23.1. `gettext` — Multilingual internationalization services¶

The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

23.1.1. GNU gettext API¶

The `gettext` module defines the following API, which is very similar to the GNU **gettext** API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

`gettext.bindtextdomain`(*domain*[, *localedir*])¶

Bind the *domain* to the locale directory *localedir*. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where *languages* is searched for in the environment variables **LANGUAGE**, **LC_ALL**, **LC_MESSAGES**, and **LANG** respectively.

If *localedir* is omitted or `None`, then the current binding for *domain* is returned. [1]

`gettext.bind_textdomain_codeset`(*domain*[, *codeset*])¶

Bind the *domain* to *codeset*, changing the encoding of strings returned by the `gettext()` family of functions. If *codeset* is omitted, then the current binding is returned.

New in version 2.4.

`gettext.textdomain`([*domain*])¶
Change or query the current global domain. If *domain* is `None`, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

`gettext.gettext`(*message*)¶
Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

`gettext.lgettext`(*message*)¶

Equivalent to `gettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

`gettext.dgettext`(*domain*, *message*)¶
Like `gettext()`, but look the message up in the specified *domain*.

`gettext.ldgettext`(*domain*, *message*)¶

Equivalent to `dgettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

`gettext.ngettext`(*singular*, *plural*, *n*)¶

Like `gettext()`, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in `.po` files and the formulas for a variety of languages.

New in version 2.3.

`gettext.lngettext`(*singular*, *plural*, *n*)¶

Equivalent to `ngettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

`gettext.dngettext`(*domain*, *singular*, *plural*, *n*)¶

Like `ngettext()`, but look the message up in the specified *domain*.

New in version 2.3.

`gettext.ldngettext`(*domain*, *singular*, *plural*, *n*)¶

Equivalent to `dngettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `bind_textdomain_codeset()`.

New in version 2.4.

Note that GNU **gettext** also defines a `dcgettext()` method, but this was deemed not useful and so it is currently unimplemented.

Here's an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print _('This is a translatable string.')
```

23.1.2. Class-based API¶

The class-based API of the `gettext` module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a "translations" class which implements the parsing of GNU `.mo` format files, and has methods for returning either standard 8-bit strings or Unicode strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function `_()`.

`gettext.find`(*domain*[, *localedir*[, *languages*[, *all*]]])¶

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what `textdomain()` takes. Optional *localedir* is as in `bindtextdomain()` Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used. [2] If *languages* is not given, then the following environment variables are searched: **LANGUAGE**, **LC_ALL**, **LC_MESSAGES**, and **LANG**. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

`find()` then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

```
localedir/language/LC_MESSAGES/domain.mo
```

The first such file name that exists is returned by `find()`. If no such file is found, then `None` is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

`gettext.translation`(*domain*[, *localedir*[, *languages*[, *class_*[, *fallback*[, *codeset*]]]]])¶

Return a `Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to `find()` to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is either *class_* if provided, otherwise `GNUTranslations`. The class's constructor must take a single file object argument. If provided, *codeset* will change the charset used to encode translated strings.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, `copy.copy()` is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises `IOError` if *fallback* is false (which is the default), and returns a `NullTranslations` instance if *fallback* is true.

Changed in version 2.4: Added the *codeset* parameter.

`gettext.install`(*domain*[, *localedir*[, *unicode*[, *codeset*[, *names*]]]])¶

This installs the function `_()` in Python's builtins namespace, based on *domain*, *localedir*, and *codeset* which are passed to the function `translation()`. The *unicode* flag is passed to the resulting translation object's `install()` method.

For the *names* parameter, please see the description of the translation object's `install()` method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this:

```
print _('This string will be translated.')
```

For convenience, you want the `_()` function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

Changed in version 2.4: Added the *codeset* parameter.

Changed in version 2.5: Added the *names* parameter.

23.1.2.1. The `NullTranslations` class¶

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is `NullTranslations`; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

class `gettext.NullTranslations`([*fp*])¶

Takes an optional file object *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through `add_fallback()`. It then calls `self._parse(fp)` if *fp* is not `None`.

`_parse`(*fp*)¶
No-op'd in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

`add_fallback`(*fallback*)¶
Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

`gettext`(*message*)¶
If a fallback has been set, forward `gettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

`lgettext`(*message*)¶

If a fallback has been set, forward `lgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.

`ugettext`(*message*)¶
If a fallback has been set, forward `ugettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

`ngettext`(*singular*, *plural*, *n*)¶

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.3.

`lngettext`(*singular*, *plural*, *n*)¶

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

New in version 2.4.

`ungettext`(*singular*, *plural*, *n*)¶

If a fallback has been set, forward `ungettext()` to the fallback. Otherwise, return the translated message as a Unicode string. Overridden in derived classes.

New in version 2.3.

`info`()¶
Return the "protected" `_info` variable.

`charset`()¶
Return the "protected" `_charset` variable.

`output_charset`()¶

Return the "protected" `_output_charset` variable, which defines the encoding used to return translated messages.

New in version 2.4.

`set_output_charset`(*charset*)¶

Change the "protected" `_output_charset` variable, which defines the encoding used to return translated messages.

New in version 2.4.

`install([unicode[, names]])`¶

If the *unicode* flag is false, this method installs `self.gettext()` into the built-in namespace, binding it to `_`. If *unicode* is true, it binds `self.ugettext()` instead. By default, *unicode* is false.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'` (bound to `self.gettext()` or `self.ugettext()` according to the *unicode* flag), `'ngettext'` (bound to `self.ngettext()` or `self.ungettext()` according to the *unicode* flag), `'lgettext'` and `'lngettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module's global namespace and so only affects calls within this module.

Changed in version 2.5: Added the *names* parameter.

23.1.2.2. The `GNUTranslations` class¶

The `gettext` module provides one additional class derived from `NullTranslations`: `GNUTranslations`. This class overrides `_parse()` to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format. It also coerces both message ids and message strings to Unicode.

`GNUTranslations` parses optional meta-data out of the translation catalog. It is convention with GNU **gettext** to include meta-data as the translation for the empty string. This meta-data is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding. The `ugettext()` method always returns a Unicode, while the `gettext()` returns an encoded 8-bit string. For the message id arguments of both methods, either Unicode strings or 8-bit strings containing only US-ASCII characters are acceptable. Note that the Unicode version of the methods (i.e. `ugettext()` and `ungettext()`) are the recommended interface to use for internationalized Python programs.

The entire set of key/value pairs are placed into a dictionary and set as the "protected" `_info` instance variable.

If the `.mo` file's magic number is invalid, or if other problems occur while reading the file, instantiating a `GNUTranslations` class can raise `IOError`.

The following methods are overridden from the base class implementation:

`GNUTranslations.gettext(message)`¶
Look up the *message* id in the catalog and return the corresponding message string, as an 8-bit string encoded with the catalog's charset encoding, if known. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's `gettext()` method. Otherwise, the *message* id is returned.

`GNUTranslations.lgettext(message)`¶

Equivalent to `gettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

New in version 2.4.

`GNUTranslations.ugettext(message)`¶
Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's `ugettext()` method. Otherwise, the *message* id is returned.

`GNUTranslations.ngettext(singular, plural, n)`¶

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is an 8-bit string encoded with the catalog's charset encoding, if known.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `ngettext()` method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

New in version 2.3.

`GNUTranslations.lngettext(singular, plural, n)`¶

Equivalent to `gettext()`, but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with `set_output_charset()`.

New in version 2.4.

`GNUTranslations.ungettext(singular, plural, n)`¶

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `ungettext()` method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ungettext(
   'There is %(num)d file in this directory',
   'There are %(num)d files in this directory',
   n) % {'num': n}
```

New in version 2.3.

23.1.2.3. Solaris message catalog support¶

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

23.1.2.4. The Catalog constructor¶

GNOME uses a version of the `gettext` module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print _('hello world')
```

For compatibility with this older module, the function `Catalog()` is an alias for the `translation()` function described above.

One difference between this module and Henstridge's: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

23.1.3. Internationalizing your programs and modules¶

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the `gettext` module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` — that is, a call to the function `_()`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

The Python distribution comes with two tools which help you generate the message catalogs once you've prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the `Tools/i18n` directory.

The **pygettext** [3] program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU **gettext** program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don't need GNU `gettext` unless you're also going to be translating C code (such as C extension modules).

pygettext generates textual Uniforum-style human readable message catalog `.pot` files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. **pygettext** is a command line script that supports a similar command line interface as **xgettext**; for details on its use, run:

```
pygettext.py --help
```

Copies of these `.pot` files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a `.po` file. Using the **msgfmt.py** [4] program (in the `Tools/i18n` directory), you take the `.po` files from your translators and generate the machine-readable `.mo` binary catalog files. The `.mo` files are what the `gettext` module uses for the actual translation processing during run-time.

How you use the `gettext` module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

23.1.3.1. Localizing your module¶

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU `gettext` API but instead the class-based API.

Let's say your module is called "spam" and the module's various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here's what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

If your translators were providing you with Unicode strings in their `.po` files, you'd instead do:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.ugettext
```

23.1.3.2. Localizing your application¶

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory or the *unicode* flag, you can pass these into the `install()` function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale', unicode=1)
```

23.1.3.3. Changing languages on the fly¶

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

23.1.3.4. Deferred translations¶

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
```

```
# ...
for a in animals:
   print a
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
   print _(a)
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify “a” as being translatable to the **pygettext** program, since it is not a string.

Another way to handle this is with the following example:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
   print _(a)
```

In this case, you are marking translatable strings with the function `N_()`, [5] which won’t conflict with any definition of `_()`. However, you will need to teach your message extraction program to look for translatable strings marked with `N_()`. **pygettext** and **xpot** both support this through the use of command line switches.

23.1.3.5. `gettext()` vs. `lgettext()`¶

In Python 2.4 the `lgettext()` family of functions were introduced. The intention of these functions is to provide an alternative which is more compliant with the current implementation of GNU gettext. Unlike `gettext()`, which returns strings encoded with the same codeset used in the translation file, `lgettext()` will return strings encoded with the preferred system encoding, as returned by `locale.getpreferredencoding()`. Also notice that Python 2.4 introduces new functions to explicitly choose the codeset used in translated strings. If a codeset is explicitly set, even `lgettext()` will return translated strings in the requested codeset, as would be expected in the GNU gettext implementation.

23.1.4. Acknowledgements¶

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

Footnotes

[1] The default locale directory is system dependent; for example, on RedHat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults; instead its default is `sys.prefix/share/locale`. For this reason, it is always best to call `bindtextdomain()` with an explicit absolute path at the start of your application.

[2] See the footnote for `bindtextdomain()` above.

[3] François Pinard has written a program called **xpot** which does a similar job. It is available as part of his **po-utils** package at http ://po-utils.progiciels-bpi.ca/.

[4] **msgfmt.py** is binary compatible with GNU **msgfmt** except that it provides a simpler, all-Python implementation. With this and **pygettext.py**, you generally won't need to install the GNU **gettext** package to internationalize your Python applications.

[5] The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

Table Of Contents