CHAPTER 11

Internationalization and Localization

Internationalization and localization are means of adapting software for non-native environments, especially for other nations and cultures.

The following are the parts of an application that might need to be localized:

- Language
- Date/time format
- Numbers such as decimal points, positioning of separators, and characters used as separators
- Time zones (UTC in internationalized environments)
- Currency
- Weights and measures

The distinction between internationalization and localization is subtle but important. *Internationalization* is the adaptation of products for potential use virtually everywhere, while *localization* is the addition of special features for use in a specific locale.

For example, in terms of language used in a Pylons application, internationalization is the process of marking up all strings that might need to be translated, and localization is the process of producing translations for a particular locale.

Pylons provides built-in support to enable you to internationalize language but leaves you to handle for yourself any other aspects of internationalization that might be appropriate for your application. In this chapter, you'll concentrate on how to internationalize and localize the strings used in your Pylons application.

Note Internationalization is often abbreviated as *I18N* (or *i18n* or *I18n*) where the number 18 refers to the number of letters omitted. Localization is often abbreviated *L10n* or *l10n* in the same manner. These abbreviations also avoid picking one spelling (*internationalisation* vs. *internationalization*) over the other.

To represent characters from multiple languages, you will need to utilize Unicode. By now you should have a good idea of what Unicode is, how to use it in Python, and in which areas of your application you need to pay specific attention to decoding and encoding Unicode data. If not, you should read the previous chapter.

Understanding the Process

Internationalizing a Pylons application involves marking every string in your application that needs to be available in more than one language with a function that will perform the necessary translation.

Localizing an application involves the following steps:

1. Running a tool to extract the strings you've marked
2. Creating a translation of the strings for each language your application will support
3. Displaying the correct translation for the current user

Before you learn how this works in a real application, I'll discuss some of the background information you will need to understand.

Marking Strings for Internationalization

Marking the strings that will need to be internationalized is actually very simple. You wrap them in a function to tell Pylons that they need to be translated. For example, if you had a controller action that looked like this:

```
def hello(self):
    return u'Hello!'
```

and you wanted to internationalize the string `Hello!`, you would do so by wrapping it in one of the Pylons translation functions. You import them like this:

```
from pylons.i18n.translation import _, ungettext
```

and use them like this:

```
def hello(self):
    return _(u'Hello!')
```

Strings to be internationalized are known as *messages* in internationalization terminology. The correct translation function to use depends on the situation. Pylons currently provides the following functions in the `pylons.i18n.translation` module:

`ugettext()` and its alias `_()`: These mark and translate a Unicode message. Developers usually choose to use `_()` rather than `ugettext()` because as well as being a well-understood convention it also saves on keystrokes.

`ungettext()`: This marks and translates a Unicode message that might have a slightly different form for the singular and the plural. You'll learn about this in the "Plural Forms" section later in the chapter.

Pylons also provides `gettext()` and `ngettext()` functions, which are equivalent to `ugettext()` and `ungettext()`, respectively, but take an ordinary Python string as their argument rather than a Unicode string. There is really no reason to use the non-Unicode versions if you have read the previous chapter and understand what Unicode is, so I won't cover the non-Unicode versions further.

When wrapping strings in the `_()` (or `ugettext()`) function, it is important not to piece sentences together manually because certain languages might need to invert the grammar. Instead, you should try to specify whole strings in one go. For example, you shouldn't do this:

```
# BAD!
msg = _("Starbug was built to last sir; this old baby's crashed ")
msg += _("more times than a ZX81.")
```

But the following is perfectly acceptable because the whole string is passed as an argument to `_()`:

```
# GOOD
msg = _("Starbug was built to last sir; this old baby's crashed "
    "more times than a ZX81.")
```

Python will automatically concatenate adjacent strings, so having two shorter strings on separate lines like this is a perfectly acceptable way to pass a long string to a function. This is surprising behavior if you haven't seen it before, but it is quite useful in this situation.

Extracting Messages and Handling Translations

As you might have guessed from the names of the functions you've already seen, Pylons internationalization support is based on GNU gettext (`http://www.gnu.org/software/gettext/`). The idea is simple: you run a tool on your source code that searches for times when you have used any of the Python I18N functions such as `_()`, `ungettext()`, and others. The tool extracts the Unicode strings passed as arguments to the functions and places them in a portable object template (`.pot`) file.

You would then generate a portable object (`.po`) file based on the portable object template for each language you wanted to support. You send these to a translator, and they will return the portable object file together with the translations of the messages for a particular locale.

Finally, the `.po` files are compiled by a tool to a machine object (`.mo`) file, which is an optimized machine-readable binary file that the Pylons internationalization tools can understand.

You can use a few different tools to extract messages from your source files and to compile the `.mo` files, but by far the most popular for use with Pylons applications is Babel (`http://babel.edgewall.org/`). Unlike the GNU gettext tool `xgettext`, Babel supports extracting translatable strings from Python templating languages (including Mako and Genshi) and has a plug-in architecture to allow it to extract messages from other types of Python source files.

Babel is comprised of two main parts:

- Tools to build and work with gettext message catalogs
- A Python interface to the Common Locale Data Repository (CLDR), providing access to various locale display names, localized number and date formatting, and so on

In the next section, you'll look at a real example that uses Babel to help in the internationalization/localization process.

Seeing It in Action

Now that you've seen the theory and understand the process, it's time to see it in action in a Pylons application. The application you'll write will simply display the greeting "Hello" in three different languages: English, Spanish, and French. The default language will be English. I'll show how to use the Pylons translator functions together with Babel to create the application.

Let's call the project `TranslateDemo` and choose to use Mako but not SQLAlchemy:

```
$ paster create --template=pylons TranslateDemo
```

Now let's add the controller:

```
$ cd TranslateDemo
$ paster controller hello
```

Edit `controllers/hello.py`, and import the `_()` function:

```
from pylons.i18n.translation import _, set_lang
```

Then update the `index()` action to look like this:

```
class HelloController(BaseController):

    def index(self):
        set_lang('es')
        return _(u'Hello!')
```

Notice that the example uses the `_()` function everywhere the string `Hello!` appears. Start the Paste HTTP server with the `--reload` option, and visit `http://localhost:5000/hello/index`:

```
$ paster serve --reload development.ini
```

You will see the following error:

```
LanguageError: IOError: [Errno 2] No translation file found for➥
domain: 'translatedemo'
```

Although the controller has been internationalized, no message catalogs are yet in place. You'll create the necessary translations next.

Using Babel

You'll need to install Babel using Easy Install:

```
$ easy_install "Babel==0.9.4"
```

You'll use Babel to extract messages to a `.pot` file in your project's `i18n` directory. First, the directory needs to be created. Don't forget to add it to your revision control system if you're using one:

```
$ mkdir translatedemo/i18n
```

Be sure to use the number `1` in the word `i18n` and not a lowercase `l`. Next, extract all the messages from the project with the following command:

```
$ python setup.py extract_messages
 extract_messages
running extract_messages
extracting messages from translatedemo/__init__.py
extracting messages from translatedemo/websetup.py
extracting messages from translatedemo/config/__init__.py
extracting messages from translatedemo/config/environment.py
extracting messages from translatedemo/config/middleware.py
extracting messages from translatedemo/config/routing.py
extracting messages from translatedemo/controllers/__init__.py
extracting messages from translatedemo/controllers/error.py
extracting messages from translatedemo/controllers/hello.py
extracting messages from translatedemo/lib/__init__.py
extracting messages from translatedemo/lib/app_globals.py
extracting messages from translatedemo/lib/base.py
extracting messages from translatedemo/lib/helpers.py
extracting messages from translatedemo/model/__init__.py
extracting messages from translatedemo/tests/__init__.py
```

```
extracting messages from translatedemo/tests/test_models.py
extracting messages from translatedemo/tests/functional/__init__.py
extracting messages from translatedemo/tests/functional/test_hello.py
writing PO template file to translatedemo/i18n/translatedemo.pot
```

As you can see, Babel searches your project for Python files looking for strings marked with the translation functions. The strings are extracted, and a new file is created called `translatedemo.pot` in your project's `translatedemo/i18n` directory. It looks like something like this:

```
# Translations template for TranslateDemo.
# Copyright (C) 2008 ORGANIZATION
# This file is distributed under the same license as the TranslateDemo project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2008.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: TranslateDemo 0.1\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2008-09-26 11:07+0100\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"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 0.9.4\n"

#: translatedemo/controllers/hello.py:18
msgid "Hello!"
msgstr ""
```

As you can see, after the heading information, it has found the one internationalized string in the `hello.py` controller. The `:18` means that this string was found on line 18 of the file.

Next you'll need to create a `.po` file for the Spanish language. You can do so with this command:

```
$ python setup.py init_catalog -l es
running init_catalog
creating catalog 'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
```

The new `translatedemo.po` file looks similar, but some of the content has been updated slightly for the Spanish language:

```
# Spanish translations for TranslateDemo.
# Copyright (C) 2008 ORGANIZATION
# This file is distributed under the same license as the TranslateDemo
# project.
# FIRST AUTHOR <EMAIL@ADDRESS>, 2008.
#
msgid ""
msgstr ""
"Project-Id-Version: TranslateDemo 0.1\n"
"Report-Msgid-Bugs-To: EMAIL@ADDRESS\n"
"POT-Creation-Date: 2008-09-26 10:35+0100\n"
"PO-Revision-Date: 2008-09-26 10:49+0100\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: es <LL@li.org>\n"
```

```
"Plural-Forms: nplurals=2; plural=(n != 1)\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: Babel 0.9.4\n"

#: translatedemo/controllers/hello.py:18
msgid "Hello!"
msgstr ""
```

Update the `.po` file with the correct details including your name, your e-mail address, and the revision date.

Edit the last line to specify the Spanish translation for the greeting `Hello!`, and save the changes, making sure you set the encoding in your editor to UTF-8 if you are using Unicode characters. (Don't worry if you can't find the ¡ character on your keyboard; this is only an example, so you can leave it out if you prefer):

```
#: translatedemo/controllers/hello.py:18
msgid "Hello!"
msgstr "¡Hola!"
```

Now that the translation is in place, you need to compile the `.po` file to a `.mo` file. Once again, Babel has a tool to help, and you access it via your project's `setup.py` file:

```
$ python setup.py compile_catalog
running compile_catalog
1 of 1 messages (100%) translated in ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po'
compiling catalog ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' to ➥
    'translatedemo/i18n/es/LC_MESSAGES/translatedemo.mo'
```

Now is a good time to test the application. Start the server with the following command:

```
$ paster serve --reload development.ini
```

Test your controller by visiting `http://localhost:5000/hello/index`. You should see the following output:

```
¡Hola!
```

Congratulations, you've internationalized and localized a Pylons application!

Supporting Multiple Languages

Supporting one language is useful, but Pylons can support multiple languages at once. To do so, you simply repeat the process of generating the `.po` and `.mo` files for the other languages. You don't need to extract the messages again, though, because the existing `.pot` file already contains all the information you need.

As an example, let's also create French and English translations:

```
$ python setup.py init_catalog -l fr
running init_catalog
creating catalog 'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
```

```
$ python setup.py init_catalog -l en
running init_catalog
creating catalog 'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
```

Modify the last lines of the `fr` catalog to look like this:

```
#: translatedemo/controllers/hello.py:18
msgid "Hello!"
msgstr "Bonjour!"
```

Since the original messages are already in English, the `en` catalog `msgstr` string can stay blank because gettext will fall back to the original.

Once you've edited the two new `.po` files, compile them to `.mo` files like this:

```
$ python setup.py compile_catalog
running compile_catalog
0 of 1 messages (0%) translated in ➥
'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po' compiling catalog ➥
'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/en/LC_MESSAGES/translatedemo.mo'
1 of 1 messages (100%) translated in ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' compiling catalog ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.mo'
1 of 1 messages (100%) translated in ➥
'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po' compiling catalog ➥
'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.mo'
```

By the end of the process, your `i18n` directory will contain these files:

```
i18n/translatedemo.pot
i18n/en/LC_MESSAGES/translatedemo.po
i18n/en/LC_MESSAGES/translatedemo.mo
i18n/es/LC_MESSAGES/translatedemo.po
i18n/es/LC_MESSAGES/translatedemo.mo
i18n/fr/LC_MESSAGES/translatedemo.po
i18n/fr/LC_MESSAGES/translatedemo.mo
```

If you look at your project's `setup.py` file, you'll see the following line:

```
package_data={'translatedemo': ['i18n/*/LC_MESSAGES/*.mo']},
```

This line ensures that all the binary message catalogs your application relies on are automatically included in any packages or egg files produced from your Pylons project.

With the changes in place, add a new action to the controller to test the different languages. First import the `get_lang()` function, which retrieves the current language being used:

```
from pylons.i18n.translation import get_lang
```

Now add the following action to the controller:

```
def multiple(self):
    resp = 'Default: %s<br />' % _(u'Hello!')
    for lang in ['fr','en','es']:
        set_lang(lang)
        resp += '%s: %s<br />' % (get_lang(), _(u'Hello!'))
    return resp
```

Start the server, and visit `http://localhost:5000/hello/multiple`. You should see the following output:

```
Default: Hello!
['fr']: Bonjour!
['en']: Hello!
['es']: ¡Hola!
```

This correctly outputs the three different languages you have prepared. The `set_lang()` function is called for each language, this changes the message catalog used so that the correct translation is produced when the `_()` function is called. Now that you know how to set the language used in a controller on the fly, let's look at how to update the message catalogs.

Updating the Catalog

You'll notice that the previous example worked even though `Hello!` is marked for translation in multiple places and not just on the line specified in the `.po` and `.pot` files. The Pylons translation functions don't use the line number a message is defined on, only the message itself, so as long as a translation of the message for the particular language exists somewhere, the message can be translated everywhere. The line numbers are used by Babel to keep track of which strings have been translated.

If you want to add a new message to the catalog, you'll need to update the `.po` and `.pot` files. Once again, Babel provides a tool to help. To demonstrate this, update the original `index()` action to look like this:

```
def index(self):
    set_lang('es')
    return _('Goodbye!')
```

You'll need to run the extract command again (I've omitted some of the output for brevity):

```
$ python setup.py extract_messages
running extract_messages
...
extracting messages from translatedemo/controllers/hello.py
...
writing PO template file to translatedemo/i18n/translatedemo.pot
```

The last lines of the file look like this:

```
#: translatedemo/controllers/hello.py:19
msgid "Goodbye!"
msgstr ""

#: translatedemo/controllers/hello.py:22 translatedemo/controllers/hello.py:25
msgid "Hello!"
msgstr ""
```

Notice that both of the lines the `Hello!` message are found on have been noted and that the `Goodbye!` message is also included.

Now run the command to update the catalogs:

```
$ python setup.py update_catalog
running update_catalog
updating catalog 'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
updating catalog 'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
updating catalog 'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po' ➥
based on 'translatedemo/i18n/translatedemo.pot'
```

You'd then update all the `.po` files and compile them to `.mo` files again, but since you are using only the Spanish language in the action in this example, you'll just edit the Spanish `.po` file. Its last few lines now look like this:

```
#: translatedemo/controllers/hello.py:19
msgid "Goodbye!"
msgstr ""

#: translatedemo/controllers/hello.py:22 translatedemo/controllers/hello.py:25
msgid "Hello!"
msgstr "¡Hola!"
```

Notice that Babel added the entry for the `Goodbye!` message but kept the existing translation for `Hello!`, updating the line numbers to reflect those in the new `.pot` file.

Update the `msgstr` line for `Goodbye!` to read as follows:

```
msgstr "¡Adiós!"
```

and recompile the catalog:

```
$ python setup.py compile_catalog
running compile_catalog
0 of 2 messages (0%) translated in ➥
'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po'
compiling catalog 'translatedemo/i18n/en/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/en/LC_MESSAGES/translatedemo.mo'
2 of 2 messages (100%) translated in ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po'
compiling catalog 'translatedemo/i18n/es/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/es/LC_MESSAGES/translatedemo.mo'
1 of 2 messages (50%) translated in ➥
'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po'
compiling catalog 'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.po' to ➥
'translatedemo/i18n/fr/LC_MESSAGES/translatedemo.mo'
```

If you start the Paste HTTP server and visit `http://localhost:5000/hello/index`, you will see the message updated to read `¡Adiós!`. You have successfully updated the message catalog without losing any of the existing translated messages.

Translations Within Templates

Although some of the messages you will want to internationalize will appear in controllers, most of them are likely to be found in your project's templates. Luckily, the internationalization and localization tools work in templates too.

Let's change the `multiple()` action to use a template instead. Update it to look like this:

```
def multiple(self):
    return render('/hello.html')
```

Then create `hello.html` in your project's `template` directory with this content:

```
<%!
    from pylons.i18n.translation import set_lang, get_lang
%>
<html>
<head>
    <title>Multiple Translations</title>
</head>
```

```
<body>
    <h1>Multiple Translations</h1>
    <p>
        Default: ${_(u'Hello!')}<br />
        % for lang in ['fr','en','es']:
            <% set_lang(lang) %>
            ${get_lang()}:  ${_(u'Hello!')}<br />
        % endfor
    </p>
</body>
</html>
```

The statement at the top is used to import the `get_lang()` and `set_lang()` functions, but Pylons automatically puts the `translator`, `ungettext()`, `_()`, and `N_()` functions into the template namespace, so you don't need to import the `_()` function in this case.

If you test the example again, you will see that the same output is produced, only this time in an HTML template with a title.

It turns out that using messages that have already been translated in a template file is easy because the Pylons translation functions are called in the same way they would be from within a controller. What is much harder is extracting the strings that need to be internationalized from a template in the first place because the extraction tools won't necessarily understand the syntax of the templating language.

One approach that was recommended in very early versions of Pylons was to run a standard tool such as `xgettext` on the Mako (or Myghty as it was then) cache directory, which, after heavy development testing, would contain a cached Python version of every template. The extraction tool will then run on the cached templates in the same way it runs on ordinary Python files. Although this process worked adequately, it was rather cumbersome. Babel provides a much better solution, which you'll learn about next.

Babel Extractors

Babel supports the concepts of extractors. These are custom functions that, when called by Babel, will perform the work of extracting internationalizable messages from a particular source file. Babel currently supports extracting gettext messages from Mako and Genshi templates as well as from Python source files.

If you look at your project's `setup.py` files, you will see the following lines commented out:

```
#message_extractors = {'translatedemo': [
#        ('**.py', 'python', None),
#        ('templates/**.mako', 'mako', None),
#        ('public/**', 'ignore', None)]},
```

If you uncomment them, Babel will know that any `.py` files should be treated as Python source code, any `.mako` files should be treated as Mako templates, and everything in the `public` directory should be ignored. You'll also want `.html` files in the `templates` directory treated as Mako source, so update these lines to look like this:

```
message_extractors = {'translatedemo': [
        ('**.py', 'python', None),
        ('templates/**.mako', 'mako', None),
        ('templates/**.html', 'mako', None),
        ('public/**', 'ignore', None)]},
```

For a project using Genshi instead of Mako, the Mako lines might be replaced with this:

```
('templates/**.html, 'genshi', None),
```

Similar options can also be set in the `setup.cfg` file if you prefer. See `http://babel.edgewall.org/wiki/Documentation/cmdline.html#extract` for details.

Once you've changed the `setup.py` file, you may need to run this command again to reinstall the package in development mode so that the changes are recognized:

```
$ python setup.py develop
```

With the changes made, you can use Babel to extract messages from the templates as well as the Python source files in your project:

```
$ python setup.py extract_messages
running extract_messages
extracting messages from translatedemo/__init__.py
extracting messages from translatedemo/websetup.py
extracting messages from translatedemo/config/__init__.py
extracting messages from translatedemo/config/environment.py
extracting messages from translatedemo/config/middleware.py
extracting messages from translatedemo/config/routing.py
extracting messages from translatedemo/controllers/__init__.py
extracting messages from translatedemo/controllers/error.py
extracting messages from translatedemo/controllers/hello.py
extracting messages from translatedemo/lib/__init__.py
extracting messages from translatedemo/lib/app_globals.py
extracting messages from translatedemo/lib/base.py
extracting messages from translatedemo/lib/helpers.py
extracting messages from translatedemo/model/__init__.py
extracting messages from translatedemo/templates/hello.html
extracting messages from translatedemo/tests/__init__.py
extracting messages from translatedemo/tests/test_models.py
extracting messages from translatedemo/tests/functional/__init__.py
extracting messages from translatedemo/tests/functional/test_hello.py
writing PO template file to translatedemo/i18n/translatedemo.pot
```

Notice this time that the `templates/hello.html` file is included in the extraction process.

Tip If you want to create a Babel extractor for a template language you use or another source file type, you should read the documentation at `http://babel.edgewall.org/wiki/Documentation/messages.html#writing-extraction-methods`.

Setting the Language in the Config File

Pylons supports defining the default language to be used in the config file. Set a `lang` variable to the desired default language in your `development.ini` file, and Pylons will automatically call `set_lang()` with that language at the beginning of every request.

For example, to set the default language to French, you would add `lang = fr` to your `development.ini` file:

```
[app:main]
use = egg:translatedemo
lang = fr
```

If you are running the server with the `--reload` option, the server will automatically restart if you change the `development.ini` file. Otherwise, restart the server manually, and the output would this time be as follows:

```
Default: Bonjour!
fr: Bonjour!
en: Hello!
es: ¡Hola!
```

Using a Session to Store the User's Language

In a real application, the language to be used is likely to be set on each request so that each user sees pages in their own language. In this section, I'll show you one way of setting this up.

You'll remember from Chapter 9 that each controller can have a `__before__()` method, which is run before each controller action. This is a great place to set the language to be used for that controller. You can also use Pylons' `session` global to store the language to be used when the user signs in. Replace the `index()` action with this:

```
def signin(self):
    # Place your sign in code here
    # Replace this with code to set the language for the signed in user
    session['lang'] = request.params.getone('lang')
    session.save()
    return 'Signed in, language set to %s.'%request.params.getone('lang')

def __before__(self):
    if 'lang' in session:
        set_lang(session['lang'])

def index(self):
    return _(u'Hello!')
```

When a user signs in, their language is set and saved in the session. On each subsequent request, the language is looked up in the session and set automatically before each action is called.

To test this, visit `http://localhost:5000/hello/signin?lang=es`. You'll see the message telling you have been signed in. Now visit `http://localhost:5000/hello/index`, and you'll see the message in the language you signed in with, which in this case is Spanish.

If you want to set the language for each request in every controller rather than dealing with each controller individually, you could write some similar code at the top of the `lib/base.py` `BaseController` class's `__call__()` method instead of in an individual controller's `__before__()` method:

```
def __call__(self, environ, start_response):
    """Invoke the Controller"""
    # WSGIController.__call__ dispatches to the Controller method
    # the request is routed to. This routing information is
    # available in environ['pylons.routes_dict']

    if 'lang' in session:
        set_lang(session['lang'])

    return WSGIController.__call__(self, environ, start_response)
```

You'll need to import the `set_lang()` function and the `session` global into that module, though:

```
from pylons.i18n.translation import set_lang
from pylons import session
```

If you delete the page controller's `__before__()` method and visit `http://localhost:5000/hello/multiple` again, notice that the default language is still the language you signed in with.

Advanced Internationalization Techniques

Now that you've seen how internationalization and localization work in practice in a Pylons application, I can show you some of the more advanced techniques that will make your internationalization and localization work easier.

Fallback Languages

You've already seen that if your code calls `_()` with a string that doesn't exist in the language catalog already being used, then the string itself will be returned. This is how the English version of the `Hello!` message has been produced in the examples so far despite that the English `.po` file doesn't contain the translation explicitly. Although this is useful, Pylons also provides a much more sophisticated mechanism that allows you to specify the order in which other message catalogs should be searched for a message if the primary language doesn't have a suitable translation.

If you have been following along with the examples so far, then French will be set as the default language in the config file, and Spanish is set as the default language in the session. Set the default language in the session to be French too by visiting `http://localhost:5000/hello/signin?lang=fr`. Now change the `index()` action to use the word `'Goodbye!'`:

```
def index(self):
    return _('Goodbye!')
```

You'll remember that you didn't add a translation for the `Goodbye!` message to the French message catalog, but you did add it to the Spanish one.

With the current language set to French, visit `http://localhost:5000/hello/index`, and you will see `Goodbye!` as the original text passed to the function is used because no French translation is available.

Now change the example to add Spanish as a fallback language like this (if you are using the `__call__()` version of the example, you will need to make the change in `lib/base.py`):

```
def __before__(self):
    add_fallback('es')
    if 'lang' in session:
        set_lang(session['lang'])
```

This tells Pylons to look for messages in the Spanish catalog if there are no translations in the French catalog. You'll need to import the `add_fallback()`function at the top of the file:

```
from pylons.i18n.translation import add_fallback
```

■ **Caution** It is important that the call to `add_fallback()` happens before the call to `set_lang()`, or the fallback will not be used.

If you run the example again, you will see the message ¡Adiós!.

You can add as many fallback languages with the add_fallback() function as you like, and they will be tested in the order you add them.

One case where using fallbacks in this way is particularly useful is when you want to display content based on the languages requested by the browser in the HTTP_ACCEPT_LANGUAGE header. Typically the browser may submit a number of languages, so it is useful to add fallbacks in the order specified by the browser so that you always try to display words in the language of preference of the user, searching the other languages in order if a translation cannot be found. The languages defined in the HTTP_ACCEPT_LANGAUGE header are available in Pylons as request.languages and can be used like this:

```
def __before__(self):
    for lang in request.languages:
        add_fallback(lang)
    if 'lang' in session:
        set_lang(session['lang'])
```

You must be sure you have the appropriate languages supported if you are going to use this approach, or you should test their existence before calling add_fallback().

Lazy Translations

Occasionally you might come across a situation when you need to translate a string when it is accessed, not when the _() or other functions are called.

Consider this example:

```
set_lang('en')
text = _(u'Hello!')

class HelloController(BaseController):

    def lazy(self):
        resp = ''
        for lang in ['fr','en','es']:
            set_lang(lang)
            resp += u'%s: %s<br />' % (get_lang(), _(u'Hello!'))
        resp += u'Text: %s<br />' % text
        return resp
```

If you run this, you get the following output:

```
['fr']: Bonjour!
['en']: Hello!
['es']: ¡Hola!
Text: Hello!
```

Notice that the text line shows Hello! even though the current language at the time the text was generated was Spanish. This is because the function _(u'Hello') just before the controller definition and after the imports is called when the default language is en, so the variable text gets the value of the English translation, even though when the string was used, the default language was Spanish.

The rule of thumb in these situations is to try to avoid using the translation functions in situations where they are not executed on each request. For situations where this isn't possible, perhaps because you are working with legacy code or with a library that doesn't support internationalization, you need to use lazy translations.

Modify the code to use lazy translations. Notice that the `text` variable is assigned its message with the `lazy_ugettext()` function:

```
from pylons.i18n import get_lang, lazy_ugettext, set_lang

set_lang('en')
text = lazy_ugettext(u'Hello!')

class HelloController(BaseController):

    def lazy(self):
        resp = ''
        for lang in ['fr','en','es']:
            set_lang(lang)
            resp += u'%s: %s<br />' % (get_lang(), _(u'Hello!'))
        resp += u'Text: %s<br />' % text
        return resp
```

This time you get the output expected:

```
['fr']: Bonjour!
['en']: Hello!
['es']: ¡Hola!
Text: ¡Hola!
```

There is one drawback to be aware of when using the lazy translation functions: they are not actually strings. This means that if our example had used the following code, it would have failed with the error `cannot concatenate 'str' and 'LazyString' objects`:

```
u'Text: ' + text + u'<br />'
```

For this reason, you should use the lazy translations only where absolutely necessary. Always ensure they are converted to strings by calling `str()` or `repr()` before they are used in operations with real strings.

Plural Forms

One thing to keep in mind when you are internationalizing an application is that other languages don't necessarily have the same plural forms as English. Although English has only two forms, singular and plural, Slovenian, for example, has singular, dual, and plural, which means that in Slovenian one thing, two things, and three things would all be treated differently! That means that the following will not work:

```
# BAD!
if n == 1:
    msg = _("There is one person here")
else:
    msg = _("There are %(num)d people here") % {'num': n}
```

Pylons provides the `ungettext()` function internationalizing plural words. It can be used as follows:

```
translated_string = ungettext(
    u'There is %(num)d person here',
    u'There are %(num)d people here',
    4
)
```

If you use 1 in the call to `ugettext()`, the first form will be used; if you use 0 or a number greater than 1, then the second form will be used. Table 11-1 lists the options.

Table 11-1. *The Results from the Original Message Catalog*

Number	Result
0	`There are %(num)d people here`
1	`There is %(num)d person here`
2	`There are %(num)d people here`

As you can see, no matter which translation is used, the result still contains the text `%(num)d`, so you can use standard Python string formatting to substitute the correct value into the translation after the correct form has been chosen for you:

```
final_string = translated_string % {'num': 4}
```

If you added the previous code to the sample project and reran the `extract_messages` tool, you would find some lines similar to these in your `.pot` file:

```
#: translatedemo/controllers/hello.py:27
#, python-format
msgid "There is %(num)d person here"
msgid_plural "There are %(num)d people here"
msgstr[0] ""
msgstr[1] ""
```

The `msgstr[0]` and `msgstr[1]` lines give you the opportunity to customize the plurals for different languages. You could then run `update_catalog` to update the `.po` files. Looking at the Spanish `.po` file, you would see this (some lines have been omitted for brevity):

```
...
"Plural-Forms: nplurals=2; plural=(n != 1)\n"
...
...
#: translatedemo/controllers/hello.py:27
#, python-format
msgid "There is %(num)d person here"
msgid_plural "There are %(num)d people here"
msgstr[0] ""
msgstr[1] ""
...
```

The `Plural-Forms` definition at the top describes which plural version to use. In Spanish there are two plurals, and the plural form should be used whenever the number of items isn't 1. The translation for the singular form should go next to `msgstr[0]`, and the translation for plural should go next to `msgstr[1]`. When you recompile the message catalog, the `ungettext()` function will return the correct form of the translation for the number passed to it.

You can find further details about how plural forms should be dealt with in the "The Format of PO Files" section of GNU gettext's manual (`http://www.gnu.org/software/gettext/manual/html_chapter/gettext_10.html#PO-Files`).

Search Engine Considerations

One issue to be aware of when using a session to determine which version of a site to display, and dynamically generating that page based on the language of the user, is that search engines will be able to search the site in the default language only. This can be a problem if you want search engines to be able to index all the pages on your web site, including the foreign-language versions. In these situations, it can be better to use a different URL for each language.

To implement this, you might set up your URLs so that the first part of the URL path represents the language to be used. For example, the English version of your site might be at `/en`, and the Japanese version might be at `/ja`. You can then use Routes to extract the first part of the URL and treat the rest of the URL as the page being requested, using the internationalization tools to produce the correct language version based on the language specified as the first part of the URL.

■ **Note** The makers of the Opera web browser used a similar technique for the Pylons-based website at `http://widgets.opera.com`. You can read more about the details of this implementation at `http://my.opera.com/WebApplications/blog/2008/03/17/search-engine-friendly`.

Summary

This chapter covered the basics of internationalizing and localizing a web application. GNU gettext is an extensive library, and the GNU gettext manual is highly recommended for more information. Although the Pylons internationalization tools don't support all the features of gettext, they do support the most commonly used ones, as you've seen in this chapter. The Babel package is also improving all the time and includes tools not covered in this chapter for providing access to various locale display names, localized number and date formatting, time zones, and more.