlines
  - Fix escaping double space with HTML
- CSV
  - Allow to control format autodetection
- XWiki
  - Added support for several XWiki formats
- RC
  - Fix parsing adjancent strings
  - Fix handling empty strings
- po2json
  - Never use empty translation in po2json

General

- Added support for removing units from a storage
- Cleaned up storage index usage
- Updated several dependencies
- Dropped version control integration from storage
- Use LRU cache for imports to improve factory performance

Contributors

This release was made possible by the following people:
Michal Čihař, Simon Urli, wojtek555, Stuart Prescott,
And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 3.0.0

Released on 15 June 2020

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Bring line wrapping closer to gettext
- XLIFF
  - Support non numeric ids on plurals in poxliff
- JSON
  - Added support for ARB files
  - Added support for go-i18n files
- Properties
  - Added support for GWT personality
  - Fix round trip of empty values without delimiter
- HTML
  - A makeover of the HTML parsing to fix several issues
- PHP
  - Add support for Laravel plurals
  - Improve round trip of some statements
- Windows RC

  - Rewritten parser using pyparsing
- l20n
  - Dropped support for deprecated format

General

- Dropped support for Python 2.7.
- Python 3.5 or newer is now required.
- Minor docs improvements.
- Several cleanups in code.
- Removed deprecated interfaces: - multistring no longer accepts encoding - search.segment is removed - pofile.extractpoline is removed - simplify_to_common no longer accepts languages parameter - getsource/setsource/gettarget/settarget methods removed from storages
- Updated requirements, lxml is no longer optional.
- Added and updated tests.
- Optional deps can be specified using pip extras.

Contributors

This release was made possible by the following people:

Michal Čihař, papeh, Yann Diorcet, Nick Schonning, Anders Kaplan, Leandro Regueiro, Javier Alfonso, Julen Ruiz Aizpuru

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.5.1

Released on 25 April 2020

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Avoid adding extra space on empty comment lines
  - Several performance improvements
- Android resources
  - Workaround broken plural handling for languages missing other tag
  - Fix setting rich content with comments
  - Fix setting target and removing markup

- YAML
  - Fixed handling of empty keys
  - Improved round trip preserving of comments and style
- TS
  - Avoid duplicating plurals definition
  - Fixed possible crash on adding new translations
- INI
  - Now supported on Python 3 thanks to iniparse support for it
- JSON
  - Allow usage of BOM in JSON files
- MO
  - Fixed context parsing
  - Fixed tests on big endian machines
- Catkeys
  - The catkeys format now has support for fingerprint calculation

Languages

- Updated plural definitions to match CLDR 36.

General

- Kept support for Python 2.7.
- Fixed py2exe support on Python 2.7.
- Minor docs improvements.
- Minor cleanups in code.
- Updated requirements.
- Added and updated tests.

Contributors

This release was made possible by the following people:

Michal Čihař, Nick Schonning, Tomáš Chvátal, Niels Sascha Reedijk.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.5.0

Released on 18 January 2020

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Avoid stripping empty lines from comments.
  - Raise error on invalid file content.
  - Fixed handling typecomments with non word chars.
  - Fixed serializing long msgidcomments.
- Properties
  - Avoid creating comment only units (issue 3928).
  - Fixed saving utf-16 Java files.
- Android resources
  - Document declaration is cloned when adding unit.
  - Fixed parsing plurals with comment.
  - Fixed setting plural with markup.
  - Fixed indentation of markup in translation.
  - Fixed XML entities handling.
- YAML
  - Quotes are preserved.
- TS
  - *ts2po* converts disambiguation notes and comments.
  - *po2ts* no longer removes consecutive linebreaks in source and translation.
- web2py
  - *@markmin* string is no longer copied to the translation.

Languages

- Added Sicilian language checks

General

- Kept support for Python 2.7.
- Added support for Python 3.8.
- Minor docs improvements.
- Minor cleanups in code.
- Updated requirements.
- Added and updated tests.

Contributors

This release was made possible by the following people:

Michal Čihař, Leandro Regueiro, Steve Mokris, Queen Vinyl Darkscratch, Matthias, David Paleino.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.4.0

Released on 24 June 2019

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Allow unicode in PO headers (issue 3896).
  - Improve finding out newline format for a file.
  - Strip UTF-8 BOM from input (issue 1640).
- XLIFF
  - Adjustments on how output is indented (issue 3424).
- Properties
  - Do not fail when parsing empty file.
- Android resources
  - Multiple adjustments so output is closer to Android Studio's.
- YAML
  - Switched to *ruamel.yaml* to simplify codebase and support YAML 1.2.
  - Added support for Ruby plurals.
  - Fixed handling dict in list (issue 3895).
  - Fixed parsing of empty YAML file.
- JSON
  - Fixed serialization of JSON arrays.
  - Placeholders are now kept in WebExtension dialect round trip conversion.
- RESX
  - Several improvements on formatting to align with Visual Studio's output.
- TS
  - Improved tags indentation.
  - Added support for new *vanished* type.
- Flat XML

  - Added support for this new format including *flatxml2po* and *po2flatxml* converters (issue 3776).
- CSV
  - No longer hardcode escape character (issue 3246).
  - Rewrote default dialect to make it more flexible.
- web2py
  - Updated converters code.
- Subtitles
  - Initialize duration on subtitle unit *__init__*.

Tools

- Tmserver: Fixed execution of unit API on Python 3.

Languages

- Updated plural definitions to CLDR 35.0.
- Removed trailing semicolon in Romanian plural definition.

Placeables

- Allow any character for Python mapping keys in *PythonFormattingPlaceable*.

API changes

- Altered storage code to have a consistent API for *removenotes*.
- Removed dependency on diff-match-patch.
- Removed embedded CherryPy wsgi server.
- Removed deprecated *has_key* implementation.

General

- Dropped no longer supported Python 3.3 and Python 3.4.
- Minor docs improvements.
- Updated requirements.
- Added and updated tests.

Contributors

This release was made possible by the following people:

Michal Čihař, Leandro Regueiro, Vinyl Darkscratch, Vitaly Novichkov, Stuart Prescott, Alex Tomkins, Darío Hereñú, BhaaL.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.3.1

Released on 06 October 2018

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Fixed parsing of files with mixed newlines *n* and *r*.
- Properties
  - Fixed escaping of serialized string for Joomla dialect.
  - Fixed loading of OS X strings dialect files having multiline strings.
- Android resources
  - Correctly handle escaping of question mark.
- PHP
  - Improved handling of unit name.
  - Handle *[]* style arrays.
  - Added support for *return* statement.
- YAML
  - Consistent handling of *int* and *bool*.
  - Fixed serialization of empty keys.
- JSON
  - Nested values ordering is now preserved.
- TMX
  - Avoid mentioning *po2tmx* in creation tool.
- RESX
  - New unit elements are now properly indented.
- INI
  - Enabled support for Python3 provided that patched *iniparse* library is available.
- RC

  - Altered to remove *r* before parsing.

API changes

- Use *backports.csv* module on Python 2 to align the behavior with Python 3 and drop many hacks.
- Removed deprecated *getoutput* methods deprecated in version 2.0.0.
- Added new deprecations:
  - Deprecated *setsource*, *getsource*, *gettarget* and *settarget* methods in favor of *source* and *target* properties for all storage classes, except *LISAunit* and its subclasses since for those these methods do actually accept additional arguments so can't just be always replaced by some property.
  - *xliffunit*: Deprecated *get_rich_source* in favor of *rich_source* property
- Defined for all unit classes the *rich_source* and *rich_target* properties without using methods. Since the old methods were private they were directly removed without deprecating them.

General

- Refactored more converters to increase readability and use a common pattern which will allow to further refactor repeated code
- Code cleanups and simplification
- Updated requirements
- Tests:
  - Added plenty of tests
  - Tests cleanups and fixes

Contributors

This release was made possible by the following people:

Leandro Regueiro, Michal Čihař, BhaaL, Mark Jansen, Stuart Prescott, David Hontecillas.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.3.0

Released on 18 March 2018

This release contains improvements and bug fixes.

Changes

Formats and Converters

- PO
  - Avoid escaping some characters (`()` / `:,`) that don't need to be escaped

  - Wrap lines on / like Gettext
  - Lines can be wrapped at specified length
  - MO units now allow to set the unit context
  - Always URL-encode locations
- PHP
  - Full rewrite of the PHP format using `phply`:
    * Fixes multiple issues
    * Brings support for new dialects: `heredoc`, `nowdoc`, short array syntax and nested arrays.
- YAML
  - Added `yaml2po` and `po2yaml` converters
  - Fixed some minor bugs
  - Documented unsupported features
- JSON
  - Refactored the storage classes to get rid of repeated shared code, reduce memory usage and get readable representation of the units.
- txt
  - Added `--no-segmentation` flag to `txt2po`

Tools

- Removed `translate.convert.poreplace`

General

- Improved support for Windows
- Refactored multiple converters to increase readability and use a common pattern which will allow to further refactor repeated code
- Tests:
  - Enabled testing on Windows
  - Added more tests
  - Plenty of tests cleanups and fixes
- Docs:
  - Updated docs on installation
  - Improved automatic generation of docs on factories

Contributors

This release was made possible by the following people:

Leandro Regueiro, Michal Čihař, Stuart Prescott, Nick Schonning, Johannes Marbach, andreistefan87, Alejandro Mantecon Guillen.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.5

Released on 01 September 2017

This release contains improvements and bug fixes.

Changes

Formats and Converters

- XLIFF
  - Fixed bug when adding new units to XLIFF store.
- JSON
  - Added support for i18next JSON dialect.
  - Improved WebExtension JSON dialect support.

Contributors

This release was made possible by the following people:

Michal Čihař, Leandro Regueiro, Ryan Northey.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.4

Released on 31 July 2017

This release contains improvements and bug fixes.

Changes

Formats and Converters

- XLIFF
  - Added support for *.xliff* extension in all converters and tools that support *.xlf* extension.
- JSON
  - Added support for nested JSON.
  - Added support for WebExtension JSON dialect.

- txt
  - *po2txt* skips obsolete and non-translatable strings.

Filters and Checks

- The *puncspace* check now strips Bidi markers chars before processing.
- Added *ReducedChecker* checker to list of checkers.

API changes

- Language and country default to *common_name* if available.
- Added function to retrieve all language classes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Leandro Regueiro, Michal Čihař, Rimas Kudelis, Ludwig Nussel, Stuart Prescott.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.3

Released on 20 June 2017

This release contains improvements and bug fixes.

Changes

- Added *MinimalChecker* and *ReducedChecker* checkers.

Contributors

This release was made possible by the following people:

Rimas Kudelis, Leandro Regueiro.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.2

Released on 20 June 2017

This release contains improvements and bug fixes.

Changes

- Fixed resolving of country names translations.

Contributors

This release was made possible by the following people:

Ryan Northey, Leandro Regueiro.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.1

Released on 19 June 2017

This release contains many improvements and bug fixes.

Major changes

- Refactored functions for resolving language/country names translation to be memory efficient.
- Improvements for ts and subtitles formats.
- Added *–preserveplaceholders* argument to *podebug*.
- Fixed Montenegrin language name.

Detailed changes

Formats and Converters

- ts
  - Write quotes as entities
  - Remove not necessary encoding/decoding to UTF-8
- Subtitles
  - Avoid errors when subtitle support is missing

Tools

- Added *–preserveplaceholders* argument to *podebug* to avoid rewriting placeholders

Languages

- Fixed Montenegrin language name.

API changes

- Refactored functions for resolving language/country names translation to be memory efficient

General

- Use gzip for packaging
- Python 3 fixes
- Added more tests

. . . and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Michal Čihař, Leandro Regueiro, Ryan Northey, Robbie Cole, Kai Pastor, Dwayne Bailey.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.2.0

Released on 15 June 2017

This release contains many improvements and bug fixes.

Major changes

- Avoid resolving external entities while parsing XML.
- Improvements for Android, ts and resx formats.
- Added support for PHP nested arrays.
- Added Kabyle language

Detailed changes

Requirements

- Updated requirements.
- Added *pycountry* recommended requirement for localized language names.

Formats and Converters

- XML formats
  - Avoid resolving external entities while parsing.
- Properties
  - Improved behavior for strings with no value.
- Android resources
  - Improved newlines handling.
  - Strip leading and trailing whitespace.

- PHP
  - Added support for nested named arrays and nested unnamed arrays.
- ts
  - Handle gracefully empty location tag.
  - Encode *po2ts* output as UTF-8.
- resx
  - Improved skeleton.
  - Fixed indent of the </data> elements.

Languages

- Added Kabyle language.

API changes

- Added functions to retrieve language and country ISO names.
- If available, *pycountry* is used first to get language names translations.

General

- Python 3 fixes
- Added more tests

. . . and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Michal Čihař, Taras Semenenko, Leandro Regueiro, Rimas Kudelis, BhaaL, Muend Belqasem, Jens Petersen.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.1.0

Released on 17 March 2017

This release contains many improvements and bug fixes.

Major changes

- Fixed *RomanianChecker* checks.
- Added an iOS checker style.
- Changed plural equations for Slovenian, Persian, Kazakh and Kyrgyz.

- Several fixes in formats and tools.

Detailed changes

Python 3 support

- Python 3.6 is now supported.

Requirements

- Updated and pinned requirements.
- Now recommended requirements pulls required requirements.

Formats and Converters

- All formats
  - *locationindex* now uses first duplicate unit in case of several units having the same location in order to keep duplicate entries in some formats when converting from PO format.
- PO
  - Only add duplicate unit if *msgcxt* is unique, in order to be able to convert monolingual formats with duplicate entries to PO.
- Properties
  - Added support for Joomla dialect.
- ts
  - Set the right context on the units.
- YAML
  - Fixed parsing of unicode values in lists.
- HTML
  - Use character offset in line for unit location in order to keep parsing repeated strings in different units.
- txt
  - Use line number on unit location to keep parsing repeated strings in different units.

Filters and Checks

- Fixed *RomanianChecker* checks.
- Added an iOS checker style to detect iOS variables styles such as `%@` and `$(VAR)`.

Tools

- *posegment* no longer outputs duplicate headers,

Languages

- Changed plural equations for Slovenian, Persian, Kazakh and Kyrgyz.

API changes

- Changed management of Xapian locks to prevent database corruption.

General

- Python 3 fixes
- Removed unused code

. . . and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Leandro Regueiro, Michal Čihař, Ryan Northey, Friedel Wolff, Olly Betts, Claude Paroz.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 2.0.0

Released on 27 January 2017

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes and optimizations for the upcoming Pootle 2.8.0 and Virtaal releases.

Major changes

- Python 3 compatibility thanks to Claude Paroz
- Dropped support for Python 2.6
- Support for new l20n format
- Translate Toolkit can now easily be installed on Windows
- Changes in storage API to expose a more standardized API

Detailed changes

Python 3 support

- Translate Toolkit went through a massive code cleanup looking forward Python 3 compatibility. There might be quirks that need to be fixed, so please test and report any issue you might find.
- Python 3.3-3.5 is now supported.

Requirements

- `lxml` requirement was raised to 3.5.0 in order to simplify code.
- Updated and pinned requirements
- Removed misleading extra requirements files

Formats and Converters

- PO
  - msgid comments (KDE style) are only parsed from msgid now.
  - Fixed parsing of PO files with first entry in unicode
  - Fixed parsing of locations with percent char
- XLIFF
  - Unaccepted ASCII control characters are now escaped in XLIFF
- DTD
  - Newlines are now skipped when parsing (issue 3390).
  - Invalid ampersands are not scrubbed anymore.
  - label+accesskey is now only extracted if it is not followed by space.
- Properties
  - Keys can contain delimiters if they are properly wrapped (issue 3275).
  - Fix control characters escaping for utf-8 encoding.
  - *po2prop* removes fully untranslated units if required
  - *po2prop* skips first entry in PO file (issue 3463)
- Mozilla .lang
  - `{ok}` marker is now more cleanly removed
  - Always output last unit followed by trailing newlines
  - Added support for headers and tag comments
  - `MAX_LENGTH` is now parsed into comment
  - File line endings are now remembered, defaulting to Unix LF
- Mozilla's l20n
  - Added this new format storage class
  - Added variants and traits support
  - Added new converters `l20n2po` and `po2l20n`
- Android
  - Unknown locales no longer produce failures.
  - Simplify newlines handling as the format now handles n and newline equally (issue 3262)
  - Moved all namespaces to <resources> element.

  - Simplified newlines handling
- ODF
  - *odf2xliff* now extracts all the text (issue 3239).
- ts
  - XML declaration is written with double quotes.
  - Self-closing for 'location' elements are not output anymore.
- JSON
  - Output now includes a trailing newline.
  - Unit ordering is maintained (issue 3394).
  - Added `--removeuntranslated` option to *po2json*
- YAML
  - YAML format support has been added.
- txt
  - *po2txt* works correctly again when `--threshold` option is passed (issue 3499)
- ical
  - Enabled this format for Python 3 too.
- TermBase eXchange (TBX)
  - *tbx2po* converter added
  - Added basic support for Parts of Speech and term definitions.
- Fixed error when writing back to the same file (issue 3419).

Filters and Checks

- Added the ability to skip some checks for some languages in specific checkers
- `accelerators` check reports an error if accelerator is present for several Indic languages in `MozillaChecker` checker.
- Added `l20nChecker` to do custom checking for Mozilla's new l20n format.
- LibreOffice checker no longer checks for Python brace format (issue 3303).
- LibreOffice validxml check correctly matches self-closing tags.
- Numbers check now handles non latin numbers. Support for non latin numbers has been added for Arabic, Assamese, Bengali and Persian languages.
- Fixed issue that prevented standard checks from being used in Pootle with default settings.
- Fixed missing attribute warning displayed when using `GnomeChecker`, `LibreOfficeChecker` and `MozillaChecker` checkers.
- Added language specific `RomanianChecker`.

Tools

- *posegment* now correctly segments Japanese strings with half width punctuation sign (issue 3280).
- *pocount* now outputs csv header in one line. It also outputs using color.
- `buildxpi` was adjusted to current Mozilla needs

Languages

- Fixed plural form for Montenegro, Macedonian, Songhay, Tajik, Slovenian and Turkish.
- Added plural forms for Bengali (Bangladesh), Konkani, Kashmiri, Sanskrit, Silesian and Yue (Cantonese).
- Added valid accelerators for Polish.
- Renamed Oriya to Odia.
- Altered Manipuri name to include its most common name Meithei.
- Added language settings for Brazilian Portuguese.
- Added Danish valid accelerators characters (issue 3487).
- Added additional special characters for Scottish Gaelic.

Setup

- Fixed Inno Setup builds allowing to easily install Translate Toolkit on Windows using the `pip` installer. Commands are compiled to .exe files.
- Updated installation instructions for Windows

API changes

- Dropped `translate.misc.dictutils.ordereddict` in favor of `collections.OrderedDict`.
- Added encoding handling in base `TranslationStore` class exposing a single API.
- Encoding detection in `TranslationStore` has been improved.
- Standardized UnitClass definition across `TranslationStore` subclasses.
- `translate.misc.multistring.multistring`:
  - Fixed list coercion to text
  - Fixed comparison regression with multistrings (issue 3404).
  - Re-added `str` method (issue 3428).
  - Fixed `__hash__` (issue 3434).

API deprecation

- Passing non-ASCII bytes to the `multistring` class has been deprecated, as well as the `encoding` argument to it. Applications should always construct `multistring` objects by passing characters (`unicode` in Python 2, `str` in Python 3), not bytes. Support for passing non-ASCII bytes will be removed in the next version.

- `TxtFile.getoutput()` and `dtdfile.getoutput()` have been deprecated. Either call `bytes(<file_instance>)` or use the `file_instance.serialize()` API if you need to get the serialized store content of a `TxtFile` or `dtdfile` instance.

General

- Dropped support for Python 2.6 since it is no longer supported by the Python Foundation. Sticking to it was making us difficult to maintain code while we move to Python 3.
- Misc docs cleanups.
- Added more tests.
- Increased Python code health.
- Legacy, deprecated and unused code cleansing:
  - Dropped code for no longer supported Python versions.
  - Removed unused code from various places across codebase.
  - The legacy `translate.search.indexing.PyLuceneIndexer1` was removed.
  - The deprecated `translate.storage.properties.find_delimiter()` was removed and replaced by the `translate.storage.properties.Dialect.find_delimiter()` class method.
  - Python scripts are now available via *console_scripts* entry point, thus allowing to drop dummy files for exposing the scripts.

. . . and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Claude Paroz, Leandro Regueiro, Dwayne Bailey, Michal Čihař, Taras Semenenko, Ryan Northey, Stuart Prescott, Kai Pastor, Julen Ruiz Aizpuru, Friedel Wolff, Hiroshi Miura, Thorbjørn Lindeijer, Melvi Ts, Jobava, Jerome Leclanche, Jakub Wilk, Adhika Setya Pramudita, Zibi Braniecki, Zdenek Juran, Yann Diorcet, Nick Shaforostoff, Jaka Kranjc, Christian Lohmaier, beernarrd.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 1.13.0

Released on 13 May 2015

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes for the upcoming Pootle 2.6.0 and 2.7.0 and Virtaal releases.

It is just over 6 months since the last release and there are many improvements across the board. A number of people contributed to this release and we've tried to credit them wherever possible (sorry if somehow we missed you).

Highlighted improvements

Major changes

- New converters for IDML format
- Support for new .Net Resource (.resx) format
- Extensive cleanup on ODF converters
- New quality checks

Formats and Converters

- IDML
  - Added the `idml2po` and `po2idml` converters.
- .Net Resource (.resx)
  - Added store to represent the format and the `resx2po` and `po2resx` converters.
- Android
  - Improved escape and unescape of Android resources with HTML markup.
  - Fixed bug in canceling whitespaces with backslash when unescaping.
- ODF
  - Removed the `--engine` option in `odf2xliff` because the `itools` third party library is no longer used.
- TS
  - Pretty print output the same as Qt Linguist (issue 1420)
- JSON
  - Dump content on memory instead of copy of parsed file (issue 3249).
- PHP
  - Correctly roundtrip PHP with spaces after array (issue 3231).
- Mozilla lang
  - Import only real comments (starting with #), not meta tags (starting with ##).
- XLIFF
  - Mark units as needing attention if sources don't match when merging units.
  - `pot2po` now also accepts files with .xliff extension
- po2moz
  - Fixed handling of files with fullstop in filename

Quality Checks

- Added quality check for Python brace format.
- Added the ability to skip some quality checks for the he, ug, zh_CN, zh_HK and zh_TW languages.
- Expanded printf quality check to support reordering boost::format positional directives.

- Expanded docstrings to include fully detailed descriptions in order to display them on Pootle.

Tools

- Removed the unnecessary dependency on `lxml` in `pretranslate` (issue 1909)

Languages

- Language plurals:
  - Fixed plural forms for ga and pt_BR languages
  - Added new plural forms for new languages
- Adjusted punctuation for zh
- Corrected "Songhay" language name

General

- Fixed bug in file discovery that prevented Pootle Pootle's terminology feature from working properly in some scenarios.
- Docs:
  - Major rewrite of releasing instructions
  - Reorganized string-related guidelines on styleguide
  - Other minor docs cleanups
- ODF code extensive cleanups:
  - Applied tons of PEP8 and style guide cleanups
  - Removed unused code
  - Removed unused test ODT file
  - Added lots of docstrings
  - Simplified code to ease maintainability and improve readability
- Dropped no longer working automatic publishing in PyPI and SourceForge
- Several changes to speed up Travis builds
- Unhid some tests

. . . and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Leandro Regueiro, Dwayne Bailey, Yaron Shahrabani, Sarah Hale, Sietse Brouwer, Jerome Leclanche, Julen Ruiz Aizpuru, Michael Andres, William Grzybowski, SirAnthony, Rafael Ferreira, Luka Kama, Francesco Lodolo, Buganini, babycaseny.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 1.12.0

Released on 12 August 2014

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes and optimizations for the upcoming Pootle 2.6.0 and Virtaal releases.

It is just over 6 months since the last release and there are many improvements across the board. A number of people contributed to this release and we've tried to credit them wherever possible (sorry if somehow we missed you).

Highlighted improvements

Major changes

- Properties and DTD formats fix a number of issues
- Massive code cleanup looking forward Python 3 compatibility
- Important changes in development process to ease testing

Formats and Converters

- Mozilla properties
  - If a unit has an associated access key entry then these are combined into a single unit
  - Encoding errors are now reported early to prevent them being masked by subsequent errors
  - Leading and trailing spaces are escaped in order to avoid losing them when using the converters
  - The `\uNN` characters are now properly handled
  - po2prop Now uses the source language accesskey if translation is missing
  - Fixed conversion of successive Gaia plural units in prop2po
- DTD
  - The `&` entity is automatically expanded when reading DTD files, and escaped back when writing them
  - Underscore character is now a valid character in entity names
  - Nonentities at end of string are now correctly handled
  - po2dtd:
    - Now uses the source language accesskey if target accesskey is missing
    - Doesn't remove stray `&` as they probably `&`
- HTML
  - The HTML5 `figcaption` tag is now localizable
  - The `title` attribute is now localizable
  - po2html now retains the untranslated attributes
- Accesskeys
  - Now accesskeys are combined using the correct case
  - Added support for accesskey after ampersand and space

- PHP
  - Fall back to default dialect after adding every new unit
  - Added support for empty array declaration when it is filled later
- Android
  - Added support for plurals
  - Text is now properly escaped when using markup
- Qt Linguist (.ts)
  - The message id attribute is added to contextname
  - Files now output the XML declaration (issue 3198)
- RC
  - RC format received some bugfixes and now ignores `TEXTINCLUDE` sections and one line comments (`//`)
- XLIFF
  - `xliff2po` now supports files with `.xliff` extension
- OS X .strings
  - Added support for UTF-8 encoded OS X strings
- Testing
  - Added new tests for the UTF-8 encoded OS X strings, Qt linguist and RC formats and the `rc2po` converter

Version Control improvements

- Added support for Subversion `.svn` directories

Checks

- Added specific checks for LibreOffice

Tools

- The `pocount` tool has now a better counting algorithm for things that look like XML

Mozilla tooling fixes

- Added support to check for bad accesskeys in .properties files
- Now the Mozilla roundtrip script can be silently run
- Added a new Gaia roundtrip script
- The `buildxpi --disable-compile-environment` option has been restored, resulting in huge speed improvements

General

- Extensive cleanup of setup script
- Some bugfixes for placeables
- Misc docs cleanups
- Recovered diff-match-patch to provide support for old third party consumers
- Minor change in placeables to correctly insert at an existing parent if appropriate
- Code cleanups:
  - Applied tons of PEP8 and style guide cleanups
  - Python 2.6 is our new minimum:
    - Removed lots of code used to support old Python versions
    - Dropped custom code in favor of Python standard libraries
    - Updated codebase to use newer libraries
    - Changed code to use newer syntax seeking Python 3 compatibility
  - Updated some third party bundled software: CherryPy, BeautifulSoup4
  - Added document to track licenses used by third party bundled code
  - Removed TODO items. Some of them were moved to the bug tracker
- Development process:
  - Added a functional test framework
  - Added dozens of new unit and functional tests
  - Expanded the tasks performed in Travis: pep8, pytest-xdist, compile all files, coveralls.io, ...

... and loads of general code cleanups and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Jerome Leclanche, Leandro Regueiro, Khaled Hosny, Javier Alfonso, Friedel Wolff, Michal Čihař, Heiki Ojasild, Julen Ruiz Aizpuru, Florian Preinstorfer, damian.golda, Zolnai Tamás, Vladimir Rusinov, Stuart Prescott, Luca De Petrillo, Kevin KIN-FOO, Henrik Saari, Dominic König.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 1.11.0

Released on 22 January 2014

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes and optimizations for the upcoming Pootle 2.5.1 and Virtaal releases.

It is just over a ten months since the last release and there are many improvements across the board. A number of people contributed to this release and we've tried to credit them wherever possible (sorry if somehow we missed you).

Highlighted improvements

Major changes

- The PO format now matches Gettext more closely
- PHP format adds a number of new features
- Support for Python 2.5 has been dropped

Formats and Converters

- Gettext PO:
  - cPO now handles fuzzy obsolete messages correctly
  - Line wrapping improvement allow PO files to more closely match Gettext
  - Optimization to increase performance
- PHP:
  - Warn about duplicate entries
  - Allow blank spaces in array declaration (issue 2646)
  - Support nested arrays (issue 2240)
- XLIFF:
  - Correctly parse XLIFF 1.2
- Properties
  - Blank source text is now always translated
  - Fuzzy units are discarded with –remove-untranslated
  - prop2po no longer drops entries that are translated the same as the source
- TMX:
  - po2tmx support comments
- Android:
  - Detect untranslatable resource strings
  - Various format improvements
- HTML:
  - Output HTML source in po2html when a unit is fuzzy (issue 3145)
- New conversion options:
  - `--timestamp` – skip conversion if the output file has a newer timestamp (Makefile-alike)
  - `--threshold` – in po2* converters this allows you to specify a percentage complete threshold. If the PO files passes this threshold then the file is output (issue 2998)
  - `--removeuntranslated` – Extend this option to po2dtd and thus po2moz – don't output untranslated text (issue 1718)

Language specific fixes

- The toolkit now supports: Sakha, N'ko, Turkish, improvements for Bengali & Hindi
- Pootle special characters are now stored on Toolkit and available for other tools to use
- Rules for language `ab` are now available for language `ab_CD`

Checks

- Spelling test improvements including speed and optimization
- Reduce false positive for the filepath test in cases of self closing tags e.g. `<br />`
- Lowered the accelerator check severity to reduce false positive impact

Mozilla tooling fixes

- Better decoding of some characters in DTD e.g » and &x0022 (")
- .lang – Improved support for untranslated entries
- `buildxpi`:
  - Can now build multiple languages at once (issue 2999)
  - Set a max product version to allow the language pack to continue to work once the browser version has moved out of Aurora channel
- Dropped native XPI building support (untested and no longer used)
- Add Mozilla plural formulas, in time we'll handle Mozilla plurals correctly

General

- Dropped support for Python 2.5 since it is no longer supported by the Python Foundation. Also sticking to it was preventing us from using features that are not supported on Python 2.5 but they are on later versions.
- Dropped psyco support – it is no longer maintained
- Use logging throughout instead of `sys.stderr`
- Lots of cleanups on docs: TBX, PHP, added Android and JSON docs
- Use requirements files for documenting all requirements and make it easy to install Translate Toolkit using `pip`
- Added some functional tests
- Improve searching to find words with hyphens
- Choose the closest repo in nested VCS
- Test suite down to zero failing tests
- Handle a broken version of python-Levenshtein
- Improve handling of escapes in wrapping

. . . and loads of internal changes to improve maintainability, remove unused imports, remove unused code and general code cleanups, some changes to ensure future Python 3 portability and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Leandro Regueiro, Alexander Dupuy, Friedel Wolff, Khaled Hosny, Michal Čihař, Jordi Mas, Stuart Prescott, Trung Ngo, Ronald Sterckx, Rail Aliev, Michael Schlenker, Martin-Zack Mekkaoui, Iskren Chernev, Luiz Fernando Ranghetti & Christian Hitz

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 1.10.0

Released on 12 March 2013

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes for the upcoming Pootle 2.5.0.

It is just over a year since the last release so there are many improvements across the board. A number of people contributed to this release and we've tried to credit them wherever possible (sorry if somehow we missed you).

Highlighted improvements

- Android format support
- Version control improvements
- Source now on Github - all our code is now on github
- Documentation - migrated all from our wiki into the code and Read The Docs
- Continuous Integration using Travis

Most important for Pootle

- Version control improvements
- Categorize pofilter checks into critical, functional, cosmetic, etc

Formats and Converters

- Android format support [Michal Čihař]
- Mozilla .lang, many improvements
- PHP support for defintions, // comments and improved whitespace preservation
- PO: X-Merge-On header to explicitly demand a conversion strategy instead of guessing
- .properties: BOMs in messages and C style comments [Roman Imankulov]
- Mac OS String formatting improved [Roman Imankulov]
- The spaces in DTD files are now preserved. For example the spaces in `<!ENTITY some.label "definition">` around the entity name `some.label` are now kept.

- The matching criterion when merging units can now be specified with the `X-Merge-On` header. Available values for this header are *location* and *id*. By default merges will be done by matching IDs. This supersedes the effects of the `X-Accelerator` header when merging and establishes an explicit way to set the desired matching criterion.

Version Control improvements

- Interface for adding files to a repository & Implement .add() for all VCSs.
- Caching of VC version info
- Don't look for VCS if it's not available
- Stop looking for VCS at a given parent
- Subversion VC tests
- Alway pass -m to 'commit' in Subversion to prevent blocking

Checks

- New OpenOffice variables style used in extensions
- Check for self-closing tags in the xmltags test [Seb M].
- GConf test fixes
- Terminology checker type for future terminology features
- Categorize pofilter checks into critical, functional, cosmetic, etc
- Added support for Objective-C %@ printf specifiers

Language specific fixes

- Correct plurals: Scottish Gaelic (gd), Irish
- Plural rules: Fulah, Brazilian Portuguese
- Punctuation rules and tests to ignore for: Burmese, Urdu, Afrikaans, Wolof

Documentation

- Moved to Git and we are now using reStructured Text and Sphinx
- Published in Read The Docs (RTD).
- Old wiki migrated to RTD.
- New clean theme for documentation and website
- API and code epydoc moved to reStructured Text.
- Translate code Style Guide written

Mozilla tooling fixes

- Mozilla specific test for dialog size settings
- Gaia properties dialect and plural handling
- Fixes and improvement to the Firefox build scripts
- Improved accesskey detection
- Improved DTD escaping for &quote, %, etc
- Improvement of DTD to align with Base classes
- Support new {{xx}} variable style introduced in PDF viewer

. . . and refactoring, PEP8, test coverage and of course many many bugfixes.

Contributors

This release was made possible by the following people:

Dwayne Bailey, Friedel Wolff, Leandro Regueiro, Julen Ruiz Aizpuru, Michal Čihař, Roman Imankulov, Alexander Dupuy, Frank Tetzel, Luiz Fernando Ranghetti, Laurette Pretorius, Jiro Matsuzawa, Henrik Saari, Luca De Petrillo, Khaled Hosny, Dave Dash & Chris Oelmueller.

And to all our bug finders and testers, a Very BIG Thank You.

Translate Toolkit 1.9.0 Released

Released on 12 April 2011

This release contains many improvements and bug fixes. While it contains many general improvements, it also specifically contains needed changes for the upcoming Pootle 2.1.6 and Virtaal 0.7.

Highlighted improvements

- Faster terminology matching
- Several small optimisations to performance and memory use
- More advanced state support (visible in pocount and Virtaal 0.7)
- Improved language detection models (+South African languages)
- Improve handling of printf variable reordering [Jacques Beaurain]
- Review of the wording of the messages of pofilter checks
- Better sentence segmentation for some non-Latin languages
- More supported formats for podebug
- Extra options for pomerge, pogrep and po2oo/xliff2oo.

The new pogrep options made this possible for GNOME.

Most relevant for Pootle

- Support for Xapian 1.2 (issue 1766) [Rimas Kudelis]
- Work around some changes introduced in Django 1.2.5/1.3

Format support

- Always use UNIX line endings for PO (even on Windows)
- XLIFF and .ts files now shows "fuzzy" only the target present
- Improved support for .ts comment as context (issue 1739)
- Support for Java properties in UTF-8 encoding
- More natural string ordering in json converter
- Improved handling of trailing spaces in Mozilla DTD files
- Removed unused support for _old_ KDE plurals in pocount

. . . and several small bugfixes

Translate Toolkit 1.8.1

Released on 19 Novermber 2010

Today the Translate team released version 1.8.1 of the Translate Toolkit. The Translate Toolkit contains many useful tools for translation, management, and quality control. It is the technology platform for Pootle, Virtaal, and other software.

This release contains many improvements and bug fixes. It is a recommended upgrade for users of Pootle and Virtaal. There were over 200 commits since version 1.8.0.

This work was made possible by volunteers and our funders:

- ANLoc, funded by IDRC

Highlighted improvements

File formats:

- A rewrite and major improvement of the html format and html2po converter
- New JSON format introduced
- Support for Universal Terminology Exchange (UTX) format
- Support for Java properties files encoded in UTF-8
- Improvements to CSV format, and improved compatibility with Excel exports
- Bug fixes to Qt .ts
- Support for XLIFF's state attributes (pocount now lists detailed state statistics)
- Minor bug fixes for PHP format

Languages and quality checks:

- Support for Persian quotations

- Major performance improvements to quality checks

Pootle will regenerate all statistics with the new Translate Toolkit installed. Read about the quality checks.

Other improvements:

- Improvements to stability of Lucene text indexing (affecting Pootle)
- Parameter for po2prop to ignore untranslated strings
- Many improvements to pot2po including Qt ts support, improved handling of extra XML namespaces in XLIFF, and performance improvements.

Further resources:

- Full feature list
- Download
- Bugs

Happy translating!

The Translate team

3.1.2 Historic releases

Translate Toolkit 1.8.0

Released on 17 August 2010

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.7.0

Released on 13 May 2010

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.6.0

Released on 3 March 2010

PO files now always have headers

Generated PO files now always contain headers. This will mainly affect the output of pofilter and pogrep. This should allow better interoperability with gettext tools, and allowed for some improvement in the code. You should still be able to use headerless files in msgmerge, although it is recommended that PO files are consistently handled with headers wherever possible.

Translate Toolkit 1.5.3

Released on 4 February 2010

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.5.2

Released on 13 January 2010

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.5.1

Released on 8 December 2009

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.5.0

Released on 25 November 2009

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.4.1

Released on 6 November 2009

CSV column header names

The names given to CSV column headers have been changed. Early releases of *csv2po* would name the columns "comment,original,translation". This was done mostly to make it easy for non-technical translators. However, comments in the command line help used terms like source and target. This release changes the column header names to "location,source,target", this aligns with terms used throughout the toolkit.

If you have CSV file generated by older versions of the toolkit then a header entry of "comment,original,translation" will be turned into a unit instead of being ignored. You can either change your CSV file to use the headers "location,source,target" or delete the header row completely. Once this is done the files will work as expected.

Translate Toolkit 1.4.0

Released on 27 August 2009

Java and Mozilla .properties

Unusual keys, separators and spacing should all be handled correctly now. Some Mozilla .properties files might now have changed. Regenerate your Mozilla l10n files from fresh POT files without any changes to your PO files to ensure that you can see and review these changes.

Hashing in podebug

The `--hash` option in *podebug* has been replaced by a format specifier %h to be able to better control the positioning of the hash value.

Translate Toolkit 1.3.0

Released on 11 February 2009

Several duplicate styles were removed as has been warned about long before. Please check the recommendations posted at the time that msgctxt was added on how to migrate.

Translate Toolkit 1.2.1

Released on 29 December 2008

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 1.2.0

Released on 24 November 2008

New formats

The toolkit now supports:

- *Qt Phrase Book (.qph)*
- *Qt .ts* v1.1

This allows reading, counting and working on these formats. The *ts2po* converter has not been changed so you will not be able to benefit from the new .ts support. However, you can use the format for translation memory, etc as its is now fully base class compliant.

Stats database change

There were some changes in the database used by pocount for storing statistics. The location of the database might also have changed, depending on what the last version is that you used. Remove the file stats.db from any of ~/.translate_toolkit, ~/.wordforge (or the corresponding directories on your Windows installation.

Valid accelerators

The *pofilter* accelerator test is now able to make use of a list of valid accelerators. This allows translators to control the behaviour of the test for their language and add or remove characters that can be used as accelerators. Please define the valid accelerators for your language and these will then be included in future releases of the toolkit. By default the old process is followed so that if you take no action then this check will continue to work as expected.

branches

These are branches that contain quite invasive changes that will most likely be merged into the main development and be released sometime in the future.

toolkit-C-po

Converting the current Python based PO parser to the Gettext C based parser for PO. This offers quite a dramatic speed improvement and conformance to the output found in Gettext itself. For most users there will be a number of changes in layout of the files as they will now conform fully to Gettext layout. The 'keep' option in `--duplicatestyle` will no longer be supported as this is not valid Gettext output.

Translate Toolkit 1.1.1

Released on 2 April 2008

Premature termination of DTD entities

Although this does not occur frequently a case emerged where some DTD entities where not fully extracted from the DTD source. This was fixed in issue 331.

We expect this change to create a few new fuzzy entries. There is no action required from the user as the next update of your PO files will bring the correct text into your translations, if you are using a translation memory your translation might be recovered from obsolete translations.

Translate Toolkit 1.1.0

Released on 22 January 2008

oo2po Help (helpcontent2) escaping fixed

OpenOffice.org Help (helpcontent2) has notoriously contained some unreadable esacping, e.g. `\\\\<tag attr=\\"value\\"\\\\>`. The escaping has been fixed and oo2po now understands helpcontent2 escaping while leaving the current GUI escape handling unaltered.

If you have not translated helpcontent2 then you are unaffected by this change. If you have translated this content then you will need to follow these instructions when upgrading.

If you follow normal procedures of creating POT files and upgrading your PO files using pot2po then your strings will not match and you will obtain files with many fuzzies. To avoid this do the following:

1. Make sure your PO files contain no fuzzy entries
2. Use po2oo from the previous release to create and SDF file

3. Upgrade to the latest Translate Toolkit with new po2oo
4. Use `po2oo -l xx-YY your.sdf po` to create a new set of PO files with correct escaping

You can choose to do this with only your helpcontent2 PO files if needed, this will allow you to leave your GUI work in its current state. Simply do the above procedure and discard all PO files except helpcontent2, then move these new helpcontent2 files into your current work.

prop2po uses developer comments

prop2po used to place comments found in the source .properties file in traditional translator comments, they should of course go into developer comments. The reason for this change is twofold, it allows these comments to be correctly managed and it is part of the process of cleaning up these formats so that they are closer to the base class and can thus work with XLIFF.

For the user there will be fairly large changes as one comment format moves to the next. It is best to *cleanup translator comments* and get your translations into a fit state, i.e. no fuzzies, and then proceed with any migrations.

moz2po no longer uses KDE comments

moz2po has traditionally used KDE style comments for storing comments aimed at translators. Many translators confuse these and try to translate them. Thus these have been moved into automatic or developer comments. The result for many people migrating Mozilla PO files will be that many strings will become fuzzy, you can avoid much of this by using pot2po which should intelligently be able to match without considering the KDE comments.

The best strategy is to get your translations into a relatively good shape before migration. You can then migrate them first to a new set of POT files generated from the same source files that the translation is based on. Eliminate all fuzzies as these should only relate to the changes in layout. Then proceed to migrate to a new set of POT files. If you cannot work against the original source files then the best would be to also first eliminate fuzzy matches before proceeding to translation. Your fuzzies will include changes in layout and changes in content so proceed carefully.

At the end of this you should have PO files that conform to the Gettext standard without KDE comments.

Read and Write MO files

You can read and write Gettext MO files (compiled PO files). Thus pocount can now count files on your filesystem and you can also compile MO files using pocompile. MO files can be compiled from either PO or XLIFF sources.

MO will now also produce correct output for msgctxt and plural forms found in PO files.

Read Qt .qm files

We can now read Qt .qm files, thus pocount can count the contents of compiled files. We cannot however write .qm files at this time.

Translate Toolkit 1.0.1

Released on 23 June 2007

pot2po will create new empty PO files if needed

From version 1.0.1, pot2po will create empty PO files corresponding to new POT files that might have been introduced. If some new POT files are present in the input to pot2po, you will see a new PO file appear in your output directory that was not in your old PO files. You will not lose any data but in the worst case you will see new files on projects that you thought were fully translated.

Translate Toolkit 1.0

Released on 1 June 2007

Improved XLIFF support

Many toolkit tools that only worked with PO files before, can now also work with XLIFF files. pogrep, pocount, pomerge, and pofilter all work with XLIFF, for example.

Pretty XML output

All XML formats should now be more human readable, and the converters to Qt .ts files should work correctly again.

Fuzzy matching in pot2po is optional

Fuzzy matching can now be entirely disabled in *pot2po* with the `--nofuzzymatching` parameter. This should make it much faster, although pot2po is **substantially** faster than earlier versions, especially if *python-Levenshtein* is installed.

Old match/Levenshtein.py* can cause name clash

The file previously called match/Levenshtein.py was renamed to lshtein.py in order to use the python-Levenshtein package mentioned above. If you follow the basic installation instructions, the old file will not be overwritten, and can cause problems. Ensure that you remove all files starting with Levenshtein.py in the installation path of the translate toolkit, usually something like /usr/lib/python2.4/site-packages/translate/search/. It could be up to three files.

PO file layout now follows Gettext more closely

The toolkits output PO format should now resemble Gettext PO files more closely. Long lines are wrapped correctly, messages with long initial lines will start with a 'msgid ""' entry. The reason for this change is to ensure that differences in files relate to content change not format change, no matter what tool you use.

To understand the problem more clearly. If a user creates POT files with e.g. *oo2po*. She then edits them in a PO editor or manipulate them with the Gettext tools. The layout of the file after manipulation was often different from the original produced by the Toolkit. Thus making it hard to tell what where content changes as opposed to layout changes.

The changes will affect you as follows:

1. They will only impact you when using the Toolkit tools.
2. You manipulate your files with a tool that follows Gettext PO layout

- your experience should now improve as the new PO files will align with your existing files
- updates should now only include real content changes not layout changes

3. You manipulate your files using Toolkit related tools or manual editing
   - your files will go through a re-layout the first time you use any of the tools
   - subsequent usage should continue as normal
   - any manipulation using Gettext tools will leave your files correctly laid out.

Our suggestion is that if you are about to suffer a major reflow that your initial merge contain only reflow and update changes. Do content changes in subsequent steps. Once you have gone through the reflow you should see no layout changes and only content changes.

Language awareness

The toolkit is gradually becoming more aware of the differences between languages. Currently this mostly affects pofilter checks (and therefore also Pootle) where tests involving punctuation and capitalisation will be more aware of the differences between English and some other languages. Provisional customisation for the following languages are in place and we will welcome more work on the language module: Amharic, Arabic, Greek, Persian, French, Armenian, Japanese, Khmer, Vietnamese, all types of Chinese.

New pofilter tests: newlines and tabs

The escapes test has been refined with two new tests, `newlines` and `tabs`. This makes identifying the errors easier and makes it easier to control the results of the tests. You shouldn't have to change your testing behaviour in any way.

Merging can change fuzzy status

pomerge now handles fuzzy states:

```
pomerge -t old -i merge -o new
```

Messages that are fuzzy in *merge* will now also be fuzzy in *new*. Similarly if a fuzzy state is present in *old* but removed in *merge* then the message in *new* will not be fuzzy.

Previously no fuzzy states were changed during a merge.

pofilter will make Mozilla accelerators a serious failure

If you use *pofilter* with the `--mozilla` option then accelerator failures will produce a serious filter error, i.e. the message will be marked as `fuzzy`. This has been done because accelerator problems in your translations have the potential to break Mozilla applications.

po2prop can output Mozilla or Java style properties

We have added the `--personality` option to allow a user to select output in either `java`, or `mozilla` style (Java property files use escaped Unicode, while Mozilla uses actual Unicode characters). This functionality was always available but was not exposed to the user and we always defaulted to the Mozilla style.

When using *po2moz* the behaviour is not changed for the user as the programs will ensure that the properties convertor uses Mozilla style.

However, when using *po2prop* the default style is now `java`, thus if you are converting a single `.properties` file as part of a Mozilla conversion you will need to add `--personality=mozilla` to your conversion. Thus:

```
po2prop -t moz.properties moz.properties.po my-moz.properties
```

Would become:

```
po2prop --personality=mozilla -t moz.properties moz.properties.po my-moz.properties
```

Note: Output in java style escaped Unicode will still be usable by Mozilla but will be harder to read.

Support for compressed files

There is some initial support for reading from and writing to compressed files. Single files compressed with gzip or bzip2 compression is supported, but not tarballs. Most tools don't support it, but pocount and the `--tm` parameter to pot2po will work with it, for example. Naturally it is slower than working with uncompressed files. Hopefully more tools can support it in future.

Translate Toolkit 0.11

Released on 24 March 2007

po2oo defaults to not check for errors

In po2oo we made the default `--filteraction=none` i.e. do nothing and don't warn. Until we have a way of clearly marking false positives we'll have to disable this functionality as there is no way to quiet the output or mark non errors. Also renamed exclude to exclude-all so that it is clearer what it does i.e. it excludes 'all' vs excludes 'serious'.

pofilter xmltags produces less false positives

In the xmltags check we handle the case where we had some false positives. E.g. "<Error>" which looks like XML/HTML but should actually be translated. These are handled by

1. identifying them as being the same length as the source text,
2. not containing any '=' sign. Thus the following would not be detected by this hack. "An <Error> occurred" -> "<Error name="bob">", but these ones need human eyes anyway.

Translate Toolkit 0.10.1

Released on 28 December 2006

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 0.10

Released on 29 August 2006

PO to XLIFF conversion

Conversion from PO to XLIFF is greatly improved in 0.10 and this was done according to the specification at http://xliff-tools.freedesktop.org/wiki/Projects/XliffPoGuide – please let us know if there are features lacking.

pot2po can replace msgmerge

pot2po has undergone major changes which means that it now respects your header entries, can resurrect obsolete messages, does fuzzy matching using *Levenshtein distance* algorithm, will correctly match messages with KDE style comments and can use an external Translation Memory. You can now use pot2po instead of Gettext's msgmerge and it can also replace *pomigrate2*. You may still want to use pomigrate2 if there where file movements between versions as pot2po can still not do intelligent matching of PO and POT files, pomigrate2 has also been adapted so that it can use pot2po as it background merging tool.

```
pomigrate2 --use-compendium --pot2po <old> <pot> <new>
```

This will migrate file with a compendium built from PO files in <*old*> and will use pot2po as its conversion engine.

.properties pretty formatting

When using templates for generating translated .properties files we will now preserve the formatting around the equal sign.

```
# Previously if the template had
property     =      value
```

```
# We output
property=translation
```

```
# We will now output
property     =      translation
```

This change ensures that there is less noise when checking differences against the template file. However, there will be quite a bit of noise when you make your first .properties commits with the new pretty layout. Our suggestion is that you make a single commit of .properties files without changes of translations to gt the formatting correct.

Translate Toolkit 0.9.2

Released on 11 August 2006

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 0.9.1

Released on 17 July 2006

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 0.9

Released on 15 June 2006

Escaping – DTD files are no longer escaped

Previously each converter handled escaping, which made it a nightmare every time we identified an escaping related error or added a new format. Escaping has now been moved into the format classes as much as possible, the result being that formats exchange Python strings and manage their own escaping.

I doing this migration we revisited some of the format migration. We found that we were escaping elements in our output DTD files. DTD's should have no escaping i.e. `\n` is a literal `\` followed by an `n` not a newline.

A result of this change is that older PO files will have different escaping to what po2moz will now expect. Probably resulting in bad output .dtd files.

We did not make this backward compatible as the fix is relatively simple and is one you would have done for any migration of your PO files.

1. Create a new set of POT files

```
moz2po -P mozilla pot
```

2. Migrate your old PO files

```
pomigrate2 old new pot
```

3. Fix all the fuzzy translations by editing your PO files
4. Use pofilter to check for escaping problems and fix them

```
pofilter -t escapes new new-check
```

5. Edit file in new-check in your PO editor

```
pomerge -t new -i new-check -o new-check
```

Migration to base class

All filters are/have been migrate to a base class. This move is so that it is easier to add new format, interchange formats and to create converters. Thus xx2po and xx2xlf become easier to create. Also adding a new format should be as simple as working towards the API exposed in the base class. An unexpected side effect will be the Pootle should be able to work directly with any base class file (although that will not be the normal Pootle operation)

We have checks in place to ensure the current operation remains correct. However, nothing is perfect and unfortunately the only way to really expose all bugs is to release this software.

If you discover a bug please report it on Bugzilla or on the Pootle mailing list. If you have the skills please check on HEAD to see if it is not already fixed and if you regard it as critical discuss on the mailing list backporting the fix

(note some fixes will not be backported because they may be too invasive for the stable branch). If you are a developer please write a test to expose the bug and a fix if possible.

Duplicate Merging in PO files – merge now the default

We added the `--duplicatestyle` option to allow duplicate messages to be merged, commented or simply appear in the PO unmerged. Initially we used the msgid_comments options as the default. This adds a KDE style comment to all affected messages which created a good balance allowing users to see duplicates in the PO file but still create a valid PO file.

'msgid_comments' was the default for 0.8 (FIXME check), however it seemed to create more confusion then it solved. Thus we have reverted to using 'merge' as the default (this then completely mimics Gettext behaviour).

As Gettext will soon introduce the msgctxt attribute we may revert to using that to manage disambiguation messages instead of KDE comments. This we feel will put us back at a good balance of usefulness and usability. We will only release this when msgctxt version of the Gettext tools are released.

.properties files no longer use escaped Unicode

The main use of the .properties converter class is to translate Mozilla files, although .properties files are actually a Java standard. The old Mozilla way, and still the Java way, of working with .properties files is to escape any Unicode characters using the `\uNNNN` convention. Mozilla now allows you to use Unicode in UTF-8 encoding for these files. Thus in 0.9 of the Toolkit we now output UTF-8 encoded properties files. Issue 193 tracks the status of this and we hope to add a feature to prop2po to restore the correct Java convention as an option.

Translate Toolkit 0.8

Released on 20 February 2006

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

Translate Toolkit 0.7.1

Released on 24 April 2004

Release notes for this release are still to be recovered, feel free to do some way back searching and create a Pull Request for this release.

3.2 History of the Translate Toolkit

This is a short history of the Translate Toolkit. In many ways written so that people who see problems in the toolkit can understand how it evolved and where it is going.

3.2.1 Conception

The toolkit was developed by David Fraser while working for Translate.org.za. Initially Translate.org.za had focussed on translating KDE into South Africa languages, this work was PO based. The next project was to translate Mozilla which used a combination of DTD and .properties files. The Mozilla project used a tool called Mozilla Translator, which mostly worked although it was not as feature rich as KBabel that was being used to manage PO translations. A

decision was made to create a set of tools that could convert the DTD and .properties files into PO files. The advantage being that translators would not need to learn new tools, that existing translations could be leveraged and that the resultant files, being bilingual, would make it easier to upgrade and manage translations.

Thus was born what initially was called the mozpotools.

3.2.2 Growth

The first problem with the tools was that it was possible to break Mozilla translations. This was a combination of the fact that translators would often translate variables such as &browserName; and that the toolkit had developed a method of folding labels and accelerators into one PO field. These breakages where presented as broken XML. Thus was born pofilter which allowed us to check the translations for problems in variables and accelerators. pomerge its sister allowed us to merge the corrections back into the main. We also developed pocount which allowed us to for the first time get a real feel of the volume of work required in translating a PO file.

3.2.3 Expansion

Of course once you can convert the convoluted Mozilla translations then you realise you can do anything. A key addition was the converter for OpenOffice.org but also added where TMX, Qt .ts, txt and OpenOffice.org SXW files.

The key being that files are converted to PO to allow translations and use of the Gettext tools and existing PO files.

3.2.4 Pootle

Initially started as a separate project to allow online translation it was soon realised that the toolkit being file based gave all the infrastructure to allow Pootle to be a wrapper around the toolkit. So a file based, web translation tool was created.

3.2.5 WordForge project

In 2006 with funding from the Open Society Institute (OSI) and IDRC the toolkit was adapted to allow many core changes. The first being to introduce the concept of a base class from which PO and XLIFF storage formats are derived. This allowed tools to be adapted to allow output to XLIFF or PO files. The tools themselves where adapted to allow them to work with the core formats XLIFF and PO as well as all base class derived formats. Thus we can count XLIFF, PO, MO and other formats.

Additional contributions during this phase where the adaptation of Pootle to use XLIFF as well as PO. The creation of tools to manage translation memory files and glossary files.

The toolkit was also adapted to make dealing with encodings, plural forms, and escaping easier and more consistent throughout the code. Many but not all of the formats where converted to the base class.

As part of the WordForge project Pootling was created which in the same way that Pootle is a web-based wrapper around the toolkit so Pootling is a GUI wrapper around the toolkit.

3.2.6 ANLoc project

The African Network for Localisation provided the opportunity for further improvements to the project. We saw the first official releases of Virtaal and massive improvements to all the translation tools.

Format support improved a lot, with several bilingual file formats now support (Wordfast TM, Qt TS, etc.), and several monolingual file formats (PHP arrays, video subtitles, Mac OS X strings, etc.).

3.2.7 The Future

The toolkit continues to evolve with clean-up focused in various areas:

- Pulling features out of Pootle that should be in the Toolkit
- Cleaning up storage classes and converters to be XLIFF/PO interchangeable
- Cleaning up the converters to use only base class features and migrating code from the converters to the storage class
- Adding storage classes as needed
- Optimisation where needed

The toolkit continues to serve as the core for the command line tools and for Pootle. Key new features:

- Process Management

3.3 License

CHAPTER 4

API Reference

This part covers any function, class or method included within the Translate Toolkit that you can use to programatically build new localization tools.

4.1 API

The Translate Toolkit provides several modules for programmers to build their own tools.

4.1.1 Module overview

The following will give you an idea about what each module is capable of.

convert

Code to convert between different storage formats for localizations.

filters

Filters that can be used on translations...

lang

Classes that represent languages and provides language-specific information.

All classes inherit from the parent class called `common`.

The type of data includes:

- Language codes
- Language name

- Plurals
- Punctuation transformation
- etc.

misc

Miscellaneous modules for translate - including modules for backward compatibility with pre-2.3 versions of Python

search

Services for searching and matching of text.

services

translate.services is part of the translate toolkit. It provides network services for interacting with the toolkit

storage

Classes that represent various storage formats for localization.

tools

Code to perform various operations, mostly on po files.

4.1.2 Module list

All the modules included in the Translated Toolkit are listed here.

convert

Code to convert between different storage formats for localizations.

acesskey

functions used to manipulate access keys in strings

class translate.convert.accesskey.**UnitMixer**(*labelsuffixes*, *accesskeysuffixes*)
Helper to mix separately defined labels and accesskeys into one unit.

match_entities(*index*)
Populates mixedentities from the index.

mix_units(*label_unit*, *accesskey_unit*, *target_unit*)
Mix the given units into the given target_unit if possible.

Might return None if no match is possible.

translate.convert.accesskey.**combine**(*label*, *accesskey*, *accesskey_marker='&'*)
Combine a label and and accesskey to form a label+accesskey string

We place an accesskey marker before the accesskey in the label and this creates a string with the two combined e.g. "File" + "F" = "&File"

The case of the accesskey is preferred unless no match is found, in which case the alternate case is used.

Parameters

- **label** (*unicode*) – a label
- **accesskey** (*unicode char*) – The accesskey

Return type unicode or None

Returns label+accesskey string or None if uncombineable

translate.convert.accesskey.**extract**(*string*, *accesskey_marker='&'*)
Extract the label and accesskey from a label+accesskey string

The function will also try to ignore &entities; which would obviously not contain accesskeys.

Parameters

- **string** (*Unicode*) – A string that might contain a label with accesskey marker
- **accesskey_marker** (*Char*) – The character that is used to prefix an access key

convert

Handles converting of files between formats (used by *translate.convert* tools).

class translate.convert.convert.**ArchiveConvertOptionParser**(*formats*, *usetemplates=False*, *usepots=False*, *description=None*, *archiveformats=None*)
ConvertOptionParser that can handle recursing into single archive files.

archiveformats maps extension to class. If the extension doesn't matter, it can be None.

If the extension is only valid for input/output/template, it can be given as (extension, filepurpose).

add_duplicates_option(*default='msgctxt'*)
Adds an option to say what to do with duplicate strings.

add_fuzzy_option(*default=False*)
Adds an option to include / exclude fuzzy translations.

add_multifile_option(*default='single'*)
Adds an option to say how to split the po/pot files.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

add_remove_untranslated_option(*default=False*)
Adds an option to remove key value from output if it is untranslated.

add_threshold_option(*default=None*)
Adds an option to output only stores where translation percentage exceeds the threshold.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under `options.output` needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

filterinputformats(*options*)
Filters input formats, processing relevant switches in options.

filteroutputoptions(*options*)
Filters output options, processing relevant switches in options.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getarchiveclass(*fileext*, *filepurpose*, *isdir=False*)
Returns the archiveclass for the given fileext and filepurpose

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions (*options*)
Get the options required to pass to the filtermethod...

gettemplatename (*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman (*option*)
returns the usage string for the given option

getusagestring (*option*)
returns the usage string for the given option

isarchive (*fileoption*, *filepurpose='input'*)
Returns whether the file option is an archive file.

isexcluded (*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive (*fileoption*, *filepurpose='input'*)
Checks if **fileoption** is a recursive file.

isvalidinputname (*inputname*)
Checks if this is a valid input filename.

mkdir (*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openarchive (*archivefilename*, *filepurpose*, ***kwargs*)
Creates an archive object for the given file.

openinputfile (*options*, *fullinputpath*)
Opens the input file.

openoutputfile (*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile (*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile (*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args (*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

potifyformat (*fileformat*)
Converts a .po to a .pot where required.

print_help (*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage (*file=None*)
outputs a manpage for the program using the help information

print_usage (*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version (*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(),

any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Run an invidividual conversion.

recursearchivefiles(*options*)
Recurse through archive files and convert files.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through archive file / directories and return files to be converted.

recursiveprocess(*options*)
Recurse through directories and convert files.

run(*argv=None*)
Parses the command line options and runs the conversion.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setpotoption()
Sets the `-P`/`--pot` option depending on input/output formats etc.

setprogressoptions()
Sets the progress options.

settimestampoption()
Sets `-S`/`--timestamp` option.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

verifyoptions(*options*)
Verifies that the options are valid (required options are present, etc).

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

class translate.convert.convert.**ConvertOptionParser**(*formats*, *usetemplates=False*, *usepots=False*, *allowmissingtemplate=False*, *description=None*)
A specialized Option Parser for convertor tools...

add_duplicates_option(*default='msgctxt'*)
Adds an option to say what to do with duplicate strings.

add_fuzzy_option(*default=False*)
Adds an option to include / exclude fuzzy translations.

add_multifile_option(*default='single'*)
Adds an option to say how to split the po/pot files.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

add_remove_untranslated_option(*default=False*)
Adds an option to remove key value from output if it is untranslated.

add_threshold_option(*default=None*)
Adds an option to output only stores where translation percentage exceeds the threshold.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

filterinputformats(*options*)
Filters input formats, processing relevant switches in options.

filteroutputoptions(*options*)
Filters output options, processing relevant switches in options.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

potifyformat(*fileformat*)
Converts a .po to a .pot where required.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Process an individual file.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
Recurse through directories and process files.

run(*argv=None*)
Parses the command line options and runs the conversion.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setpotoption()
Sets the -P/--pot option depending on input/output formats etc.

setprogressoptions()
Sets the progress options.

settimestampoption()
Sets -S/--timestamp option.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

verifyoptions(*options*)
Verifies that the options are valid (required options are present, etc).

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

class translate.convert.convert.**Replacer**(*searchstring*, *replacestring*)
An object that knows how to replace strings in files.

doreplace(*text*)
actually replace the text

searchreplaceinput(*inputfile*, *outputfile*, *templatefile*, ***kwargs*)
copies the input file to the output file, searching and replacing

searchreplacetemplate(*inputfile*, *outputfile*, *templatefile*, ***kwargs*)
Copies the template file to the output file, searching and replacing.

translate.convert.convert.**copyinput**(*inputfile*, *outputfile*, *templatefile*, ***kwargs*)
Copies the input file to the output file.

translate.convert.convert.**copytemplate**(*inputfile*, *outputfile*, *templatefile*, ***kwargs*)
Copies the template file to the output file.

translate.convert.convert.**should_output_store**(*store*, *threshold*)
Check if the percent of translated source words more than or equal to the given threshold.

csv2po

Convert Comma-Separated Value (.csv) files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/csv2po.html for examples and usage instructions.

translate.convert.csv2po.**convertcsv**(*inputfile*, *outputfile*, *templatefile*, *charset=None*, *columnorder=None*, *duplicatestyle='msgctxt'*)
reads in inputfile using csvl10n, converts using csv2po, writes to outputfile

class translate.convert.csv2po.**csv2po**(*templatepo=None*, *charset=None*, *duplicatestyle='keep'*)
a class that takes translations from a .csv file and puts them in a .po file

convertstore(*thecsvfile*)
converts a csvfile to a pofile, and returns it. uses templatepo if given at construction

convertunit(*csvunit*)
converts csv unit to po unit

handlecsvunit(*csvunit*)
handles reintegrating a csv unit into the .po file

makeindex()
makes indexes required for searching...

translate.convert.csv2po.**replacestrings**(*source*, **pairs*)
Use `pairs` of `(original, replacement)` to replace text found in `source`.

Parameters

- **source** (*String*) – String to on which `pairs` of strings are to be replaced
- ***pairs** (*One or more tuples of (original, replacement)*) – Strings to be matched and replaced

Returns String with `*pairs` of strings replaced

csv2tbx

Convert Comma-Separated Value (.csv) files to a TermBase eXchange (.tbx) glossary file

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/csv2tbx.html for examples and usage instructions

translate.convert.csv2tbx.**convertcsv**(*inputfile*, *outputfile*, *templatefile*, *charset=None*, *columnorder=None*)
reads in inputfile using csvl10n, converts using csv2tbx, writes to outputfile

class translate.convert.csv2tbx.**csv2tbx**(*charset=None*)
a class that takes translations from a .csv file and puts them in a .tbx file

convertfile(*csvfile*)
converts a csvfile to a tbxfile, and returns it. uses templatepo if given at construction

dtd2po

Convert a Mozilla .dtd UTF-8 localization format to a Gettext PO localization file.

Uses the po and dtd modules, and the dtd2po convertor class which is in this module You can convert back to .dtd using po2dtd.py.

translate.convert.dtd2po.**convertdtd**(*inputfile*, *outputfile*, *templatefile*, *pot=False*, *duplicatestyle='msgctxt'*)
reads in inputfile and templatefile using dtd, converts using dtd2po, writes to outputfile

translate.convert.dtd2po.**is_css_entity**(*entity*)
Says if the given entity is likely to contain CSS that should not be translated.

factory

Factory methods to convert supported input files to supported translatable files.

exception translate.convert.factory.**UnknownExtensionError**(*afile*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception translate.convert.factory.**UnsupportedConversionError**(*in_ext=None*, *out_ext=None*, *templ_ext=None*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

html2po

Convert HTML files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/html2po.html for examples and usage instructions.

translate.convert.html2po.**converthtml**(*inputfile*, *outputfile*, *templates*, *includeuntagged=False*, *pot=False*, *duplicatestyle='msgctxt'*, *keepcomments=False*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

ical2po

Convert iCalendar files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ical2po.html for examples and usage instructions.

class translate.convert.ical2po.**ical2po**(*input_file*, *output_file*, *template_file=None*, *blank_msgstr=False*, *duplicate_style='msgctxt'*)
Convert one or two iCalendar files to a single PO file.

SourceStoreClass
alias of *translate.storage.ical.icalfile*

TargetStoreClass
alias of *translate.storage.pypo.pofile*

TargetUnitClass
alias of *translate.storage.pypo.pounit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

merge_stores()
Convert two source format files to a target format file.

run()
Run the converter.

translate.convert.ical2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*)
Wrapper around converter.

ini2po

Convert .ini files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ini2po.html for examples and usage instructions.

class translate.convert.ini2po.**ini2po**(*input_file*, *output_file*, *template_file=None*, *blank_msgstr=False*, *duplicate_style='msgctxt'*, *dialect='default'*)
Convert one or two INI files to a single PO file.

SourceStoreClass
alias of *translate.storage.ini.inifile*

TargetStoreClass
alias of *translate.storage.pypo.pofile*

TargetUnitClass
alias of *translate.storage.pypo.pounit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

merge_stores()
Convert two source format files to a target format file.

run()
Run the converter.

translate.convert.ini2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*, *dialect='default'*)
Wrapper around converter.

json2po

Convert JSON files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/json2po.html for examples and usage instructions.

translate.convert.json2po.**convertjson**(*input_file*, *output_file*, *template_file*, *pot=False*, *duplicatestyle='msgctxt'*, *dialect='default'*, *filter=None*)
Reads in *input_file* using jsonl10n, converts using *json2po*, writes to *output_file*.

class translate.convert.json2po.**json2po**
Convert a JSON file to a PO file

convert_store(*input_store*, *duplicatestyle='msgctxt'*)
Converts a JSON file to a PO file

convert_unit(*input_unit*, *commenttype*)
Converts a JSON unit to a PO unit

Returns None if empty or not for translation

merge_store(*template_store*, *input_store*, *blankmsgstr=False*, *duplicatestyle='msgctxt'*)
Converts two JSON files to a PO file

moz2po

Convert Mozilla .dtd and .properties files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/moz2po.html for examples and usage instructions.

mozfunny2prop

Converts additional Mozilla files to properties files.

translate.convert.mozfunny2prop.**inc2po**(*inputfile*, *outputfile*, *templatefile*, *encoding=None*, *pot=False*, *duplicatestyle='msgctxt'*)
wraps prop2po but converts input/template files to properties first

translate.convert.mozfunny2prop.**inc2prop**(*lines*)
convert a .inc file with #defines in it to a properties file

translate.convert.mozfunny2prop.**it2po**(*inputfile*, *outputfile*, *templatefile*, *encoding='cp1252'*, *pot=False*, *duplicatestyle='msgctxt'*)
wraps prop2po but converts input/template files to properties first

translate.convert.mozfunny2prop.**it2prop**(*lines*, *encoding='cp1252'*)
convert a pseudo-properties .it file to a conventional properties file

mozlang2po

Convert Mozilla .lang files to Gettext PO localization files.

class translate.convert.mozlang2po.**lang2po**(*input_file*, *output_file*, *template_file=None*, *blank_msgstr=False*, *duplicate_style='msgctxt'*, *encoding='utf-8'*)
Convert one Mozilla .lang file to a single PO file.

SourceStoreClass
alias of *translate.storage.mozilla_lang.LangStore*

TargetStoreClass
alias of *translate.storage.pypo.pofile*

TargetUnitClass
alias of *translate.storage.pypo.pounit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

merge_stores()
Convert two source format files to a target format file.

run()
Run the converter.

translate.convert.mozlang2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*, *encoding='utf-8'*)
Wrapper around converter.

odf2xliff

Convert OpenDocument (ODF) files to XLIFF localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/odf2xliff.html for examples and usage instructions.

translate.convert.odf2xliff.**convertodf**(*inputfile*, *outputfile*, *templates*)
Convert an ODF package to XLIFF.

oo2po

Convert an OpenOffice.org (SDF) localization file to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/oo2po.html for examples and usage instructions.

translate.convert.oo2po.**convertoo**(*inputfile*, *outputfile*, *templates*, *pot=False*, *sourcelanguage=None*, *targetlanguage=None*, *duplicatestyle='msgid_comment'*, *multifilestyle='single'*)
reads in stdin using inputstore class, converts using convertorclass, writes to stdout

translate.convert.oo2po.**verifyoptions**(*options*)
verifies the commandline options

oo2xliff

Convert an OpenOffice.org (SDF) localization file to XLIFF localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/oo2po.html for examples and usage instructions.

translate.convert.oo2xliff.**convertoo**(*inputfile*, *outputfile*, *templates*, *pot=False*, *sourcelanguage=None*, *targetlanguage=None*, *duplicatestyle='msgctxt'*, *multifilestyle='single'*)
reads in stdin using inputstore class, converts using convertorclass, writes to stdout

translate.convert.oo2xliff.**verifyoptions**(*options*)
verifies the commandline options

php2po

Convert PHP localization files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/php2po.html for examples and usage instructions.

class translate.convert.php2po.**php2po**(*input_file*, *output_file*, *template_file=None*, *blank_msgstr=False*, *duplicate_style='msgctxt'*)

Convert one or two PHP files to a single PO file.

SourceStoreClass
alias of *translate.storage.php.phpfile*

TargetStoreClass
alias of *translate.storage.pypo.pofile*

TargetUnitClass
alias of *translate.storage.pypo.pounit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

merge_stores()
Convert two source format files to a target format file.

run()
Run the converter.

translate.convert.php2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*)

Wrapper around converter.

po2csv

Convert Gettext PO localization files to Comma-Separated Value (.csv) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/csv2po.html for examples and usage instructions.

translate.convert.po2csv.**convertcsv**(*inputfile*, *outputfile*, *templatefile*, *columnorder=None*)
reads in inputfile using po, converts using po2csv, writes to outputfile

po2dtd

Converts a Gettext PO file to a UTF-8 encoded Mozilla .dtd file.

translate.convert.po2dtd.**applytranslation**(*entity*, *dtdunit*, *inputunit*, *mixedentities*)
applies the translation for entity in the po unit to the dtd unit

class translate.convert.po2dtd.**po2dtd**(*android=False*, *remove_untranslated=False*)
this is a convertor class that creates a new dtd file based on a po file without a template

class translate.convert.po2dtd.**redtd**(*dtdfile*, *android=False*, *remove_untranslated=False*)
this is a convertor class that creates a new dtd based on a template using translations in a po

po2html

Convert Gettext PO localization files to HTML files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/html2po.html for examples and usage instructions.

translate.convert.po2html.**converthtml**(*inputfile*, *outputfile*, *templatefile*, *includefuzzy=False*, *outputthreshold=None*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

class translate.convert.po2html.**po2html**
po2html can take a po file and generate html. best to give it a template file otherwise will just concat msgstrs

mergestore(*inputstore*, *templatetext*, *includefuzzy*)
converts a file to .po format

po2ical

Convert Gettext PO localization files to iCalendar files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ical2po.html for examples and usage instructions.

class translate.convert.po2ical.**po2ical**(*input_file*, *output_file*, *template_file=None*, *include_fuzzy=False*, *output_threshold=None*)
Convert a PO file and a template iCalendar file to a iCalendar file.

SourceStoreClass
alias of *translate.storage.pypo.pofile*

TargetStoreClass
alias of *translate.storage.ical.icalfile*

TargetUnitClass
alias of *translate.storage.ical.icalunit*

merge_stores()
Convert a source file to a target file using a template file.

Source file is in source format, while target and template files use target format.

run()
Run the converter.

translate.convert.po2ical.**run_converter**(*inputfile*, *outputfile*, *templatefile=None*, *includefuzzy=False*, *outputthreshold=None*)
Wrapper around converter.

po2ini

Convert Gettext PO localization files to .ini files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ini2po.html for examples and usage instructions.

class translate.convert.po2ini.**po2ini**(*input_file*, *output_file*, *template_file=None*, *include_fuzzy=False*, *output_threshold=None*, *dialect='default'*)
Convert a PO file and a template INI file to a INI file.

SourceStoreClass
alias of *translate.storage.pypo.pofile*

TargetStoreClass
alias of *translate.storage.ini.inifile*

TargetUnitClass
alias of *translate.storage.ini.iniunit*

merge_stores()
Convert a source file to a target file using a template file.

Source file is in source format, while target and template files use target format.

run()
Run the converter.

translate.convert.po2ini.**run_converter**(*inputfile*, *outputfile*, *templatefile=None*, *includefuzzy=False*, *dialect='default'*, *outputthreshold=None*)
Wrapper around converter.

po2json

Convert Gettext PO localization files to JSON files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/json2po.html for examples and usage instructions.

po2mozlang

Convert Gettext PO localization files to Mozilla .lang files.

class translate.convert.po2mozlang.**po2lang**(*input_file*, *output_file*, *template_file=None*, *include_fuzzy=False*, *output_threshold=None*, *mark_active=True*)
Convert a PO file to a Mozilla .lang file.

SourceStoreClass
alias of *translate.storage.pypo.pofile*

TargetStoreClass
alias of *translate.storage.mozilla_lang.LangStore*

TargetUnitClass
alias of *translate.storage.mozilla_lang.LangUnit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

run()
Run the converter.

translate.convert.po2mozlang.**run_converter**(*inputfile*, *outputfile*, *templatefile=None*, *includefuzzy=False*, *mark_active=True*, *outputthreshold=None*)
Wrapper around converter.

po2moz

Convert Gettext PO localization files to Mozilla .dtd and .properties files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/moz2po.html for examples and usage instructions.

class translate.convert.po2moz.**MozConvertOptionParser**(*formats*, *usetemplates=False*, *usepots=False*, *description=None*)

add_duplicates_option(*default='msgctxt'*)
Adds an option to say what to do with duplicate strings.

add_fuzzy_option(*default=False*)
Adds an option to include / exclude fuzzy translations.

add_multifile_option(*default='single'*)
Adds an option to say how to split the po/pot files.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

add_remove_untranslated_option(*default=False*)
Adds an option to remove key value from output if it is untranslated.

add_threshold_option(*default=None*)
Adds an option to output only stores where translation percentage exceeds the threshold.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

filterinputformats(*options*)
Filters input formats, processing relevant switches in options.

filteroutputoptions(*options*)
Filters output options, processing relevant switches in options.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

potifyformat(*fileformat*)
Converts a .po to a .pot where required.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Process an individual file.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
recurse through directories and convert files

run(*argv=None*)
Parses the command line options and runs the conversion.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setpotoption()
Sets the `-P`/`--pot` option depending on input/output formats etc.

setprogressoptions()
Sets the progress options.

settimestampoption()
Sets `-S`/`--timestamp` option.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
splits a inputpath into name and extension

Special adaptation to handle po2moz case where extensions are e.g. properties.po

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

verifyoptions(*options*)
Verifies that the options are valid (required options are present, etc).

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

po2oo

Convert Gettext PO localization files to an OpenOffice.org (SDF) localization file.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/oo2po.html for examples and usage instructions.

po2php

Convert Gettext PO localization files to PHP localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/php2po.html for examples and usage instructions.

po2prop

Convert Gettext PO localization files to Java/Mozilla .properties files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/prop2po.html for examples and usage instructions.

translate.convert.po2prop.**applytranslation**(*key*, *propunit*, *inunit*, *mixedkeys*)
applies the translation for key in the po unit to the prop unit

translate.convert.po2prop.**convertmozillaprop**(*inputfile*, *outputfile*, *templatefile*, *includefuzzy=False*, *remove_untranslated=False*, *outputthreshold=None*)

Mozilla specific convertor function

translate.convert.po2prop.**convertstrings**(*inputfile*, *outputfile*, *templatefile*, *personality='strings'*, *includefuzzy=False*, *encoding=None*, *outputthreshold=None*, *remove_untranslated=False*)

.strings specific convertor function

po2rc

Convert Gettext PO localization files back to Windows Resource (.rc) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/rc2po.html for examples and usage instructions.

translate.convert.po2rc.**is_iterable_but_not_string**(*o*)

Check if object is iterable but not a string.

po2resx

Convert Gettext PO localisation files to .Net Resource (.resx) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/resx2po.html for examples and usage instructions.

po2sub

Convert Gettext PO localization files to subtitle files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/sub2po.html for examples and usage instructions.

po2symb

Convert Gettext PO localization files to Symbian translation files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/symb2po.html for examples and usage instructions.

po2tiki

Convert Gettext PO files to TikiWiki's language.php files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/tiki2po.html for examples and usage instructions.

class translate.convert.po2tiki.**po2tiki**(*input_file*, *output_file*, *template_file=None*)

Convert a PO file and a template TikiWiki file to a TikiWiki file.

SourceStoreClass

alias of *translate.storage.pypo.pofile*

TargetStoreClass
alias of *translate.storage.tiki.TikiStore*

TargetUnitClass
alias of *translate.storage.tiki.TikiUnit*

convert_store()
Convert a single source format file to a target format file.

convert_unit(*unit*)
Convert a source format unit to a target format unit.

run()
Run the converter.

translate.convert.po2tiki.**run_converter**(*inputfile*, *outputfile*, *template=None*)
Wrapper around converter.

po2tmx

Convert Gettext PO localization files to a TMX (Translation Memory eXchange) file.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/po2tmx.html for examples and usage instructions.

class translate.convert.po2tmx.**TmxOptionParser**(*formats*, *usetemplates=False*, *usepots=False*, *description=None*, *archiveformats=None*)

add_duplicates_option(*default='msgctxt'*)
Adds an option to say what to do with duplicate strings.

add_fuzzy_option(*default=False*)
Adds an option to include / exclude fuzzy translations.

add_multifile_option(*default='single'*)
Adds an option to say how to split the po/pot files.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

add_remove_untranslated_option(*default=False*)
Adds an option to remove key value from output if it is untranslated.

add_threshold_option(*default=None*)
Adds an option to output only stores where translation percentage exceeds the threshold.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under `options.output` needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

filterinputformats(*options*)
Filters input formats, processing relevant switches in options.

filteroutputoptions(*options*)
Filters output options, processing relevant switches in options.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getarchiveclass(*fileext*, *filepurpose*, *isdir=False*)
Returns the archiveclass for the given fileext and filepurpose

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isarchive (*fileoption*, *filepurpose='input'*)
Returns whether the file option is an archive file.

isexcluded (*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive (*fileoption*, *filepurpose='input'*)
Checks if **fileoption** is a recursive file.

isvalidinputname (*inputname*)
Checks if this is a valid input filename.

mkdir (*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openarchive (*archivefilename*, *filepurpose*, ***kwargs*)
Creates an archive object for the given file.

openinputfile (*options*, *fullinputpath*)
Opens the input file.

openoutputfile (*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile (*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile (*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args (*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

potifyformat (*fileformat*)
Converts a .po to a .pot where required.

print_help (*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage (*file=None*)
outputs a manpage for the program using the help information

print_usage (*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version (*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile (*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Run an invidividual conversion.

recursearchivefiles (*options*)
Recurse through archive files and convert files.

recurseinputfilelist (*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through archive file / directories and return files to be converted.

recursiveprocess(*options*)
Recurse through directories and convert files.

run(*argv=None*)
Parses the command line options and runs the conversion.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setpotoption()
Sets the `-P`/`--pot` option depending on input/output formats etc.

setprogressoptions()
Sets the progress options.

settimestampoption()
Sets `-S`/`--timestamp` option.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

verifyoptions(*options*)
Verifies that the options are valid (required options are present, etc).

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

translate.convert.po2tmx.**convertpo**(*inputfile*, *outputfile*, *templatefile*, *sourcelanguage='en'*, *targetlanguage=None*, *comment=None*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

po2ts

Convert Gettext PO localization files to Qt Linguist (.ts) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ts2po.html for examples and usage instructions.

translate.convert.po2ts.**convertpo**(*inputfile*, *outputfile*, *templatefile*, *context*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

po2txt

Convert Gettext PO localization files to plain text (.txt) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/txt2po.html for examples and usage instructions.

class translate.convert.po2txt.**po2txt**(*input_file*, *output_file*, *template_file=None*, *include_fuzzy=False*, *output_threshold=None*, *encoding='utf-8'*, *wrap=None*)
po2txt can take a po file and generate txt.

best to give it a template file otherwise will just concat msgstrs

convert_store()
Convert a source file to a target file.

merge_stores()
Convert a source file to a target file using a template file.

Source file is in source format, while target and template files use target format.

run()
Run the converter.

wrapmessage(*message*)
rewraps text as required

translate.convert.po2txt.**run_converter**(*inputfile*, *outputfile*, *templatefile=None*, *wrap=None*, *includefuzzy=False*, *encoding='utf-8'*, *outputthreshold=None*)
Wrapper around converter.

po2web2py

Convert GNU/gettext PO files to web2py translation dictionaries (.py).

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/web2py2po.html for examples and usage instructions.

po2wordfast

Convert Gettext PO localization files to a Wordfast translation memory file.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/po2wordfast.html for examples and usage instructions.

class translate.convert.po2wordfast.**WfOptionParser**(*formats*, *usetemplates=False*, *usepots=False*, *description=None*, *archiveformats=None*)

add_duplicates_option(*default='msgctxt'*)
Adds an option to say what to do with duplicate strings.

add_fuzzy_option(*default=False*)
Adds an option to include / exclude fuzzy translations.

add_multifile_option(*default='single'*)
Adds an option to say how to split the po/pot files.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

add_remove_untranslated_option(*default=False*)
Adds an option to remove key value from output if it is untranslated.

add_threshold_option(*default=None*)
Adds an option to output only stores where translation percentage exceeds the threshold.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under `options.output` needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

filterinputformats(*options*)
Filters input formats, processing relevant switches in options.

filteroutputoptions(*options*)
Filters output options, processing relevant switches in options.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getarchiveclass(*fileext*, *filepurpose*, *isdir=False*)
Returns the archiveclass for the given fileext and filepurpose

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isarchive(*fileoption*, *filepurpose='input'*)
Returns whether the file option is an archive file.

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if **fileoption** is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openarchive(*archivefilename*, *filepurpose*, ***kwargs*)
Creates an archive object for the given file.

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

potifyformat(*fileformat*)
Converts a .po to a .pot where required.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Run an invididual conversion.

recursearchivefiles(*options*)
Recurse through archive files and convert files.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through archive file / directories and return files to be converted.

recursiveprocess(*options*)
Recurse through directories and convert files.

run(*argv=None*)
Parses the command line options and runs the conversion.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)

- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setpotoption()
Sets the `-P`/`--pot` option depending on input/output formats etc.

setprogressoptions()
Sets the progress options.

settimestampoption()
Sets `-S`/`--timestamp` option.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

verifyoptions(*options*)
Verifies that the options are valid (required options are present, etc).

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

translate.convert.po2wordfast.**convertpo**(*inputfile*, *outputfile*, *templatefile*, *sourcelanguage='en'*, *targetlanguage=None*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

po2xliff

Convert Gettext PO localization files to XLIFF localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/xliff2po.html for examples and usage instructions.

translate.convert.po2xliff.**convertpo**(*inputfile*, *outputfile*, *templatefile*)
reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

po2yaml

Convert Gettext PO localization files to YAML files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/yaml2po.html for examples and usage instructions.

class translate.convert.po2yaml.**po2yaml**(*input_file*, *output_file*, *template_file=None*, *include_fuzzy=False*, *output_threshold=None*)

Convert a PO file and a template YAML file to a YAML file.

SourceStoreClass
alias of *translate.storage.pypo.pofile*

TargetStoreClass
alias of translate.storage.yaml.YAMLFile

TargetUnitClass
alias of translate.storage.yaml.YAMLUnit

convert_unit(*unit*)
Convert a source format unit to a target format unit.

merge_stores()
Convert a source file to a target file using a template file.

Source file is in source format, while target and template files use target format.

run()
Run the converter.

translate.convert.po2yaml.**run_converter**(*inputfile*, *outputfile*, *templatefile=None*, *includefuzzy=False*, *outputthreshold=None*)

Wrapper around converter.

pot2po

Convert template files (like .pot or template .xlf files) to translation files, preserving existing translations.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pot2po.html for examples and usage instructions.

translate.convert.pot2po.**convert_stores**(*input_store*, *template_store*, *temp_store=None*, *tm=None*, *min_similarity=75*, *fuzzymatching=True*, ***kwargs*)

Actual conversion function, works on stores not files, returns a properly initialized pretranslated output store, with structure based on input_store, metadata based on template_store, migrates old translations from template_store and pretranslating from TM.

translate.convert.pot2po.**convertpot**(*input_file*, *output_file*, *template_file*, *tm=None*, *min_similarity=75*, *fuzzymatching=True*, *classes=None*, *classes_str=None*, ***kwargs*)

Main conversion function.

prop2mozfunny

Converts properties files to additional Mozilla format files.

translate.convert.prop2mozfunny.**po2inc**(*inputfile*, *outputfile*, *templatefile*, *encoding=None*, *includefuzzy=False*, *remove_untranslated=False*, *outputthreshold=None*)

wraps po2prop but converts outputfile to properties first

translate.convert.prop2mozfunny.**po2ini**(*inputfile*, *outputfile*, *templatefile*, *encoding='UTF-8'*, *includefuzzy=False*, *remove_untranslated=False*, *outputthreshold=None*)

wraps po2prop but converts outputfile to properties first using UTF-8 encoding

translate.convert.prop2mozfunny.**po2it**(*inputfile*, *outputfile*, *templatefile*, *encoding='cp1252'*, *includefuzzy=False*, *remove_untranslated=False*, *outputthreshold=None*)
wraps po2prop but converts outputfile to properties first

translate.convert.prop2mozfunny.**prop2inc**(*pf*)
convert a properties file back to a .inc file with #defines in it

translate.convert.prop2mozfunny.**prop2it**(*pf*)
convert a properties file back to a pseudo-properties .it file

prop2po

Convert Java/Mozilla .properties files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/prop2po.html for examples and usage instructions.

exception translate.convert.prop2po.**DiscardUnit**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

translate.convert.prop2po.**convertmozillaprop**(*inputfile*, *outputfile*, *templatefile*, *pot=False*, *duplicatestyle='msgctxt'*)
Mozilla specific convertor function

translate.convert.prop2po.**convertprop**(*inputfile*, *outputfile*, *templatefile*, *personality='java'*, *pot=False*, *duplicatestyle='msgctxt'*, *encoding=None*)
reads in inputfile using properties, converts using prop2po, writes to outputfile

translate.convert.prop2po.**convertstrings**(*inputfile*, *outputfile*, *templatefile*, *personality='strings'*, *pot=False*, *duplicatestyle='msgctxt'*, *encoding=None*)
.strings specific convertor function

class translate.convert.prop2po.**prop2po**(*personality='java'*, *blankmsgstr=False*, *duplicatestyle='msgctxt'*)
convert a .properties file to a .po file for handling the translation.

convertpropunit(*store*, *unit*, *commenttype*, *mixbucket='properties'*)
Converts a unit from store to a po unit, keeping track of mixed names along the way.

`mixbucket` can be specified to indicate if the given unit is part of the template or the translated file.

convertstore(*thepropfile*)
converts a .properties file to a .po file...

convertunit(*propunit*, *commenttype*)
Converts a .properties unit to a .po unit. Returns None if empty or not for translation.

fold_gaia_plurals(*postore*)
Fold the multiple plural units of a gaia file into a gettext plural.

fold_gwt_plurals(*postore*)
Fold the multiple plural units of a gwt file into a gettext plural.

mergestore(*origpropfile*, *translatedpropfile*)
converts two .properties files to a .po file...

rc2po

Convert Windows RC files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/rc2po.html for examples and usage instructions.

translate.convert.rc2po.**convertrc**(*input_file*, *output_file*, *template_file*, *pot=False*, *duplicatestyle='msgctxt'*, *charset=None*, *lang=None*, *sublang=None*)
reads in input_file using rc, converts using rc2po, writes to output_file

class translate.convert.rc2po.**rc2po**
Convert a .rc file to a .po file for handling the translation.

convert_store(*input_store*, *duplicatestyle='msgctxt'*)
converts a .rc file to a .po file...

convert_unit(*input_unit*, *commenttype*)
Converts a .rc unit to a .po unit. Returns None if empty or not for translation.

merge_store(*template_store*, *input_store*, *blankmsgstr=False*, *duplicatestyle='msgctxt'*)
converts two .rc files to a .po file...

resx2po

Convert .Net Resource (.resx) to Gettext PO localisation files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/resx2po.html for examples and usage instructions.

class translate.convert.resx2po.**resx2po**
Convert a RESX file to a PO file for handling translation

convert_store(*input_store*, *duplicatestyle='msgctxt'*)
Converts a RESX file to a PO file

convert_unit(*input_unit*, *commenttype*)
Converts a RESX unit to a PO unit @return: None if empty or not for translation

merge_store(*template_store*, *input_store*, *blankmsgstr=False*, *duplicatestyle='msgctxt'*)
Converts two RESX files to a PO file

sub2po

Convert subtitle files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/sub2po.html for examples and usage instructions.

translate.convert.sub2po.**convert_store**(*input_store*, *duplicatestyle='msgctxt'*)
converts a subtitle file to a .po file...

translate.convert.sub2po.**convert_unit**(*input_unit*, *commenttype*)
Converts a subtitle unit to a .po unit. Returns None if empty or not for translation.

translate.convert.sub2po.**convertsub**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*)
Reads in *input_file* using translate.subtitles, converts using `sub2po`, writes to *output_file*.

translate.convert.sub2po.**merge_store**(*template_store*, *input_store*, *blankmsgstr=False*, *duplicatestyle='msgctxt'*)

converts two subtitle files to a .po file...

symb2po

Convert Symbian localisation files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/symb2po.html for examples and usage instructions.

tiki2po

Convert TikiWiki's language.php files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/tiki2po.html for examples and usage instructions.

translate.convert.tiki2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *includeunused=False*)

Wrapper around converter.

class translate.convert.tiki2po.**tiki2po**(*input_file*, *output_file*, *template_file=None*, *include_unused=False*)

Convert one or two TikiWiki's language.php files to a single PO file.

SourceStoreClass

alias of *translate.storage.tiki.TikiStore*

TargetStoreClass

alias of *translate.storage.pypo.pofile*

TargetUnitClass

alias of *translate.storage.pypo.pounit*

convert_store()

Convert a single source format file to a target format file.

convert_unit(*unit*)

Convert a source format unit to a target format unit.

run()

Run the converter.

ts2po

Convert Qt Linguist (.ts) files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/ts2po.html for examples and usage instructions.

translate.convert.ts2po.**convertts**(*inputfile*, *outputfile*, *templates*, *pot=False*, *duplicatestyle='msgctxt'*)

reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

txt2po

Convert plain text (.txt) files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/txt2po.html for examples and usage instructions.

translate.convert.txt2po.**run_converter**(*input_file*, *output_file*, *template_file=None*, *duplicatestyle='msgctxt'*, *encoding='utf-8'*, *flavour=None*, *no_segmentation=False*)

Wrapper around converter.

class translate.convert.txt2po.**txt2po**(*input_file*, *output_file*, *template_file=None*, *duplicate_style='msgctxt'*, *encoding='utf-8'*, *flavour=None*, *no_segmentation=False*)

Convert one plain text (.txt) file to a single PO file.

SourceStoreClass

alias of *translate.storage.txt.TxtFile*

TargetStoreClass

alias of *translate.storage.pypo.pofile*

TargetUnitClass

alias of *translate.storage.pypo.pounit*

convert_store()

Convert a single source format file to a target format file.

merge_stores()

Convert two source format files to a target format file.

run()

Run the converter.

web2py2po

Convert web2py translation dictionaries (.py) to GNU/gettext PO files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/web2py2po.html for examples and usage instructions.

xliff2odf

Convert XLIFF translation files to OpenDocument (ODF) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/odf2xliff.html for examples and usage instructions.

translate.convert.xliff2odf.**convertxliff**(*input_file*, *output_file*, *template*)

Create a translated ODF using an ODF template and a XLIFF file.

translate.convert.xliff2odf.**write_odf**(*template*, *output_file*, *dom_trees*)

Write the translated ODF package.

The resulting ODF package is a copy of the template ODF package, with the translatable files replaced by their translated versions.

xliff2oo

Convert XLIFF localization files to an OpenOffice.org (SDF) localization file.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/oo2po.html for examples and usage instructions.

xliff2po

Convert XLIFF localization files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/xliff2po.html for examples and usage instructions.

`translate.convert.xliff2po.`**`convertxliff`**(*inputfile*, *outputfile*, *templates*, *duplicatestyle='msgctxt'*)

reads in stdin using fromfileclass, converts using convertorclass, writes to stdout

yaml2po

Convert YAML files to Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/yaml2po.html for examples and usage instructions.

`translate.convert.yaml2po.`**`run_converter`**(*input_file*, *output_file*, *template_file=None*, *pot=False*, *duplicatestyle='msgctxt'*)

Wrapper around converter.

`class` `translate.convert.yaml2po.`**`yaml2po`**(*input_file*, *output_file*, *template_file=None*, *blank_msgstr=False*, *duplicate_style='msgctxt'*)

Convert one or two YAML files to a single PO file.

`SourceStoreClass`

alias of `translate.storage.yaml.YAMLFile`

`TargetStoreClass`

alias of *translate.storage.pypo.pofile*

`TargetUnitClass`

alias of *translate.storage.pypo.pounit*

`convert_store`()

Convert a single source format file to a target format file.

`convert_unit`(*unit*)

Convert a source format unit to a target format unit.

`merge_stores`()

Convert two source format files to a target format file.

`run`()

Run the converter.

filters

Filters that can be used on translations...

autocorrect

A set of autocorrect functions that fix common punctuation and space problems automatically

translate.filters.autocorrect.**correct**(*source*, *target*)
Runs a set of easy and automatic corrections

Current corrections include:

- Ellipses - align target to use source form of ellipses (either three dots or the Unicode ellipses characters)
- Missing whitespace and start or end of the target
- Missing punction (.:?) at the end of the target

checks

This is a set of validation checks that can be performed on translation units.

Derivatives of UnitChecker (like StandardUnitChecker) check translation units, and derivatives of TranslationChecker (like StandardChecker) check (source, target) translation pairs.

When adding a new test here, please document and explain their behaviour on the *pofilter tests* page.

class translate.filters.checks.**CCLicenseChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If ([{ or }]) appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits (*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting (*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing (*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace (*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers (*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs (*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged (*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated (*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls (*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars (*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables (*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags (*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**CheckerConfig**(*targetlanguage=None*, *accelmarkers=None*, *varmatches=None*, *notranslatewords=None*, *musttranslatewords=None*, *validchars=None*, *punctuation=None*, *endpunctuation=None*, *ignoretags=None*, *canchangetags=None*, *criticaltests=None*, *credit_sources=None*)

Object representing the configuration of a checker.

update(*otherconfig*)
Combines the info in `otherconfig` into this config object.

updatetargetlanguage(*langcode*)
Updates the target language in the config to the given target language and sets its script.

updatevalidchars(*validchars*)
Updates the map that eliminates valid characters.

class translate.filters.checks.**DrupalChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `( [ {` or `} ] )` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits (*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting (*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing (*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses `[...]` in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace (*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers (*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs (*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged (*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated (*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls (*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars (*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables (*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags (*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

exception translate.filters.checks.**FilterFailure**(*messages*)

This exception signals that a Filter didn't pass, and gives an explanation or a comment.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.filters.checks.**GnomeChecker**(***kwargs*)

accelerators(*str1*, *str2*)

Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)

Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)

Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)

Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name

Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)

Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)

Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes " to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. info@example.com are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as ? or ! are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as \n \ to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as /home/user1 have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. your_user_name/path/to/filename.conf.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from str1.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. rgb() or getEntity.Name() are not translated.

gconf(*str1*, *str2*)
Checks if we have any gconf config settings translated.

Gconf settings should not be translated so this check checks that gconf settings such as "name" or "modification_date" are not translated in the translation. It allows you to change the surrounding quotes but will ensure that the setting values remain untranslated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as "_: comment\n". New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using --musttranslatefile.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of \n newlines (and variants such as \r\n) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using --notranslatefile.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig (*config*)
Sets the accelerator list.

setsuggestionstore (*store*)
Sets the filename that a checker should use for evaluating suggestions.

short (*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting (*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck (*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be trans-

lated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**IOSChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace (*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes (*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths (*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list (*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options(*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf(*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally

they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig (*config*)
Sets the accelerator list.

setsuggestionstore (*store*)
Sets the filename that a checker should use for evaluating suggestions.

short (*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc(*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**KdeChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as ? or ! are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as \n \ to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as /home/user1 have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. your_user_name/path/to/filename.conf.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from str1.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. rgb() or getEntity.Name() are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as "_: comment\n". New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long (*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords (*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines (*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords (*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers (*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of `run_test()`.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a `FilterFailure` as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig (*config*)
Sets the accelerator list.

setsuggestionstore (*store*)
Sets the filename that a checker should use for evaluating suggestions.

short (*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting (*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc(*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**L20nChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

For Mozilla we lower the severity to cosmetic, and for some languages it also ensures accelerators are absent in the target string since some languages do not use accelerators, for example Indic languages.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits (*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

dialogsizes (*str1*, *str2*)
Checks that dialog sizes are not translated.

This is a Mozilla specific test. Mozilla uses a language called XUL to define dialogues and screens. This can make use of CSS to specify properties of the dialogue. These properties include things such as the width and height of the box. The size might need to be changed if the dialogue size changes due to longer translations. Thus translators can change these settings. But you are only meant to change the number not translate the words 'width' or 'height'. This check capture instances where these are translated. It will also catch other types of errors in these units.

doublequoting (*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing (*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines (*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords (*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers (*str1*, *str2*)
Checks that numbers are not translated.

Special handling for Mozilla to ignore entries that are dialog sizes.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount(*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have

specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs (*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged (*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

Special handling for Mozilla to ignore entries that are dialog sizes.

untranslated (*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls (*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars (*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**LibreOfficeChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `( [ {` or `} ] )` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Not used in LibreOffice

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig (*config*)
Sets the accelerator list.

setsuggestionstore (*store*)
Sets the filename that a checker should use for evaluating suggestions.

short (*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting (*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck (*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

validxml(*str1*, *str2*)
Check that all XML/HTML open/close tags has close/open pair in the translation.

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**MinimalChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace (*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes (*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths (*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list (*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options(*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf(*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally

they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing(*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc(*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat(*str1*, *str2*)
Checks whether python brace format strings match.

run_filters(*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount(*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc(*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**MozillaChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

For Mozilla we lower the severity to cosmetic, and for some languages it also ensures accelerators are absent in the target string since some languages do not use accelerators, for example Indic languages.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If ([{ or }]) appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert #-#-#-#-# into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

dialogsizes(*str1*, *str2*)
Checks that dialog sizes are not translated.

This is a Mozilla specific test. Mozilla uses a language called XUL to define dialogues and screens. This can make use of CSS to specify properties of the dialogue. These properties include things such as the width and height of the box. The size might need to be changed if the dialogue size changes due to longer translations. Thus translators can change these settings. But you are only meant to change the number not translate the words 'width' or 'height'. This check capture instances where these are translated. It will also catch other types of errors in these units.

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes " to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments (*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long (*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords (*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines (*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords (*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers (*str1*, *str2*)
Checks that numbers are not translated.

Special handling for Mozilla to ignore entries that are dialog sizes.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc(*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat(*str1*, *str2*)
Checks whether python brace format strings match.

run_filters(*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount(*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the ' character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. '., this might not be detected properly by the check.

spellcheck (*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs (*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged (*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

Special handling for Mozilla to ignore entries that are dialog sizes.

untranslated (*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls (*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration

information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**OpenOfficeChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If ([{ or }]) appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert #-#-#-#-# into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes " to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. info@example.com are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as ? or ! are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of \n newlines (and variants such as \r\n) and reports and error if they differ.

notranslatewords (*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using --notranslatefile.

numbers (*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options (*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as --help, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. --file=FILE, then the test will check that the parameter has been translated.

printf (*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: %d, %5.2f, %100s, etc. The test can also manage variables-reordering using the %1$s syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals(*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc(*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**ReducedChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes " to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. info@example.com are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as ? or ! are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as \n \ to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as /home/user1 have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. your_user_name/path/to/filename.conf.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from str1.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. rgb() or getEntity.Name() are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as "_: comment\n". New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using --musttranslatefile.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of \n newlines (and variants such as \r\n) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using --notranslatefile.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options(*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf(*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing(*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc(*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat(*str1*, *str2*)
Checks whether python brace format strings match.

run_filters(*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount(*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting (*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck (*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

exception translate.filters.checks.**SeriousFilterFailure**(*messages*)
This exception signals that a Filter didn't pass, and the bad translation might break an application (so the string will be marked fuzzy)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.filters.checks.**StandardChecker**(*checkerconfig=None*, *excludefilters=None*, *limitfilters=None*, *errorhandler=None*)
The basic test suite for source -> target translations.

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `([{` or `}])` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords (*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails (*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc (*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace (*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes (*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n \` to ensure that if they exist in the original string you also have them in the translation.

filepaths (*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list (*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options(*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf(*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally

they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing (*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc (*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat (*str1*, *str2*)
Checks whether python brace format strings match.

run_filters (*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test (*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount (*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig (*config*)
Sets the accelerator list.

setsuggestionstore (*store*)
Sets the filename that a checker should use for evaluating suggestions.

short (*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps (*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting(*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck(*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps(*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc(*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace(*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs(*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged(*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**StandardUnitChecker**(*checkerconfig=None*, *excludefilters=None*, *limitfilters=None*, *errorhandler=None*)
The standard checks for common checks on translation units.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

hassuggestion(*unit*)
Checks if there is at least one suggested translation for this unit.

If a message has a suggestion (an alternate translation stored in alt-trans units in XLIFF and .pending files in PO) then these will be extracted. This is used by Pootle and is probably only useful in pofilter when using XLIFF files.

isfuzzy(*unit*)
Check if the unit has been marked fuzzy.

If a message is marked fuzzy in the PO file then it is extracted. Note this is different from `--fuzzy` and `--nofuzzy` options which specify whether tests should be performed against messages marked fuzzy.

isreview(*unit*)
Check if the unit has been marked review.

If you have made use of the 'review' flags in your translations:

```
# (review) reason for review
# (pofilter) testname: explanation for translator
```

Then if a message is marked for review in the PO file it will be extracted. Note this is different from `--review` and `--noreview` options which specify whether tests should be performed against messages already marked as under review.

nplurals(*unit*)
Checks for the correct number of noun forms for plural translations.

This uses the plural information in the language module of the Translate Toolkit. This is the same as the Gettext nplural value. It will check that the number of plurals required is the same as the number supplied in your translation.

run_filters(*unit*, *categorised=False*)
Run all the tests in this suite.

Return type Dictionary

Returns

Content of the dictionary is as follows:

```
{'testname': { 'message': message_or_exception, 'category': failure_
↪category } }
```

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

class translate.filters.checks.**TeeChecker**(*checkerconfig=None*, *excludefilters=None*, *limitfilters=None*, *checkerclasses=None*, *errorhandler=None*, *languagecode=None*)
A Checker that controls multiple checkers.

categories = {}
Categories where each checking function falls into Function names are used as keys, categories are the values

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns a dictionary of available filters, including/excluding those in the given lists.

run_filters(*unit*, *categorised=False*)
Run all the tests in the checker's suites.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

class translate.filters.checks.**TermChecker**(***kwargs*)

accelerators(*str1*, *str2*)
Checks whether accelerators are consistent between the two strings.

This test is capable of checking the different type of accelerators that are used in different projects, like Mozilla or KDE. The test will pick up accelerators that are missing and ones that shouldn't be there.

See accelerators on the localization guide for a full description on accelerators.

acronyms(*str1*, *str2*)
Checks that acronyms that appear are unchanged.

If an acronym appears in the original this test will check that it appears in the translation. Translating acronyms is a language decision but many languages leave them unchanged. In that case this test is useful for tracking down translations of the acronym and correcting them.

blank(*str1*, *str2*)
Checks whether a translation is totally blank.

This will check to see if a translation has inadvertently been translated as blank i.e. as spaces. This is different from untranslated which is completely empty. This test is useful in that if something is translated as " " it will appear to most tools as if it is translated.

brackets(*str1*, *str2*)
Checks that the number of brackets in both strings match.

If `( [ {` or `} ] )` appear in the original this will check that the same number appear in the translation.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

compendiumconflicts(*str1*, *str2*)
Checks for Gettext compendium conflicts (#-#-#-#-#).

When you use msgcat to create a PO compendium it will insert `#-#-#-#-#` into entries that are not consistent. If the compendium is used later in a message merge then these conflicts will appear in your translations. This test quickly extracts those for correction.

credits(*str1*, *str2*)
Checks for messages containing translation credits instead of normal translations.

Some projects have consistent ways of giving credit to translators by having a unit or two where translators can fill in their name and possibly their contact details. This test allows you to find these units easily to check that they are completed correctly and also disables other tests that might incorrectly get triggered for these units (such as urls, emails, etc.)

doublequoting(*str1*, *str2*)
Checks whether doublequoting is consistent between the two strings.

Checks on double quotes `"` to ensure that you have the same number in both the original and the translated string. This tests takes into account that several languages use different quoting characters, and will test for them instead.

doublespacing(*str1*, *str2*)
Checks for bad double-spaces by comparing to original.

This will identify if you have [space][space] in when you don't have it in the original or it appears in the original but not in your translation. Some of these are spurious and how you correct them depends on the conventions of your language.

doublewords(*str1*, *str2*)
Checks for repeated words in the translation.

Words that have been repeated in a translation will be highlighted with this test e.g. "the the", "a a". These are generally typos that need correcting. Some languages may have valid repeated words in their structure, in that case either ignore those instances or switch this test off.

emails(*str1*, *str2*)
Checks that emails are not translated.

Generally you should not be translating email addresses. This check will look to see that email addresses e.g. `info@example.com` are not translated. In some cases of course you should translate the address but generally you shouldn't.

endpunc(*str1*, *str2*)
Checks whether punctuation at the end of the strings match.

This will ensure that the ending of your translation has the same punctuation as the original. E.g. if it ends in :[space] then so should yours. It is useful for ensuring that you have ellipses [...] in all your translations, not simply three separate full-stops. You may pick up some errors in the original: feel free to keep your translation and notify the programmers. In some languages, characters such as `?` or `!` are always preceded by a space e.g. [space]? — do what your language customs dictate. Other false positives you will notice are, for example, if through changes in word-order you add "), etc. at the end of the sentence. Do not change these: your language word-order takes precedence.

It must be noted that if you are tempted to leave out [full-stop] or [colon] or add [full-stop] to a sentence, that often these have been done for a reason, e.g. a list where fullstops make it look cluttered. So, initially match them with the English, and make changes once the program is being used.

This check is aware of several language conventions for punctuation characters, such as the custom question marks for Greek and Arabic, Devanagari Danda, full-width punctuation for CJK languages, etc. Support for your language can be added easily if it is not there yet.

endwhitespace(*str1*, *str2*)
Checks whether whitespace at the end of the strings matches.

Operates the same as endpunc but is only concerned with whitespace. This filter is particularly useful for those strings which will evidently be followed by another string in the program, e.g. [Password:] or [Enter your username:]. The whitespace is an inherent part of the string. This filter makes sure you don't miss those important but otherwise invisible spaces!

If your language uses full-width punctuation (like Chinese), the visual spacing in the character might be enough without an added extra space.

escapes(*str1*, *str2*)
Checks whether escaping is consistent between the two strings.

Checks escapes such as `\n` `\` to ensure that if they exist in the original string you also have them in the translation.

filepaths(*str1*, *str2*)
Checks that file paths have not been translated.

Checks that paths such as `/home/user1` have not been translated. Generally you do not translate a file path, unless it is being used as an example, e.g. `your_user_name/path/to/filename.conf`.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

functions(*str1*, *str2*)
Checks that function names are not translated.

Checks that function names e.g. `rgb()` or `getEntity.Name()` are not translated.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

kdecomments(*str1*, *str2*)
Checks to ensure that no KDE style comments appear in the translation.

KDE style translator comments appear in PO files as `"_: comment\n"`. New translators often translate the comment. This test tries to identify instances where the comment has been translated.

long(*str1*, *str2*)
Checks whether a translation is much longer than the original string.

This is most useful in the special case where the translation is multiple characters long while the source text is only 1 character long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

musttranslatewords(*str1*, *str2*)
Checks that words configured as definitely translatable don't appear in the translation.

If for instance in your language you decide that you must translate 'OK' then this test will flag any occurrences of 'OK' in the translation if it appeared in the source string. You must specify a file containing all of the *must translate* words using `--musttranslatefile`.

newlines(*str1*, *str2*)
Checks whether newlines are consistent between the two strings.

Counts the number of `\n` newlines (and variants such as `\r\n`) and reports and error if they differ.

notranslatewords(*str1*, *str2*)
Checks that words configured as untranslatable appear in the translation too.

Many brand names should not be translated, this test allows you to easily make sure that words like: Word, Excel, Impress, Calc, etc. are not translated. You must specify a file containing all of the *no translate* words using `--notranslatefile`.

numbers(*str1*, *str2*)
Checks whether numbers of various forms are consistent between the two strings.

You will see some errors where you have either written the number in full or converted it to the digit in your translation. Also changes in order will trigger this error.

options(*str1*, *str2*)
Checks that command line options are not translated.

In messages that contain command line options, such as `--help`, this test will check that these remain untranslated. These could be translated in the future if programs can create a mechanism to allow this, but currently they are not translated. If the options has a parameter, e.g. `--file=FILE`, then the test will check that the parameter has been translated.

printf(*str1*, *str2*)
Checks whether printf format strings match.

If the printf formatting variables are not identical, then this will indicate an error. Printf statements are used by programs to format output in a human readable form (they are placeholders for variable data). They allow you to specify lengths of string variables, string padding, number padding, precision, etc. Generally they will look like this: `%d`, `%5.2f`, `%100s`, etc. The test can also manage variables-reordering using the `%1$s` syntax. The variables' type and details following data are tested to ensure that they are strictly identical, but they may be reordered.

See also printf Format String.

puncspacing(*str1*, *str2*)
Checks for bad spacing after punctuation.

In the case of [full-stop][space] in the original, this test checks that your translation does not remove the space. It checks also for [comma], [colon], etc.

Some languages don't use spaces after common punctuation marks, especially where full-width punctuation marks are used. This check will take that into account.

purepunc(*str1*, *str2*)
Checks that strings that are purely punctuation are not changed.

This extracts strings like + or − as these usually should not be changed.

pythonbraceformat(*str1*, *str2*)
Checks whether python brace format strings match.

run_filters(*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

sentencecount(*str1*, *str2*)
Checks that the number of sentences in both strings match.

Adds the number of sentences to see that the sentence count is the same between the original and translated string. You may not always want to use this test, if you find you often need to reformat your translation, because the original is badly-expressed, or because the structure of your language works better that way. Do what works best for your language: it's the meaning of the original you want to convey, not the exact way it was written in the English.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

short(*str1*, *str2*)
Checks whether a translation is much shorter than the original string.

This is most useful in the special case where the translation is 1 characters long while the source text is multiple characters long. Otherwise, we use a general ratio that will catch very big differences but is set conservatively to limit the number of false positives.

simplecaps(*str1*, *str2*)
Checks the capitalisation of two strings isn't wildly different.

This will pick up many false positives, so don't be a slave to it. It is useful for identifying translations that don't start with a capital letter (upper-case letter) when they should, or those that do when they shouldn't. It will also highlight sentences that have extra capitals; depending on the capitalisation convention of your language, you might want to change these to Title Case, or change them all to normal sentence case.

simpleplurals (*str1*, *str2*)
Checks for English style plural(s) for you to review.

This test will extract any message that contains words with a final "(s)" in the source text. You can then inspect the message, to check that the correct plural form has been used for your language. In some languages, plurals are made by adding text at the beginning of words, making the English style messy. In this case, they often revert to the plural form. This test allows an editor to check that the plurals used are correct. Be aware that this test may create a number of false positives.

For languages with no plural forms (only one noun form) this test will simply test that nothing like "(s)" was used in the translation.

singlequoting (*str1*, *str2*)
Checks whether singlequoting is consistent between the two strings.

The same as doublequoting but checks for the `'` character. Because this is used in contractions like it's and in possessive forms like user's, this test can output spurious errors if your language doesn't use such forms. If a quote appears at the end of a sentence in the translation, i.e. `'.`, this might not be detected properly by the check.

spellcheck (*str1*, *str2*)
Checks words that don't pass a spell check.

This test will check for misspelled words in your translation. The test first checks for misspelled words in the original (usually English) text, and adds those to an exclusion list. The advantage of this exclusion is that many words that are specific to the application will not raise errors e.g. program names, brand names, function names.

The checker works with PyEnchant. You need to have PyEnchant installed as well as a dictionary for your language (for example, one of the Hunspell or aspell dictionaries). This test will only work if you have specified the `--language` option.

The pofilter error that is created, lists the misspelled word, plus suggestions returned from the spell checker. That makes it easy for you to identify the word and select a replacement.

startcaps (*str1*, *str2*)
Checks that the message starts with the correct capitalisation.

After stripping whitespace and common punctuation characters, it then checks to see that the first remaining character is correctly capitalised. So, if the sentence starts with an upper-case letter, and the translation does not, an error is produced.

This check is entirely disabled for many languages that don't make a distinction between upper and lower case. Contact us if this is not yet disabled for your language.

startpunc (*str1*, *str2*)
Checks whether punctuation at the beginning of the strings match.

Operates as endpunc but you will probably see fewer errors.

startwhitespace (*str1*, *str2*)
Checks whether whitespace at the beginning of the strings matches.

As in endwhitespace but you will see fewer errors.

tabs (*str1*, *str2*)
Checks whether tabs are consistent between the two strings.

Counts the number of `\t` tab markers and reports an error if they differ.

unchanged (*str1*, *str2*)
Checks whether a translation is basically identical to the original string.

This checks to see if the translation isn't just a copy of the English original. Sometimes, this is what you want, but other times you will detect words that should have been translated.

untranslated(*str1*, *str2*)
Checks whether a string has been translated at all.

This check is really only useful if you want to extract untranslated strings so that they can be translated independently of the main work.

urls(*str1*, *str2*)
Checks that URLs are not translated.

This checks only basic URLs (http, ftp, mailto etc.) not all URIs (e.g. afp, smb, file). Generally, you don't want to translate URLs, unless they are example URLs (http://your_server.com/filename.html). If the URL is for configuration information, then you need to query the developers about placing configuration information in PO files. It shouldn't really be there, unless it is very clearly marked: such information should go into a configuration file.

validchars(*str1*, *str2*)
Checks that only characters specified as valid appear in the translation.

Often during character conversion to and from UTF-8 you get some strange characters appearing in your translation. This test presents a simple way to try and identify such errors.

This test will only run of you specify the `--validcharsfile` command line option. This file contains all the characters that are valid in your language. You must use UTF-8 encoding for the characters in the file.

If the test finds any characters not in your valid characters file then the test will print the character together with its Unicode value (e.g. 002B).

variables(*str1*, *str2*)
Checks whether variables of various forms are consistent between the two strings.

This checks to make sure that variables that appear in the original also appear in the translation. It can handle variables from projects like KDE or OpenOffice. It does not at the moment cope with variables that use the reordering syntax of Gettext PO files.

xmltags(*str1*, *str2*)
Checks that XML/HTML tags have not been translated.

This check finds the number of tags in the source string and checks that the same number are in the translation. If the counts don't match then either the tag is missing or it was mistakenly translated by the translator, both of which are errors.

The check ignores tags or things that look like tags that cover the whole string e.g. `<Error>` but will produce false positives for things like `An <Error> occurred` as here `Error` should be translated. It also will allow translation of the *alt* attribute in e.g. `<img src="bob.png" alt="Image description">` or similar translatable attributes in OpenOffice.org help files.

class translate.filters.checks.**TranslationChecker**(*checkerconfig=None*, *excludefilters=None*, *limitfilters=None*, *errorhandler=None*)
A checker that passes source and target strings to the checks, not the whole unit.

This provides some speedup and simplifies testing.

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

run_filters(*unit*, *categorised=False*)
Do some optimisation by caching some data of the unit for the benefit of *run_test()*.

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

class translate.filters.checks.**UnitChecker**(*checkerconfig=None*, *excludefilters=None*, *limitfilters=None*, *errorhandler=None*)
Parent Checker class which does the checking based on functions available in derived classes.

categories = None
Categories where each checking function falls into Function names are used as keys, categories are the values

checker_name
Extract checker name, for example 'mozilla' from MozillaChecker.

filteraccelerators_by_list(*str1*, *acceptlist=None*)
Filter out accelerators from `str1`.

get_ignored_filters()
Return checker's additional filters for current language.

getfilters(*excludefilters=None*, *limitfilters=None*)
Returns dictionary of available filters, including/excluding those in the given lists.

run_filters(*unit*, *categorised=False*)
Run all the tests in this suite.

Return type Dictionary

Returns

Content of the dictionary is as follows:

```
{'testname': { 'message': message_or_exception, 'category': failure_
↪category } }
```

run_test(*test*, *unit*)
Runs the given test on the given unit.

Note that this can raise a *FilterFailure* as part of normal operation.

setconfig(*config*)
Sets the accelerator list.

setsuggestionstore(*store*)
Sets the filename that a checker should use for evaluating suggestions.

translate.filters.checks.**batchruntests**(*pairs*)
Runs test on a batch of string pairs.

translate.filters.checks.**intuplelist**(*pair*, *list*)
Tests to see if pair == (a,b,c) is in list, but handles None entries in list as wildcards (only allowed in positions "a" and "c"). We take a shortcut by only considering "c" if "b" has already matched.

translate.filters.checks.**runtests**(*str1*, *str2*, *ignorelist=()*)
Verifies that the tests pass for a pair of strings.

translate.filters.checks.**tagname**(*string*)
Returns the name of the XML/HTML tag in string

translate.filters.checks.**tagproperties**(*strings*, *ignore*)
Returns all the properties in the XML/HTML tag string as (tagname, propertyname, propertyvalue), but ignore those combinations specified in ignore.

decoration

functions to get decorative/informative text out of strings...

translate.filters.decoration.**countaccelerators**(*accelmarker*, *acceptlist=None*)
returns a function that counts the number of accelerators marked with the given marker

translate.filters.decoration.**findaccelerators**(*str1*, *accelmarker*, *acceptlist=None*)
returns all the accelerators and locations in str1 marked with a given marker

translate.filters.decoration.**findmarkedvariables**(*str1*, *startmarker*, *endmarker*, *ignorelist=[]*)
returns all the variables and locations in str1 marked with a given marker

translate.filters.decoration.**getaccelerators**(*accelmarker*, *acceptlist=None*)
returns a function that gets a list of accelerators marked using accelmarker

translate.filters.decoration.**getemails**(*str1*)
returns the email addresses that are in a string

translate.filters.decoration.**getfunctions**(*str1*)
returns the functions() that are in a string, while ignoring the trailing punctuation in the given parameter

translate.filters.decoration.**getnumbers**(*str1*)
returns any numbers that are in the string

translate.filters.decoration.**geturls**(*str1*)
returns the URIs in a string

translate.filters.decoration.**getvariables**(*startmarker*, *endmarker*)
returns a function that gets a list of variables marked using startmarker and endmarker

translate.filters.decoration.**ispurepunctuation**(*str1*)
checks whether the string is entirely punctuation

translate.filters.decoration.**isvalidaccelerator**(*accelerator*, *acceptlist=None*)
returns whether the given accelerator character is valid

Parameters

- **accelerator** (*character*) – A character to be checked for accelerator validity
- **acceptlist** (*String*) – A list of characters that are permissible as accelerators

Return type Boolean

Returns True if the supplied character is an acceptable accelerator

translate.filters.decoration.**puncend**(*str1*, *punctuation*)
returns all the punctuation from the end of the string

translate.filters.decoration.**puncstart**(*str1*, *punctuation*)
returns all the punctuation from the start of the string

translate.filters.decoration.**spaceend**(*str1*)
returns all the whitespace from the end of the string

translate.filters.decoration.**spacestart**(*str1*)
returns all the whitespace from the start of the string

helpers

a set of helper functions for filters...

translate.filters.helpers.**countmatch**(*str1*, *str2*, *countstr*)
checks whether countstr occurs the same number of times in str1 and str2

translate.filters.helpers.**countsmatch**(*str1*, *str2*, *countlist*)
checks whether each element in countlist occurs the same number of times in str1 and str2

translate.filters.helpers.**filtercount**(*str1*, *func*)
returns the number of characters in str1 that pass func

translate.filters.helpers.**filtertestmethod**(*testmethod*, *strfilter*)
returns a version of the testmethod that operates on filtered strings using strfilter

translate.filters.helpers.**funcmatch**(*str1*, *str2*, *func*, **args*)
returns whether the result of func is the same for str1 and str2

translate.filters.helpers.**funcsmatch**(*str1*, *str2*, *funclist*)
checks whether the results of each func in funclist match for str1 and str2

translate.filters.helpers.**multifilter**(*str1*, *strfilters*, **args*)
passes str1 through a list of filters

translate.filters.helpers.**multifiltertestmethod**(*testmethod*, *strfilters*)
returns a version of the testmethod that operates on filtered strings using strfilter

pofilter

Perform quality checks on Gettext PO, XLIFF and TMX localization files.

Snippet files are created whenever a test fails. These can be examined, corrected and merged back into the originals using pomerge.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pofilter.html for examples and usage instructions and http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pofilter_tests.html for full descriptions of all tests.

class translate.filters.pofilter.**FilterOptionParser**(*formats*)
A specialized Option Parser for filter tools...

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

build_checkerconfig(*options*)
Prepare the checker config from the given options. This is mainly factored out for the sake of unit tests.

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

parse_noinput(*option*, *opt*, *value*, *parser*, **args*, ***kwargs*)
This sets an option to *True*, but also sets input to - to prevent an error.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Process an individual file.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
Recurse through directories and process files.

run()
Parses the arguments, and runs recursiveprocess with the resulting options.

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions()
Sets the progress options.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

translate.filters.pofilter.**runfilter**(*inputfile*, *outputfile*, *templatefile*, *checkfilter=None*)
Reads in inputfile, filters using checkfilter, writes to outputfile.

prefilters

Filters that strings can be passed through before certain tests.

translate.filters.prefilters.**filteraccelerators**(*accelmarker*)
Returns a function that filters accelerators marked using *accelmarker* from a strings.

Parameters **accelmarker** (*string*) – Accelerator marker character

Return type Function

Returns fn(str1, acceplist=None)

translate.filters.prefilters.**filtervariables**(*startmarker*, *endmarker*, *varfilter*)
Returns a function that filters variables marked using *startmarker* and *endmarker* from a string.

Parameters

- **startmarker** (*string*) – Start of variable marker
- **endmarker** (*string*) – End of variable marker
- **varfilter** (*Function*) – fn(variable, startmarker, endmarker)

Return type Function

Returns fn(str1)

translate.filters.prefilters.**filterwordswithpunctuation**(*str1*)
Goes through a list of known words that have punctuation and removes the punctuation from them.

translate.filters.prefilters.**removekdecomments**(*str1*)
Remove KDE-style PO comments.

KDE comments start with `_: [space]` and end with a literal `\n`. Example:

```
"_: comment\n"
```

translate.filters.prefilters.**varname**(*variable*, *startmarker*, *endmarker*)
Variable filter that returns the variable name without the marking punctuation.

Note: Currently this function simply returns *variable* unchanged, no matter what **marker*'s are set to.

Return type String

Returns Variable name with the supplied *startmarker* and *endmarker* removed.

translate.filters.prefilters.**varnone**(*variable*, *startmarker*, *endmarker*)
Variable filter that returns an empty string.

Return type String

Returns Empty string

spelling

An API to provide spell checking for use in checks or elsewhere.

lang

Classes that represent languages and provides language-specific information.

All classes inherit from the parent class called *common*.

The type of data includes:

- Language codes
- Language name

- Plurals
- Punctuation transformation
- etc.

af

This module represents the Afrikaans language.

See also:

http://en.wikipedia.org/wiki/Afrikaans_language

class translate.lang.af.**af**
This class represents Afrikaans.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Modify this for the indefinite article ('n).

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

translate.lang.af.**cyr2lat = {'': 'Jo', '': 'A', '': 'B', '': 'W', '': 'G', '': 'D', '':**
Mapping of Cyrillic to Latin letters for transliteration in Afrikaans

translate.lang.af.**tranliterate_cyrillic**(*text*)
Convert Cyrillic text to Latin according to the AWS transliteration rules.

am

This module represents the Amharic language.

See also:

http://en.wikipedia.org/wiki/Amharic_language

class translate.lang.am.**am**
This class represents Amharic.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ar

This module represents the Arabic language.

See also:

http://en.wikipedia.org/wiki/Arabic_language

class translate.lang.ar.**ar**
This class represents Arabic.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

bn

This module represents the Bengali language.

See also:

http://en.wikipedia.org/wiki/Bengali_language

class translate.lang.bn.**bn**
This class represents Bengali.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

code_or

This module represents the Odia language.

See also:

https://en.wikipedia.org/wiki/Odia_language

class translate.lang.code_or.**code_or**
This class represents Odia.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

common

This module contains all the common features for languages.

Supported features:

- language code (km, af)
- language name (Khmer, Afrikaans)
- Plurals
  - Number of plurals (nplurals)
  - Plural equation
- pofilter tests to ignore

Segmentation:

- characters
- words
- sentences

Punctuation:

- End of sentence
- Start of sentence
- Middle of sentence
- Quotes
  - single
  - double
- Valid characters
- Accelerator characters
- Special characters
- Direction (rtl or ltr)

TODOs and Ideas for possible features:

- Language-Team information
- Segmentation
  - phrases

class translate.lang.common.**Common**
This class is the common parent class for all language classes.

CJKpunc = ''
These punctuation marks are used in certain circumstances with CJK languages.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

checker = None
A language specific checker instance (see filters.checks).

This doesn't need to be supplied, but will be used if it exists.

code = ''
The ISO 639 language code, possibly with a country specifier or other modifier.

Examples:

```
km
pt_BR
sr_YU@Latn
```

commonpunc = '.,;:!?-@#$%^*_()[]{}/\\\'`"<>'
These punctuation marks are common in English and most languages that use latin script.

ethiopicpunc = ''
These punctuation marks are used by several Ethiopic languages.

fullname = ''
The full (English) name of this language.

Dialect codes should have the form of:

- Khmer
- Portugese (Brazil)
- TODO: sr_YU@Latn?

ignoretests = {}
Dictionary of tests to ignore in some or all checkers.

Keys are checker names and values are list of names for the ignored tests in the checker. A special 'all' checker name can be used to tell that the tests must be ignored in all the checkers.

Listed checkers to ignore tests on must be lowercase strings for the checker name, for example "mozilla" for MozillaChecker or "libreoffice" for LibreOfficeChecker.

indicpunc = ''
These punctuation marks are used by several Indic languages.

invertedpunc = '¿¡'
Inverted punctuation sometimes used at the beginning of sentences in Spanish, Asturian, Galician, and Catalan.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

listseperator = ', '
This string is used to separate lists of textual elements. Most languages probably can stick with the default comma, but Arabic and some Asian languages might want to override this.

miscpunc = '...±°¹²³·©®×£¥€'
The middle dot (·) is used by Greek and Georgian.

mozilla_pluralequation = '0'
This of languages that has different plural formula in Mozilla than the standard one in Gettext.

nplurals = 0
The number of plural forms of this language.

0 is not a valid value - it must be overridden. Any positive integer is valid (it should probably be between 1 and 6)

See also:

translate.lang.data

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

numbertuple = ()
A tuple of number transformation rules that can be used by numbertranslate().

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

pluralequation = '0'
The plural equation for selection of plural forms.

This is used for PO files to fill into the header.

See also:

Gettext manual, *translate.lang.data*

puncdict = {}
A dictionary of punctuation transformation rules that can be used by punctranslate().

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

punctuation = '.,;:!?-@#$%^*_()[]{}/\\\'`"<>‘’“”„‹ ›«»¿¡÷...±°¹²³·©®×£¥€'
We include many types of punctuation here, simply since this is only meant to determine if something is punctuation. Hopefully we catch some languages which might not be represented with modules. Most languages won't need to override this.

quotes = '‘’“”„‹ ›«»'
These are different quotation marks used by various languages.

rtlpunc = '÷'
These punctuation marks are used by Arabic and Persian, for example.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

sentenceend = '.!?...'
These marks can indicate a sentence end. Once again we try to account for many languages. Most langauges won't need to override this.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

specialchars = ''
Characters used by the language that might not be easy to input with common keyboard layouts

validaccel = None
Characters that can be used as accelerators (access keys) i.e. Alt+X where X is the accelerator. These can include combining diacritics as long as they are accessible from the users keyboard in a single keystroke, but normally they would be at least precomposed characters. All characters, lower and upper, are included in the list.

validdoublewords = []
Some languages allow double words in certain cases. This is a dictionary of such words.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

data

This module stores information and functionality that relates to plurals.

translate.lang.data.**cldr_plural_categories = ['zero', 'one', 'two', 'few', 'many', 'other']**
List of plural tags generated from CLDR 32.0.1 using https://github.com/WeblateOrg/language-data

translate.lang.data.**expansion_factors = {'af': 0.1, 'ar': -0.09, 'es': 0.21, 'fr': 0.28**
Source to target string length expansion factors.

translate.lang.data.**forceunicode**(*string*)
Ensures that the string is in unicode.

Parameters **string** (*Unicode, String*) – A text string

Returns String converted to Unicode and normalized as needed.

Return type Unicode

translate.lang.data.**get_country_iso_name**(*country_code*)
Return country ISO name.

translate.lang.data.**get_language_iso_fullname**(*language_code*)
Return language ISO fullname.

If language code is not a simple ISO 639 code, then we try to split into a two part language code (ISO 639 and ISO 3166).

translate.lang.data.**get_language_iso_name**(*language_code*)
Return language ISO name.

translate.lang.data.**gettext_country**(*langcode=None*)
Returns a gettext function to translate country names into the given language, or the system language if no language is specified.

translate.lang.data.**gettext_domain**(*langcode*, *domain*, *localedir=None*)
Returns a gettext function for given iso domain

translate.lang.data.**gettext_lang**(*langcode=None*)
Returns a gettext function to translate language names into the given language, or the system language if no language is specified.

translate.lang.data.**languagematch**(*languagecode*, *otherlanguagecode*)
matches a languagecode to another, ignoring regions in the second

translate.lang.data.**languages = {'ach': ('Acholi', 2, 'n > 1'), 'af': ('Afrikaans', 2, '**
Dictionary of language data. The language code is the dictionary key (which may contain country codes and modifiers). The value is a tuple: (Full name in English from iso-codes, nplurals, plural equation).

Note that the English names should not be used in user facing places - it should always be passed through the function returned from tr_lang(), or at least passed through _fix_language_name().

translate.lang.data.**normalize**(*string*, *normal_form='NFC'*)
Return a unicode string in its normalized form

Parameters

- **string** – The string to be normalized
- **normal_form** – NFC (default), NFD, NFKC, NFKD

Returns Normalized string

translate.lang.data.**normalized_unicode**(*string*)
Forces the string to unicode and does normalization.

translate.lang.data.**scripts = {'Beng': ['bn', 'mni'], 'Deva': ['anp', 'bho', 'brx', 'doi**
Dictionary of scripts data. The dictionary keys are ISO 15924 script codes, and script names where scripts are missing from standard. The value is a list of codes for languages using that script.

This is mainly used to alter the behavior of some checks (the accelerators one for example).

translate.lang.data.**simplercode**(*code*)
This attempts to simplify the given language code by ignoring country codes, for example.

See also:

- http://www.rfc-editor.org/rfc/bcp/bcp47.txt
- http://www.rfc-editor.org/rfc/rfc4646.txt
- http://www.rfc-editor.org/rfc/rfc4647.txt
- http://www.w3.org/International/articles/language-tags/

translate.lang.data.**simplify_to_common**(*language_code*)
Simplify language code to the most commonly used form for the language, stripping country information for languages that tend not to be localized differently for different countries

translate.lang.data.**tr_lang**(*langcode=None*)
Gives a function that can translate a language name, even in the form `"language (country)"`, into the language with iso code langcode, or the system language if no language is specified.

de

This module represents the German language.

See also:

http://en.wikipedia.org/wiki/German_language

class translate.lang.de.**de**
This class represents German.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

el

This module represents the Greek language.

See also:

http://en.wikipedia.org/wiki/Greek_language

class translate.lang.el.**el**
This class represents Greek.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

es

This module represents the Spanish language.

Note: As it only has special case code for initial inverted punctuation, it could also be used for Asturian, Galician, or Catalan.

---

class translate.lang.es.**es**
This class represents Spanish.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Implement some extra features for inverted punctuation.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

factory

This module provides a factory to instantiate language classes.

translate.lang.factory.**get_all_languages**()
Return all language classes.

translate.lang.factory.**getlanguage**
This returns a language class.

Parameters **code** – The ISO 639 language code

fa

This module represents the Persian language.

See also:

http://en.wikipedia.org/wiki/Persian_language

class translate.lang.fa.**fa**
This class represents Persian.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Implement “French” quotation marks.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

fi

This module represents the Finnish language.

class translate.lang.fi.**fi**
This class represents Finnish.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

fr

This module represents the French language.

See also:

http://en.wikipedia.org/wiki/French_language

class translate.lang.fr.**fr**
This class represents French.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Implement some extra features for quotation marks.

Known shortcomings:

- % and $ are not touched yet for fear of variables
- Double spaces might be introduced

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

gu

This module represents the Gujarati language.

See also:

http://en.wikipedia.org/wiki/Gujarati_language

class translate.lang.gu.**gu**
This class represents Gujarati.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

he

This module represents the Hebrew language.

See also:

http://en.wikipedia.org/wiki/Hebrew_language

class translate.lang.he.**he**
This class represents Hebrew.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

hi

This module represents the Hindi language.

See also:

http://en.wikipedia.org/wiki/Hindi_language

class translate.lang.hi.**hi**
This class represents Hindi.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

hy

This module represents the Armenian language.

See also:

http://en.wikipedia.org/wiki/Armenian_language

class translate.lang.hy.**hy**
This class represents Armenian.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

identify

This module contains functions for identifying languages based on language models.

ja

This module represents the Japanese language.

See also:

http://en.wikipedia.org/wiki/Japanese_language

class translate.lang.ja.**ja**
This class represents Japanese.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

km

This module represents the Khmer language.

See also:

http://en.wikipedia.org/wiki/Khmer_language

class translate.lang.km.**km**
This class represents Khmer.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

khmerpunc = ''
These marks are only used for Khmer.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

kn

This module represents the Kannada language.

See also:

http://en.wikipedia.org/wiki/Kannada_language

class translate.lang.kn.**kn**
This class represents Kannada.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ko

This module represents the Korean language.

See also:

http://en.wikipedia.org/wiki/Korean_language

class translate.lang.ko.**ko**
This class represents Korean.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ml

This module represents the Malayalam language.

See also:

http://en.wikipedia.org/wiki/Malayalam_language

class translate.lang.ml.**ml**
This class represents Malayalam.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

mr

This module represents the Marathi language.

See also:

http://en.wikipedia.org/wiki/Marathi_language

class translate.lang.mr.**mr**
This class represents Marathi.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ne

This module represents the Nepali language.

See also:

http://en.wikipedia.org/wiki/Nepali_language

class translate.lang.ne.**ne**
This class represents Nepali.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ngram

Ngram models for language guessing.

Note: Orignal code from http://thomas.mangin.me.uk/data/source/ngram.py

pa

This module represents the Punjabi language.

See also:

http://en.wikipedia.org/wiki/Punjabi_language

class translate.lang.pa.**pa**
This class represents Punjabi.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

poedit

Functions to manage Poedit's language features.

Note: The ISO 639 maps are from Poedit's isocode.cpp (v1.4.2) to ensure that we match currently released versions of Poedit.

translate.lang.poedit.**dialects = {'Chinese': {'CHINA': 'zh_CN', 'None': 'zh_CN', 'TAIWAN**
Language dialects based on ISO 3166 country names, 'None' is the default fallback

translate.lang.poedit.**isocode**(*language*, *country=None*)
Returns a language code for the given Poedit language name.

Poedit uses language and country names in the PO header entries:

- X-Poedit-Language
- X-Poedit-Country

This function converts the supplied language name into the required ISO 639 code. If needed, in the case of *dialects*, the country name is used to create an xx_YY style dialect code.

Parameters

- **language** (*String*) – Language name
- **country** (*String*) – Country name

Returns ISO 639 language code

Return type String

translate.lang.poedit.**lang_codes = {'aa': 'Afar', 'ab': 'Abkhazian', 'ae': 'Avestan', 'a**
ISO369 codes and names as used by Poedit. Mostly these are identical to ISO 639, but there are some differences.

translate.lang.poedit.**lang_names = {'(Afan) Oromo': 'om', 'Abkhazian': 'ab', 'Afar': 'aa**
Reversed *lang_codes*

si

This module represents the Sinhala language.

See also:

http://en.wikipedia.org/wiki/Sinhala_language

class translate.lang.si.**si**
This class represents Sinhala.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

st

This module represents the Southern Sotho language.

class translate.lang.st.**st**
This class represents Southern Sotho.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

sv

This module represents the the Swedish language.

See also:

http://en.wikipedia.org/wiki/Swedish_language

class translate.lang.sv.**sv**
This class represents Swedish.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ta

This module represents the Tamil language.

See also:

http://en.wikipedia.org/wiki/Tamil_language

class translate.lang.ta.**ta**
This class represents Tamil.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

team

Module to guess the language ISO code based on the 'Language-Team' entry in the header of a Gettext PO file.

translate.lang.team.**LANG_TEAM_CONTACT_SNIPPETS = {'af': ('i18n@af.org.za', 'Petri Jooste'}**
Language codes with snippets of contact information that can be used to uniquely identify the language

translate.lang.team.**guess_language**(*team_string*)
Gueses the language of a PO file based on the Language-Team entry

te

This module represents the Telugu language.

See also:

http://en.wikipedia.org/wiki/Telugu_language

class translate.lang.te.**te**
This class represents Telugu.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

th

This module represents the Thai language.

See also:

http://en.wikipedia.org/wiki/Thai_language

class translate.lang.th.**th**
This class represents Thai.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ug

This module represents the Uyghur language.

See also:

http://en.wikipedia.org/wiki/Uyghur_language

class translate.lang.ug.**ug**
This class represents Uyghur.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

ur

This module represents the Urdu language.

See also:

http://en.wikipedia.org/wiki/Urdu_language

class translate.lang.ur.**ur**
This class represents Urdu.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

vi

This module represents the Vietnamese language.

See also:

http://en.wikipedia.org/wiki/Vietnamese_language

class translate.lang.vi.**vi**
This class represents Vietnamese.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Implement some extra features for quotation marks.

Known shortcomings:

- % and $ are not touched yet for fear of variables
- Double spaces might be introduced

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

zh

This module represents the Chinese language (Both tradisional and simplified).

See also:

http://en.wikipedia.org/wiki/Chinese_language

class translate.lang.zh.**zh**
This class represents Chinese.

classmethod alter_length(*text*)
Converts the given string by adding or removing characters as an estimation of translation length (with English assumed as source language).

classmethod capsstart(*text*)
Determines whether the text starts with a capital letter.

classmethod character_iter(*text*)
Returns an iterator over the characters in text.

classmethod characters(*text*)
Returns a list of characters in text.

classmethod length_difference(*length*)
Returns an estimate to a likely change in length relative to an English string of length length.

classmethod numbertranslate(*text*)
Converts the numbers in a string according to the rules of the language.

classmethod numstart(*text*)
Determines whether the text starts with a numeric value.

classmethod punctranslate(*text*)
Converts the punctuation in a string according to the rules of the language.

classmethod sentence_iter(*text*, *strip=True*)
Returns an iterator over the sentences in text.

classmethod sentences(*text*, *strip=True*)
Returns a list of sentences in text.

classmethod word_iter(*text*)
Returns an iterator over the words in text.

classmethod words(*text*)
Returns a list of words in text.

misc

Miscellaneous modules for translate - including modules for backward compatibility with pre-2.3 versions of Python

dictutils

Implements a case-insensitive (on keys) dictionary and order-sensitive dictionary

class translate.misc.dictutils.**cidict**(*fromdict=None*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*)
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*E*) → None.
Update D from E: for k in E.keys(): D[k] = E[k]

values() → an object providing a view on D's values

file_discovery

translate.misc.file_discovery.**get_abs_data_filename**(*path_parts*, *basedirs=None*)
Get the absolute path to the given file- or directory name in the current running application's data directory.

Parameters **path_parts** (*list*) – The path parts that can be joined by `os.path.join()`.

multistring

Supports a hybrid Unicode string that can also have a list of alternate strings in the strings attribute

class translate.misc.multistring.**multistring**(**args*, ***kwargs*)

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace(*old*, *new*, *count=None*)
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace remove.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper()
Return a copy of the string converted to uppercase.

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

optrecurse

class translate.misc.optrecurse.**ManHelpFormatter**(*indent_increment=0*, *max_help_position=0*, *width=80*, *short_first=1*)

format_option_strings(*option*)
Return a comma-separated list of option strings & metavariables.

class translate.misc.optrecurse.**ManPageOption**(**opts*, ***attrs*)

take_action(*action*, *dest*, *opt*, *value*, *values*, *parser*)
take_action that can handle manpage as well as standard actions

class translate.misc.optrecurse.**RecursiveOptionParser**(*formats*, *usetemplates=False*, *allowmissingtemplate=False*, *description=None*)
A specialized Option Parser for recursing through directories.

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
Parses the command line options, handling implicit input/output args.

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile (*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Process an individual file.

recurseinputfilelist (*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles (*options*)
Recurse through directories and return files to be processed.

recursiveprocess (*options*)
Recurse through directories and process files.

run ()
Parses the arguments, and runs recursiveprocess with the resulting options...

set_usage (*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions ()
Sets the errorlevel options.

setformats (*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption ()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions ()
Sets the progress options.

splitext (*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext (*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext (*templatepath*)
Splits a *templatepath* into name and extension.

templateexists (*options*, *templatepath*)
Returns whether the given template exists...

warning (*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

ourdom

module that provides modified DOM functionality for our needs

Note that users of ourdom should ensure that no code might still use classes directly from minidom, like minidom.Element, minidom.Document or methods such as minidom.parseString, since the functionality provided here will not be in those objects.

class translate.misc.ourdom.**Document**

documentElement
Top-level element of this document.

firstChild
First child node, or None.

lastChild
Last child node, or None.

localName
Namespace-local name of this node.

class translate.misc.ourdom.**Element**(*tagName*, *namespaceURI=None*, *prefix=None*, *localName=None*)

attributes
NamedNodeMap of attributes on the element.

firstChild
First child node, or None.

lastChild
Last child node, or None.

localName
Namespace-local name of this element.

class translate.misc.ourdom.**ExpatBuilderNS**(*options=None*)

createParser()
Create a new namespace-handling parser.

getParser()
Return the parser object, creating a new one if needed.

install(*parser*)
Insert the namespace-handlers onto the parser.

parseFile(*file*)
Parse a document from a file object, returning the document node.

parseString(*string*)
Parse a document from a string, returning the document node.

reset()
Free all data structures used during DOM construction.

start_namespace_decl_handler(*prefix*, *uri*)
Push this namespace declaration on our storage.

translate.misc.ourdom.**getElementsByTagName_helper**(*parent*, *name*, *dummy=None*)
A reimplementation of getElementsByTagName as an iterator.

Note that this is not compatible with getElementsByTagName that returns a list, therefore, the class below exposes this through yieldElementsByTagName

translate.misc.ourdom.**getnodetext**(*node*)
returns the node's text by iterating through the child nodes

translate.misc.ourdom.**parse**(*file*, *parser=None*, *bufsize=None*)
Parse a file into a DOM by filename or file object.

translate.misc.ourdom.**parseString**(*string*, *parser=None*)
Parse a file into a DOM from a string.

translate.misc.ourdom.**searchElementsByTagName_helper**(*parent*, *name*, *onlysearch*)
limits the search to within tags occuring in onlysearch

translate.misc.ourdom.**writexml_helper**(*self*, *writer*, *indent=''*, *addindent=''*, *newl=''*)
A replacement for writexml that formats it like typical XML files. Nodes are intendented but text nodes, where whitespace can be significant, are not indented.

progressbar

Progress bar utilities for reporting feedback on the progress of an application.

class translate.misc.progressbar.**DotsProgressBar**
An ultra-simple progress indicator that just writes a dot for each action

show(*verbosemessage*)
show a dot for progress :-)

class translate.misc.progressbar.**HashProgressBar**(**args*, ***kwargs*)
A ProgressBar which knows how to go back to the beginning of the line.

show(*verbosemessage*)
displays the progress bar

class translate.misc.progressbar.**MessageProgressBar**(**args*, ***kwargs*)
A ProgressBar that just writes out the messages without any progress display

show(*verbosemessage*)
displays the progress bar

class translate.misc.progressbar.**NoProgressBar**
An invisible indicator that does nothing.

show(*verbosemessage*)
show nothing for progress :-)

class translate.misc.progressbar.**ProgressBar**(*minValue=0*, *maxValue=100*, *totalWidth=50*)
A plain progress bar that doesn't know very much about output.

show(*verbosemessage*)
displays the progress bar

class translate.misc.progressbar.**VerboseProgressBar**(**args*, ***kwargs*)

show(*verbosemessage*)
displays the progress bar

quote

String processing utilities for extracting strings with various kinds of delimiters

translate.misc.quote.**entitydecode**(*source*, *name2codepoint*)
Decode source using entities from name2codepoint.

Parameters

- **source** (*unicode*) – Source string to decode
- **name2codepoint** (dict()) – Dictionary mapping entity names (without the the leading & or the trailing ;) to code points

translate.misc.quote.**entityencode**(*source*, *codepoint2name*)
Encode source using entities from codepoint2name.

Parameters

- **source** (*unicode*) – Source string to encode
- **codepoint2name** (dict()) – Dictionary mapping code points to entity names (without the the leading & or the trailing ;)

translate.misc.quote.**escapecontrols**(*source*)
escape control characters in the given string

translate.misc.quote.**extract**(*source*, *startdelim*, *enddelim*, *escape=None*, *startinstring=False*, *allowreentry=True*)
Extracts a doublequote-delimited string from a string, allowing for backslash-escaping returns tuple of (quoted string with quotes, still in string at end).

translate.misc.quote.**extractwithoutquotes**(*source*, *startdelim*, *enddelim*, *escape=None*, *startinstring=False*, *includeescapes=True*, *allowreentry=True*)
Extracts a doublequote-delimited string from a string, allowing for backslash-escaping includeescapes can also be a function that takes the whole escaped string and returns the replaced version.

translate.misc.quote.**find_all**(*searchin*, *substr*)
Returns a list of locations where substr occurs in searchin locations are not allowed to overlap

translate.misc.quote.**htmlentitydecode**(*source*)
Decode source using HTML entities e.g. © -> ©.

Parameters **source** (*unicode*) – Source string to decode

translate.misc.quote.**htmlentityencode**(*source*)
Encode source using HTML entities e.g. © -> ©

Parameters **source** (*unicode*) – Source string to encode

translate.misc.quote.**java_utf8_properties_encode**(*source*)
Encodes source in the escaped-unicode encoding used by java utf-8 .properties files.

translate.misc.quote.**javapropertiesencode**(*source*)
Encodes source in the escaped-unicode encoding used by Java .properties files

translate.misc.quote.**mozillaescapemarginspaces**(*source*)
Escape leading and trailing spaces for Mozilla .properties files.

translate.misc.quote.**propertiesdecode**(*source*)
Decodes source from the escaped-unicode encoding used by .properties files.

Java uses Latin1 by default, and Mozilla uses UTF-8 by default.

Since the .decode("unicode-escape") routine decodes everything, and we don't want to we reimplemented the algorithm from Python Objects/unicode.c in Python and modify it to retain escaped control characters.

wsgi

Wrapper to launch the bundled CherryPy server.

translate.misc.wsgi.**launch_server**(*host*, *port*, *app*, ***kwargs*)
Use cheroot WSGI server, a multithreaded scallable server.

xml_helpers

Helper functions for working with XML.

translate.misc.xml_helpers.**getText**(*node*, *xml_space='preserve'*)
Extracts the plain text content out of the given node.

This method checks the xml:space attribute of the given node, and takes an optional default to use in case nothing is specified in this node.

translate.misc.xml_helpers.**getXMLlang**(*node*)
Gets the xml:lang attribute on node

translate.misc.xml_helpers.**getXMLspace**(*node*, *default=None*)
Gets the xml:space attribute on node

translate.misc.xml_helpers.**namespaced**(*namespace*, *name*)
Returns name in Clark notation within the given namespace.

For example namespaced("source") in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

translate.misc.xml_helpers.**normalize_space**(*text*)
Normalize the given text for implementation of `xml:space="default"`.

translate.misc.xml_helpers.**normalize_xml_space**(*node*, *xml_space*, *remove_start=False*)
normalize spaces following the nodes xml:space, or alternatively the given xml_space parameter.

translate.misc.xml_helpers.**reindent**(*elem*, *level=0*, *indent=' '*, *max_level=4*, *skip=None*, *toplevel=True*, *leaves=None*)
Adjust indentation to match specification.

Each nested tag is identified by indent string, up to max_level depth, possibly skipping tags listed in skip.

translate.misc.xml_helpers.**setXMLlang**(*node*, *lang*)
Sets the xml:lang attribute on node

translate.misc.xml_helpers.**setXMLspace**(*node*, *value*)
Sets the xml:space attribute on node

translate.misc.xml_helpers.**string_xpath = string()**
Return a non-normalized string in the node subtree

translate.misc.xml_helpers.**string_xpath_normalized = normalize-space()**
Return a (space) normalized string in the node subtree

translate.misc.xml_helpers.**xml_preserve_ancestors = ancestor-or-self::*[attribute::xml:spac**
All ancestors with xml:space='preserve'

translate.misc.xml_helpers.**xml_space_ancestors = ancestor-or-self::*/attribute::xml:space**
All xml:space attributes in the ancestors

search

Services for searching and matching of text.

lshtein

A class to calculate a similarity based on the Levenshtein distance.

See http://en.wikipedia.org/wiki/Levenshtein_distance.

If available, the python-Levenshtein will be used which will provide better performance as it is implemented natively.

translate.search.lshtein.**distance**(*a*, *b*, *stopvalue=0*)
Same as python_distance in functionality. This uses the fast C version if we detected it earlier.

Note that this does not support arbitrary sequence types, but only string types.

translate.search.lshtein.**native_distance**(*a*, *b*, *stopvalue=0*)
Same as python_distance in functionality. This uses the fast C version if we detected it earlier.

Note that this does not support arbitrary sequence types, but only string types.

translate.search.lshtein.**python_distance**(*a*, *b*, *stopvalue=-1*)
Calculates the distance for use in similarity calculation. Python version.

match

Class to perform translation memory matching from a store of translation units.

class translate.search.match.**matcher**(*store*, *max_candidates=10*, *min_similarity=75*, *max_length=70*, *comparer=None*, *usefuzzy=False*)
A class that will do matching and store configuration for the matching process.

buildunits(*candidates*)
Builds a list of units conforming to base API, with the score in the comment.

extendtm(*units*, *store=None*, *sort=True*)
Extends the memory with extra unit(s).

Parameters

- **units** – The units to add to the TM.
- **store** – Optional store from where some metadata can be retrieved and associated with each unit.
- **sort** – Optional parameter that can be set to False to supress sorting of the candidates list. This should probably only be used in *matcher.inittm()*.

getstartlength(*min_similarity*, *text*)
Calculates the minimum length we are interested in. The extra fat is because we don't use plain character distance only.

getstoplength(*min_similarity*, *text*)
Calculates a length beyond which we are not interested. The extra fat is because we don't use plain character distance only.

inittm(*stores*, *reverse=False*)
Initialises the memory for later use. We use simple base units for speedup.

matches(*text*)
Returns a list of possible matches for given source text.

Parameters **text** (*String*) – The text that will be search for in the translation memory

Return type list

Returns a list of units with the source and target strings from the translation memory. If `self.addpercentage` is *True* (default) the match quality is given as a percentage in the notes.

setparameters(*max_candidates=10*, *min_similarity=75*, *max_length=70*)
Sets the parameters without reinitialising the tm. If a parameter is not specified, it is set to the default, not ignored

usable(*unit*)
Returns whether this translation unit is usable for TM

translate.search.match.**sourcelen**(*unit*)
Returns the length of the source string.

class translate.search.match.**terminologymatcher**(*store*, *max_candidates=10*, *min_similarity=75*, *max_length=500*, *comparer=None*)
A matcher with settings specifically for terminology matching.

buildunits(*candidates*)
Builds a list of units conforming to base API, with the score in the comment.

extendtm(*units*, *store=None*, *sort=True*)
Extends the memory with extra unit(s).

Parameters

- **units** – The units to add to the TM.
- **store** – Optional store from where some metadata can be retrieved and associated with each unit.
- **sort** – Optional parameter that can be set to False to supress sorting of the candidates list. This should probably only be used in *matcher.inittm()*.

getstartlength(*min_similarity*, *text*)
Calculates the minimum length we are interested in. The extra fat is because we don't use plain character distance only.

getstoplength(*min_similarity*, *text*)
Calculates a length beyond which we are not interested. The extra fat is because we don't use plain character distance only.

inittm(*store*)
Normal initialisation, but convert all source strings to lower case

matches(*text*)
Normal matching after converting text to lower case. Then replace with the original unit to retain comments, etc.

setparameters(*max_candidates=10*, *min_similarity=75*, *max_length=70*)
Sets the parameters without reinitialising the tm. If a parameter is not specified, it is set to the default, not ignored

usable(*unit*)
Returns whether this translation unit is usable for terminology.

translate.search.match.**unit2dict**(*unit*)
converts a pounit to a simple dict structure for use over the web

terminology

A class that does terminology matching

services

translate.services is part of the translate toolkit. It provides network services for interacting with the toolkit

tmserver

A translation memory server using tmdb for storage, communicates with clients using JSON over HTTP.

class translate.services.tmserver.**TMServer**(*tmdbfile*, *tmfiles*, *max_candidates=3*, *min_similarity=75*, *max_length=1000*, *prefix=''*, *source_lang=None*, *target_lang=None*)
A RESTful JSON TM server.

storage

Classes that represent various storage formats for localization.

base

Base classes for storage interfaces.

class translate.storage.base.**DictStore**(*unitclass=None*, *encoding=None*)

UnitClass
alias of *TranslationUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type *TranslationUnit*

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (*TranslationUnit*) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type *TranslationUnit* or None

findunits(*source*)
Find the units with the given source string.

Return type *TranslationUnit* or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*data*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (*TranslationUnit*) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.base.**DictUnit**(*source=None*)

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement *TranslationUnit.addlocation()*.

> **Warning:** This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

> **Note:** Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

> **Note:** Plural forms might be combined.

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
```

(continues on next page)

(continued from previous page)

```
<StringElem([<StringElem(['bar'])>])>,
<StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

exception translate.storage.base.**ParseError**(*inner_exc*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.storage.base.**TranslationStore**(*unitclass=None*, *encoding=None*)
Base class for stores for multiple translation units of type UnitClass.

Extensions = None
A list of file extentions associated with this store type

Mimetypes = None
A list of MIME types associated with this store type

Name = 'Base translation store'
The human usable name of this store type

UnitClass
The class of units that will be instantiated and used by this class

alias of *TranslationUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type *TranslationUnit*

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (*TranslationUnit*) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type *TranslationUnit* or None

findunits(*source*)
Find the units with the given source string.

Return type *TranslationUnit* or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*data*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (*TranslationUnit*) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

suggestions_in_format = False
Indicates if format can store suggestions and alternative translation for a unit

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.base.**TranslationUnit**(*source=None*)
Base class for translation units.

Our concept of a *translation unit* is influenced heavily by XLIFF.

As such most of the method- and variable names borrows from XLIFF terminology.

A translation unit consists of the following:

- A *source* string. This is the original translatable text.
- A *target* string. This is the translation of the *source*.
- Zero or more *notes* on the unit. Notes would typically be some comments from a translator on the unit, or some comments originating from the source code.
- Zero or more *locations*. Locations indicate where in the original source code this unit came from.

- Zero or more *errors*. Some tools (eg. *pofilter*) can run checks on translations and produce error messages.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

---

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement *TranslationUnit.addlocation()*.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_parsers = []
A list of functions to use for parsing a string into a rich string tree.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

benchmark

class translate.storage.benchmark.**TranslateBenchmarker**(*test_dir*, *storeclass*)
class to aid in benchmarking Translate Toolkit stores

clear_test_dir()
removes the given directory

create_sample_files(*num_dirs*, *files_per_dir*, *strings_per_file*, *source_words_per_string*, *target_words_per_string*)
creates sample files for benchmarking

parse_files(*file_dir=None*)
parses all the files in the test directory into memory

parse_placeables()
parses placeables

bundleprojstore

class translate.storage.bundleprojstore.**BundleProjectStore**(*fname*)
Represents a translate project bundle (zip archive).

append_file(*afile*, *fname*, *ftype='trans'*, *delete_orig=False*)
Append the given file to the project with the given filename, marked to be of type `ftype` ('src', 'trans', 'tgt').

Parameters **delete_orig** – If `True`, as set by `convert_forward()`, `afile` is deleted after appending, if possible.

Note: For this implementation, the appended file will be deleted from disk if `delete_orig` is `True`.

---

cleanup()
Clean up our mess: remove temporary files.

get_file(*fname*)
Retrieve a project file (source, translation or target file) from the project archive.

get_filename_type(*fname*)
Get the type of file ('src', 'trans', 'tgt') with the given name.

get_proj_filename(*realfname*)
Try and find a project file name for the given real file name.

load(*zipname*)
Load the bundle project from the zip file of the given name.

remove_file(*fname*, *ftype=None*)
Remove the file with the given project name from the project.

save(*filename=None*)
Save all project files to the bundle zip file.

sourcefiles
Read-only access to `self._sourcefiles`.

targetfiles
Read-only access to `self._targetfiles`.

transfiles
Read-only access to `self._transfiles`.

update_file(*pfname*, *infile*)
Updates the file with the given project file name with the contents of `infile`.

Returns the results from `BundleProjStore.append_file()`.

exception translate.storage.bundleprojstore.**InvalidBundleError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

catkeys

Manage the Haiku catkeys translation format

The Haiku catkeys format is the translation format used for localisation of the Haiku operating system.

It is a bilingual base class derived format with *CatkeysFile* and *CatkeysUnit* providing file and unit level access. The file format is described here: http://www.haiku-os.org/blog/pulkomandy/2009-09-24_haiku_locale_kit_translator_handbook

Implementation The implementation covers the full requirements of a catkeys file. The files are simple Tab Separated Value (TSV) files that can be read by Microsoft Excel and other spreadsheet programs. They use the .txt extension which does make it more difficult to automatically identify such files.

The dialect of the TSV files is specified by *CatkeysDialect*.

Encoding The files are UTF-8 encoded.

Header *CatkeysHeader* provides header management support.

Escaping catkeys seem to escape things like in C++ (strings are just extracted from the source code unchanged, it seems.

Functions allow for `_escape()` and `_unescape()`.

class translate.storage.catkeys.**CatkeysDialect**
Describe the properties of a catkeys generated TAB-delimited file.

class translate.storage.catkeys.**CatkeysFile**(*inputfile=None*, ***kwargs*)
A catkeys translation memory file

UnitClass
alias of *CatkeysUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type `TranslationUnit`

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*newlang*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.catkeys.**CatkeysHeader**(*header=None*)
A catkeys translation memory header

setchecksum(*checksum*)
Set the checksum for the file

settargetlanguage(*newlang*)
Set a human readable target language

class translate.storage.catkeys.**CatkeysUnit**(*source=None*)
A catkeys translation memory unit

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

dict
Get the dictionary of values for a catkeys line

getcontext()
Get the message context.

getdict()
Get the dictionary of values for a catkeys line

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*present=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setdict(*newdict*)
Set the dictionary of values for a catkeys line

Parameters **newdict** (*Dict*) – a new dictionary with catkeys line elements

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.catkeys.**FIELDNAMES = ['source', 'context', 'comment', 'target']**
Field names for a catkeys TU

translate.storage.catkeys.**FIELDNAMES_HEADER = ['version', 'language', 'mimetype', 'checksu**
Field names for the catkeys header

translate.storage.catkeys.**FIELDNAMES_HEADER_DEFAULTS = {'checksum': '', 'language': '',**
Default or minimum header entries for a catkeys file

cpo

csvl10n

classes that hold units of comma-separated values (.csv) files (csvunit) or entire files (csvfile) for use with localisation

class translate.storage.csvl10n.**DefaultDialect**

class translate.storage.csvl10n.**csvfile**(*inputfile=None*, *fieldnames=None*, *encoding='auto'*)

This class represents a .csv file with various lines. The default format contains three columns: location, source, target

UnitClass
alias of *csvunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*csvsrc*, *sample_length=1024*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write to file

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.csvl10n.**csvunit**(*source=None*)

add_spreadsheet_escapes(*source*, *target*)
add common spreadsheet escapes to two strings

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

match_header()
see if unit might be a header

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

remove_spreadsheet_escapes(*source*, *target*)
remove common spreadsheet escapes from two strings

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*value*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.csvl10n.**detect_header**(*inputfile*, *dialect*, *fieldnames*)
Test if file has a header or not, also returns number of columns in first row

translate.storage.csvl10n.**valid_fieldnames**(*fieldnames*)
Check if fieldnames are valid, that is at least one field is identified as the source.

directory

This module provides functionality to work with directories.

class translate.storage.directory.**Directory**(*dir=None*)
This class represents a directory.

file_iter()
Iterator over (dir, filename) for all files in this directory.

getfiles()
Returns a list of (dir, filename) tuples for all the file names in this directory.

getunits()
List of all the units in all the files in this directory.

scanfiles()
Populate the internal file data.

unit_iter()
Iterator over all the units in all the files in this directory.

dtd

Classes that hold units of .dtd files (*dtdunit*) or entire files (*dtdfile*).

These are specific .dtd files for localisation used by mozilla.

Specifications The following information is provided by Mozilla:

Specification

There is a grammar for entity definitions, which isn't really precise, as the spec says. There's no formal specification for DTD files, it's just "whatever makes this work" basically. The whole piece is clearly not the strongest point of the xml spec

XML elements are allowed in entity values. A number of things that are allowed will just break the resulting document, Mozilla forbids these in their DTD parser.

Dialects There are two dialects:

- Regular DTD
- Android DTD

Both dialects are similar, but the Android DTD uses some particular escapes that regular DTDs don't have.

Escaping in regular DTD In DTD usually there are characters escaped in the entities. In order to ease the translation some of those escaped characters are unescaped when reading from, or converting, the DTD, and that are escaped again when saving, or converting to a DTD.

In regular DTD the following characters are usually or sometimes escaped:

- The % character is escaped using % or % or %
- The " character is escaped using "
- The ' character is escaped using ' (partial roundtrip)
- The & character is escaped using &
- The < character is escaped using < (not yet implemented)
- The > character is escaped using > (not yet implemented)

Besides the previous ones there are a lot of escapes for a huge number of characters. This escapes usually have the form of &#NUMBER; where NUMBER represents the numerical code for the character.

There are a few particularities in DTD escaping. Some of the escapes are not yet implemented since they are not really necessary, or because its implementation is too hard.

A special case is the ' escaping using ' which doesn't provide a full roundtrip conversion in order to support some special Mozilla DTD files.

Also the " character is never escaped in the case that the previous character is = (the sequence =" is present on the string) in order to avoid escaping the " character indicating an attribute assignment, for example in a href attribute for an a tag in HTML (anchor tag).

Escaping in Android DTD It has the sames escapes as in regular DTD, plus this ones:

- The ' character is escaped using ' or ' or u0027
- The " character is escaped using "

`translate.storage.dtd.`**`accesskeysuffixes = ('.accesskey', '.accessKey', '.akey')`**
Accesskey Suffixes: entries with this suffix may be combined with labels ending in `labelsuffixes` into accelerator notation

class `translate.storage.dtd.`**dtdfile**(*inputfile=None*, *android=False*)
A .dtd file made up of dtdunits.

UnitClass
alias of *dtdunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type `TranslationUnit`

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*dtdsrc*)
read the source code of a dtd file in and include them as dtdunits in self.units

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write content to file

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.dtd.**dtdunit**(*source=''*, *android=False*)
An entity definition from a DTD file (and any associated comments).

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Set the entity to the given "location".

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
Return the entity as location (identifier).

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput()
convert the dtd entity back to string form

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
returns whether this dtdunit doesn't actually have an entity definition

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

parse(*dtdsrc*)
read the first dtd element from the source code into this object, return linesprocessed

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*new_id*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

source
gets the unquoted source string

target
gets the unquoted target string

unit_iter()
Iterator that only returns this unit.

translate.storage.dtd.**labelsuffixes = ('.label', '.title')**
Label suffixes: entries with this suffix are able to be comibed with accesskeys found in in entries ending with `accesskeysuffixes`

translate.storage.dtd.**quoteforandroid**(*source*)
Escapes a line for Android DTD files.

translate.storage.dtd.**quotefordtd**(*source*)
Quotes and escapes a line for regular DTD files.

translate.storage.dtd.**removeinvalidamps**(*name*, *value*)
Find and remove ampersands that are not part of an entity definition.

A stray & in a DTD file can break an application's ability to parse the file. In Mozilla localisation this is very important and these can break the parsing of files used in XUL and thus break interface rendering. Tracking down the problem is very difficult, thus by removing potential broken ampersand and warning the users we can ensure that the output DTD will always be parsable.

Parameters

- **name** (*String*) – Entity name
- **value** (*String*) – Entity text value

Return type String

Returns Entity value without bad ampersands

translate.storage.dtd.**unquotefromandroid**(*source*)
Unquotes a quoted Android DTD definition.

translate.storage.dtd.**unquotefromdtd**(*source*)
unquotes a quoted dtd definition

_factory_classes

Py2exe can't find stuff that we import dynamically, so we have this file just for the sake of the Windows installer to easily pick up all the stuff that we need and ensure they make it into the installer.

factory

factory methods to build real storage objects that conform to base.py

translate.storage.factory.**getclass**(*storefile*, *localfiletype=None*, *ignore=None*, *classes=None*, *classes_str=None*, *hiddenclasses=None*)
Factory that returns the applicable class for the type of file presented. Specify ignore to ignore some part at the back of the name (like .gz).

translate.storage.factory.**getobject**(*storefile*, *localfiletype=None*, *ignore=None*, *classes=None*, *classes_str=None*, *hiddenclasses=None*)

Factory that returns a usable object for the type of file presented.

Parameters **storefile** (*file or str or* TranslationStore) – File object or file name.

Specify ignore to ignore some part at the back of the name (like .gz).

translate.storage.factory.**supported_files**()

Returns data about all supported files

Returns list of type that include (name, extensions, mimetypes)

Return type list

fpo

html

module for parsing html files for translation

class translate.storage.html.**POHTMLParser**(*includeuntaggeddata=None*, *inputfile=None*, *callback=None*)

UnitClass

alias of *htmlunit*

add_unit_to_index(*unit*)

Add a unit to source and location idexes

addsourceunit(*source*)

Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)

Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

close()

Handle any buffered data.

detect_encoding(*text*, *default_encodings=None*)

Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

do_encoding(*htmlsrc*)

Return the html text properly encoded based on a charset.

fallback_detection(*text*)

Simple detection based on BOM in case chardet is not available.

feed(*data*)

Feed data to the parser.

Call this as often as you want, with as little or as much text as you want (may include 'n').

findid(*id*)

find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

get_starttag_text()
Return full source of start tag: '<...>'.

getids(*filename=None*)
return a list of unit ids

getpos()
Return current line number and offset.

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

guess_encoding(*htmlsrc*)
Returns the encoding of the html text.

We look for 'charset=' within a meta tag to do this.

handle_charref(*name*)
Handle entries in the form &#NNNN; e.g. ⃡

handle_entityref(*name*)
Handle named entities of the form &aaaa; e.g. ’

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*htmlsrc*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

reset()
Reset this instance. Loses all unprocessed data.

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.html.**htmlfile**(*includeuntaggeddata=None*, *inputfile=None*, *callback=None*)

EMPTY_HTML_ELEMENTS = ['area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input', 'li
An empty element is an element that cannot have any child nodes (i.e., nested elements or text nodes). In HTML, using a closing tag on an empty element is usually invalid. Reference https://developer.mozilla.org/en-US/docs/Glossary/Empty_element

TRANSLATABLE_ATTRIBUTES = ['abbr', 'alt', 'lang', 'summary', 'title', 'value']
Text from these HTML attributes will be extracted as translation units. Note: the content attribute of meta tags is a special case.

TRANSLATABLE_ELEMENTS = ['address', 'article', 'aside', 'blockquote', 'caption', 'dd',
These HTML elements (tags) will be extracted as translation units, unless they lack translatable text content. In case one translatable element is embedded in another, the outer translation unit will be split into the parts before and after the inner translation unit.

TRANSLATABLE_METADATA = ['description', 'keywords']
Document metadata from meta elements with these names will be extracted as translation units. Reference https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta/name

UnitClass
alias of *htmlunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

close()
Handle any buffered data.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

do_encoding(*htmlsrc*)
Return the html text properly encoded based on a charset.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

feed(*data*)
Feed data to the parser.

Call this as often as you want, with as little or as much text as you want (may include 'n').

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

get_starttag_text()
Return full source of start tag: '<...>'.

getids(*filename=None*)
return a list of unit ids

getpos()
Return current line number and offset.

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

guess_encoding(*htmlsrc*)
Returns the encoding of the html text.

We look for 'charset=' within a meta tag to do this.

handle_charref(*name*)
Handle entries in the form &#NNNN; e.g. ⃡

handle_entityref(*name*)
Handle named entities of the form &aaaa; e.g. ’

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*htmlsrc*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

reset()
Reset this instance. Loses all unprocessed data.

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.html.**htmlunit**(*source=None*)
A unit of translatable/localisable HTML content

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

ical

Class that manages iCalender files for translation.

iCalendar files follow the RFC2445 specification.

The iCalendar specification uses the following naming conventions:

- Component: an event, journal entry, timezone, etc
- Property: a property of a component: summary, description, start time, etc
- Attribute: an attribute of a property, e.g. language

The following are localisable in this implementation:

- VEVENT component: SUMMARY, DESCRIPTION, COMMENT and LOCATION properties

While other items could be localised this is not seen as important until use cases arise. In such a case simply adjusting the component.name and property.name lists to include these will allow expanded localisation.

LANGUAGE Attribute While the iCalendar format allows items to have a language attribute this is not used. The reason being that for most of the items that we localise they are only allowed to occur zero or once. Thus 'summary' would ideally be present in multiple languages in one file, the format does not allow such multiple entries. This is unfortunate as it prevents the creation of a single multilingual iCalendar file.

Future Format Support As this format used vobject which supports various formats including vCard it is possible to expand this format to understand those if needed.

class translate.storage.ical.**icalfile**(*inputfile=None*, ***kwargs*)
An ical file

UnitClass
alias of *icalunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.ical.**icalunit**(*source=None*, ***kwargs*)
An ical entry that is translatable

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.

- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

ini

Class that manages .ini files for translation

a comment ; a comment

[Section] a = a string b : a string

class translate.storage.ini.**Dialect**
Base class for differentiating dialect options and functions

class translate.storage.ini.**DialectDefault**

class translate.storage.ini.**DialectInno**

class translate.storage.ini.**inifile**(*inputfile=None*, *dialect='default'*, ***kwargs*)
An INI file

UnitClass
alias of *iniunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type `TranslationUnit`

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
Parse the given file or file source string.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.ini.**iniunit**(*source=None*, ***kwargs*)
A INI file entry

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.ini.**register_dialect**(*dialect*)
Decorator that registers the dialect.

jsonl10n

Class that manages JSON data files for translation

JSON is an acronym for JavaScript Object Notation, it is an open standard designed for human-readable data interchange.

JSON basic types:

- Number (integer or real)
- String (double-quoted Unicode with backslash escaping)
- Boolean (true or false)
- Array (an ordered sequence of values, comma-separated and enclosed in square brackets)
- Object (a collection of key:value pairs, comma-separated and enclosed in curly braces)
- null

Example:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
```

(continues on next page)

(continued from previous page)

```
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
          "type": "home",
          "number": "212 555-1234"
        },
        {
          "type": "fax",
          "number": "646 555-4567"
        }
    ]
}
```

TODO:

- Handle \u and other escapes in Unicode
- Manage data type storage and conversion. True –> "True" –> True

class translate.storage.jsonl10n.**ARBJsonFile**(*inputfile=None*, *filter=None*, ***kwargs*)

ARB JSON file

See following URLs for doc:

https://github.com/google/app-resource-bundle/wiki/ApplicationResourceBundleSpecification https://flutter.dev/docs/development/accessibility-and-localization/internationalization#appendix-using-the-dart-intl-tools

UnitClass

alias of *ARBJsonUnit*

add_unit_to_index(*unit*)

Add a unit to source and location idexes

addsourceunit(*source*)

Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)

Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)

Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)

Simple detection based on BOM in case chardet is not available.

findid(*id*)

find unit with matching id by checking id_index

findunit(*source*)

Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)

Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**ARBJsonUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, *metadata=None*, ***kwargs*)

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement TranslationUnit.addlocation().

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.jsonl10n.**GoI18NJsonFile**(*inputfile=None*, *filter=None*, ***kwargs*)
go-i18n JSON file

See following URLs for doc:

https://github.com/nicksnyder/go-i18n https://godoc.org/github.com/nicksnyder/go-i18n/v2

UnitClass
alias of *GoI18NJsonUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type `TranslationUnit`

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**GoI18NJsonUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, ***kwargs*)

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement TranslationUnit.addlocation().

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.jsonl10n.**I18NextFile**(*inputfile=None*, *filter=None*, ***kwargs*)
A i18next v3 format, this is nested JSON with several additions.

See https://www.i18next.com/

UnitClass
alias of *I18NextUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**I18NextUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, ***kwargs*)

A i18next v3 format, JSON with plurals.

See https://www.i18next.com/

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement TranslationUnit.addlocation().

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.jsonl10n.**JsonFile**(*inputfile=None*, *filter=None*, ***kwargs*)
A JSON file

UnitClass
alias of *JsonUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**JsonNestedFile**(*inputfile=None*, *filter=None*, ***kwargs*)

A JSON file with nested keys

UnitClass
alias of *JsonNestedUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**JsonNestedUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, ***kwargs*)

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

---

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.jsonl10n.**JsonUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, ***kwargs*)
A JSON entry

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.jsonl10n.**WebExtensionJsonFile**(*inputfile=None*, *filter=None*, ***kwargs*)
WebExtension JSON file

See following URLs for doc:

https://developer.chrome.com/extensions/i18n https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Internationalization

UnitClass
alias of *WebExtensionJsonUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parse the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.jsonl10n.**WebExtensionJsonUnit**(*source=None*, *item=None*, *notes=None*, *placeholders=None*, ***kwargs*)

DefaultDict
alias of collections.OrderedDict

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

getvalue()
Returns dictionary for serialization.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

lisa

Parent class for LISA standards (TMX, TBX, XLIFF)

class translate.storage.lisa.**LISAfile**(*inputfile=None*, *sourcelanguage='en'*, *targetlanguage=None*, ***kwargs*)

A class representing a file store for one of the LISA file formats.

UnitClass
alias of *LISAunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Method to be overridden to initialise headers, etc.

addsourceunit(*source*)
Adds and returns a new unit with the given string as first entry.

addunit(*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out=None*)
Converts to a string containing the file's XML

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.lisa.**LISAunit**(*source*, *empty=False*, ***kwargs*)
A single unit in the file. Provisional work is done to make several languages possible.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

createlanguageNode(*lang*, *text*, *purpose=None*)
Returns a xml Element setup with given parameters to represent a single language entry. Has to be overridden.

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
Returns a list of all nodes that contain per language information.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *`getlocations()`*).

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the "target" string (second language), or alternatively appends to the list

unit_iter()
Iterator that only returns this unit.

mo

Module for parsing Gettext .mo files for translation.

The coding of .mo files was produced from Gettext documentation, Pythons msgfmt.py and by observing and testing existing .mo files in the wild.

The hash algorithm is implemented for MO files, this should result in faster access of the MO file. The hash is optional for Gettext and is not needed for reading or writing MO files, in this implementation it is always on and does produce sometimes different results to Gettext in very small files.

class translate.storage.mo.**mofile**(*inputfile=None*, ***kwargs*)
A class representing a .mo file.

UnitClass
alias of *mounit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getheaderplural()
Returns the nplural and plural values from the header.

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Return the project based on information in the header.

The project is determined in the following sequence:

1. Use the 'X-Project-Style' entry in the header.
2. Use 'Report-Msgid-Bug-To' entry
3. Use the 'X-Accelerator' entry
4. Use the Project ID
5. Analyse the file itself (not yet implemented)

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Return the target language based on information in the header.

The target language is determined in the following sequence:

1. Use the 'Language' entry in the header.
2. Poedit's custom headers.
3. Analysing the 'Language-Team' entry.

getunits()
Return a list of all units in this store.

header()
Returns the header element, or None. Only the first element is allowed to be a header. Note that this could still return an empty header element, if present.

init_headers(*charset='UTF-8'*, *encoding='8bit'*, ***kwargs*)
sets default values for po headers

isempty()
Return True if the object doesn't contain any translation units.

makeheader(***kwargs*)
Create a header for the given filename.

Check .makeheaderdict() for information on parameters.

makeheaderdict(*charset='CHARSET'*, *encoding='ENCODING'*, *project_id_version=None*, *pot_creation_date=None*, *po_revision_date=None*, *last_translator=None*, *language_team=None*, *mime_version=None*, *plural_forms=None*, *report_msgid_bugs_to=None*, ***kwargs*)
Create a header dictionary with useful defaults.

pot_creation_date can be None (current date) or a value (datetime or string) po_revision_date can be None (form), False (=pot_creation_date), True (=now), or a value (datetime or string)

Returns Dictionary with the header items

Return type dict of strings

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

mergeheaders(*otherstore*)
Merges another header with this header.

This header is assumed to be the template.

parse(*input*)
parses the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

parseheader()
Parses the PO header and returns the interpreted values as a dictionary.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Output a string representation of the MO data file

setprojectstyle(*project_style*)
Set the project in the header.

Parameters **project_style** (*str*) – the new project

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*lang*)
Set the target language in the header.

This removes any custom Poedit headers if they exist.

Parameters **lang** (*str*) – the new target language code

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

updatecontributor(*name*, *email=None*)
Add contribution comments if necessary.

updateheader(*add=False*, ***kwargs*)
Updates the fields in the PO style header.

This will create a header if add == True.

updateheaderplural(*nplurals*, *plural*)
Update the Plural-Form PO header.

class translate.storage.mo.**mounit**(*source=None*, ***kwargs*)
A class representing a .mo translation message.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

> **Warning:** This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

> **Note:** Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

> **Note:** Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Is this a header entry?

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Is this message translateable?

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.mo.**mounpack**(*filename='messages.mo'*)
Helper to unpack Gettext MO files into a Python string

mozilla_lang

A class to manage Mozilla .lang files.

See https://github.com/mozilla-l10n/langchecker/wiki/.lang-files-format for specifications on the format.

class translate.storage.mozilla_lang.**LangStore**(*inputfile=None*, *mark_active=False*, ***kwargs*)
We extend TxtFile, since that has a lot of useful stuff for encoding

UnitClass
alias of *LangUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*lines*)
Read in text lines and create txtunits from the blocks of text

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.mozilla_lang.**LangUnit**(*source=None*)
This is just a normal unit with a weird string output

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

odf_io

odf_shared

omegat

Manage the OmegaT glossary format

OmegaT glossary format is used by the OmegaT computer aided translation tool.

It is a bilingual base class derived format with *OmegaTFile* and *OmegaTUnit* providing file and unit level access.

Format Implementation The OmegaT glossary format is a simple Tab Separated Value (TSV) file with the columns: source, target, comment.

The dialect of the TSV files is specified by *OmegaTDialect*.

Encoding The files are either UTF-8 or encoded using the system default. UTF-8 encoded files use the .utf8 extension while system encoded files use the .tab extension.

translate.storage.omegat.**OMEGAT_FIELDNAMES = ['source', 'target', 'comment']**
Field names for an OmegaT glossary unit

class translate.storage.omegat.**OmegaTDialect**
Describe the properties of an OmegaT generated TAB-delimited glossary file.

class translate.storage.omegat.**OmegaTFile**(*inputfile=None*, ***kwargs*)
An OmegaT glossary file

UnitClass
alias of *OmegaTUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parsese the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.omegat.**OmegaTFileTab**(*inputfile=None*, ***kwargs*)
An OmegaT glossary file in the default system encoding

UnitClass
alias of *OmegaTUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parsese the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.omegat.**OmegaTUnit**(*source=None*)
An OmegaT glossary unit

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement TranslationUnit.addlocation().

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

dict
Get the dictionary of values for a OmegaT line

getcontext()
Get the message context.

getdict()
Get the dictionary of values for a OmegaT line

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
```

(continues on next page)

(continued from previous page)

```
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setdict(*newdict*)
Set the dictionary of values for a OmegaT line

Parameters **newdict** (*Dict*) – a new dictionary with OmegaT line elements

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

oo

Classes that hold units of .oo files (oounit) or entire files (oofile).

These are specific .oo files for localisation exported by OpenOffice.org - SDF format (previously knows as GSI files).

The behaviour in terms of escaping is explained in detail in the programming comments.

translate.storage.oo.**escape_help_text**(*text*)
Escapes the help text as it would be in an SDF file.

<, >, " are only escaped in <[[:lower:]]> tags. Some HTML tags make it in in lowercase so those are dealt with. Some OpenOffice.org help tags are not escaped.

translate.storage.oo.**escape_text**(*text*)
Escapes SDF text to be suitable for unit consumption.

translate.storage.oo.**int2byte**()
S.pack(v1, v2, ...) -> bytes

Return a bytes object containing values v1, v2, ... packed according to the format string S.format. See help(struct) for more on format strings.

translate.storage.oo.**makekey**(*ookey*, *long_keys*)
converts an oo key tuple into a unique identifier

Parameters

- **ookey** (*tuple*) – an oo key
- **long_keys** (*Boolean*) – Use long keys

Return type str

Returns unique ascii identifier

translate.storage.oo.**normalizefilename**(*filename*)
converts any non-alphanumeric (standard roman) characters to _

class translate.storage.oo.**oofile**(*input=None*)
this represents an entire .oo file

UnitClass
alias of *oounit*

addline(*thisline*)
adds a parsed line to the file

getoutput(*skip_source=False*, *fallback_lang=None*)
converts all the lines back to tab-delimited form

parse(*input*)
parses lines and adds them to the file

serialize(*out*, *skip_source=False*, *fallback_lang=None*)
convert to a string. double check that unicode is handled

class translate.storage.oo.**ooline**(*parts=None*)
this represents one line, one translation in an .oo file

getkey()
get the key that identifies the resource

getoutput()
return a line in tab-delimited form

getparts()
return a list of parts in this line

gettext()
Obtains the text column and handle escaping.

setparts(*parts*)
create a line from its tab-delimited parts

settext(*text*)
Sets the text column and handle escaping.

text
Obtains the text column and handle escaping.

class translate.storage.oo.**oomultifile**(*filename*, *mode=None*, *multifilestyle='single'*)
this takes a huge GSI file and represents it as multiple smaller files...

createsubfileindex()
reads in all the lines and works out the subfiles

getoofile(*subfile*)
returns an oofile built up from the given subfile's lines

getsubfilename(*line*)
looks up the subfile name for the line

getsubfilesrc(*subfile*)
returns the list of lines matching the subfile

listsubfiles()
returns a list of subfiles in the file

openinputfile(*subfile*)
returns a pseudo-file object for the given subfile

openoutputfile(*subfile*)
returns a pseudo-file object for the given subfile

class translate.storage.oo.**oounit**
this represents a number of translations of a resource

addline(*line*)
add a line to the oounit

getoutput(*skip_source=False*, *fallback_lang=None*)
return the lines in tab-delimited form

translate.storage.oo.**unescape_help_text**(*text*)
Unescapes normal text to be suitable for writing to the SDF file.

translate.storage.oo.**unescape_text**(*text*)
Unescapes SDF text to be suitable for unit consumption.

class translate.storage.oo.**unormalizechar**(*normalchars*)

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

placeables

This module implements basic functionality to support placeables.

A placeable is used to represent things like:

1. Substitutions

   For example, in ODF, footnotes appear in the ODF XML where they are defined; so if we extract a paragraph with some footnotes, the translator will have a lot of additional XML to with; so we separate the footnotes out into separate translation units and mark their positions in the original text with placeables.

2. Hiding of inline formatting data

   The translator doesn't want to have to deal with all the weird formatting conventions of wherever the text came from.

3. Marking variables

   This is an old issue - translators translate variable names which should remain untranslated. We can wrap placeables around variable names to avoid this.

The placeables model follows the XLIFF standard's list of placeables. Please refer to the XLIFF specification to get a better understanding.

base

Contains base placeable classes with names based on XLIFF placeables. See the XLIFF standard for more information about what the names mean.

class translate.storage.placeables.base.**Bpt**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the

parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first (*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset (*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset (*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode (*encoding='utf-8'*)
More `unicode` class emulation.

find (*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with (*x*)
Find all elements in the current sub-tree containing `x`.

flatten (*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data (*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem (*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert (*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between (*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf ()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first (*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map (*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**Ept**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**Ph**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**It**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**G**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**Bx**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**Ex**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**X**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.base.**Sub**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

general

Contains general placeable implementations. That is placeables that does not fit into any other sub-category.

class translate.storage.placeables.general.**AltAttrPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

Placeable for the "alt=..." attributes inside XML tags.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

classmethod parse(*pstr*)
A parser method to extract placeables from a string based on a regular expression. Use this function as the `@parse()` method of a placeable class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.general.**XMLEntityPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)
Placeable handling XML entities (`&xxxxx;`-style entities).

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

classmethod parse(*pstr*)
A parser method to extract placeables from a string based on a regular expression. Use this function as the `@parse()` method of a placeable class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.general.**XMLTagPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)
Placeable handling XML tags.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the

parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first (*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset (*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset (*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode (*encoding='utf-8'*)
More `unicode` class emulation.

find (*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with (*x*)
Find all elements in the current sub-tree containing `x`.

flatten (*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data (*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem (*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert (*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between (*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf ()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first (*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map (*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

classmethod parse(*pstr*)
A parser method to extract placeables from a string based on a regular expression. Use this function as the `@parse()` method of a placeable class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

interfaces

This file contains abstract (semantic) interfaces for placeable implementations.

class translate.storage.placeables.interfaces.**BasePlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

Base class for all placeables.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.interfaces.**InvisiblePlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.interfaces.**MaskingPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.interfaces.**ReplacementPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.interfaces.**SubflowPlaceable**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten (*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data (*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem (*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert (*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between (*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf ()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first (*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map (*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree (*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune ()
Remove unnecessary nodes to make the tree optimal.

remove_type (*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate ()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

lisa

parse

Contains the `parse` function that parses normal strings into StringElem- based "rich" string element trees.

translate.storage.placeables.parse.**parse**(*tree*, *parse_funcs*)
Parse placeables from the given string or sub-tree by using the parsing functions provided.

The output of this function is **heavily** dependent on the order of the parsing functions. This is because of the algorithm used.

An over-simplification of the algorithm: the leaves in the `StringElem` tree are expanded to the output of the first parsing function in `parse_funcs`. The next level of recursion is then started on the new set of leaves with the used parsing function removed from `parse_funcs`.

Parameters **tree** (*unicode|StringElem*) – The string or string element sub-tree to parse.

strelem

Contains the base *StringElem* class that represents a node in a parsed rich-string tree. It is the base class of all placeables.

exception translate.storage.placeables.strelem.**ElementNotFoundError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.storage.placeables.strelem.**StringElem**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)
This class represents a sub-tree of a string parsed into a rich structure. It is also the base class of all placeables.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

has_content = True
Whether this string can have sub-elements.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

iseditable = True
Whether this string should be changable by the user. Not used at the moment.

isfragile = False
Whether this element should be deleted in its entirety when partially deleted. Only checked when `iseditable = False`

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

istranslatable = True
Whether this string is translatable into other languages.

isvisible = True
Whether this string should be visible to the user. Not used at the moment.

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

classmethod parse(*pstr*)
Parse an instance of this class from the start of the given string. This method should be implemented by any sub-class that wants to parseable by *translate.storage.placeables.parse*.

Parameters **pstr** (*unicode*) – The string to parse into an instance of this class.

Returns An instance of the current class, or `None` if the string not parseable by this class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

renderer = None
An optional function that returns the Unicode representation of the string.

sub = []
The sub-elements that make up this this string.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

terminology

Contains the placeable that represents a terminology term.

class translate.storage.placeables.terminology.**TerminologyPlaceable**(**args*, ***kwargs*)
Terminology distinguished from the rest of a string by being a placeable.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

matchers = []
A list of matcher objects to use to identify terminology.

classmethod parse(*pstr*)
Parse an instance of this class from the start of the given string. This method should be implemented by any sub-class that wants to parseable by *translate.storage.placeables.parse*.

Parameters **pstr** (*unicode*) – The string to parse into an instance of this class.

Returns An instance of the current class, or `None` if the string not parseable by this class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

translations = []
The available translations for this placeable.

xliff

Contains XLIFF-specific placeables.

class translate.storage.placeables.xliff.**Bpt**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**Ept**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**X**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**Bx**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**Ex**(*id=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**G**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**It**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**Sub**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**Ph**(*sub=None*, *id=None*, *rid=None*, *xid=None*, ***kwargs*)

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

class translate.storage.placeables.xliff.**UnknownXML**(*sub=None*, *id=None*, *rid=None*, *xid=None*, *xml_node=None*, ***kwargs*)
Placeable for unrecognized or umimplemented XML nodes. It's main purpose is to preserve all associated XML data.

apply_to_strings(*f*)
Apply `f` to all actual strings in the tree.

Parameters **f** – Must take one (str or unicode) argument and return a string or unicode.

copy()
Returns a copy of the sub-tree. This should be overridden in sub-classes with more data.

Note: `self.renderer` is **not** copied.

---

delete_range(*start_index*, *end_index*)
Delete the text in the range given by the string-indexes `start_index` and `end_index`.

Partial nodes will only be removed if they are editable.

Returns A `StringElem` representing the removed sub-string, the parent node from which it was deleted as well as the offset at which it was deleted from. `None` is returned for the parent value if the root was deleted. If the parent and offset values are not `None`, `parent.insert(offset, deleted)` effectively undoes the delete.

depth_first(*filter=None*)
Returns a list of the nodes in the tree in depth-first order.

elem_at_offset(*offset*)
Get the `StringElem` in the tree that contains the string rendered at the given offset.

elem_offset(*elem*)
Find the offset of `elem` in the current tree.

This cannot be reliably used if `self.renderer` is used and even less so if the rendering function renders the string differently upon different calls. In Virtaal the `StringElemGUI.index()` method is used as replacement for this one.

Returns The string index where element `e` starts, or -1 if `e` was not found.

encode(*encoding='utf-8'*)
More `unicode` class emulation.

find(*x*)
Find sub-string `x` in this string tree and return the position at which it starts.

find_elems_with(*x*)
Find all elements in the current sub-tree containing `x`.

flatten(*filter=None*)
Flatten the tree by returning a depth-first search over the tree's leaves.

get_index_data(*index*)
Get info about the specified range in the tree.

Returns

A dictionary with the following items:

- *elem*: The element in which `index` resides.
- *index*: Copy of the `index` parameter
- *offset*: The offset of `index` into `'elem'`.

get_parent_elem(*child*)
Searches the current sub-tree for and returns the parent of the `child` element.

insert(*offset*, *text*, *preferred_parent=None*)
Insert the given text at the specified offset of this string-tree's string (Unicode) representation.

insert_between(*left*, *right*, *text*)
Insert the given text between the two parameter `StringElem`s.

isleaf()
Whether or not this instance is a leaf node in the `StringElem` tree.

A node is a leaf node if it is a `StringElem` (not a sub-class) and contains only sub-elements of type `str` or `unicode`.

Return type bool

iter_depth_first(*filter=None*)
Iterate through the nodes in the tree in dept-first order.

map(*f*, *filter=None*)
Apply `f` to all nodes for which `filter` returned `True` (optional).

classmethod parse(*pstr*)
Parse an instance of this class from the start of the given string. This method should be implemented by any sub-class that wants to parseable by *translate.storage.placeables.parse*.

Parameters **pstr** (*unicode*) – The string to parse into an instance of this class.

Returns An instance of the current class, or `None` if the string not parseable by this class.

print_tree(*indent=0*, *verbose=False*)
Print the tree from the current instance's point in an indented manner.

prune()
Remove unnecessary nodes to make the tree optimal.

remove_type(*ptype*)
Replace nodes with type `ptype` with base `StringElem`s, containing the same sub-elements. This is only applicable to elements below the element tree root node.

translate()
Transform the sub-tree according to some class-specific needs. This method should be either overridden in implementing sub-classes or dynamically replaced by specific applications.

Returns The transformed Unicode string representing the sub-tree.

php

Classes that hold units of PHP localisation files *phpunit* or entire files *phpfile*. These files are used in translating many PHP based applications.

Only PHP files written with these conventions are supported:

```
<?php
$lang['item'] = "vale";  # Array of values
$some_entity = "value";  # Named variables
define("ENTITY", "value");
$lang = array(
   'item1' => 'value1'    ,   #Supports space before comma
   'item2' => 'value2',
);
$lang = array(    # Nested arrays
   'item1' => 'value1',
   'item2' => array(
      'key' => 'value'    ,   #Supports space before comma
      'key2' => 'value2',
   ),
);
```

Nested arrays without key for nested array are not supported:

```
<?php
$lang = array(array('key' => 'value'));
```

The working of PHP strings and specifically the escaping conventions which differ between single quote (‘) and double quote (“) characters are implemented as outlined in the PHP documentation for the String type.

class translate.storage.php.**LaravelPHPFile**(*inputfile=None*, ***kwargs*)

UnitClass
alias of *LaravelPHPUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object’s list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*phpsrc*)
Read the source of a PHP file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Convert the units back to lines.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.php.**LaravelPHPUnit**(*source=''*)

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput(*indent=''*, *name=None*)
Convert the unit back into formatted lines for a php file.

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Return whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.php.**phpdecode**(*text*, *quotechar="'"*)
Convert PHP escaped string to a Python string.

translate.storage.php.**phpencode**(*text*, *quotechar="'"*)
Convert Python string to PHP escaping.

The encoding is implemented for 'single quote' and "double quote" syntax.

heredoc and nowdoc are not implemented and it is not certain whether this would ever be needed for PHP localisation needs.

class translate.storage.php.**phpfile**(*inputfile=None*, ***kwargs*)
This class represents a PHP file, made up of phpunits.

UnitClass
alias of *phpunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*phpsrc*)
Read the source of a PHP file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Convert the units back to lines.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.php.**phpunit**(*source=''*)
A unit of a PHP file: a name, a value, and any comments associated.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput(*indent=''*, *name=None*)
Convert the unit back into formatted lines for a php file.

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Return whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.php.**wrap_production**(*func*)
Decorator for production functions to store lexer positions.

pocommon

translate.storage.pocommon.**extract_msgid_comment**(*text*)
The one definitive way to extract a msgid comment out of an unescaped unicode string that might contain it.

Return type unicode

class translate.storage.pocommon.**pofile**(*inputfile=None*, *noheader=False*, ***kwargs*)

UnitClass
alias of *translate.storage.base.TranslationUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getheaderplural()
Returns the nplural and plural values from the header.

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Return the project based on information in the header.

The project is determined in the following sequence:

1. Use the 'X-Project-Style' entry in the header.
2. Use 'Report-Msgid-Bug-To' entry
3. Use the 'X-Accelerator' entry
4. Use the Project ID
5. Analyse the file itself (not yet implemented)

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Return the target language based on information in the header.

The target language is determined in the following sequence:

1. Use the 'Language' entry in the header.
2. Poedit's custom headers.
3. Analysing the 'Language-Team' entry.

getunits()
Return a list of all units in this store.

header()
Returns the header element, or None. Only the first element is allowed to be a header. Note that this could still return an empty header element, if present.

init_headers(*charset='UTF-8'*, *encoding='8bit'*, ***kwargs*)
sets default values for po headers

isempty()
Return True if the object doesn't contain any translation units.

makeheader(***kwargs*)
Create a header for the given filename.

Check .makeheaderdict() for information on parameters.

makeheaderdict(*charset='CHARSET'*, *encoding='ENCODING'*, *project_id_version=None*, *pot_creation_date=None*, *po_revision_date=None*, *last_translator=None*, *language_team=None*, *mime_version=None*, *plural_forms=None*, *report_msgid_bugs_to=None*, ***kwargs*)
Create a header dictionary with useful defaults.

pot_creation_date can be None (current date) or a value (datetime or string) po_revision_date can be None (form), False (=pot_creation_date), True (=now), or a value (datetime or string)

Returns Dictionary with the header items

Return type dict of strings

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

mergeheaders(*otherstore*)
Merges another header with this header.

This header is assumed to be the template.

parse(*data*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

parseheader()
Parses the PO header and returns the interpreted values as a dictionary.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project in the header.

Parameters **project_style** (*str*) – the new project

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*lang*)
Set the target language in the header.

This removes any custom Poedit headers if they exist.

Parameters **lang** (*str*) – the new target language code

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

updatecontributor(*name*, *email=None*)
Add contribution comments if necessary.

updateheader(*add=False*, ***kwargs*)
Updates the fields in the PO style header.

This will create a header if add == True.

updateheaderplural(*nplurals*, *plural*)
Update the Plural-Form PO header.

class translate.storage.pocommon.**pounit**(*source=None*)

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*present=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review. Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

translate.storage.pocommon.**quote_plus**(*text*)
Quote the query fragment of a URL; replacing ' ' with '+'

translate.storage.pocommon.**unquote_plus**(*text*)
unquote('%7e/abc+def') -> '~/abc def'

poheader

class that handles all header functions for a header in a po file

translate.storage.poheader.**parseheaderstring**(*input*)
Parses an input string with the definition of a PO header and returns the interpreted values as a dictionary.

class translate.storage.poheader.**poheader**
This class implements functionality for manipulation of po file headers. This class is a mix-in class and useless on its own. It must be used from all classes which represent a po file

getheaderplural()
Returns the nplural and plural values from the header.

getprojectstyle()
Return the project based on information in the header.

The project is determined in the following sequence:

1. Use the 'X-Project-Style' entry in the header.
2. Use 'Report-Msgid-Bug-To' entry
3. Use the 'X-Accelerator' entry
4. Use the Project ID
5. Analyse the file itself (not yet implemented)

gettargetlanguage()
Return the target language based on information in the header.

The target language is determined in the following sequence:

1. Use the 'Language' entry in the header.
2. Poedit's custom headers.
3. Analysing the 'Language-Team' entry.

header()
Returns the header element, or None. Only the first element is allowed to be a header. Note that this could still return an empty header element, if present.

init_headers(*charset='UTF-8'*, *encoding='8bit'*, ***kwargs*)
sets default values for po headers

makeheader(***kwargs*)
Create a header for the given filename.

Check .makeheaderdict() for information on parameters.

makeheaderdict(*charset='CHARSET'*, *encoding='ENCODING'*, *project_id_version=None*, *pot_creation_date=None*, *po_revision_date=None*, *last_translator=None*, *language_team=None*, *mime_version=None*, *plural_forms=None*, *report_msgid_bugs_to=None*, ***kwargs*)
Create a header dictionary with useful defaults.

pot_creation_date can be None (current date) or a value (datetime or string) po_revision_date can be None (form), False (=pot_creation_date), True (=now), or a value (datetime or string)

Returns Dictionary with the header items

Return type dict of strings

mergeheaders(*otherstore*)
Merges another header with this header.

This header is assumed to be the template.

parseheader()
Parses the PO header and returns the interpreted values as a dictionary.

setprojectstyle(*project_style*)
Set the project in the header.

Parameters **project_style** (*str*) – the new project

settargetlanguage(*lang*)
Set the target language in the header.

This removes any custom Poedit headers if they exist.

Parameters **lang** (*str*) – the new target language code

updatecontributor(*name*, *email=None*)
Add contribution comments if necessary.

updateheader(*add=False*, ***kwargs*)
Updates the fields in the PO style header.

This will create a header if add == True.

updateheaderplural(*nplurals*, *plural*)
Update the Plural-Form PO header.

translate.storage.poheader.**tzstring**()
Returns the timezone as a string in the format [+-]0000, eg +0200.

Return type str

translate.storage.poheader.**update**(*existing*, *add=False*, ***kwargs*)
Update an existing header dictionary with the values in kwargs, adding new values only if add is true.

Returns Updated dictionary of header entries

Return type dict of strings

poparser

translate.storage.poparser.**decode_header**(*unit*, *decode*)
The header has been arbitrarily decoded with a single-byte encoding. We re-encode it to decode values with the proper encoding defined in the header (using decode_list above).

translate.storage.poparser.**read_obsolete_lines**(*parse_state*)
Read all the lines belonging to the current unit if obsolete.

translate.storage.poparser.**read_prevmsgid_lines**(*parse_state*)
Read all the lines belonging starting with #|. These lines contain the previous msgid and msgctxt info. We strip away the leading '#| ' and read until we stop seeing #|.

po

A class loader that will load C or Python implementations of the PO class depending on the USECPO variable.

Use the environment variable `USECPO=2` (or `USECPO=1`) to choose the C implementation which uses Gettext's libgettextpo for high parsing speed. Otherwise the local Python based parser is used (slower but very well tested).

poxliff

XLIFF classes specifically suited for handling the PO representation in XLIFF.

This way the API supports plurals as if it was a PO file, for example.

class translate.storage.poxliff.**PoXliffFile**(**args*, ***kwargs*)
a file for the po variant of Xliff files

UnitClass
alias of *PoXliffUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Initialise the file header.

addplural(*source*, *target*, *filename*, *createifmissing=False*)
This method should now be unnecessary, but is left for reference

addsourceunit(*source*, *filename='NoName'*, *createifmissing=False*)
adds the given trans-unit to the last used body node if the filename has changed it uses the slow method instead (will create the nodes required if asked). Returns success

addunit(*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

createfilenode(*filename*, *sourcelanguage='en-US'*, *datatype='po'*)
creates a filenode with the given filename. All parameters are needed for XLIFF compliance.

creategroup(*filename='NoName'*, *createifmissing=False*, *restype=None*)
adds a group tag into the specified file

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getbodynode(*filenode*, *createifmissing=False*)
finds the body node for the given filenode

getdatatype(*filename=None*)
Returns the datatype of the stored file. If no filename is given, the datatype of the first file is given.

getdate(*filename=None*)
Returns the date attribute for the file.

If no filename is given, the date of the first file is given. If the date attribute is not specified, None is returned.

Returns Date attribute of file

Return type Date or None

getfilename(*filenode*)
returns the name of the given file

getfilenames()
returns all filenames in this XLIFF file

getfilenode(*filename*, *createifmissing=False*)
finds the filenode with the given name

getheadernode(*filenode*, *createifmissing=False*)
finds the header node for the given filenode

getheaderplural()
Returns the nplural and plural values from the header.

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

header()
Returns the header element, or None. Only the first element is allowed to be a header. Note that this could still return an empty header element, if present.

init_headers(*charset='UTF-8'*, *encoding='8bit'*, ***kwargs*)
sets default values for po headers

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeheader(***kwargs*)
Create a header for the given filename.

Check .makeheaderdict() for information on parameters.

makeheaderdict(*charset='CHARSET'*, *encoding='ENCODING'*, *project_id_version=None*, *pot_creation_date=None*, *po_revision_date=None*, *last_translator=None*, *language_team=None*, *mime_version=None*, *plural_forms=None*, *report_msgid_bugs_to=None*, ***kwargs*)

Create a header dictionary with useful defaults.

pot_creation_date can be None (current date) or a value (datetime or string) po_revision_date can be None (form), False (=pot_creation_date), True (=now), or a value (datetime or string)

Returns Dictionary with the header items

Return type dict of strings

makeindex()

Indexes the items in this store. At least .sourceindex should be useful.

merge_on

The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

mergeheaders(*otherstore*)

Merges another header with this header.

This header is assumed to be the template.

namespaced(*name*)

Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)

Populates this object from the given xml string

classmethod parsefile(*storefile*)

Reads the given file (or opens the given filename) and parses back to an object.

parseheader()

Parses the PO header and returns the interpreted values as a dictionary.

classmethod parsestring(*storestring*)

Parses the string to return the correct file object

remove_unit_from_index(*unit*)

Remove a unit from source and locaton indexes

removedefaultfile()

We want to remove the default file-tag as soon as possible if we know if still present and empty.

removeunit(*unit*)

Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a string containing the file's XML

setfilename(*filenode*, *filename*)
set the name of the given file

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*language*)
Set the source language for this store.

settargetlanguage(*language*)
Set the target language for this store.

switchfile(*filename*, *createifmissing=False*)
Adds the given trans-unit (will create the nodes required if asked).

Returns Success

Return type Boolean

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

updatecontributor(*name*, *email=None*)
Add contribution comments if necessary.

updateheader(*add=False*, ***kwargs*)
Updates the fields in the PO style header.

This will create a header if add == True.

updateheaderplural(*nplurals*, *plural*)
Update the Plural-Form PO header.

class translate.storage.poxliff.**PoXliffUnit**(*source=None*, *empty=False*, ***kwargs*)
A class to specifically handle the plural units created from a po file.

addalttrans(*txt*, *origin=None*, *lang=None*, *sourcetxt=None*, *matchquality=None*)
Adds an alt-trans tag and alt-trans components to the unit.

Parameters **txt** (*String*) – Alternative translation of the source text.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in a "note" tag

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

correctorigin(*node*, *origin*)
Check against node tag's origin (e.g note or alt-trans)

createcontextgroup(*name*, *contexts=None*, *purpose=None*)
Add the context group to the trans-unit with contexts a list with (type, text) tuples describing each context.

createlanguageNode(*lang*, *text*, *purpose*)
Returns an xml Element setup with given parameters.

delalttrans(*alternative*)
Removes the supplied alternative from the list of alt-trans tags

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

get_rich_target(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

getalttrans(*origin=None*)
Returns <alt-trans> for the given origin as a list of units. No origin means all alternatives.

getautomaticcomments()
Returns the automatic comments (x-po-autocomment), which corresponds to the #. style po comments.

getcontext()
Get the message context.

getcontextgroups(*name*)
Returns the contexts in the context groups with the specified name

geterrors()
Get all error messages.

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
We override this to get source and target nodes.

getlocations()
Returns all the references (source locations)

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *`getlocations()`*).

getrestype()
returns the restype attribute in the trans-unit tag

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

gettranslatorcomments()
Returns the translator comments (x-po-trancomment), which corresponds to the # style po comments.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isapproved()
States whether this unit is approved.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
States whether this unit needs to be reviewed

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markapproved(*value=True*)
Mark this unit as approved.

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*id*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the target string to the given value.

unit_iter()
Iterator that only returns this unit.

project

class translate.storage.project.**Project**(*projstore=None*)
Manages a project store as well as the processes involved in a project workflow.

add_source(*srcfile*, *src_fname=None*)
Proxy for `self.store.append_sourcefile()`.

add_source_convert(*srcfile*, *src_fname=None*, *convert_options=None*, *extension=None*)
Convenience method that calls *add_source()* and *convert_forward()* and returns the results from both.

close()
Proxy for `self.store.close()`.

convert_forward(*input_fname*, *template=None*, *output_fname=None*, ***options*)
Convert the given input file to the next type in the process:

Source document (eg. ODT) -> Translation file (eg. XLIFF) -> Translated document (eg. ODT).

Parameters

- **input_fname** (*basestring*) – The project name of the file to convert
- **convert_options** (*Dictionary (optional)*) – Passed as-is to `translate.convert.factory.convert()`.

Returns 2-tuple the converted file object and its project name.

export_file(*fname*, *destfname*)
Export the file with the specified filename to the given destination. This method will raise *FileNotInProjectError* via the call to *get_file()* if *fname* is not found in the project.

get_file(*fname*)
Proxy for `self.store.get_file()`.

get_proj_filename(*realfname*)
Proxy for `self.store.get_proj_filename()`.

get_real_filename(*projfname*)
Try and find a real file name for the given project file name.

remove_file(*projfname*, *ftype=None*)
Proxy for `self.store.remove_file()`.

save(*filename=None*)
Proxy for `self.store.save()`.

update_file(*proj_fname*, *infile*)
Proxy for `self.store.update_file()`.

projstore

exception translate.storage.projstore.**FileExistsInProjectError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception translate.storage.projstore.**FileNotInProjectError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.storage.projstore.**ProjectStore**
Basic project file container.

append_file(*afile*, *fname*, *ftype='trans'*, *delete_orig=False*)
Append the given file to the project with the given filename, marked to be of type `ftype` ('src', 'trans', 'tgt').

Parameters **delete_orig** (*bool*) – Whether or not the original (given) file should be deleted after being appended. This is set to `True` by `convert_forward()` . Not used in this class.

get_file(*fname*, *mode='rb'*)
Retrieve the file with the given name from the project store.

The file is looked up in the `self._files` dictionary. The values in this dictionary may be `None`, to indicate that the file is not cacheable and needs to be retrieved in a special way. This special way must be defined in this method of sub-classes. The value may also be a string, which indicates that it is a real file accessible via `open`.

Parameters **mode** (*str*) – The mode in which to re-open the file (if it is closed).

get_filename_type(*fname*)
Get the type of file ('src', 'trans', 'tgt') with the given name.

get_proj_filename(*realfname*)
Try and find a project file name for the given real file name.

load(**args*, ***kwargs*)
Load the project in some way. Undefined for this (base) class.

remove_file(*fname*, *ftype=None*)
Remove the file with the given project name from the project. If the file type ('src', 'trans' or 'tgt') is not given, it is guessed.

save (*filename=None*, **args*, ***kwargs*)
Save the project in some way. Undefined for this (base) class.

sourcefiles
Read-only access to `self._sourcefiles`.

targetfiles
Read-only access to `self._targetfiles`.

transfiles
Read-only access to `self._transfiles`.

update_file (*pfname*, *infile*)
Remove the project file with name `pfname` and add the contents from `infile` to the project under the same file name.

Returns the results from *`ProjectStore.append_file()`*.

properties

Classes that hold units of .properties, and similar, files that are used in translating Java, Mozilla, MacOS and other software.

The *`propfile`* class is a monolingual class with *`propunit`* providing unit level access.

The .properties store has become a general key value pair class with *`Dialect`* providing the ability to change the behaviour of the parsing and handling of the various dialects.

Currently we support:

- Java .properties
- Mozilla .properties
- Adobe Flex files
- MacOS X .strings files
- Skype .lang files
- XWiki .properties

The following provides references and descriptions of the various dialects supported:

Java Java .properties are supported completely except for the ability to drop pairs that are not translated.

The following .properties file description gives a good references to the .properties specification.

Properties file may also hold Java MessageFormat messages. No special handling is provided in this storage class for MessageFormat, but this may be implemented in future.

All delimiter types, comments, line continuations and spaces handling in delimeters are supported.

Mozilla Mozilla files use '=' as a delimiter, are UTF-8 encoded and thus don't need \u escaping. Any \U values will be converted to correct Unicode characters.

Strings Mac OS X strings files are implemented using these two articles as references.

Flex Adobe Flex files seem to be normal .properties files but in UTF-8 just like Mozilla files. This page provides the information used to implement the dialect.

Skype Skype .lang files seem to be UTF-16 encoded .properties files.

XWiki XWiki translations files are standard Java .properties but with specific escaping support for simple quotes, and support of missing translations. This XWiki document provides the information used to implement the dialect.

A simple summary of what is permissible follows.

Comments supported:

```
# a comment
// a comment (only at the beginning of a line)

# The following are # escaped to render in docs
# ! is standard but not widely supported
#! a comment
# /* is non-standard but used on some implementations
#/* a comment (not across multiple lines) */
```

Name and Value pairs:

```
# Delimiters
key = value
key : value
# Whitespace delimiter
# key[sp]value

# Space in key and around value
\ key\ = \ value

# Note that the b and c are escaped for reST rendering
b = a string with escape sequences \\t \\n \\r \\\\ \\" \\' \\ (space) \u0123
c = a string with a continuation line \\
    continuation line

# Special cases
# key with no value
//key (escaped; doesn't render in docs)
# value no key (extractable in prop2po but not mergeable in po2prop)
=value

# .strings specific
"key" = "value";
```

class translate.storage.properties.**Dialect**

Settings for the various behaviours in key=value files.

classmethod encode(*string*, *encoding=None*)

Encode the string

classmethod find_delimiter(*line*)

Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)

Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectFlex**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectGaia**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectGwt**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectJava**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectJavaUtf16**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectJavaUtf8**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectJoomla**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectMozilla**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectSkype**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectStrings**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectStringsUtf8**

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**DialectXWiki**
XWiki dialect is mainly a Java properties behaviour but with special handling of simple quotes: they are escaped by doubling them when an argument on the form "{X}" is provided, X being a number.

classmethod encode(*string*, *encoding=None*)
Encode the string

classmethod find_delimiter(*line*)
Find the type and position of the delimiter in a property line.

Property files can be delimited by "=", ":" or whitespace (space for now). We find the position of each delimiter, then find the one that appears first.

Parameters

- **line** (*str*) – A properties line
- **delimiters** (*list*) – valid delimiters

Returns delimiter character and offset within *line*

Return type Tuple (delimiter char, Offset Integer)

classmethod key_strip(*key*)
Strip unneeded characters from the key

classmethod value_strip(*value*)
Strip unneeded characters from the value

class translate.storage.properties.**XWikiFullPage**(**args*, ***kwargs*)
Represents a full XWiki Page translation: this file does not contains properties but its whole content needs to be translated. More information on https://dev.xwiki.org/xwiki/bin/view/Community/XWiki%20Translations%20Formats/#HXWikiFullContentTranslation

UnitClass
alias of *xwikiunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**XWikiPageProperties**(**args*, ***kwargs*)
Represents an XWiki Page containing translation properties as described in https://dev.xwiki.org/xwiki/bin/view/Community/XWiki%20Translations%20Formats/#HXWikiPageProperties

UnitClass
alias of *xwikiunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

translate.storage.properties.**accesskeysuffixes = ('.accesskey', '.accessKey', '.akey')**
Accesskey Suffixes: entries with this suffix may be combined with labels ending in labelsuffixes into accelerator notation

class translate.storage.properties.**gwtfile**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

translate.storage.properties.**is_comment_end**(*line*)
Determine whether a *line* ends a new multi-line comment.

Parameters **line** (*unicode*) – A properties line

Returns True if line ends a new multi-line comment

Return type bool

translate.storage.properties.**is_comment_one_line**(*line*)
Determine whether a *line* is a one-line comment.

Parameters **line** (*unicode*) – A properties line

Returns True if line is a one-line comment

Return type bool

translate.storage.properties.**is_comment_start**(*line*)
Determine whether a *line* starts a new multi-line comment.

Parameters **line** (*unicode*) – A properties line

Returns True if line starts a new multi-line comment

Return type bool

translate.storage.properties.**is_line_continuation**(*line*)
Determine whether *line* has a line continuation marker.

.properties files can be terminated with a backslash () indicating that the 'value' continues on the next line. Continuation is only valid if there are an odd number of backslashses (an even number would result in a set of N/2 slashes not an escape)

Parameters **line** (*str*) – A properties line

Returns Does *line* end with a line continuation

Return type Boolean

class translate.storage.properties.**javafile**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**javautf16file**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**javautf8file**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**joomlafile**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

translate.storage.properties.**labelsuffixes = ('.label', '.title')**
Label suffixes: entries with this suffix are able to be comibed with accesskeys found in in entries ending with `accesskeysuffixes`

class translate.storage.properties.**propfile**(*inputfile=None*, *personality='java'*, *encoding=None*)
this class represents a .properties file, made up of propunits

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**proppluralunit**(*source=''*, *personality='java'*)

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural(*key=None*)
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
returns whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.properties.**propunit**(*source=''*, *personality='java'*)
An element of a properties file i.e. a name and value, and any comments associated.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

---

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

classmethod get_missing_part()
Return the string representing a missing translation.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput()
Convert the element back into formatted lines for a .properties file

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
returns whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

classmethod represents_missing(*line*)
The line represents a missing translation

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

classmethod strip_missing_part(*line*)
Remove the missing prefix from the line.

unit_iter()
Iterator that only returns this unit.

translate.storage.properties.**register_dialect**(*dialect*)
Decorator that registers the dialect.

class translate.storage.properties.**stringsfile**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**stringsutf8file**(**args*, ***kwargs*)

UnitClass
alias of *propunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**xwikifile**(**args*, ***kwargs*)

UnitClass
alias of *xwikiunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*propsrc*)
Read the source of a properties file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.properties.**xwikiunit**(*source=''*, *personality='xwiki'*)

Represents an XWiki translation unit. The difference with a propunit is twofold:

1. the dialect used is xwiki for simple quote escape handling
2. missing translations are output with a dedicated "### Missing: " prefix

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement TranslationUnit.addlocation().

> **Warning:** This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

classmethod get_missing_part()
Return the string representing a missing translation.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

> **Note:** Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput()
Convert the element back into formatted lines for a .properties file

gettargetlen()
Returns the length of the target string.

Return type Integer

> **Note:** Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
returns whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

classmethod represents_missing(*line*)
Return true if the line represents a missing translation

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

classmethod strip_missing_part(*line*)
Remove the missing prefix from the line.

unit_iter()
Iterator that only returns this unit.

pypo

Classes that hold units of Gettext .po files (pounit) or entire files (pofile).

class translate.storage.pypo.**PoWrapper**(*width=77*, *replace_whitespace=False*, *expand_tabs=False*, *drop_whitespace=False*)

fill(*text : string*) → string
Reformat the single paragraph in 'text' to fit in lines of no more than 'self.width' columns, and return a new string containing the entire wrapped paragraph.

wrap(*text : string*) → [string]
Reformat the single paragraph in 'text' so it fits in lines of no more than 'self.width' columns, and return a list of wrapped lines. Tabs in 'text' are expanded with string.expandtabs(), and all other whitespace characters (including newline) are converted to space.

translate.storage.pypo.**escapeforpo**(*line*)
Escapes a line for po format. assumes no occurs in the line.

param line unescaped text

translate.storage.pypo.**lsep = '\n#: '**
Separator for #: entries

class translate.storage.pypo.**pofile**(*inputfile=None*, *width=None*, ***kwargs*)
A .po file containing various units

UnitClass
alias of *pounit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

decode(*lines*)
decode any non-unicode strings in lines with self.encoding

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

encode(*lines*)
encode any unicode strings in lines in self.encoding

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getheaderplural()
Returns the nplural and plural values from the header.

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Return the project based on information in the header.

The project is determined in the following sequence:

1. Use the 'X-Project-Style' entry in the header.
2. Use 'Report-Msgid-Bug-To' entry
3. Use the 'X-Accelerator' entry

4. Use the Project ID
5. Analyse the file itself (not yet implemented)

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Return the target language based on information in the header.

The target language is determined in the following sequence:

1. Use the 'Language' entry in the header.
2. Poedit's custom headers.
3. Analysing the 'Language-Team' entry.

getunits()
Return a list of all units in this store.

header()
Returns the header element, or None. Only the first element is allowed to be a header. Note that this could still return an empty header element, if present.

init_headers(*charset='UTF-8'*, *encoding='8bit'*, ***kwargs*)
sets default values for po headers

isempty()
Return True if the object doesn't contain any translation units.

makeheader(***kwargs*)
Create a header for the given filename.

Check .makeheaderdict() for information on parameters.

makeheaderdict(*charset='CHARSET'*, *encoding='ENCODING'*, *project_id_version=None*, *pot_creation_date=None*, *po_revision_date=None*, *last_translator=None*, *language_team=None*, *mime_version=None*, *plural_forms=None*, *report_msgid_bugs_to=None*, ***kwargs*)
Create a header dictionary with useful defaults.

pot_creation_date can be None (current date) or a value (datetime or string) po_revision_date can be None (form), False (=pot_creation_date), True (=now), or a value (datetime or string)

Returns Dictionary with the header items

Return type dict of strings

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

mergeheaders(*otherstore*)
Merges another header with this header.

This header is assumed to be the template.

parse(*input*)
Parses the given file or file source string.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

parseheader()
Parses the PO header and returns the interpreted values as a dictionary.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeduplicates(*duplicatestyle='merge'*)
Make sure each msgid is unique ; merge comments etc from duplicates into original

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write to file

setprojectstyle(*project_style*)
Set the project in the header.

Parameters **project_style** (*str*) – the new project

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*lang*)
Set the target language in the header.

This removes any custom Poedit headers if they exist.

Parameters **lang** (*str*) – the new target language code

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

updatecontributor(*name*, *email=None*)
Add contribution comments if necessary.

updateheader(*add=False*, ***kwargs*)
Updates the fields in the PO style header.

This will create a header if add == True.

updateheaderplural(*nplurals*, *plural*)
Update the Plural-Form PO header.

class translate.storage.pypo.**pounit**(*source=None*, *wrapper=None*, ***kwargs*)

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

addlocation(*location*)
Add a location to sourcecomments in the PO unit

Parameters **location** (*String*) – Text location e.g. 'file.c:23' does not include #:

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
This is modeled on the XLIFF method.

See *translate.storage.xliff.xliffunit.addnote()*

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getalttrans()
Return a list of alternate units.

Previous msgid and current msgstr is combined to form a single alternative unit.

getcontext()
Get the message context.

geterrors()
Get all error messages.

getid()
Returns a unique identifier for this unit.

getlocations()
Get a list of locations from sourcecomments in the PO unit

rtype: List return: A list of the locations with '#: ' stripped

getnotes(*origin=None*)
Return comments based on origin value.

Parameters **origin** – programmer, developer, source code, translator or None

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasmarkedcomment(*commentmarker*)
Check whether the given comment marker is present.

These should appear as:

```
# (commentmarker) ...
```

hasplural()
returns whether this pounit contains plural strings...

hastypecomment(*typecomment*, *parsed=None*)
Check whether the given type comment is present

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Makes this unit obsolete

markfuzzy(*present=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review. Adds an optional explanation as a note.

merge(*otherpo*, *overwrite=False*, *comments=True*, *authoritative=False*)
Merges the otherpo (with the same msgid) into this one.

Overwrite non-blank self.msgstr only if overwrite is True merge comments only if comments is True

msgidcomment
Extract KDE style msgid comments from the unit.

Return type String

Returns Returns the extracted msgidcomments found in this unit's msgid.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

prev_source
Returns the unescaped msgid

removenotes(*origin=None*)
Remove all the translator's notes (other comments)

resurrect()
Makes an obsolete unit normal

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settypecomment(*typecomment*, *present=True*)
Alters whether a given typecomment is present

source
Returns the unescaped msgid

target
Returns the unescaped msgstr

unit_iter()
Iterator that only returns this unit.

translate.storage.pypo.**quoteforpo**(*text*, *wrapper_obj=None*)
Quotes the given text for a PO file, returning quoted and escaped lines

translate.storage.pypo.**splitlines**(*text*)
Split lines based on first newline char.

Can not use univerzal newlines as they match any newline like character inside text and that breaks on files with unix newlines and LF chars inside comments.

The code looks for first msgid and looks for newline used after it. This should safely cover weird newlines used in comments or filenames, while properly parsing po files with any newlines.

translate.storage.pypo.**unescape**(*line*)
Unescape the given line.

Quotes on either side should already have been removed.

qm

Module for parsing Qt .qm files.

Note: Based on documentation from Gettext's .qm implementation (see *write-qt.c*) and on observation of the output of lrelease.

Note: Certain deprecated section tags are not implemented. These will break and print out the missing tag. They are easy to implement and should follow the structure in 03 (Translation). We could find no examples that use these so we'd rather leave it unimplemented until we actually have test data.

Note: Many .qm files are unable to be parsed as they do not have the source text. We assume that since they use a hash table to lookup the data there is actually no need for the source text. It seems however that in Qt4's lrelease all data is included in the resultant .qm file.

Note: We can only parse, not create, a .qm file. The main issue is that we need to implement the hashing algorithm (which seems to be identical to the Gettext hash algorithm). Unlike Gettext it seems that the hash is required, but that has not been validated.

Note: The code can parse files correctly. But it could be cleaned up to be more readable, especially the part that breaks the file into sections.

http://qt.gitorious.org/+kde-developers/qt/kde-qt/blobs/master/tools/linguist/shared/qm.cpp Plural information QLocale languages

class translate.storage.qm.**qmfile**(*inputfile=None*, ***kwargs*)

A class representing a .qm file.

UnitClass

alias of *qmunit*

add_unit_to_index(*unit*)

Add a unit to source and location idexes

addsourceunit(*source*)

Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)

Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)

Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)

Simple detection based on BOM in case chardet is not available.

findid(*id*)

find unit with matching id by checking id_index

findunit(*source*)

Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)

Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)

return a list of unit ids

getprojectstyle()

Get the project type for this store.

getsourcelanguage()

Get the source language for this store.

gettargetlanguage()

Get the target language for this store.

getunits()

Return a list of all units in this store.

isempty()

Return True if the object doesn't contain any translation units.

makeindex()

Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
Parses the given file or file source string.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Output a string representation of the .qm data file

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.qm.**qmunit**(*source=None*)
A class representing a .qm translation message.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

---

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.

- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

`translate.storage.qm.`**qmunpack**(*file_='messages.qm'*)
Helper to unpack Qt .qm files into a Python string

qph

Module for handling Qt Linguist Phrase Book (.qph) files.

Extract from the Qt Linguist Manual: Translators: .qph Qt Phrase Book Files are human-readable XML files containing standard phrases and their translations. These files are created and updated by Qt Linguist and may be used by any number of projects and applications.

A DTD to define the format does not seem to exist, but the following code provides the reference implementation for the Qt Linguist product.

class translate.storage.qph.**QphFile**(*inputfile=None*, *sourcelanguage='en'*, *targetlanguage=None*, ***kwargs*)

Class representing a QPH file store.

UnitClass
alias of *QphUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Method to be overridden to initialise headers, etc.

addsourceunit(*source*)
Adds and returns a new unit with the given string as first entry.

addunit(*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this .qph file.

We don't implement setsourcelanguage as users really shouldn't be altering the source language in .qph files, it should be set correctly by the extraction tools.

Returns ISO code e.g. af, fr, pt_BR

Return type String

gettargetlanguage()
Get the target language for this .qph file.

Returns ISO code e.g. af, fr, pt_BR

Return type String

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the XML document to the file *out*.

We have to override this to ensure mimic the Qt convention:

- no XML declaration

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this .qph file to *targetlanguage*.

Parameters **targetlanguage** (*String*) – ISO code e.g. af, fr, pt_BR

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.qph.**QphUnit**(*source*, *empty=False*, ***kwargs*)
A single term in the qph file.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in a "definition" tag

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

createlanguageNode(*lang*, *text*, *purpose*)
Returns an xml Element setup with given parameters.

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
We override this to get source and target nodes.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *`getlocations()`*).

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the "target" string (second language), or alternatively appends to the list

unit_iter()
Iterator that only returns this unit.

rc

Classes that hold units of .rc files (*rcunit*) or entire files (*rcfile*) used in translating Windows Resources.

translate.storage.rc.**escape_to_python**(*string*)
Escape a given .rc string into a valid Python string.

translate.storage.rc.**escape_to_rc**(*string*)
Escape a given Python string into a valid .rc string.

translate.storage.rc.**generate_dialog_caption_name**(*block_type*, *identifier*)
Return the name generated for a caption of a dialog.

translate.storage.rc.**generate_dialog_control_name**(*block_type*, *block_id*, *control_type*, *identifier*)
Return the name generated for a control of a dialog.

translate.storage.rc.**generate_menu_pre_name**(*block_type*, *block_id*)
Return the pre-name generated for elements of a menu.

translate.storage.rc.**generate_menuitem_name**(*pre_name*, *block_type*, *identifier*)
Return the name generated for a menuitem of a popup.

translate.storage.rc.**generate_popup_caption_name**(*pre_name*)
Return the name generated for a caption of a popup.

translate.storage.rc.**generate_popup_pre_name**(*pre_name*, *caption*)
Return the pre-name generated for subelements of a popup.

Parameters

- **pre_name** – The pre_name that already have the popup.
- **caption** – The caption (whitout quotes) of the popup.

Returns The subelements pre-name based in the pre-name of the popup and its caption.

translate.storage.rc.**generate_stringtable_name**(*identifier*)
Return the name generated for a stringtable element.

translate.storage.rc.**rc_statement**()
Generate a RC statement parser that can be used to parse a RC file

Return type pyparsing.ParserElement

class translate.storage.rc.**rcfile**(*inputfile=None*, *lang=None*, *sublang=None*, ***kwargs*)
This class represents a .rc file, made up of rcunits.

UnitClass
alias of *rcunit*

add_popup_units(*pre_name*, *popup*)
Transverses the popup tree making new units as needed.

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*rcsrc*)
Read the source of a .rc file in and include them as units.

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the units back to file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.rc.**rcunit**(*source=''*, ***kwargs*)
A unit of an rc file

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

getoutput()
Convert the element back into formatted lines for a .rc file.

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Returns whether this is a blank element, containing only comments.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

statistics

Module to provide statistics and related functionality.

class translate.storage.statistics.**Statistics**(*sourcelanguage='en'*, *targetlanguage='en'*, *checkerstyle=None*)
Manages statistics for storage objects.

classifyunit(*unit*)
Returns a list of the classes that the unit belongs to.

Parameters **unit** – the unit to classify

classifyunits()
Makes a dictionary of which units fall into which classifications.

This method iterates over all units.

countwords()
Counts the source and target words in each of the units.

fuzzy_unitcount()
Returns the number of fuzzy units.

fuzzy_units()
Return a list of fuzzy units.

get_source_text(*units*)
Joins the unit source strings in a single string of text.

getunits()
Returns a list of all units in this object.

reclassifyunit(*item*)
Updates the classification of a unit in self.classification.

Parameters **item** – an integer that is an index in .getunits().

source_wordcount()
Returns the number of words in the source text.

translated_unitcount()
Returns the number of translated units.

translated_units()
Return a list of translated units.

translated_wordcount()
Returns the number of translated words in this object.

untranslated_unitcount()
Returns the number of untranslated units.

untranslated_units()
Return a list of untranslated units.

untranslated_wordcount()
Returns the number of untranslated words in this object.

wordcount(*text*)
Returns the number of words in the given text.

statsdb

Module to provide a cache of statistics in a database.

class translate.storage.statsdb.**StatsCache**
An object instantiated as a singleton for each statsfile that provides access to the database cache from a pool of StatsCache objects.

con = None
This cache's connection

cur = None
The current cursor

filechecks(*filename*, *checker*, *store=None*)
Retrieves the error statistics for the given file if possible, otherwise delegates to cachestorechecks().

filestatestats(*filename*, *store=None*, *extended=False*)
Return a dictionary of unit stats mapping sets of unit indices with those states

filestats(*filename*, *checker*, *store=None*, *extended=False*)
Return a dictionary of property names mapping sets of unit indices with those properties.

filetotals(*filename*, *store=None*, *extended=False*)
Retrieves the statistics for the given file if possible, otherwise delegates to cachestore().

unitstats(*filename*, *_lang=None*, *store=None*)
Return a dictionary of property names mapping to arrays which map unit indices to property values.

Please note that this is different from filestats, since filestats supplies sets of unit indices with a given property, whereas this method supplies arrays which map unit indices to given values.

translate.storage.statsdb.**emptyfiletotals**()
Returns a dictionary with all statistics initalised to 0.

translate.storage.statsdb.**statefordb**(*unit*)
Returns the numeric database state for the unit.

translate.storage.statsdb.**transaction**(*f*)
Modifies f to commit database changes if it executes without exceptions. Otherwise it rolls back the database.

ALL publicly accessible methods in StatsCache MUST be decorated with this decorator.

translate.storage.statsdb.**wordsinunit**(*unit*)
Counts the words in the unit's source and target, taking plurals into account. The target words are only counted if the unit is translated.

subtitles

Class that manages subtitle files for translation.

This class makes use of the subtitle functionality of gaupol.

See also:

gaupol/agents/open.py::open_main

A patch to gaupol is required to open utf-8 files successfully.

class translate.storage.subtitles.**AdvSubStationAlphaFile**(**args*, ***kwargs*)
specialized class for SubRipFile's only

UnitClass
alias of *SubtitleUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
parse the given file

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.subtitles.**MicroDVDFile**(**args*, ***kwargs*)
specialized class for SubRipFile's only

UnitClass
alias of *SubtitleUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
parse the given file

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.subtitles.**SubRipFile**(**args*, ***kwargs*)
specialized class for SubRipFile's only

UnitClass
alias of *SubtitleUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type `TranslationUnit` or None

findunits(*source*)
Find the units with the given source string.

Return type `TranslationUnit` or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
parse the given file

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.subtitles.**SubStationAlphaFile**(**args*, ***kwargs*)
specialized class for SubRipFile's only

UnitClass
alias of *SubtitleUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
parse the given file

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.subtitles.**SubtitleFile**(*inputfile=None*, ***kwargs*)
A subtitle file

UnitClass
alias of *SubtitleUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
parse the given file

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.subtitles.**SubtitleUnit**(*source=None*, ***kwargs*)
A subtitle entry that is translatable

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

symbian

tbx

module for handling TBX glossary files

class translate.storage.tbx.**tbxfile**(*inputfile=None*, *sourcelanguage='en'*, *targetlanguage=None*, ***kwargs*)

Class representing a TBX file store.

UnitClass
alias of *tbxunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Initialise headers with TBX specific things.

addsourceunit(*source*)
Adds and returns a new unit with the given string as first entry.

addunit(*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out=None*)
Converts to a string containing the file's XML

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.tbx.**tbxunit**(*source*, *empty=False*, ***kwargs*)
A single term in the TBX file. Provisional work is done to make several languages possible.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in a "note" tag

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

createlanguageNode(*lang*, *text*, *purpose*)
returns a langset xml Element setup with given parameters

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
Returns a list of all nodes that contain per language information.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the "target" string (second language), or alternatively appends to the list

unit_iter()
Iterator that only returns this unit.

tiki

Class that manages TikiWiki files for translation. Tiki files are <strike>ugly and inconsistent</strike> formatted as a single large PHP array with several special sections identified by comments. Example current as of 2008-12-01:

```
<?php
  // Many comments at the top
  $lang=Array(
  // ### Start of unused words
  "aaa" => "zzz",
  // ### end of unused words

  // ### start of untranslated words
  // "bbb" => "yyy",
  // ### end of untranslated words

  // ### start of possibly untranslated words
  "ccc" => "xxx",
  // ### end of possibly untranslated words

  "ddd" => "www",
  "###end###"=>"###end###");
?>
```

In addition there are several auto-generated //-style comments scattered through the page and array, some of which matter when being parsed.

This has all been gleaned from the TikiWiki source. As far as I know no detailed documentation exists for the tiki language.php files.

class translate.storage.tiki.**TikiStore**(*inputfile=None*)
Represents a tiki language.php file.

UnitClass
alias of *TikiUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
Parse the given input into source units.

Parameters **input** – the source, either a string or filehandle

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Will return a formatted tiki-style language.php file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.tiki.**TikiUnit**(*source=None*, ***kwargs*)
A tiki unit entry.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Location is defined by the comments in the file. This function will only set valid locations.

Parameters **location** – Where the string is located in the file. Must be a valid location.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
Returns the a list of the location(s) of the string.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

tmdb

Module to provide a translation memory database.

exception translate.storage.tmdb.**LanguageError**(*value*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

tmx

module for parsing TMX translation memeory files

class translate.storage.tmx.**tmxfile**(*inputfile=None*, *sourcelanguage='en'*, *targetlanguage=None*, ***kwargs*)
Class representing a TMX file store.

UnitClass
alias of *tmxunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Method to be overridden to initialise headers, etc.

addsourceunit(*source*)
Adds and returns a new unit with the given string as first entry.

addtranslation(*source*, *srclang*, *translation*, *translang*, *comment=None*)
addtranslation method for testing old unit tests

addunit(*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out=None*)
Converts to a string containing the file's XML

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*sourcetext*, *sourcelang=None*, *targetlang=None*)
method to test old unit tests

unit_iter()
Iterator over all the units in this store.

class translate.storage.tmx.**tmxunit**(*source*, *empty=False*, ***kwargs*)
A single unit in the TMX file.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in a "note" tag.

The origin parameter is ignored

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

copy()
Make a copy of the translation unit.

We don't want to make a deep copy - this could duplicate the whole XML tree. For now we just serialise and reparse the unit's XML.

createlanguageNode(*lang*, *text*, *purpose*)
returns a langset xml Element setup with given parameters

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

getcontext()
Get the message context.

geterrors()
Get all error messages.

getid()
Returns the identifier for this unit. The optional tuid property is used if available, otherwise we inherit .getid(). Note that the tuid property is only mandated to be unique from TMX 2.0.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
Returns a list of all nodes that contain per language information.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see `getlocations()`).

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the "target" string (second language), or alternatively appends to the list

unit_iter()
Iterator that only returns this unit.

trados

Manage the Trados .txt Translation Memory format

A Trados file looks like this:

```
<TrU>
<CrD>18012000, 13:18:35
<CrU>CAROL-ANN
<UsC>0
<Seg L=EN_GB>Association for Road Safety \endash  Conference
<Seg L=DE_DE>Tagung der Gesellschaft für Verkehrssicherheit
</TrU>
<TrU>
<CrD>18012000, 13:19:14
<CrU>CAROL-ANN
<UsC>0
<Seg L=EN_GB>Road Safety Education in our Schools
<Seg L=DE_DE>Verkehrserziehung an Schulen
</TrU>
```

translate.storage.trados.**TRADOS_TIMEFORMAT = '%d%m%Y, %H:%M:%S'**
Time format used by Trados .txt

translate.storage.trados.**RTF_ESCAPES = {'\\-': '\xad', '_': '-', '\\bullet': '•', '\\e**
RTF control to Unicode map. See http://msdn.microsoft.com/en-us/library/aa140283(v=office.10).aspx

translate.storage.trados.**escape**(*text*)
Convert Unicode string to Trodas escapes

translate.storage.trados.**unescape**(*text*)
Convert Trados text to normal Unicode string

class translate.storage.trados.**TradosTxtDate**(*newtime=None*)
Manages the timestamps in the Trados .txt format of DDMMYYY, hh:mm:ss

get_time()
Get the time_struct object

get_timestring()
Get the time in the Trados time format

set_time(*newtime*)
Set the time_struct object

Parameters **newtime** (*time.time_struct*) – a new time object

set_timestring(*timestring*)
Set the time_struct object using a Trados time formated string

Parameters **timestring** (*String*) – A Trados time string (DDMMYYYY, hh:mm:ss)

time
Get the time_struct object

timestring
Get the time in the Trados time format

class translate.storage.trados.**TradosUnit**(*source=None*)

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

class translate.storage.trados.**TradosTxtTmFile**(*inputfile=None*, ***kwargs*)
A Trados translation memory file

UnitClass
alias of *TradosUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parser to process the given source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

ts2

Module for handling Qt linguist (.ts) files.

This will eventually replace the older ts.py which only supports the older format. While converters haven't been updated to use this module, we retain both.

TS file format 4.3, 4.8, 5. Example.

Specification of the valid variable entries, 2

class translate.storage.ts2.**tsfile**(**args*, ***kwargs*)
Class representing a TS file store.

UnitClass
alias of *tsunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Method to be overridden to initialise headers, etc.

addsourceunit(*source*)
Adds and returns a new unit with the given string as first entry.

addunit(*unit*, *new=True*, *contextname=None*, *comment=None*, *createifmissing=True*)
Adds the given unit to the last used body node (current context).

If the contextname is specified, switch to that context (creating it if allowed by createifmissing).

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this .ts file.

The 'sourcelanguage' attribute was only added to the TS format in Qt v4.5. We return 'en' if there is no sourcelanguage set.

We don't implement setsourcelanguage as users really shouldn't be altering the source language in .ts files, it should be set correctly by the extraction tools.

Returns ISO code e.g. af, fr, pt_BR

Return type String

gettargetlanguage()
Get the target language for this .ts file.

Returns ISO code e.g. af, fr, pt_BR

Return type String

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Write the XML document to a file.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this .ts file to *targetlanguage*.

Parameters **targetlanguage** (*String*) – ISO code e.g. af, fr, pt_BR

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.ts2.**tsunit**(*source*, *empty=False*, ***kwargs*)
A single term in the TS file.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in the appropriate *comment* tag

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

createlanguageNode(*lang*, *text*, *purpose*)
Returns an xml Element setup with given parameters.

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
We override this to get source and target nodes.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
States whether this unit needs to be reviewed

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*value*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the "target" string (second language), or alternatively appends to the list

statemap = {'obsolete': -100, 'unfinished': 30, '': 100, None: 100}
This maps the unit "type" attribute to state.

unit_iter()
Iterator that only returns this unit.

ts

Module for parsing Qt .ts files for translation.

Currently this module supports the old format of .ts files. Some applictaions use the newer .ts format which are documented here: TS file format 4.3, Example

Specification of the valid variable entries, 2

txt

This class implements the functionality for handling plain text files, or similar wiki type files.

Supported formats are

- Plain text
- dokuwiki
- MediaWiki

class translate.storage.txt.**TxtFile**(*inputfile=None*, *flavour=None*, *no_segmentation=False*, ***kwargs*)
This class represents a text file, made up of txtunits

UnitClass
alias of *TxtUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*lines*)
Read in text lines and create txtunits from the blocks of text

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.txt.**TxtUnit**(*source=''*, ***kwargs*)
This class represents a block of text from a text file

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

getcontext()
Get the message context.

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

target
gets the unquoted target string

unit_iter()
Iterator that only returns this unit.

utx

Manage the Universal Terminology eXchange (UTX) format

UTX is a format for terminology exchange, designed it seems with Machine Translation (MT) as it's primary consumer. The format is created by the Asia-Pacific Association for Machine Translation (AAMT).

It is a bilingual base class derived format with *UtxFile* and *UtxUnit* providing file and unit level access.

The format can manage monolingual dictionaries but these classes don't implement that.

Specification The format is implemented according to UTX v1.0 (No longer available from their website. The current UTX version may be downloaded instead).

Format Implementation The UTX format is a Tab Seperated Value (TSV) file in UTF-8. The first two lines are headers with subsequent lines containing a single source target definition.

Encoding The files are UTF-8 encoded with no BOM and CR+LF line terminators.

class translate.storage.utx.**UtxDialect**
Describe the properties of an UTX generated TAB-delimited dictionary file.

class translate.storage.utx.**UtxFile**(*inputfile=None*, ***kwargs*)
A UTX dictionary file

UnitClass
alias of *UtxUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parsese the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.utx.**UtxHeader**
A UTX header entry

A UTX header is a single line that looks like this:: #UTX-S <version>; < source language >/< target language>; <date created>; <optional fields (creator, license, etc.)>

Where::

- UTX-S version is currently 1.00.
- Source language/target language: ISO 639, 3166 formats. In the case of monolingual dictionary, target language should be omitted.
- Date created: ISO 8601 format
- Optional fields (creator, license, etc.)

class translate.storage.utx.**UtxUnit**(*source=None*)

A UTX dictionary unit

adderror(*errorname*, *errortext*)

Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)

Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)

Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)

Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)

Build a native unit from a foreign unit, preserving as much information as possible.

dict

Get the dictionary of values for a UTX line

getcontext()

Get the message context.

getdict()

Get the dictionary of values for a UTX line

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.
- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setdict(*newdict*)
Set the dictionary of values for a UTX line

Parameters **newdict** (*Dict*) – a new dictionary with UTX line elements

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

wordfast

Manage the Wordfast Translation Memory format

Wordfast TM format is the Translation Memory format used by the Wordfast computer aided translation tool.

It is a bilingual base class derived format with *WordfastTMFile* and *WordfastUnit* providing file and unit level access.

Wordfast is a computer aided translation tool. It is an application built on top of Microsoft Word and is implemented as a rather sophisticated set of macros. Understanding that helps us understand many of the seemingly strange choices around this format including: encoding, escaping and file naming.

Implementation The implementation covers the full requirements of a Wordfast TM file. The files are simple Tab Separated Value (TSV) files that can be read by Microsoft Excel and other spreadsheet programs. They use the .txt extension which does make it more difficult to automatically identify such files.

The dialect of the TSV files is specified by *WordfastDialect*.

Encoding The files are UTF-16 or ISO-8859-1 (Latin1) encoded. These choices are most likely because Microsoft Word is the base editing tool for Wordfast.

The format is tab separated so we are able to detect UTF-16 vs Latin-1 by searching for the occurance of a UTF-16 tab character and then continuing with the parsing.

Timestamps *WordfastTime* allows for the correct management of the Wordfast YYYYMMDD~HHMMSS timestamps. However, timestamps on individual units are not updated when edited.

Header *WordfastHeader* provides header management support. The header functionality is fully implemented through observing the behaviour of the files in real use cases, input from the Wordfast programmers and public documentation.

Escaping Wordfast TM implements a form of escaping that covers two aspects:

1. Placeable: bold, formating, etc. These are left as is and ignored. It is up to the editor and future placeable implementation to manage these.
2. Escapes: items that may confuse Excel or translators are escaped as `&'XX;`. These are fully implemented and are converted to and from Unicode. By observing behaviour and reading documentation we where able to observe all possible escapes. Unfortunately the escaping differs slightly between Windows and

Mac version. This might cause errors in future. Functions allow for <_wf_to_char> and back to Wordfast escape (<_char_to_wf>).

Extended Attributes The last 4 columns allow users to define and manage extended attributes. These are left as is and are not directly managed byour implemenation.

translate.storage.wordfast.**TAB_UTF16 = b'\x00\t'**
The tab t character as it would appear in UTF-16 encoding

translate.storage.wordfast.**WF_ESCAPE_MAP = (("&'26;", '&'), ("&'82;", ','), ("&'85;", '...**
Mapping of Wordfast &'XX; escapes to correct Unicode characters

translate.storage.wordfast.**WF_FIELDNAMES = ['date', 'user', 'reuse', 'src-lang', 'source',**
Field names for a Wordfast TU

translate.storage.wordfast.**WF_FIELDNAMES_HEADER = ['date', 'userlist', 'tucount', 'src-lang**
Field names for the Wordfast header

translate.storage.wordfast.**WF_FIELDNAMES_HEADER_DEFAULTS = {'attr1list': '', 'attr2list'**
Default or minimum header entries for a Wordfast file

translate.storage.wordfast.**WF_TIMEFORMAT = '%Y%m%d~%H%M%S'**
Time format used by Wordfast

class translate.storage.wordfast.**WordfastDialect**
Describe the properties of a Wordfast generated TAB-delimited file.

class translate.storage.wordfast.**WordfastHeader**(*header=None*)
A wordfast translation memory header

getheader()
Get the header dictionary

header
Get the header dictionary

class translate.storage.wordfast.**WordfastTMFile**(*inputfile=None*, ***kwargs*)
A Wordfast translation memory file

UnitClass
alias of *WordfastUnit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addsourceunit(*source*)
Add and returns a new unit with the given source string.

Return type TranslationUnit

addunit(*unit*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

detect_encoding(*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection(*text*)
Simple detection based on BOM in case chardet is not available.

findid(*id*)
find unit with matching id by checking id_index

findunit(*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits(*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getids(*filename=None*)
return a list of unit ids

getprojectstyle()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

parse(*input*)
parsese the given file or file source string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Convert the string representation back to an object.

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a bytes representation that can be parsed back using *parsestring()*. *out* should be an open file-like objects to write to.

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*sourcelanguage*)
Set the source language for this store.

settargetlanguage(*targetlanguage*)
Set the target language for this store.

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.wordfast.**WordfastTime**(*newtime=None*)
Manages time stamps in the Wordfast format of YYYYMMDD~hhmmss

get_time()
Get the time_struct object

get_timestring()
Get the time in the Wordfast time format

set_time(*newtime*)
Set the time_struct object

Parameters **newtime** (*time.time_struct*) – a new time object

set_timestring(*timestring*)
Set the time_sturct object using a Wordfast time formated string

Parameters **timestring** (*String*) – A Wordfast time string (YYYMMDD~hhmmss)

time
Get the time_struct object

timestring
Get the time in the Wordfast time format

class translate.storage.wordfast.**WordfastUnit**(*source=None*)
A Wordfast translation memory unit

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

Parameters

- **errorname** (*string*) – A single word to id the error.
- **errortext** (*string*) – The text describing the error.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Adds a note (comment).

Parameters

- **text** (*string*) – Usually just a sentence or two.
- **origin** (*string*) – Specifies who/where the comment comes from. Origin can be one of the following text strings: - 'translator' - 'developer', 'programmer', 'source code' (synonyms)

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

dict
Get the dictionary of values for a Wordfast line

getcontext()
Get the message context.

getdict()
Get the dictionary of values for a Wordfast line

geterrors()
Get all error messages.

Return type Dictionary

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *getlocations()*).

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
Indicates whether this unit needs review.

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Parameters

- **needsreview** – Defaults to True.

- **explanation** – Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

multistring_to_rich(*mulstring*)
Convert a multistring to a list of "rich" string trees:

```
>>> target = multistring(['foo', 'bar', 'baz'])
>>> TranslationUnit.multistring_to_rich(target)
[<StringElem([<StringElem(['foo'])>])>,
 <StringElem([<StringElem(['bar'])>])>,
 <StringElem([<StringElem(['baz'])>])>]
```

removenotes(*origin=None*)
Remove all the translator's notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Convert a "rich" string tree to a `multistring`:

```
>>> from translate.storage.placeables.interfaces import X
>>> rich = [StringElem(['foo', X(id='xxx', sub=[' ']), 'bar'])]
>>> TranslationUnit.rich_to_multistring(rich)
multistring('foo bar')
```

setcontext(*context*)
Set the message context

setdict(*newdict*)
Set the dictionary of values for a Wordfast line

Parameters **newdict** (*Dict*) – a new dictionary with Wordfast line elements

setid(*value*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

unit_iter()
Iterator that only returns this unit.

workflow

A workflow is defined by a set of states that a translation unit can be in and the (allowed) transitions between these states. A state is defined by a range between -128 and 127, indicating its level of "completeness". The range is closed at the beginning and open at the end. That is, if a workflow contains states A, B and C where A < B < C, a unit with state number n is in state A if A <= n < B, state B if B <= n < C or state C if C <= n < MAX.

A value of 0 is typically the "empty" or "new" state with negative values reserved for states like "obsolete" or "do not use".

Format specific workflows should be defined in such a way that the numeric state values correspond to similar states. For example state 0 should be "untranslated" in PO and "new" or "empty" in XLIFF, state 100 should be "translated" in PO and "final" in XLIFF. This allows formats to implicitly define similar states.

exception translate.storage.workflow.**InvalidStateObjectError**(*obj*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception translate.storage.workflow.**NoInitialStateError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class translate.storage.workflow.**StateEnum**
Only contains the constants for default states.

exception translate.storage.workflow.**StateNotInWorkflowError**(*state*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception translate.storage.workflow.**TransitionError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception translate.storage.workflow.**WorkflowError**

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

xliff

Module for handling XLIFF files for translation.

The official recommendation is to use the extention .xlf for XLIFF files.

class translate.storage.xliff.**xlifffile**(**args*, ***kwargs*)
Class representing a XLIFF file store.

UnitClass
alias of *xliffunit*

add_unit_to_index(*unit*)
Add a unit to source and location idexes

addheader()
Initialise the file header.

addsourceunit(*source*, *filename='NoName'*, *createifmissing=False*)
adds the given trans-unit to the last used body node if the filename has changed it uses the slow method instead (will create the nodes required if asked). Returns success

addunit (*unit*, *new=True*)
Append the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (TranslationUnit) – The unit that will be added.

createfilenode (*filename*, *sourcelanguage=None*, *targetlanguage=None*, *datatype='plaintext'*)
creates a filenode with the given filename. All parameters are needed for XLIFF compliance.

creategroup (*filename='NoName'*, *createifmissing=False*, *restype=None*)
adds a group tag into the specified file

detect_encoding (*text*, *default_encodings=None*)
Try to detect a file encoding from *text*, using either the chardet lib or by trying to decode the file.

fallback_detection (*text*)
Simple detection based on BOM in case chardet is not available.

findid (*id*)
find unit with matching id by checking id_index

findunit (*source*)
Find the unit with the given source string.

Return type TranslationUnit or None

findunits (*source*)
Find the units with the given source string.

Return type TranslationUnit or None

getbodynode (*filenode*, *createifmissing=False*)
finds the body node for the given filenode

getdatatype (*filename=None*)
Returns the datatype of the stored file. If no filename is given, the datatype of the first file is given.

getdate (*filename=None*)
Returns the date attribute for the file.

If no filename is given, the date of the first file is given. If the date attribute is not specified, None is returned.

Returns Date attribute of file

Return type Date or None

getfilename (*filenode*)
returns the name of the given file

getfilenames ()
returns all filenames in this XLIFF file

getfilenode (*filename*, *createifmissing=False*)
finds the filenode with the given name

getheadernode (*filenode*, *createifmissing=False*)
finds the header node for the given filenode

getids (*filename=None*)
return a list of unit ids

getprojectstyle ()
Get the project type for this store.

getsourcelanguage()
Get the source language for this store.

gettargetlanguage()
Get the target language for this store.

getunits()
Return a list of all units in this store.

initbody()
Initialises self.body so it never needs to be retrieved from the XML again.

isempty()
Return True if the object doesn't contain any translation units.

makeindex()
Indexes the items in this store. At least .sourceindex should be useful.

merge_on
The matching criterion to use when merging on.

Returns The default matching criterion for all the subclasses.

Return type string

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

parse(*xml*)
Populates this object from the given xml string

classmethod parsefile(*storefile*)
Reads the given file (or opens the given filename) and parses back to an object.

classmethod parsestring(*storestring*)
Parses the string to return the correct file object

remove_unit_from_index(*unit*)
Remove a unit from source and locaton indexes

removedefaultfile()
We want to remove the default file-tag as soon as possible if we know if still present and empty.

removeunit(*unit*)
Remove the given unit to the object's list of units.

This method should always be used rather than trying to modify the list manually.

Parameters **unit** (`TranslationUnit`) – The unit that will be added.

require_index()
make sure source index exists

save()
Save to the file that data was originally read from, if available.

savefile(*storefile*)
Write the string representation to the given file (or filename).

serialize(*out*)
Converts to a string containing the file's XML

setfilename(*filenode*, *filename*)
set the name of the given file

setprojectstyle(*project_style*)
Set the project type for this store.

setsourcelanguage(*language*)
Set the source language for this store.

settargetlanguage(*language*)
Set the target language for this store.

suggestions_in_format = True
xliff units have alttrans tags which can be used to store suggestions

switchfile(*filename*, *createifmissing=False*)
Adds the given trans-unit (will create the nodes required if asked).

Returns Success

Return type Boolean

translate(*source*)
Return the translated string for a given source string.

Return type String or None

unit_iter()
Iterator over all the units in this store.

class translate.storage.xliff.**xliffunit**(*source*, *empty=False*, ***kwargs*)
A single term in the xliff file.

addalttrans(*txt*, *origin=None*, *lang=None*, *sourcetxt=None*, *matchquality=None*)
Adds an alt-trans tag and alt-trans components to the unit.

Parameters **txt** (*String*) – Alternative translation of the source text.

adderror(*errorname*, *errortext*)
Adds an error message to this unit.

addlocation(*location*)
Add one location to the list of locations.

Note: Shouldn't be implemented if the format doesn't support it.

---

addlocations(*location*)
Add a location or a list of locations.

Note: Most classes shouldn't need to implement this, but should rather implement `TranslationUnit.addlocation()`.

Warning: This method might be removed in future.

addnote(*text*, *origin=None*, *position='append'*)
Add a note specifically in a "note" tag

classmethod buildfromunit(*unit*)
Build a native unit from a foreign unit, preserving as much information as possible.

correctorigin(*node*, *origin*)
Check against node tag's origin (e.g note or alt-trans)

createcontextgroup(*name*, *contexts=None*, *purpose=None*)
Add the context group to the trans-unit with contexts a list with (type, text) tuples describing each context.

createlanguageNode(*lang*, *text*, *purpose*)
Returns an xml Element setup with given parameters.

delalttrans(*alternative*)
Removes the supplied alternative from the list of alt-trans tags

getNodeText(*languageNode*, *xml_space='preserve'*)
Retrieves the term from the given `languageNode`.

get_rich_target(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

getalttrans(*origin=None*)
Returns <alt-trans> for the given origin as a list of units. No origin means all alternatives.

getcontext()
Get the message context.

getcontextgroups(*name*)
Returns the contexts in the context groups with the specified name

geterrors()
Get all error messages.

getid()
A unique identifier for this unit.

Return type string

Returns an identifier for this unit that is unique in the store

Derived classes should override this in a way that guarantees a unique identifier for each unit in the store.

getlanguageNode(*lang=None*, *index=None*)
Retrieves a `languageNode` either by language or by index.

getlanguageNodes()
We override this to get source and target nodes.

getlocations()
A list of source code locations.

Return type List

Note: Shouldn't be implemented if the format doesn't support it.

---

getnotes(*origin=None*)
Returns all notes about this unit.

It will probably be freeform text or something reasonable that can be synthesised by the format. It should not include location comments (see *`getlocations()`*).

getrestype()
returns the restype attribute in the trans-unit tag

gettarget(*lang=None*)
retrieves the "target" text (second entry), or the entry in the specified language, if it exists

gettargetlen()
Returns the length of the target string.

Return type Integer

Note: Plural forms might be combined.

---

getunits()
This unit in a list.

hasplural()
Tells whether or not this specific unit has plural strings.

infer_state()
Empty method that should be overridden in sub-classes to infer the current state(_n) of the unit from its current state.

isapproved()
States whether this unit is approved.

isblank()
Used to see if this unit has no source or target string.

Note: This is probably used more to find translatable units, and we might want to move in that direction rather and get rid of this.

---

isfuzzy()
Indicates whether this unit is fuzzy.

isheader()
Indicates whether this unit is a header.

isobsolete()
indicate whether a unit is obsolete

isreview()
States whether this unit needs to be reviewed

istranslatable()
Indicates whether this unit can be translated.

This should be used to distinguish real units for translation from header, obsolete, binary or other blank units.

istranslated()
Indicates whether this unit is translated.

This should be used rather than deducing it from .target, to ensure that other classes can implement more functionality (as XLIFF does).

makeobsolete()
Make a unit obsolete

markapproved(*value=True*)
Mark this unit as approved.

markfuzzy(*value=True*)
Marks the unit as fuzzy or not.

markreviewneeded(*needsreview=True*, *explanation=None*)
Marks the unit to indicate whether it needs review.

Adds an optional explanation as a note.

merge(*otherunit*, *overwrite=False*, *comments=True*, *authoritative=False*)
Do basic format agnostic merging.

classmethod multistring_to_rich(*mstr*)
Override `TranslationUnit.multistring_to_rich()` which is used by the `rich_source` and `rich_target` properties.

namespaced(*name*)
Returns name in Clark notation.

For example `namespaced("source")` in an XLIFF document might return:

```
{urn:oasis:names:tc:xliff:document:1.1}source
```

This is needed throughout lxml.

removenotes(*origin=None*)
Remove all the translator notes.

rich_source

See also:

rich_to_multistring(), *multistring_to_rich()*

rich_target

See also:

rich_to_multistring(), *multistring_to_rich()*

classmethod rich_to_multistring(*elem_list*)
Override `TranslationUnit.rich_to_multistring()` which is used by the `rich_source` and `rich_target` properties.

setcontext(*context*)
Set the message context

setid(*id*)
Sets the unique identified for this unit.

only implemented if format allows ids independant from other unit properties like source or context

settarget(*target*, *lang='xx'*, *append=False*)
Sets the target string to the given value.

unit_iter()
Iterator that only returns this unit.

xml_extract

extract

class translate.storage.xml_extract.extract.**ParseState**(*no_translate_content_elements*, *inline_elements={}*, *nsmap={}*)

Maintain constants and variables used during the walking of a DOM tree (via the function apply).

class translate.storage.xml_extract.extract.**Translatable**(*placeable_name*, *xpath*, *dom_node*, *source*, *is_inline=False*)

A node corresponds to a translatable element. A node may have children, which correspond to placeables.

has_translatable_text

Check if it contains any chunk of text with more than whitespace.

If not, then there's nothing to translate.

translate.storage.xml_extract.extract.**build_idml_store**(*odf_file*, *store*, *parse_state*, *store_adder=None*)

Build a store for the given IDML file.

translate.storage.xml_extract.extract.**build_store**(*odf_file*, *store*, *parse_state*, *store_adder=None*)

Build a store for the given XML file.

translate.storage.xml_extract.extract.**make_postore_adder**(*store*, *id_maker*, *file-name*)

Return a function which, when called with a Translatable will add a unit to 'store'. The placeables will be represented as strings according to 'placeable_quoter'.

translate.storage.xml_extract.extract.**process_translatable**(*dom_node*, *state*)

Process a translatable DOM node.

Any translatable content present in a child node is treated as a placeable.

generate

translate.storage.xml_extract.generate.**find_dom_root**(*parent_dom_node*, *dom_node*)

See also:

find_placeable_dom_tree_roots()

translate.storage.xml_extract.generate.**find_placeable_dom_tree_roots**(*unit_node*)

For an inline placeable, find the root DOM node for the placeable in its parent.

Consider the diagram. In this pseudo-ODF example, there is an inline span element. However, the span is contained in other tags (which we never process). When splicing the template DOM tree (that is, the DOM which comes from the XML document we're using to generate a translated XML document), we'll need to move DOM sub-trees around and we need the roots of these sub-trees:

```
<p> This is text \/                <- Paragraph containing an inline placeable
                 <blah>            <- Inline placeable's root (which we want to␣
↪find)
                 ...               <- Any number of intermediate DOM nodes
                 <span> bold text  <- The inline placeable's Translatable
                                      holds a reference to this DOM node
```

translate.storage.xml_extract.generate.**get_xliff_source_target_doms**(*unit*)
Return a tuple with unit source and target DOM objects.

This method is method is meant to provide a way to retrieve the DOM objects for the unit source and target for XLIFF stores.

translate.storage.xml_extract.generate.**replace_dom_text**(*make_parse_state*, *dom_retriever=<function get_xliff_source_target_doms>*, *process_translatable=<function process_translatable>*)
Return a function:

```
action: etree_Element x base.TranslationUnit -> None
```

which takes a dom_node and a translation unit. The dom_node is rearranged according to rearrangement of placeables in unit.target (relative to their positions in unit.source).

misc

translate.storage.xml_extract.misc.**compose_mappings**(*left*, *right*)
Given two mappings left: A -> B and right: B -> C, create a hash result_map: A -> C. Only values in left (i.e. things from B) which have corresponding keys in right will have their keys mapped to values in right.

translate.storage.xml_extract.misc.**parse_tag**(*full_tag*)

```
>>> parse_tag('{urn:oasis:names:tc:opendocument:xmlns:office:1.0}document-content
↪')
('urn:oasis:names:tc:opendocument:xmlns:office:1.0', 'document-content')
>>> parse_tag('document-content')
('', 'document-content')
```

translate.storage.xml_extract.misc.**reduce_tree**(*f*, *parent_unit_node*, *unit_node*, *get_children*, **state*)
Enumerate a tree, applying f to in a pre-order fashion to each node.

parent_unit_node contains the parent of unit_node. For the root of the tree, parent_unit_node == unit_node.

get_children is a single argument function applied to a unit_node to get a list/iterator to its children.

state is used by f to modify state information relating to whatever f does to the tree.

unit_tree

translate.storage.xml_extract.unit_tree.**build_unit_tree**(*store*, *filename=None*)
Enumerate a translation store and build a tree with XPath components as nodes and where a node contains a unit if a path from the root of the tree to the node containing the unit, is equal to the XPath of the unit.

The tree looks something like this:

```
root
   `- ('document-content', 1)
      `- ('body', 2)
         |- ('text', 1)
```

(continues on next page)

(continued from previous page)

```
|  `- ('p', 1)
|     `- <reference to a unit>
|- ('text', 2)
|  `- ('p', 1)
|     `- <reference to a unit>
`- ('text', 3)
   `- ('p', 1)
      `- <reference to a unit>
```

xpath_breadcrumb

class translate.storage.xml_extract.xpath_breadcrumb.**XPathBreadcrumb**

A class which is used to build XPath-like paths as a DOM tree is walked. It keeps track of the number of times which it has seen a certain tag, so that it will correctly create indices for tags.

Initially, the path is empty. Thus >>> xb = XPathBreadcrumb() >>> xb.xpath “”

Suppose we walk down a DOM node for the tag <foo> and we want to record this, we simply do >>> xb.start_tag(‘foo’)

Now, the path is no longer empty. Thus >>> xb.xpath foo[0]

Now suppose there are two <bar> tags under the tag <foo> (that is <foo><bar></bar><bar></bar><foo>), then the breadcrumb will keep track of the number of times it sees <bar>. Thus

```
>>> xb.start_tag('bar')
>>> xb.xpath
foo[0]/bar[0]
>>> xb.end_tag()
>>> xb.xpath
foo[0]
>>> xb.start_tag('bar')
>>> xb.xpath
foo[0]/bar[1]
```

xml_name

class translate.storage.xml_name.**XmlNamer**(*dom_node*)

Initialize me with a DOM node or a DOM document node (the toplevel node you get when parsing an XML file). Then use me to generate fully qualified XML names.

```
>>> xml = '<office:document-styles xmlns:office=
↪"urn:oasis:names:tc:opendocument:xmlns:office:1.0"></office>'
>>> from lxml import etree
>>> namer = XmlNamer(etree.fromstring(xml))
>>> namer.name('office', 'blah')
{urn:oasis:names:tc:opendocument:xmlns:office:1.0}blah
>>> namer.name('office:blah')
{urn:oasis:names:tc:opendocument:xmlns:office:1.0}blah
```

I can also give you XmlNamespace objects if you give me the abbreviated namespace name. These are useful if you need to reference a namespace continuously.

```
>>> office_ns = name.namespace('office')
>>> office_ns.name('foo')
{urn:oasis:names:tc:opendocument:xmlns:office:1.0}foo
```

zip

This module provides functionality to work with zip files.

class translate.storage.zip.**ZIPFile**(*filename=None*)
This class represents a ZIP file like a directory.

file_iter()
Iterator over (dir, filename) for all files in this directory.

getfiles()
Returns a list of (dir, filename) tuples for all the file names in this directory.

getunits()
List of all the units in all the files in this directory.

scanfiles()
Populate the internal file data.

unit_iter()
Iterator over all the units in all the files in this zip file.

tools

Code to perform various operations, mostly on po files.

build_tmdb

Import units from translations files into tmdb.

phppo2pypo

Convert PHP format .po files to Python format .po files.

translate.tools.phppo2pypo.**convertphp2py**(*inputfile*, *outputfile*, *template=None*)
Converts from PHP .po format to Python .po format

Parameters

- **inputfile** – file handle of the source
- **outputfile** – file handle to write to
- **template** – unused

translate.tools.phppo2pypo.**main**(*argv=None*)
Converts PHP .po files to Python .po files.

poclean

Produces a clean file from an unclean file (Trados/Wordfast) by stripping out the tw4win indicators.

This does not convert an RTF file to PO/XLIFF, but produces the target file with only the target text in from a text version of the RTF.

translate.tools.poclean.**cleanfile**(*thefile*)
cleans the given file

translate.tools.poclean.**cleanunit**(*unit*)
cleans the targets in the given unit

translate.tools.poclean.**runclean**(*inputfile*, *outputfile*, *templatefile*)
reads in inputfile, cleans, writes to outputfile

pocompile

Compile XLIFF and Gettext PO localization files into Gettext MO (Machine Object) files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pocompile.html for examples and usage instructions.

translate.tools.pocompile.**convertmo**(*inputfile*, *outputfile*, *templatefile*, *includefuzzy=False*)
reads in a base class derived inputfile, converts using pocompile, writes to outputfile

poconflicts

Conflict finder for Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/poconflicts.html for examples and usage instructions.

class translate.tools.poconflicts.**ConflictOptionParser**(*formats*, *usetemplates=False*, *allowmissingtemplate=False*, *description=None*)
a specialized Option Parser for the conflict tool...

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

buildconflictmap()
work out which strings are conflicting

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

clean(*string*, *options*)
returns the cleaned string that contains the text to be matched

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

flatten(*text*, *joinchar*)
flattens text to just be words

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

outputconflicts(*options*)
saves the result of the conflict match

parse_args(*args=None*, *values=None*)
parses the command line options, handling implicit input/output args

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*)
process an individual file

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
recurse through directories and process files

run()
Parses the arguments, and runs recursiveprocess with the resulting options...

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions()
Sets the progress options.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

pocount

Count strings and words for supported localization files.

These include: XLIFF, TMX, Gettex PO and MO, Qt .ts and .qm, Wordfast TM, etc

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pocount.html for examples and usage instructions.

class translate.tools.pocount.**ConsoleColor**
Class to implement color mode.

translate.tools.pocount.**calcstats_old**(*filename*)
This is the previous implementation of calcstats() and is left for comparison and debuging purposes.

translate.tools.pocount.**summarize**(*title*, *stats*, *style=0*, *indent=8*, *incomplete_only=False*)
Print summary for a .po file in specified format.

Parameters

- **title** – name of .po file

- **stats** – array with translation statistics for the file specified
- **indent** – indentation of the 2nd column (length of longest filename)
- **incomplete_only** (*Boolean*) – omit fully translated files

Return type Boolean

Returns 1 if counting incomplete files (incomplete_only=True) and the file is completely translated, 0 otherwise

podebug

Insert debug messages into XLIFF and Gettext PO localization files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/podebug.html for examples and usage instructions.

translate.tools.podebug.**convertpo**(*inputfile*, *outputfile*, *templatefile*, *format=None*, *rewritestyle=None*, *ignoreoption=None*, *preserveplaceholders=None*)

Reads in inputfile, changes it to have debug strings, writes to outputfile.

pogrep

Grep XLIFF, Gettext PO and TMX localization files.

Matches are output to snippet files of the same type which can then be reviewed and later merged using *pomerge*.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pogrep.html for examples and usage instructions.

class translate.tools.pogrep.**GrepMatch**(*unit*, *part='target'*, *part_n=0*, *start=0*, *end=0*)

Just a small data structure that represents a search match.

class translate.tools.pogrep.**GrepOptionParser**(*formats*, *usetemplates=False*, *allowmissingtemplate=False*, *description=None*)

a specialized Option Parser for the grep tool...

add_option(*Option*)

add_option(opt_str, ..., kwarg=val, ...)

check_values(*values : Values, args : [string]*)

-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)

Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)

Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()

Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args(*args=None*, *values=None*)
parses the command line options, handling implicit input/output args

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*, *fulloutputpath*, *fulltemplatepath*)
Process an individual file.

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
Recurse through directories and process files.

run()
parses the arguments, and runs recursiveprocess with the resulting options

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats(*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions()
Sets the progress options.

splitext(*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext(*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext(*templatepath*)
Splits a *templatepath* into name and extension.

templateexists(*options*, *templatepath*)
Returns whether the given template exists...

warning(*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

translate.tools.pogrep.**find_matches**(*unit*, *part*, *strings*, *re_search*)
Return the GrepFilter objects where re_search matches in strings.

translate.tools.pogrep.**real_index**(*string*, *nfc_index*)
Calculate the real index in the unnormalized string that corresponds to the index nfc_index in the normalized string.

translate.tools.pogrep.**rungrep**(*inputfile*, *outputfile*, *templatefile*, *checkfilter*)
reads in inputfile, filters using checkfilter, writes to outputfile

pomerge

Merges XLIFF and Gettext PO localization files.

Snippet file produced by e.g. *pogrep* and updated by a translator can be merged back into the original files.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pomerge.html for examples and usage instructions.

translate.tools.pomerge.**mergestores**(*store1*, *store2*, *mergeblanks*, *mergefuzzy*, *mergecomments*)
Take any new translations in store2 and write them into store1.

translate.tools.pomerge.**str2bool**(*option*)
Convert a string value to boolean

Parameters **option** (*String*) – yes, true, 1, no, false, 0

Return type Boolean

porestructure

Restructure Gettxt PO files produced by *poconflicts* into the original directory tree for merging using *pomerge*.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pomerge.html for examples and usage instructions.

class translate.tools.porestructure.**SplitOptionParser**(*formats*, *usetemplates=False*, *allowmissingtemplate=False*, *description=None*)

a specialized Option Parser for posplit

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions (*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions (*options*)
Get the options required to pass to the filtermethod...

gettemplatename (*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman (*option*)
returns the usage string for the given option

getusagestring (*option*)
returns the usage string for the given option

isexcluded (*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive (*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname (*inputname*)
Checks if this is a valid input filename.

mkdir (*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile (*options*, *fullinputpath*)
Opens the input file.

openoutputfile (*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile (*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile (*options*, *fulloutputpath*)
Opens a temporary output file.

parse_args (*args=None*, *values=None*)
parses the command line options, handling implicit input/output args

print_help (*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage (*file=None*)
outputs a manpage for the program using the help information

print_usage (*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version (*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile (*options*, *fullinputpath*)
process an individual file

recurseinputfilelist (*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles (*options*)
Recurse through directories and return files to be processed.

recursiveprocess (*options*)
recurse through directories and process files

run ()
Parses the arguments, and runs recursiveprocess with the resulting options...

set_usage (*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions ()
Sets the errorlevel options.

setformats (*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption ()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions ()
Sets the progress options.

splitext (*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext (*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext (*templatepath*)
Splits a *templatepath* into name and extension.

templateexists (*options*, *templatepath*)
Returns whether the given template exists...

warning (*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

posegment

Segment Gettext PO, XLIFF and TMX localization files at the sentence level.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/posegment.html for examples and usage instructions.

translate.tools.posegment.**segmentfile**(*inputfile*, *outputfile*, *templatefile*, *sourcelanguage='en'*, *targetlanguage=None*, *stripspaces=True*, *onlyaligned=False*)
reads in inputfile, segments it then, writes to outputfile

poswap

Builds a new translation file with the target of the input language as source language.

Note: Ensure that the two po files correspond 100% to the same pot file before using this.

To translate Kurdish (ku) through French:

```
poswap -i fr/ -t ku -o fr-ku
```

To convert the fr-ku files back to en-ku:

```
poswap --reverse -i fr/ -t fr-ku -o en-ku
```

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/poswap.html for examples and usage instructions.

translate.tools.poswap.**convertpo**(*inputpofile*, *outputpotfile*, *template*, *reverse=False*)
reads in inputpofile, removes the header, writes to outputpotfile.

translate.tools.poswap.**swapdir**(*store*)
Swap the source and target of each unit.

poterminology

Create a terminology file by reading a set of .po or .pot files to produce a pootle-terminology.pot.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/poterminology.html for examples and usage instructions.

class translate.tools.poterminology.**TerminologyOptionParser**(*formats*, *usetemplates=False*, *allowmissingtemplate=False*, *description=None*)
a specialized Option Parser for the terminology tool...

add_option(*Option*)
add_option(opt_str, ..., kwarg=val, ...)

check_values(*values : Values, args : [string]*)
-> (values : Values, args : [string])

Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

checkoutputsubdir(*options*, *subdir*)
Checks to see if subdir under options.output needs to be created, creates if neccessary.

define_option(*option*)
Defines the given option, replacing an existing one of the same short name if neccessary...

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don't get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(*msg : string*)
Print a usage message incorporating 'msg' to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

finalizetempoutputfile(*options*, *outputfile*, *fulloutputpath*)
Write the temp outputfile to its final destination.

format_manpage()
returns a formatted manpage

getformathelp(*formats*)
Make a nice help string for describing formats...

getfullinputpath(*options*, *inputpath*)
Gets the absolute path to an input file.

getfulloutputpath(*options*, *outputpath*)
Gets the absolute path to an output file.

getfulltemplatepath(*options*, *templatepath*)
Gets the absolute path to a template file.

getoutputname(*options*, *inputname*, *outputformat*)
Gets an output filename based on the input filename.

getoutputoptions(*options*, *inputpath*, *templatepath*)
Works out which output format and processor method to use...

getpassthroughoptions(*options*)
Get the options required to pass to the filtermethod...

gettemplatename(*options*, *inputname*)
Gets an output filename based on the input filename.

getusageman(*option*)
returns the usage string for the given option

getusagestring(*option*)
returns the usage string for the given option

isexcluded(*options*, *inputpath*)
Checks if this path has been excluded.

isrecursive(*fileoption*, *filepurpose='input'*)
Checks if fileoption is a recursive file.

isvalidinputname(*inputname*)
Checks if this is a valid input filename.

mkdir(*parent*, *subdir*)
Makes a subdirectory (recursively if neccessary).

openinputfile(*options*, *fullinputpath*)
Opens the input file.

openoutputfile(*options*, *fulloutputpath*)
Opens the output file.

opentemplatefile(*options*, *fulltemplatepath*)
Opens the template file (if required).

opentempoutputfile(*options*, *fulloutputpath*)
Opens a temporary output file.

outputterminology(*options*)
saves the generated terminology glossary

parse_args(*args=None*, *values=None*)
parses the command line options, handling implicit input/output args

print_help(*file : file = stdout*)
Print an extended help message, listing all options and any help text provided with them, to 'file' (default stdout).

print_manpage(*file=None*)
outputs a manpage for the program using the help information

print_usage(*file : file = stdout*)
Print the usage message for the current program (self.usage) to 'file' (default stdout). Any occurrence of the string "%prog" in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.

print_version(*file : file = stdout*)
Print the version message for this program (self.version) to 'file' (default stdout). As with print_usage(), any occurrence of "%prog" in self.version is replaced by the current program's name. Does nothing if self.version is empty or undefined.

processfile(*fileprocessor*, *options*, *fullinputpath*)
process an individual file

recurseinputfilelist(*options*)
Use a list of files, and find a common base directory for them.

recurseinputfiles(*options*)
Recurse through directories and return files to be processed.

recursiveprocess(*options*)
recurse through directories and process files

run()
parses the arguments, and runs recursiveprocess with the resulting options

set_usage(*usage=None*)
sets the usage string - if usage not given, uses getusagestring for each option

seterrorleveloptions()
Sets the errorlevel options.

setformats (*formats*, *usetemplates*)
Sets the format options using the given format dictionary.

Parameters **formats** (*Dictionary or iterable*) – The dictionary *keys* should be:

- Single strings (or 1-tuples) containing an input format (if not *usetemplates*)
- Tuples containing an input format and template format (if *usetemplates*)
- Formats can be *None* to indicate what to do with standard input

The dictionary *values* should be tuples of outputformat (string) and processor method.

setmanpageoption ()
creates a manpage option that allows the optionparser to generate a manpage

setprogressoptions ()
Sets the progress options.

splitext (*pathname*)
Splits *pathname* into name and ext, and removes the extsep.

Parameters **pathname** (*string*) – A file path

Returns root, ext

Return type tuple

splitinputext (*inputpath*)
Splits an *inputpath* into name and extension.

splittemplateext (*templatepath*)
Splits a *templatepath* into name and extension.

templateexists (*options*, *templatepath*)
Returns whether the given template exists...

warning (*msg*, *options=None*, *exc_info=None*)
Print a warning message incorporating 'msg' to stderr and exit.

pretranslate

Fill localization files with suggested translations based on translation memory and existing translations.

See: http://docs.translatehouse.org/projects/translate-toolkit/en/latest/commands/pretranslate.html for examples and usage instructions.

translate.tools.pretranslate.**match_fuzzy** (*input_unit*, *matchers*)
Return a fuzzy match from a queue of matchers.

translate.tools.pretranslate.**match_source** (*input_unit*, *template_store*)
Returns a matching unit from a template. matching based on unit id

translate.tools.pretranslate.**match_template_id** (*input_unit*, *template_store*)
Returns a matching unit from a template. matching based on unit id

translate.tools.pretranslate.**match_template_location** (*input_unit*, *template_store*)
Returns a matching unit from a template. matching based on locations

translate.tools.pretranslate.**memory** (*tmfiles*, *max_candidates=1*, *min_similarity=75*, *max_length=1000*)
Returns the TM store to use. Only initialises on first call.

translate.tools.pretranslate.**pretranslate_file**(*input_file*, *output_file*, *template_file*, *tm=None*, *min_similarity=75*, *fuzzymatching=True*)
Pretranslate any factory supported file with old translations and translation memory.

translate.tools.pretranslate.**pretranslate_store**(*input_store*, *template_store*, *tm=None*, *min_similarity=75*, *fuzzymatching=True*)
Do the actual pretranslation of a whole store.

translate.tools.pretranslate.**pretranslate_unit**(*input_unit*, *template_store*, *matchers=None*, *mark_reused=False*, *merge_on='id'*)
Pretranslate a unit or return unchanged if no translation was found.

Parameters

- **input_unit** – Unit that will be pretranslated.
- **template_store** – Fill input unit with units matching in this store.
- **matchers** – List of fuzzy *matcher* objects.
- **mark_reused** – Whether to mark old translations as reused or not.
- **merge_on** – Where will the merge matching happen on.

pydiff

diff tool like GNU diff, but lets you have special options that are useful in dealing with PO files

class translate.tools.pydiff.**DirDiffer**(*fromdir*, *todir*, *options*)
generates diffs between directories

isexcluded(*difffile*)
checks if the given filename has been excluded from the diff

writediff(*outfile*)
writes the actual diff to the given file

class translate.tools.pydiff.**FileDiffer**(*fromfile*, *tofile*, *options*)
generates diffs between files

get_from_lines(*group*)
returns the lines referred to by group, from the fromfile

get_to_lines(*group*)
returns the lines referred to by group, from the tofile

unified_diff(*group*)
takes the group of opcodes and generates a unified diff line by line

writediff(*outfile*)
writes the actual diff to the given file

translate.tools.pydiff.**main**()
main program for pydiff

pypo2phppo

Convert Python format .po files to PHP format .po files.

translate.tools.pypo2phppo.**convertpy2php**(*inputfile*, *outputfile*, *template=None*)
Converts from Python .po to PHP .po

Parameters

- **inputfile** – file handle of the source
- **outputfile** – file handle to write to
- **template** – unused

translate.tools.pypo2phppo.**main**(*argv=None*)
Converts from Python .po to PHP .po

Python Module Index

t

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z