gettext — Message Catalogs

Purpose: Message catalog API for internationalization.

The `gettext` module provides a pure-Python implementation compatible with the GNU **gettext** library for message translation and catalog management. The tools available with the Python source distribution enable you to extract messages from a set of source files, build a message catalog containing translations, and use that message catalog to display an appropriate message for the user at runtime.

Message catalogs can be used to provide internationalized interfaces for a program, showing messages in a language appropriate to the user. They can also be used for other message customizations, including “skinning” an interface for different wrappers or partners.

Note

Although the standard library documentation says all of the necessary tools are included with Python, `pygettext.py` failed to extract messages wrapped in the `ngettext` call, even with the appropriate command line options. These examples use `xgettext` from the GNU **gettext** tool set, instead.

Translation Workflow Overview

The process for setting up and using translations includes five steps.

1. *Identify and mark up literal strings in the source code that contain messages to translate.*

   Start by identifying the messages within the program source that need to be translated, and marking the literal strings so the extraction program can find them.

2. *Extract the messages.*

   After the translatable strings in the source are identified, use `xgettext` to extract them and create a `.pot` file, or *translation template*. The template is a text file with copies of all of the strings identified and placeholders for their translations.

3. *Translate the messages.*

   Give a copy of the `.pot` file to the translator, changing the extension to `.po`. The `.po` file is an editable source file used as input for the compilation step. The translator should update the header text in the file and provide translations for all of the strings.

4. *“Compile” the message catalog from the translation.*

   When the translator sends back the completed `.po` file, compile the text file to the binary catalog format using `msgfmt`. The binary format is used by the runtime catalog lookup code.

5. *Load and activate the appropriate message catalog at runtime.*

   The final step is to add a few lines to the application to configure and load the message catalog and install the translation function. There are a couple of ways to do that, with associated trade-offs.

The rest of this section will examine those steps in a little more detail, starting with the code modifications needed.

Creating Message Catalogs from Source Code

`gettext` works by looking up literal strings in a database of translations, and pulling out the appropriate translated string. The usual pattern is to bind the appropriate lookup function to the name “_” (a single underscore character) so that the code is not cluttered with a lot of calls to functions with longer names.

The message extraction program, `xgettext`, looks for messages embedded in calls to the catalog lookup functions. It understands different source languages and uses an appropriate parser for each. If the lookup functions are aliased, or extra functions are added, give `xgettext` the names of additional symbols to consider when extracting messages.

This script has a single message ready to be translated.

```
# gettext_example.py

import gettext

# Set up message catalog access
t = gettext.translation(
    'example_domain', 'locale',
    fallback=True,
)
_ = t.gettext

print(_('This message is in the script.'))
```

The text "`This message is in the script.`" is the message to be substituted from the catalog. Fallback mode is enabled, so if the script is run without a message catalog, the in-lined message is printed.

```
$ python3 gettext_example.py

This message is in the script.
```

The next step is to extract the message and create the `.pot` file, using `pygettext.py` or `xgettext`.

```
$ xgettext -o example.pot gettext_example.py
```

The output file produced contains the following content.

```
# example.pot

# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2018-03-18 16:20-0400\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: gettext_example.py:19
msgid "This message is in the script."
msgstr ""
```

Message catalogs are installed into directories organized by *domain* and *language*. The domain is provided by the application or library, and is usually a unique value like the application name. In this case, the domain in `gettext_example.py` is `example_domain`. The language value is provided by the user's environment at runtime, through one of the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, or `LANG`, depending on their configuration and platform. These examples were all run with the language set to `en_US`.

Now that the template is ready, the next step is to create the required directory structure and copy the template in to the right spot. The `locale` directory inside the PyMOTW source tree will serve as the root of the message catalog directory for these examples, but it is typically better to use a directory accessible system-wide so that all users have access to the message catalogs. The full path to the catalog input source is `$localedir/$language/LC_MESSAGES/$domain.po`, and the actual catalog has the filename extension `.mo`.

The catalog is created by copying `example.pot` to `locale/en_US/LC_MESSAGES/example.po` and editing it to change the values in the header and set the alternate messages. The result is shown next.

```
# locale/en_US/LC_MESSAGES/example.po

# Messages from gettext_example.py.
```

```
# Copyright (C) 2009 Doug Hellmann
# Doug Hellmann <doug@doughellmann.com>, 2016.
#
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW-3\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug@doughellmann.com>\n"
"POT-Creation-Date: 2016-01-24 13:04-0500\n"
"PO-Revision-Date: 2016-01-24 13:04-0500\n"
"Last-Translator: Doug Hellmann <doug@doughellmann.com>\n"
"Language-Team: US English <doug@doughellmann.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"


#: gettext_example.py:16
msgid "This message is in the script."
msgstr "This message is in the en_US catalog."
```

The catalog is built from the `.po` file using `msgformat`.

```
$ cd locale/en_US/LC_MESSAGES; msgfmt -o example.mo example.po
```

The domain in `gettext_example.py` is `example_domain`, but the file is called `example.pot`. To have `gettext` find the right translation file, the names need to match.

```
# gettext_example_corrected.py

t = gettext.translation(
    'example', 'locale',
    fallback=True,
)
```

Now when the script is run, the message from the catalog is printed instead of the in-line string.

```
$ python3 gettext_example_corrected.py

This message is in the en_US catalog.
```

Finding Message Catalogs at Runtime

As described earlier, the *locale directory* containing the message catalogs is organized based on the language with catalogs named for the *domain* of the program. Different operating systems define their own default value, but `gettext` does not know all of these defaults. It uses a default locale directory of `sys.prefix + '/share/locale'`, but most of the time it is safer to always explicitly give a `localedir` value than to depend on this default being valid. The `find()` function is responsible for locating an appropriate message catalog at runtime.

```
# gettext_find.py

import gettext

catalogs = gettext.find('example', 'locale', all=True)
print('Catalogs:', catalogs)
```

The language portion of the path is taken from one of several environment variables that can be used to configure localization features (`LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`). The first variable found to be set is used. Multiple languages can be selected by separating the values with a colon (`:`). To see how that works, use a second message catalog to run a few experiments.

```
$ cd locale/en_CA/LC_MESSAGES; msgfmt -o example.mo example.po
$ cd ../../..
$ python3 gettext_find.py

Catalogs: ['locale/en_US/LC_MESSAGES/example.mo']

$ LANGUAGE=en_CA python3 gettext_find.py
```

```
Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo']

$ LANGUAGE=en_CA:en_US python3 gettext_find.py

Catalogs: ['locale/en_CA/LC_MESSAGES/example.mo',
'locale/en_US/LC_MESSAGES/example.mo']

$ LANGUAGE=en_US:en_CA python3 gettext_find.py

Catalogs: ['locale/en_US/LC_MESSAGES/example.mo',
'locale/en_CA/LC_MESSAGES/example.mo']
```

Although `find()` shows the complete list of catalogs, only the first one in the sequence is actually loaded for message lookups.

```
$ python3 gettext_example_corrected.py

This message is in the en_US catalog.

$ LANGUAGE=en_CA python3 gettext_example_corrected.py

This message is in the en_CA catalog.

$ LANGUAGE=en_CA:en_US python3 gettext_example_corrected.py

This message is in the en_CA catalog.

$ LANGUAGE=en_US:en_CA python3 gettext_example_corrected.py

This message is in the en_US catalog.
```

Plural Values

While simple message substitution will handle most translation needs, `gettext` treats pluralization as a special case. Depending on the language, the difference between the singular and plural forms of a message may vary only by the ending of a single word, or the entire sentence structure may be different. There may also be different forms depending on the level of plurality. To make managing plurals easier (and, in some cases, possible), there is a separate set of functions for asking for the plural form of a message.

```
# gettext_plural.py

from gettext import translation
import sys

t = translation('plural', 'locale', fallback=False)
num = int(sys.argv[1])
msg = t.ngettext('{num} means singular.',
                 '{num} means plural.',
                 num)

# Still need to add the values to the message ourself.
print(msg.format(num=num))
```

Use `ngettext()` to access the plural substitution for a message. The arguments are the messages to be translated and the item count.

```
$ xgettext -L Python -o plural.pot gettext_plural.py
```

Since there are alternate forms to be translated, the replacements are listed in an array. Using an array allows translations for languages with multiple plural forms (for example, Polish has different forms indicating the relative quantity).

```
# plural.pot

# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
```

```
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2018-03-18 16:20-0400\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"

#: gettext_plural.py:15
#, python-brace-format
msgid "{num} means singular."
msgid_plural "{num} means plural."
msgstr[0] ""
msgstr[1] ""
```

In addition to filling in the translation strings, the library needs to be told about the way plurals are formed so it knows how to index into the array for any given count value. The line `"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"` includes two values to replace manually. `nplurals` is an integer indicating the size of the array (the number of translations used) and `plural` is a C language expression for converting the incoming quantity to an index in the array when looking up the translation. The literal string `n` is replaced with the quantity passed to `ungettext()`.

For example, English includes two plural forms. A quantity of `0` is treated as plural ("0 bananas"). The `Plural-Forms` entry is:

```
Plural-Forms: nplurals=2; plural=n != 1;
```

The singular translation would then go in position 0, and the plural translation in position 1.

```
# locale/en_US/LC_MESSAGES/plural.po

# Messages from gettext_plural.py
# Copyright (C) 2009 Doug Hellmann
# This file is distributed under the same license
# as the PyMOTW package.
# Doug Hellmann <doug@doughellmann.com>, 2016.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PyMOTW-3\n"
"Report-Msgid-Bugs-To: Doug Hellmann <doug@doughellmann.com>\n"
"POT-Creation-Date: 2016-01-24 13:04-0500\n"
"PO-Revision-Date: 2016-01-24 13:04-0500\n"
"Last-Translator: Doug Hellmann <doug@doughellmann.com>\n"
"Language-Team: en_US <doug@doughellmann.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=n != 1;"

#: gettext_plural.py:15
#, python-format
msgid "{num} means singular."
msgid_plural "{num} means plural."
msgstr[0] "In en_US, {num} is singular."
msgstr[1] "In en_US, {num} is plural."
```

Running the test script a few times after the catalog is compiled will demonstrate how different values of N are converted to indexes for the translation strings.

```
$ cd locale/en_US/LC_MESSAGES/; msgfmt -o plural.mo plural.po
$ cd ../../..
$ python3 gettext_plural.py 0
```

```
In en_US, 0 is plural.

$ python3 gettext_plural.py 1

In en_US, 1 is singular.

$ python3 gettext_plural.py 2

In en_US, 2 is plural.
```

Application vs. Module Localization

The scope of a translation effort defines how `gettext` is installed and used with a body of code.

Application Localization

For application-wide translations, it is acceptable for the author to install a function like `ngettext()` globally using the `__builtins__` namespace, because they have control over the top-level of the application's code.

```
# gettext_app_builtin.py
```

```
import gettext

gettext.install(
    'example',
    'locale',
    names=['ngettext'],
)

print(_('This message is in the script.'))
```

The `install()` function binds `gettext()` to the name `_()` in the `__builtins__` namespace. It also adds `ngettext()` and other functions listed in `names`.

Module Localization

For a library or individual module, modifying `__builtins__` is not a good idea because it may introduce conflicts with an application global value. Instead, import or re-bind the names of translation functions by hand at the top of the module.

```
# gettext_module_global.py

import gettext

t = gettext.translation(
    'example',
    'locale',
    fallback=False,
)
_ = t.gettext
ngettext = t.ngettext

print(_('This message is in the script.'))
```

Switching Translations

The earlier examples all use a single translation for the duration of the program. Some situations, especially web applications, need to use different message catalogs at different times, without exiting and resetting the environment. For those cases, the class-based API provided in `gettext` will be more convenient. The API calls are essentially the same as the global calls described in this section, but the message catalog object is exposed and can be manipulated directly, so that multiple catalogs can be used.

See also

- Standard library documentation for gettext
- `locale` – Other localization tools.
- GNU gettext – The message catalog formats, API, etc. for this module are all based on the original gettext package

from GNU. The catalog file formats are compatible, and the command line scripts have similar options (if not identical). The GNU gettext manual has a detailed description of the file formats and describes GNU versions of the tools for working with them.
- Plural forms – Handling of plural forms of words and sentences in different languages.
- Internationalizing Python – A paper by Martin von Löwis about techniques for internationalization of Python applications.
- Django Internationalization – Another good source of information on using gettext, including real-life examples.

Internationalization and Localization

locale — Cultural Localization API

Quick Links

This page was last updated 2016-12-18.

Navigation

Internationalization and Localization
locale — Cultural Localization API

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?