

23.1. `gettext` — Multilingual internationalization services

Source code: [Lib/gettext.py](#)

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=None)`

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=None)`

Bind the *domain* to *codeset*, changing the encoding of byte strings returned by the `lgettext()`, `ldgettext()`, `lngettext()` and `ldngettext()` functions. If *codeset* is omitted, then the current binding is returned.

`gettext.textdomain(domain=None)`

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.dgettext(domain, message)`

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

`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.

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

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

`gettext.lgettext(message)`

`gettext.ldgettext(domain, message)`

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

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

Equivalent to the corresponding functions without the `l` prefix (`gettext()`, `dgettext()`, `ngettext()` and `dngettext()`), but the translation is returned as a byte string encoded in the preferred system encoding if no other encoding was explicitly set with `bind_textdomain_codeset()`.

Warning: These functions should be avoided in Python 3, because they return encoded bytes. It's much better to use alternatives which return Unicode strings instead, since most Python applications will want to manipulate human readable text as strings instead of bytes. Further, it's possible that you may get unexpected Unicode-related exceptions if there are encoding problems

with the translated strings. It is possible that the `l*()` functions will be deprecated in future Python versions due to their inherent problems and limitations.

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/director
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 strings. Instances of this “translations” class can also install themselves in the built-in namespace as the function `_()`.

`gettext.find(domain, localedir=None, languages=None, all=False)`

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=None, languages=None, class_=None, fallback=False, codeset=None)`

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 in the `gettext()` and `gettext()` methods.

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 `OSError` if *fallback* is false (which is the default), and returns a `NullTranslations` instance if *fallback* is true.

Changed in version 3.3: `IOError` used to be raised instead of `OSError`.

`gettext.install(domain, localedir=None, codeset=None, names=None)`

This installs the function `_()` in Python's builtins namespace, based on *domain*, *localedir*, and *codeset* which are passed to the function `translation()`.

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.

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=None*)

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 *message*. Overridden in derived classes.

`ngettext(singular, plural, n)`

If a fallback has been set, forward `ngettext()` to the fallback. Otherwise, return *singular* if *n* is 1; return *plural* otherwise. Overridden in derived classes.

`lgettext(message)`

`lgettext(singular, plural, n)`

Equivalent to [gettext\(\)](#) and [ngettext\(\)](#), but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with [set_output_charset\(\)](#). Overridden in derived classes.

Warning: These methods should be avoided in Python 3. See the warning for the [lgettext\(\)](#) function.

`info()`

Return the “protected” `_info` variable.

`charset()`

Return the encoding of the message catalog file.

`output_charset()`

Return the encoding used to return translated messages in `gettext()` and `lgettext()`.

set_output_charset(charset)

Change the encoding used to return translated messages.

install(names=None)

This method installs `gettext()` into the built-in namespace, binding it to `_`.

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', 'ngettext', '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.

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.

`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, else ASCII encoding is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

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, the major version number is unexpected, or if other problems occur while reading the file, instantiating a `GNUTranslations` class can raise `OSError`.

`class gettext.GNUTranslations`

The following methods are overridden from the base class implementation:

gettext(*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 `gettext()` method. Otherwise, the *message* id is returned.

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 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 `ngettext()` 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.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext(*message*)

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

Equivalent to `gettext()` and `ngettext()`, but the translation is returned as a byte string encoded in the preferred system encoding if no encoding was explicitly set with `set_output_charset()`.

Warning: These methods should be avoided in Python 3. See the warning for the `lgettext()` function.

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.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. [Babel](#) is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard's program called **xpot** does a similar job and is available as part of his [po-utils package](#).

(Python also includes pure-Python versions of these programs, called **pygettext.py** and **msgfmt.py**; some Python distributions will install them for you. **pygettext.py** is similar to **xgettext**, but only understands Python source code and cannot handle other programming languages such as C or C++. **pygettext.py** supports a command-line interface similar to **xgettext**; for details on its use, run `pygettext.py --help`. **msgfmt.py** is binary compatible with GNU **msgfmt**. With these two programs, you may not need the GNU **gettext** package to internationalize your Python applications.)

xgettext, **pygettext**, and similar tools generate `.po` files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these `.po` files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a `<language-name>.po` file that's compiled into a machine-readable `.mo` binary catalog file using the **msgfmt** program. The `.mo` files are used by the [gettext](#) module for the actual translation processing at 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.gettext
```

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, you can pass it into the `install()` function:

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

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 **gettext** program, because the parameter is not a string literal.

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_()`, 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_()`. **xgettext**, **pygettext**, **pybabel extract**, and **xpot** all support this through the use of the `-k` command-line switch. The choice of `N_()` here is totally arbitrary; it could have just as easily been `MarkThisStringForTranslation()`.

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.