translationstring Documentation

Release 1.4.dev0

Pylons Developers

August 29, 2017

Contents

A library used by various Pylons Project packages for internationalization (i18n) duties.

This package provides a *translation string* class, a *translation string factory* class, translation and pluralization primitives, and a utility that helps *Chameleon* templates use translation facilities of this package. It does not depend on *Babel*, but its translation and pluralization services are meant to work best when provided with an instance of the `babel.support.Translations` class.

CHAPTER 1

Translation Strings

While you write your software, you can insert specialized markup into your Python code that makes it possible for the system to translate text values into the languages used by your application's users. This markup generates a *translation string*. A translation string is an object that behave mostly like a normal Unicode object, except that it also carries around extra information related to its job as part of a higher-level system's translation machinery.

Note: Using a translation string can be thought of as equivalent to using a "lazy string" object in other i18n systems.

Using The `TranslationString` Class

The most primitive way to create a translation string is to use the *translationstring.TranslationString* callable:

```
from translationstring import TranslationString
ts = TranslationString('Add')
```

This creates a Unicode-like object that is a *translationstring.TranslationString*.

Note: For people familiar with Zope internationalization, a TranslationString is a lot like a `zope.i18nmessageid.Message` object. It is not a subclass, however.

The first argument to *translationstring.TranslationString* is the `msgid`; it is required. It represents the key into the translation mappings provided by a particular localization. The `msgid` argument must be a Unicode object or an ASCII string. The msgid may optionally contain *replacement markers*. For instance:

```
from translationstring import TranslationString
ts = TranslationString('Add ${number}')
```

Within the string above, `${stuff}` is a replacement marker. It will be replaced by whatever is in the *mapping* for a translation string when the *translationstring.TranslationString.interpolate()* method is called. The mapping may be supplied at the same time as the replacement marker itself:

```
from translationstring import TranslationString
ts = TranslationString('Add ${number}', mapping={'number':1})
```

You can also create a new translation string instance with a mapping using the standard python %-operator:

```
from translationstring import TranslationString
ts = TranslationString('Add ${number}') % {'number': 1}
```

You may interpolate a translation string with a mapping:

```
from translationstring import TranslationString
ts = TranslationString('Add ${number}', mapping={'number':1})
result = ts.interpolate()
```

The above `result` will be `Add 1`.

Any number of replacement markers can be present in the msgid value, any number of times. Only markers which can be replaced by the values in the *mapping* will be replaced at translation time. The others will not be interpolated and will be output literally.

Replacement markers may also be spelled without squiggly braces:

```
from translationstring import TranslationString
ts = TranslationString('Add $number', mapping={'number':1})
```

The `Add $number` msgid above is equivalent to `Add ${number}`.

A translation string should also usually carry a *domain*. The domain represents a translation category to disambiguate it from other translations of the same msgid, in case they conflict.

```
from translationstring import TranslationString
ts = TranslationString('Add ${number}', mapping={'number':1},
                       domain='form')
```

The above translation string named a domain of `form`. A *translator* function (see *Translation*) will often use the domain to locate the right translator file on the filesystem which contains translations for a given domain. In this case, if it were trying to translate to our msgid to German, it might try to find a translation from a *gettext* file within a *translation directory* like this one:

```
locale/de/LC_MESSAGES/form.mo
```

In other words, it would want to take translations from the `form.mo` translation file in the German language.

Finally, the TranslationString constructor accepts a `default` argument. If a `default` argument is supplied, it replaces usages of the `msgid` as the *default value* for the translation string. When `default` is `None`, the `msgid` value passed to a TranslationString is used as an implicit message identifier. Message identifiers are matched with translations in translation files, so it is often useful to create translation strings with "opaque" message identifiers unrelated to their default text:

```
from translationstring import TranslationString
ts = TranslationString('add-number', default='Add ${number}',
                       domain='form', mapping={'number':1})
```

When a `default` value is used, the default may contain replacement markers and the msgid should not contain replacement markers.

Using the `TranslationStringFactory` Class

Another way to generate a translation string is to use the *translationstring.TranslationStringFactory* object. This object is a *translation string factory*. Basically a translation string factory presets the `domain` value of any *translation string* generated by using it. For example:

```
from translationstring import TranslationStringFactory
_ = TranslationStringFactory('bfg')
ts = _('add-number', default='Add ${number}', mapping={'number':1})
```

Note: We assigned the translation string factory to the name _. This is a convention which will be supported by translation file generation tools.

After assigning _ to the result of a *translationstring.TranslationStringFactory()*, the subsequent result of calling _ will be a *translationstring.TranslationString* instance. Even though a `domain` value was not passed to _ (as would have been necessary if the *translationstring.TranslationString* constructor were used instead of a translation string factory), the `domain` attribute of the resulting translation string will be `bfg`. As a result, the previous code example is completely equivalent (except for spelling) to:

```
from translationstring import TranslationString as _
ts = _('add-number', default='Add ${number}', mapping={'number':1},
       domain='bfg')
```

You can set up your own translation string factory much like the one provided above by using the *translationstring.TranslationStringFactory* class. For example, if you'd like to create a translation string factory which presets the `domain` value of generated translation strings to `form`, you'd do something like this:

```
from translationstring import TranslationStringFactory
_ = TranslationStringFactory('form')
ts = _('add-number', default='Add ${number}', mapping={'number':1})
```

Note: For people familiar with Zope internationalization, a TranslationStringFactory is a lot like a `zope.i18nmessageid.MessageFactoy` object. It is not a subclass, however.

Pickleability

Translation strings may be pickled and unpickled.

CHAPTER 2

Translation

translationstring provides a function named *translationstring.Translator()* which is used to create a *translator* object.

It is called like so:

```
import gettext
from translationstring import Translator
translations = gettext.translations(.. the right arguments ...)
translator = Translator(translations)
```

The `translations` argument is required; it should be an object supporting *at least* the Python `gettext.NullTranslations` API but ideally the `babel.support.Translations` API, which has support for domain lookups like dugettext.

The callable returned accepts three arguments: a translation string `tstring` (required), `domain` (optional), and `mapping` (optional). When called, it will translate the `tstring` translation string to a `unicode` object using the `translations` object provided and interpolate the result.

```
from gettext import translations
from translationstring import Translator
from translationstring import TranslationString

t = translations(.. the right arguments ...)
translator = Translator(t)
ts = TranslationString('Add ${number}', domain='foo', mapping={'number':1})
translator(ts)
```

If `translations` is `None`, the result of interpolation of the msgid or default value of the translation string is returned.

The translation function can also deal with plain Unicode objects. The optional `domain` argument can be used to specify or override the domain of the `tstring` argument (useful when `tstring` is a normal string rather than a translation string). The optional `mapping` argument can specify the interpolation mapping, useful when the `tstring` argument is not a translation string. If `tstring` is a translation string its mapping data, if present, is combined with the data from the `mapping` argument.

```
from gettext import translations
from translationstring import Translator
from translationstring import TranslationString

t = translations(.. the right arguments ...)
translator = Translator(t)
translator('Add ${number}', domain='foo', mapping={'number':1})
```

The *translationstring.Translator()* function accepts an additional optional argument named `policy`. `policy` should be a callable which accepts three arguments: `translations`, `tstring` and `domain`. It must perform the actual translation lookup. If `policy` is `None`, the *translationstring.dugettext_policy()* policy will be used.

CHAPTER 3

Pluralization

translationstring.Pluralizer() provides a gettext "plural forms" pluralization service.

It is called like so:

```
import gettext
from translationstring import Pluralizer
translations = gettext.translations(.. the right arguments ...)
pluralizer = Pluralizer(translations)
```

The `translations` argument is required; it should be an object supporting *at least* the Python `gettext.NullTranslations` API but ideally the `babel.support.Translations` API, which has support for domain lookups like dungettext.

The object returned will be a callable which has the following signature:

```
def pluralizer(singular, plural, n, domain=None, mapping=None):
    """ Pluralize """
```

The `singular` and `plural` arguments passed may be translation strings or unicode strings. `n` represents the number of elements. `domain` is the translation domain to use to do the pluralization, and `mapping` is the interpolation mapping that should be used on the result. The pluralizer will return the plural form or the singular form, translated, as necessary.

Note: if the objects passed are translation strings, their domains and mappings are ignored. The domain and mapping arguments must be used instead. If the `domain` is not supplied, a default domain is used (usually `messages`).

If `translations` is `None`, a `gettext.NullTranslations` object is created for the pluralizer to use.

The *translationstring.Pluralizer()* function accepts an additional optional argument named `policy`. `policy` should be a callable which accepts five arguments: `translations`, `singular` and `plural`, `n` and `domain`. It must perform the actual pluralization lookup. If `policy` is `None`, the *translationstring.dungettext_policy()* policy will be used.

CHAPTER 4

Chameleon Translate Function Support

translationstring.ChameleonTranslate() is a function which returns a callable suitable for use as the `translate` argument to various `PageTemplate*` constructors.

```
from chameleon.zpt.template import PageTemplate
from translationstring import ChameleonTranslate
from translationstring import Translator
import gettext

translations = gettext.translations(...)
translator = Translator(translations)
translate = ChameleonTranslate(translate)
pt = PageTemplate('<html></html>', translate=translate)
```

The `translator` provided should be a callable which accepts a single argument `translation_string` (a *translationstring.TranslationString* instance) which returns a `unicode` object as a translation; usually the result of calling *translationstring.Translator()*. `translator` may also optionally be `None`, in which case no translation is performed (the `msgid` or `default` value is returned untranslated).

CHAPTER 5

API Documentation

class `TranslationString`

The constructor for a *translation string*. A translation string is a Unicode-like object that has some extra metadata.

This constructor accepts one required argument named `msgid`. `msgid` must be the *message identifier* for the translation string. It must be a `unicode` object or a `str` object encoded in the default system encoding.

Optional keyword arguments to this object's constructor include `domain`, `default`, and `mapping`.

`domain` represents the *translation domain*. By default, the translation domain is `None`, indicating that this translation string is associated with the default translation domain (usually `messages`).

`default` represents an explicit *default text* for this translation string. Default text appears when the translation string cannot be translated. Usually, the `msgid` of a translation string serves double duty as its default text. However, using this option you can provide a different default text for this translation string. This feature is useful when the default of a translation string is too complicated or too long to be used as a message identifier. If `default` is provided, it must be a `unicode` object or a `str` object encoded in the default system encoding (usually means ASCII). If `default` is `None` (its default value), the `msgid` value used by this translation string will be assumed to be the value of `default`.

`mapping`, if supplied, must be a dictionary-like object which represents the replacement values for any *translation string replacement marker* instances found within the `msgid` (or `default`) value of this translation string.

`context` represents the translation context. By default, the translation context is `None`.

After a translation string is constructed, it behaves like most other `unicode` objects; its `msgid` value will be displayed when it is treated like a `unicode` object. Only when its `ugettext` method is called will it be translated.

Its default value is available as the `default` attribute of the object, its *translation domain* is available as the `domain` attribute, and the `mapping` is available as the `mapping` attribute. The object otherwise behaves much like a Unicode string.

`interpolate`(*translated=None*)

Interpolate the value `translated` which is assumed to be a Unicode object containing zero or more

replacement markers (`$foo` or `${bar}`) using the `mapping` dictionary attached to this instance. If the `mapping` dictionary is empty or `None`, no interpolation is performed.

If `translated` is `None`, interpolation will be performed against the `default` value.

`TranslationStringFactory` (*factory_domain*)
Create a factory which will generate translation strings without requiring that each call to the factory be passed a `domain` value. A single argument is passed to this class' constructor: `domain`. This value will be used as the `domain` values of *`translationstring.TranslationString`* objects generated by the `__call__` of this class. The `msgid`, `mapping`, and `default` values provided to the `__call__` method of an instance of this class have the meaning as described by the constructor of the *`translationstring.TranslationString`*

`ChameleonTranslate` (*translator*)
When necessary, use the result of calling this function as a Chameleon template 'translate' function (e.g. the `translate` argument to the `chameleon.zpt.template.PageTemplate` constructor) to allow our translation machinery to drive template translation. A single required argument `translator` is passsed. The `translator` provided should be a callable which accepts a single argument `translation_string` (a *`translationstring.TranslationString`* instance) which returns a `unicode` object as a translation. `translator` may also optionally be `None`, in which case no translation is performed (the `msgid` or `default` value is returned untranslated).

`Translator` (*translations=None*, *policy=None*)
Return a translator object based on the `translations` and `policy` provided. `translations` should be an object supporting *at least* the Python `gettext.NullTranslations` API but ideally the `babel.support.Translations` API, which has support for domain lookups like dugettext.

`policy` should be a callable which accepts three arguments: `translations`, `tstring` and `domain`. It must perform the actual translation lookup. If `policy` is `None`, the *`translationstring.dugettext_policy()`* policy will be used.

The callable returned accepts three arguments: `tstring` (required), `domain` (optional) and `mapping` (optional). When called, it will translate the `tstring` translation string to a `unicode` object using the `translations` provided. If `translations` is `None`, the result of interpolation of the default value is returned. The optional `domain` argument can be used to specify or override the domain of the `tstring` (useful when `tstring` is a normal string rather than a translation string). The optional `mapping` argument can specify or override the `tstring` interpolation mapping, useful when the `tstring` argument is a simple string instead of a translation string.

`dugettext_policy` (*translations*, *tstring*, *domain*, *context*)
A translator policy function which assumes the use of a `babel.support.Translations` translations object, which supports the dugettext API; fall back to ugettext.

`ugettext_policy` (*translations*, *tstring*, *domain*, *context*)
A translator policy function which unconditionally uses the `ugettext` API on the translations object.

`Pluralizer` (*translations=None*, *policy=None*)
Return a pluralizer object based on the `translations` and `policy` provided. `translations` should be an object supporting *at least* the Python `gettext.NullTranslations` API but ideally the `babel.support.Translations` API, which has support for domain lookups like dugettext.

`policy` should be a callable which accepts five arguments: `translations`, `singular` and `plural`, `n` and `domain`. It must perform the actual pluralization lookup. If `policy` is `None`, the *`translationstring.dungettext_policy()`* policy will be used.

The object returned will be a callable which has the following signature:

```
def pluralizer(singular, plural, n, domain=None, mapping=None):
    ...
```

The `singular` and `plural` objects passed may be translation strings or unicode strings. `n` represents the number of elements. `domain` is the translation domain to use to do the pluralization, and `mapping` is the interpolation mapping that should be used on the result. Note that if the objects passed are translation strings, their domains and mappings are ignored. The domain and mapping arguments must be used instead. If the `domain` is not supplied, a default domain is used (usually `messages`).

dungettext_policy(*translations*, *singular*, *plural*, *n*, *domain*, *context*)
A pluralizer policy function which assumes the use of the `babel.support.Translations` class, which supports the dungettext API; falls back to ungettext.

ungettext_policy(*translations*, *singular*, *plural*, *n*, *domain*, *context*)
A pluralizer policy function which unconditionally uses the `ungettext` API on the translations object.

CHAPTER 6

Glossary

Babel A collection of tools for internationalizing Python applications.

Chameleon chameleon is templating language written and maintained by Malthe Borch.

Gettext The GNU gettext library, used by the *translationstring* locale translation machinery.

Message Identifier An unchanging string that is the identifier for a particular translation string. For example, you may have a translation string which has the `default` "the fox jumps over the lazy dog", but you might give this translation string a message identifier of `foxdog` to reduce the chances of minor spelling or wording changes breaking your translations. The message identifier of a *translation string* is represented as its `msgid` argument.

Translation Directory A translation directory is a *gettext* translation directory. It contains language folders, which themselves contain `LC_MESSAGES` folders, which contain `.mo` files. Each `.mo` file represents a set of translations for a language in a *translation domain*. The name of the `.mo` file (minus the .mo extension) is the translation domain name.

Translation Domain A string representing the "context" in which a particular translation was made. For example the word "java" might be translated differently if the translation domain is "programming-languages" than would be if the translation domain was "coffee". Every *translation string* has an associated translation domain.

Translation String An instance of *translationstring.TranslationString*, which is a class that behaves like a Unicode string, but has several extra attributes such as `domain`, `msgid`, and `mapping` for use during translation. Translation strings are usually created by hand within software, but are sometimes created on the behalf of the system for automatic template translation. For more information, see *API Documentation*.

Translation String Factory A factory for generating *translation string* objects which predefines a *translation domain*.

Translator A callable which receives a *translation string* and returns a translated Unicode object for the purposes of internationalization.

CHAPTER 7

Index and Glossary

- *Glossary*
- genindex
- modindex
- search

Python Module Index

t

Index