- Python Cookbook

- Comments Off
- Chapters

Table of Contents

Chapter 10. Modules and Packages

Modules and packages are the core of any large project, and the Python installation itself. This chapter focuses on common programming techniques involving modules and packages, such as how to organize packages, splitting large modules into multiple files, and creating namespace packages. Recipes that allow you to customize the operation of the `import` statement itself are also given.

Making a Hierarchical Package of Modules

Problem

You want to organize your code into a package consisting of a hierarchical collection of modules.

Solution

Making a package structure is simple. Just organize your code as you wish on the file-system and make sure that every directory defines an *__init__.py* file. For example:

```
    graphics/
        __init__.py
        primitive/
             __init__.py
             line.py
             fill.py
             text.py
        formats/
             __init__.py
             png.py
             jpg.py
```

Once you have done this, you should be able to perform various `import` statements, such as the following:

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

Discussion

Defining a hierarchy of modules is as easy as making a directory structure on the filesystem. The purpose of the *__init__.py* files is to include optional initialization code that runs as different levels of a package are encountered. For example, if you have the statement `import graphics`, the file *graphics/__init__.py* will be imported and form the contents of the `graphics` namespace. For an import such as `import graphics.formats.jpg`, the files *graphics/__init__.py* and *graphics/formats/__init__.py* will both be imported prior to the final import of the *graphics/formats/jpg.py* file.

More often that not, it's fine to just leave the *__init__.py* files empty. However, there are certain situations where they might include code. For example, an *__init__.py* file can be used to automatically load submodules like this:

```
# graphics/formats/__init__.py

from . import jpg
from . import png
```

For such a file, a user merely has to use a single `import graphics.formats` instead of a separate import for `graphics.formats.jpg` and `graphics.formats.png`.

Other common uses of *__init__.py* include consolidating definitions from multiple files into a single logical namespace, as is sometimes done when splitting modules. This is discussed in "Splitting a Module into Multiple Files".

Astute programmers will notice that Python 3.3 still seems to perform package imports even if no

__init__.py files are present. If you don't define *__init__.py*, you actually create what's known as a "namespace package," which is described in "Making Separate Directories of Code Import Under a Common Namespace". All things being equal, include the *__init__.py* files if you're just starting out with the creation of a new package.

Controlling the Import of Everything

Problem

You want precise control over the symbols that are exported from a module or package when a user uses the `from module import *` statement.

Solution

Define a variable `__all__` in your module that explicitly lists the exported names. For example:

```
# somemodule.py

def spam():
    pass

def grok():
    pass

blah = 42

# Only export 'spam' and 'grok'
__all__ = ['spam', 'grok']
```

Discussion

Although the use of `from module import *` is strongly discouraged, it still sees frequent use in modules that define a large number of names. If you don't do anything, this form of import will export all names that don't start with an underscore. On the other hand, if `__all__` is defined, then only the names explicitly listed will be exported.

If you define `__all__` as an empty list, then nothing will be exported. An `AttributeError` is raised on import if `__all__` contains undefined names.

Importing Package Submodules Using Relative Names

Problem

You have code organized as a package and want to import a submodule from one of the other package submodules without hardcoding the package name into the import statement.

Solution

To import modules of a package from other modules in the same package, use a package-relative import. For example, suppose you have a package `mypackage` organized as follows on the filesystem:

```
    mypackage/
        __init__.py
        A/
            __init__.py
            spam.py
            grok.py
        B/
            __init__.py
            bar.py
```

If the module `mypackage.A.spam` wants to import the module `grok` located in the same directory, it should include an import statement like this:

```
# mypackage/A/spam.py

from . import grok
```

If the same module wants to import the module `B.bar` located in a different directory, it can use an import statement like this:

```
# mypackage/A/spam.py

from ..B import bar
```

Both of the import statements shown operate relative to the location of the *spam.py* file and do not include the top-level package name.

Discussion

Inside packages, imports involving modules in the same package can either use fully specified absolute names or a relative imports using the syntax shown. For example:

```
# mypackage/A/spam.py

from mypackage.A import grok      # OK
from . import grok                # OK
import grok                       # Error (not found)
```

The downside of using an absolute name, such as `mypackage.A`, is that it hardcodes the top-level package name into your source code. This, in turn, makes your code more brittle and hard to work with if you ever want to reorganize it. For example, if you ever changed the name of the package, you would have to go through all of your files and fix the source code. Similarly, hardcoded names make it difficult for someone else to move the code around. For example, perhaps someone wants to install two different versions of a package, differentiating them only by name. If relative imports are used, it would all work fine, whereas everything would break with absolute names.

The `.` and `..` syntax on the `import` statement might look funny, but think of it as specifying a directory name. `.` means look in the current directory and `..B` means look in the *../B* directory. This syntax only works with the `from` form of import. For example:

```
from . import grok          # OK
import .grok                # ERROR
```

Although it looks like you could navigate the filesystem using a relative import, they are not allowed to escape the directory in which a package is defined. That is, combinations of dotted name patterns that

would cause an import to occur from a non-package directory cause an error.

Finally, it should be noted that relative imports only work for modules that are located inside a proper package. In particular, they do not work inside simple modules located at the top level of scripts. They also won't work if parts of a package are executed directly as a script. For example:

```
% python3 mypackage/A/spam.py      # Relative imports fail
```

On the other hand, if you execute the preceding script using the `-m` option to Python, the relative imports will work properly. For example:

```
% python3 -m mypackage.A.spam      # Relative imports work
```

For more background on relative package imports, see PEP 328.

Splitting a Module into Multiple Files

Problem

You have a module that you would like to split into multiple files. However, you would like to do it without breaking existing code by keeping the separate files unified as a single logical module.

Solution

A program module can be split into separate files by turning it into a package. Consider the following simple module:

```
# mymodule.py

class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

Suppose you want to split *mymodule.py* into two files, one for each class definition. To do that, start by replacing the *mymodule.py* file with a directory called *mymodule*. In that directory, create the following files:

```
    mymodule/
        __init__.py
        a.py
        b.py
```

In the *a.py* file, put this code:

```
# a.py

class A:
    def spam(self):
        print('A.spam')
```

In the *b.py* file, put this code:

```
# b.py

from .a import A

class B(A):
    def bar(self):
        print('B.bar')
```

Finally, in the *__init__.py* file, glue the two files together:

```
# __init__.py

from .a import A
from .b import B
```

If you follow these steps, the resulting `mymodule` package will appear to be a single logical module:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

Discussion

The primary concern in this recipe is a design question of whether or not you want users to work with a lot of small modules or just a single module. For example, in a large code base, you could just break everything up into separate files and make users use a lot of `import` statements like this:

```
from mymodule.a import A
from mymodule.b import B
...
```

This works, but it places more of a burden on the user to know where the different parts are located. Often, it's just easier to unify things and allow a single import like this:

```
from mymodule import A, B
```

For this latter case, it's most common to think of `mymodule` as being one large source file. However, this recipe shows how to stitch multiple files together into a single logical namespace. The key to doing this is to create a package directory and to use the *__init__.py* file to glue the parts together.

When a module gets split, you'll need to pay careful attention to cross-filename references. For instance, in this recipe, class `B` needs to access class `A` as a base class. A package-relative import `from .a import A` is used to get it.

Package-relative imports are used throughout the recipe to avoid hardcoding the top-level module name into the source code. This makes it easier to rename the module or move it around elsewhere later (see "Importing Package Submodules Using Relative Names").

One extension of this recipe involves the introduction of "lazy" imports. As shown, the *__init__.py* file imports all of the required subcomponents all at once. However, for a very large module, perhaps you only want to load components as they are needed. To do that, here is a slight variation of *__init__.py*:

```
# __init__.py

def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

In this version, classes `A` and `B` have been replaced by functions that load the desired classes when they are first accessed. To a user, it won't look much different. For example:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

The main downside of lazy loading is that inheritance and type checking might break. For example, you might have to change your code slightly:

```
if isinstance(x, mymodule.A):       # Error
   ...

if isinstance(x, mymodule.a.A):    # Ok
   ...
```

For a real-world example of lazy loading, look at the source code for *multiprocessing/__init__.py* in the standard library.

Making Separate Directories of Code Import Under a Common Namespace

Problem

You have a large base of code with parts possibly maintained and distributed by different people. Each part is organized as a directory of files, like a package. However, instead of having each part installed as a separated named package, you would like all of the parts to join together under a common package prefix.

Solution

Essentially, the problem here is that you would like to define a top-level Python package that serves as a namespace for a large collection of separately maintained subpackages. This problem often arises in large application frameworks where the framework developers want to encourage users to distribute plug-ins or add-on packages.

To unify separate directories under a common namespace, you organize the code just like a normal Python package, but you omit *__init__.py* files in the directories where the components are going to join together.

To illustrate, suppose you have two different directories of Python code like this:

```
    foo-package/
        spam/
             blah.py

    bar-package/
        spam/
             grok.py
```

In these directories, the name `spam` is being used as a common namespace. Observe that there is no *__init__.py* file in either directory.

Now watch what happens if you add both `foo-package` and `bar-package` to the Python module path and try some imports:

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah
>>> import spam.grok
>>>
```

You'll observe that, by magic, the two different package directories merge together and you can import either `spam.blah` or `spam.grok`. It just works.

Discussion

The mechanism at work here is a feature known as a "namespace package." Essentially, a namespace package is a special kind of package designed for merging different directories of code together under a common namespace, as shown. For large frameworks, this can be useful, since it allows parts of a framework to be broken up into separately installed downloads. It also enables people to easily make third-party add-ons and other extensions to such frameworks.

The key to making a namespace package is to make sure there are no *__init__.py* files in the top-level directory that is to serve as the common namespace. The missing *__init__.py* file causes an interesting thing to happen on package import. Instead of causing an error, the interpreter instead starts creating a list of all directories that happen to contain a matching package name. A special namespace package module is then created and a read-only copy of the list of directories is stored in its `__path__` variable. For example:

```
>>> import spam
>>> spam.__path__
_NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

The directories on `__path__` are used when locating further package subcomponents (e.g., when importing `spam.grok` or `spam.blah`).

An important feature of namespace packages is that anyone can extend the namespace with their own code. For example, suppose you made your own directory of code like this:

```
    my-package/
         spam/
             custom.py
```

If you added your directory of code to `sys.path` along with the other packages, it would just seamlessly merge together with the other `spam` package directories:

```
>>> import spam.custom
>>> import spam.grok
>>> import spam.blah
>>>
```

As a debugging tool, the main way that you can tell if a package is serving as a namespace package is to check its `__file__` attribute. If it's missing altogether, the package is a namespace. This will also be indicated in the representation string by the word "namespace":

```
>>> spam.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>> spam
<module 'spam' (namespace)>
>>>
```

Further information about namespace packages can be found in PEP 420.

Reloading Modules

Problem

You want to reload an already loaded module because you've made changes to its source.

Solution

To reload a previously loaded module, use `imp.reload()`. For example:

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

Discussion

Reloading a module is something that is often useful during debugging and development, but which is generally never safe in production code due to the fact that it doesn't always work as you expect.

Under the covers, the `reload()` operation wipes out the contents of a module's underlying dictionary and refreshes it by re-executing the module's source code. The identity of the module object itself remains unchanged. Thus, this operation updates the module everywhere that it has been imported in a program.

However, `reload()` does not update definitions that have been imported using statements such as `from module import name`. To illustrate, consider the following code:

```
# spam.py

def bar():
```

```
    print('bar')

def grok():
    print('grok')
```

Now start an interactive session:

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

Without quitting Python, go edit the source code to *spam.py* so that the function `grok()` looks like this:

```
def grok():
    print('New grok')
```

Now go back to the interactive session, perform a reload, and try this experiment:

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok()             # Notice old output
grok
>>> spam.grok()        # Notice new output
New grok
>>>
```

In this example, you'll observe that there are two versions of the `grok()` function loaded. Generally, this is not what you want, and is just the sort of thing that eventually leads to massive headaches.

For this reason, reloading of modules is probably something to be avoided in production code. Save it for debugging or for interactive sessions where you're experimenting with the interpreter and trying things out.

Making a Directory or Zip File Runnable As a Main Script

Problem

You have a program that has grown beyond a simple script into an application involving multiple files. You'd like to have some easy way for users to run the program.

Solution

If your application program has grown into multiple files, you can put it into its own directory and add a *__main__.py* file. For example, you can create a directory like this:

```
    myapplication/
         spam.py
```

```
        bar.py
        grok.py
        __main__.py
```

If *__main__.py* is present, you can simply run the Python interpreter on the top-level directory like this:

```
bash % python3 myapplication
```

The interpreter will execute the *__main__.py* file as the main program.

This technique also works if you package all of your code up into a zip file. For example:

```
bash % ls
spam.py    bar.py   grok.py   __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

Discussion

Creating a directory or zip file and adding a *__main__.py* file is one possible way to package a larger Python application. It's a little bit different than a package in that the code isn't meant to be used as a standard library module that's installed into the Python library. Instead, it's just this bundle of code that you want to hand someone to execute.

Since directories and zip files are a little different than normal files, you may also want to add a supporting shell script to make execution easier. For example, if the code was in a file named *myapp.zip*, you could make a top-level script like this:

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

Reading Datafiles Within a Package

Problem

Your package includes a datafile that your code needs to read. You need to do this in the most portable way possible.

Solution

Suppose you have a package with files organized as follows:

```
mypackage/
    __init__.py
    somedata.dat
    spam.py
```

Now suppose the file *spam.py* wants to read the contents of the file *somedata.dat*. To do it, use the following code:

```
# spam.py

import pkgutil
```

```
data = pkgutil.get_data(__package__, 'somedata.dat')
```

The resulting variable `data` will be a byte string containing the raw contents of the file.

Discussion

To read a datafile, you might be inclined to write code that uses built-in I/O functions, such as `open()`. However, there are several problems with this approach.

First, a package has very little control over the current working directory of the interpreter. Thus, any I/O operations would have to be programmed to use absolute filenames. Since each module includes a `__file__` variable with the full path, it's not impossible to figure out the location, but it's messy.

Second, packages are often installed as *.zip* or *.egg* files, which don't preserve the files in the same way as a normal directory on the filesystem. Thus, if you tried to use `open()` on a datafile contained in an archive, it wouldn't work at all.

The `pkgutil.get_data()` function is meant to be a high-level tool for getting a datafile regardless of where or how a package has been installed. It will simply "work" and return the file contents back to you as a byte string.

The first argument to `get_data()` is a string containing the package name. You can either supply it directly or use a special variable, such as `__package__`. The second argument is the relative name of the file within the package. If necessary, you can navigate into different directories using standard Unix filename conventions as long as the final directory is still located within the package.

Adding Directories to sys.path

Problem

You have Python code that can't be imported because it's not located in a directory listed in `sys.path`. You would like to add new directories to Python's path, but don't want to hardwire it into your code.

Solution

There are two common ways to get new directories added to `sys.path`. First, you can add them through the use of the `PYTHONPATH` environment variable. For example:

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct  4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

In a custom application, this environment variable could be set at program startup or through a shell script of some kind.

The second approach is to create a *.pth* file that lists the directories like this:

```
# myapplication.pth
/some/dir
/other/dir
```

This *.pth* file needs to be placed into one of Python's *site-packages* directories, which are typically located at */usr/local/lib/python3.3/site-packages* or *~/.local/lib/python3.3/site-packages*. On interpreter startup, the directories listed in the *.pth* file will be added to *sys.path* as long as they exist on the filesystem. Installation of a *.pth* file might require administrator access if it's being added to the system-wide Python interpreter.

Discussion

Faced with trouble locating files, you might be inclined to write code that manually adjusts the value of `sys.path`. For example:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

Although this "works," it is extremely fragile in practice and should be avoided if possible. Part of the problem with this approach is that it adds hardcoded directory names to your source. This can cause maintenance problems if your code ever gets moved around to a new location. It's usually much better to configure the path elsewhere in a manner that can be adjusted without making source code edits.

You can sometimes work around the problem of hardcoded directories if you carefully construct an appropriate absolute path using module-level variables, such as `__file__`. For example:

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__'), 'src'))
```

This adds an *src* directory to the path where that directory is located in the same directory as the code that's executing the insertion step.

The *site-packages* directories are the locations where third-party modules and packages normally get installed. If your code was installed in that manner, that's where it would be placed. Although *.pth* files for configuring the path must appear in *site-packages*, they can refer to any directories on the system that you wish. Thus, you can elect to have your code in a completely different set of directories as long as those directories are included in a *.pth* file.

Importing Modules Using a Name Given in a String

Problem

You have the name of a module that you would like to import, but it's being held in a string. You would like to invoke the `import` command on the string.

Solution

Use the `importlib.import_module()` function to manually import a module or part of a package where the name is given as a string. For example:

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

`import_module` simply performs the same steps as `import`, but returns the resulting module object back to you as a result. You just need to store it in a variable and use it like a normal module afterward.

If you are working with packages, `import_module()` can also be used to perform relative imports. However, you need to give it an extra argument. For example:

```
import importlib

# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

Discussion

The problem of manually importing modules with `import_module()` most commonly arises when writing code that manipulates or wraps around modules in some way. For example, perhaps you're implementing a customized importing mechanism of some kind where you need to load a module by name and perform patches to the loaded code.

In older code, you will sometimes see the built-in `__import__()` function used to perform imports. Although this works, `importlib.import_module()` is usually easier to use.

See "Loading Modules from a Remote Machine Using Import Hooks" for an advanced example of customizing the import process.

Loading Modules from a Remote Machine Using Import Hooks

Problem

You would like to customize Python's import statement so that it can transparently load modules from a remote machine.

Solution

First, a serious disclaimer about security. The idea discussed in this recipe would be wholly bad without some kind of extra security and authentication layer. That said, the main goal is actually to take a deep dive into the inner workings of Python's `import` statement. If you get this recipe to work and understand the inner workings, you'll have a solid foundation of customizing `import` for almost any other purpose. With that out of the way, let's carry on.

At the core of this recipe is a desire to extend the functionality of the `import` statement. There are several approaches for doing this, but for the purposes of illustration, start by making the following directory of Python code:

```
testcode/
```

```
    spam.py
    fib.py
    grok/
        __init__.py
        blah.py
```

The content of these files doesn't matter, but put a few simple statements and functions in each file so you can test them and see output when they're imported. For example:

```
# spam.py
print("I'm spam")

def hello(name):
    print('Hello %s' % name)

# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# grok/__init__.py
print("I'm grok.__init__")

# grok/blah.py
print("I'm grok.blah")
```

The goal here is to allow remote access to these files as modules. Perhaps the easiest way to do this is to publish them on a web server. Simply go to the *testcode* directory and run Python like this:

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

Leave that server running and start up a separate Python interpreter. Make sure you can access the remote files using `urllib`. For example:

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

>>>
```

Loading source code from this server is going to form the basis for the remainder of this recipe. Specifically, instead of manually grabbing a file of source code using `urlopen()`, the `import` statement will be customized to do it transparently behind the scenes.

The first approach to loading a remote module is to create an explicit loading function for doing it. For example:

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

This function merely downloads the source code, compiles it into a code object using `compile()`, and executes it in the dictionary of a newly created module object. Here's how you would use the function:

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py'>
>>>
```

As you can see, it "works" for simple modules. However, it's not plugged into the usual `import` statement, and extending the code to support more advanced constructs, such as packages, would require additional work.

A much slicker approach is to create a custom importer. The first way to do this is to create what's known as a meta path importer. Here is an example:

```
# urlimport.py

import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Debugging
import logging
log = logging.getLogger(__name__)

# Get links from a given URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
```

```
            if tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))

    links = set()
    try:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
    except Exception as e:
        log.debug('Could not get links. %s', e)
    log.debug('links: %r', links)
    return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._links   = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]

        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Check link cache
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl)
                log.debug('find_module: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return loader

        # A normal module
        filename = basename + '.py'
        if filename in self._links[baseurl]:
            log.debug('find_module: module %r found', fullname)
            return self._loaders[baseurl]
        else:
            log.debug('find_module: module %r not found', fullname)
```

```
            return None

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links.clear()

# Module Loader for a URL
class UrlModuleLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

    # Required method
    def load_module(self, fullname):
        code = self.get_code(fullname)
        mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
        mod.__file__ = self.get_filename(fullname)
        mod.__loader__ = self
        mod.__package__ = fullname.rpartition('.')[0]
        exec(code, mod.__dict__)
        return mod

    # Optional extensions
    def get_code(self, fullname):
        src = self.get_source(fullname)
        return compile(src, self.get_filename(fullname), 'exec')

    def get_data(self, path):
        pass

    def get_filename(self, fullname):
        return self._baseurl + '/' + fullname.split('.')[-1] + '.py'

    def get_source(self, fullname):
        filename = self.get_filename(fullname)
        log.debug('loader: reading %r', filename)
        if filename in self._source_cache:
            log.debug('loader: cached %r', filename)
            return self._source_cache[filename]
        try:
            u = urlopen(filename)
            source = u.read().decode('utf-8')
            log.debug('loader: %r loaded', filename)
            self._source_cache[filename] = source
            return source
        except (HTTPError, URLError) as e:
            log.debug('loader: %r failed.  %s', filename, e)
            raise ImportError("Can't load %s" % filename)


    def is_package(self, fullname):
        return False

# Package loader for a URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
```

```
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Utility functions for installing/uninstalling the loader
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)
```

Here is an interactive session showing how to use the preceding code:

```
>>> # importing currently fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Load the importer and retry (it works)
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

This particular solution involves installing an instance of a special finder object `UrlMetaFinder` as the last entry in `sys.meta_path`. Whenever modules are imported, the finders in `sys.meta_path` are consulted in order to locate the module. In this example, the `UrlMetaFinder` instance becomes a finder of last resort that's triggered when a module can't be found in any of the normal locations.

As for the general implementation approach, the `UrlMetaFinder` class wraps around a user-specified URL. Internally, the finder builds sets of valid links by scraping them from the given URL. When imports are made, the module name is compared against this set of known links. If a match can be found, a separate `UrlModuleLoader` class is used to load source code from the remote machine and create the resulting module object. One reason for caching the links is to avoid unnecessary HTTP requests on repeated imports.

The second approach to customizing import is to write a hook that plugs directly into the `sys.path`

variable, recognizing certain directory naming patterns. Add the following class and support functions to *urlimport.py*:

```
# urlimport.py

# ... include previous code above ...


# Path finder class for a URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')
        basename = parts[-1]
        # Check link cache
        if self._links is None:
            self._links = []     # See discussion
            self._links = _get_links(self._baseurl)

        # Check if it's a package
        if basename in self._links:
            log.debug('find_loader: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                log.debug('find_loader: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_loader: %r is a namespace package', fullname)
                loader = None
            return (loader, [fullurl])

        # A normal module
        filename = basename + '.py'
        if filename in self._links:
            log.debug('find_loader: module %r found', fullname)
            return (self._loader, [])
        else:
            log.debug('find_loader: module %r not found', fullname)
            return (None, [])

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links = None

# Check path to see if it looks like a URL
_url_path_cache = {}
def handle_url(path):
    if path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if path in _url_path_cache:
            finder = _url_path_cache[path]
        else:
            finder = UrlPathFinder(path)
```

```
            _url_path_cache[path] = finder
        return finder
    else:
        log.debug('Handle path? %s. [No]', path)

def install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def remove_path_hook():
    sys.path_hooks.remove(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')
```

To use this path-based finder, you simply add URLs to `sys.path`. For example:

```
>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

The key to this last example is the `handle_url()` function, which is added to the `sys.path_hooks` variable. When the entries on `sys.path` are being processed, the functions in `sys.path_hooks` are invoked. If any of those functions return a finder object, that finder is used to try to load modules for that entry on `sys.path`.

It should be noted that the remotely imported modules work exactly like any other module. For instance:

```
>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
```

```
>>> print(inspect.getsource(fib))
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

>>>
```

Discussion

Before discussing this recipe in further detail, it should be emphasized that Python's module, package, and import mechanism is one of the most complicated parts of the entire language—often poorly understood by even the most seasoned Python programmers unless they've devoted effort to peeling back the covers. There are several critical documents that are worth reading, including the documentation for the `importlib` module and PEP 302. That documentation won't be repeated here, but some essential highlights will be discussed.

First, if you want to create a new module object, you use the `imp.new_module()` function. For example:

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

Module objects usually have a few expected attributes, including `__file__` (the name of the file that the module was loaded from) and `__package__` (the name of the enclosing package, if any).

Second, modules are cached by the interpreter. The module cache can be found in the dictionary `sys.modules`. Because of this caching, it's common to combine caching and module creation together into a single step. For example:

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

The main reason for doing this is that if a module with the given name already exists, you'll get the already created module instead. For example:

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
```

```
>>>
```

Since creating modules is easy, it is straightforward to write simple functions, such as the `load_module()` function in the first part of this recipe. A downside of this approach is that it is actually rather tricky to handle more complicated cases, such as package imports. In order to handle a package, you would have to reimplement much of the underlying logic that's already part of the normal `import` statement (e.g., checking for directories, looking for *__init__.py* files, executing those files, setting up paths, etc.). This complexity is one of the reasons why it's often better to extend the `import` statement directly rather than defining a custom function.

Extending the `import` statement is straightforward, but involves a number of moving parts. At the highest level, `import` operations are processed by a list of "meta-path" finders that you can find in the list `sys.meta_path`. If you output its value, you'll see the following:

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

When executing a statement such as `import fib`, the interpreter walks through the finder objects on `sys.meta_path` and invokes their `find_module()` method in order to locate an appropriate module loader. It helps to see this by experimentation, so define the following class and try the following:

```
>>> class Finder:
...     def find_module(self, fullname, path):
...             print('Looking for', fullname, path)
...             return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder())   # Insert as first entry
>>> import math
Looking for math None
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

Notice how the `find_module()` method is being triggered on every import. The role of the `path` argument in this method is to handle packages. When packages are imported, it is a list of the directories that are found in the package's `__path__` attribute. These are the paths that need to be checked to find package subcomponents. For example, notice the path setting for `xml.etree` and `xml.etree.ElementTree`:

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
```

```
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>
```

The placement of the finder on `sys.meta_path` is critical. Remove it from the front of the list to the end of the list and try more imports:

```
>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime
```

Now you don't see any output because the imports are being handled by other entries in `sys.meta_path`. In this case, you would only see it trigger when nonexistent modules are imported:

```
>>> import fib
Looking for fib None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>
```

The fact that you can install a finder to catch unknown modules is the key to the `UrlMetaFinder` class in this recipe. An instance of `UrlMetaFinder` is added to the end of `sys.meta_path`, where it serves as a kind of importer of last resort. If the requested module name can't be located by any of the other import mechanisms, it gets handled by this finder. Some care needs to be taken when handling packages. Specifically, the value presented in the `path` argument needs to be checked to see if it starts with the URL registered in the finder. If not, the submodule must belong to some other finder and should be ignored.

Additional handling of packages is found in the `UrlPackageLoader` class. This class, rather than importing the package name, tries to load the underlying *__init__.py* file. It also sets the module `__path__` attribute. This last part is critical, as the value set will be passed to subsequent `find_module()` calls when loading package submodules.

The path-based import hook is an extension of these ideas, but based on a somewhat different mechanism. As you know, `sys.path` is a list of directories where Python looks for modules. For example:

```
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python33.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/usr/local/lib/...3.3/site-packages']
>>>
```

Each entry in `sys.path` is additionally attached to a finder object. You can view these finders by looking

at sys.path_importer_cache:

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
 '/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
 '/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
 '/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collections'),
 '/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodings'),
 '/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dynload'),
 '/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-darwin'),
 '/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site-packages'),
 '/usr/local/lib/python33.zip': None}
>>>
```

sys.path_importer_cache tends to be much larger than sys.path because it records finders for all known directories where code is being loaded. This includes subdirectories of packages which usually aren't included on sys.path.

To execute import fib, the directories on sys.path are checked in order. For each directory, the name fib is presented to the associated finder found in sys.path_importer_cache. This is also something that you can investigate by making your own finder and putting an entry in the cache. Try this experiment:

```
>>> class Finder:
...     def find_loader(self, name):
...             print('Looking for', name)
...             return (None, [])
...
>>> import sys
>>> # Add a "debug" entry to the importer cache
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>
```

Here, you've installed a new cache entry for the name debug and installed the name debug as the first entry on sys.path. On all subsequent imports, you see your finder being triggered. However, since it returns (None, []), processing simply continues to the next entry.

The population of sys.path_importer_cache is controlled by a list of functions stored in sys.path_hooks. Try this experiment, which clears the cache and adds a new path checking function to sys.path_hooks:

```
>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> import fib
Checked debug
Checking .
```

```
Checking /usr/local/lib/python33.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

As you can see, the `check_path()` function is being invoked for every entry on `sys.path`. However, since an `ImportError` exception is raised, nothing else happens (checking just moves to the next function on `sys.path_hooks`).

Using this knowledge of how `sys.path` is processed, you can install a custom path checking function that looks for filename patterns, such as URLs. For instance:

```
>>> def check_url(path):
...     if path.startswith('http://'):
...             return Finder()
...     else:
...             raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib             # Finder output!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

This is the key mechanism at work in the last part of this recipe. Essentially, a custom path checking function has been installed that looks for URLs in `sys.path`. When they are encountered, a new `UrlPathFinder` instance is created and installed into `sys.path_importer_cache`. From that point forward, all import statements that pass through that part of `sys.path` will try to use your custom finder.

Package handling with a path-based importer is somewhat tricky, and relates to the return value of the `find_loader()` method. For simple modules, `find_loader()` returns a tuple `(loader, None)` where `loader` is an instance of a loader that will import the module.

For a normal package, `find_loader()` returns a tuple `(loader, path)` where `loader` is the loader instance that will import the package (and execute *__init__.py*) and path is a list of the directories that will make up the initial setting of the package's `__path__` attribute. For example, if the base URL was `http://localhost:15000` and a user executed `import grok`, the path returned by `find_loader()` would be `[ 'http://localhost:15000/grok' ]`.

The `find_loader()` must additionally account for the possibility of a namespace package. A namespace package is a package where a valid package directory name exists, but no *__init__.py* file can be found. For this case, `find_loader()` must return a tuple `(None, path)` where `path` is a list of directories that

would have made up the package's `__path__` attribute had it defined an *__init__.py* file. For this case, the import mechanism moves on to check further directories on `sys.path`. If more namespace packages are found, all of the resulting paths are joined together to make a final namespace package. See "Making Separate Directories of Code Import Under a Common Namespace" for more information on namespace packages.

There is a recursive element to package handling that is not immediately obvious in the solution, but also at work. All packages contain an internal path setting, which can be found in `__path__` attribute. For example:

```
>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

As mentioned, the setting of `__path__` is controlled by the return value of the `find_loader()` method. However, the subsequent processing of `__path__` is also handled by the functions in `sys.path_hooks`. Thus, when package subcomponents are loaded, the entries in `__path__` are checked by the `handle_url()` function. This causes new instances of `UrlPathFinder` to be created and added to `sys.path_importer_cache`.

One remaining tricky part of the implementation concerns the behavior of the `handle_url()` function and its interaction with the `_get_links()` function used internally. If your implementation of a finder involves the use of other modules (e.g., `urllib.request`), there is a possibility that those modules will attempt to make further imports in the middle of the finder's operation. This can actually cause `handle_url()` and other parts of the finder to get executed in a kind of recursive loop. To account for this possibility, the implementation maintains a cache of created finders (one per URL). This avoids the problem of creating duplicate finders. In addition, the following fragment of code ensures that the finder doesn't respond to any import requests while it's in the processs of getting the initial set of links:

```
        # Check link cache
        if self._links is None:
            self._links = []     # See discussion
            self._links = _get_links(self._baseurl)
```

You may not need this checking in other implementations, but for this example involving URLs, it was required.

Finally, the `invalidate_caches()` method of both finders is a utility method that is supposed to clear internal caches should the source code change. This method is triggered when a user invokes `importlib.invalidate_caches()`. You might use it if you want the URL importers to reread the list of links, possibly for the purpose of being able to access newly added files.

In comparing the two approaches (modifying `sys.meta_path` or using a path hook), it helps to take a high-level view. Importers installed using `sys.meta_path` are free to handle modules in any manner that they wish. For instance, they could load modules out of a database or import them in a manner that is radically different than normal module/package handling. This freedom also means that such importers need to do more bookkeeping and internal management. This explains, for instance, why the implementation of `UrlMetaFinder` needs to do its own caching of links, loaders, and other details. On the other hand, path-based hooks are more narrowly tied to the processing of `sys.path`. Because of the connection to `sys.path`, modules loaded with such extensions will tend to have the same features as

normal modules and packages that programmers are used to.

Assuming that your head hasn't completely exploded at this point, a key to understanding and experimenting with this recipe may be the added logging calls. You can enable logging and try experiments such as this:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

Last, but not least, spending some time sleeping with PEP 302 and the documentation for `importlib` under your pillow may be advisable.

Patching Modules on Import

Problem

You want to patch or apply decorators to functions in an existing module. However, you only want to do it if the module actually gets imported and used elsewhere.

Solution

The essential problem here is that you would like to carry out actions in response to a module being loaded. Perhaps you want to trigger some kind of callback function that would notify you when a module was loaded.

This problem can be solved using the same import hook machinery discussed in "Loading Modules from a Remote Machine Using Import Hooks". Here is a possible solution:

```
# postimport.py

import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)
```

```
class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module

def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())
```

To use this code, you use the `when_imported()` decorator. For example:

```
>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads?  Are you crazy?')
...
>>>
>>> import threading
Threads?  Are you crazy?
>>>
```

As a more practical example, maybe you want to apply decorators to existing definitions, such as shown here:

```
from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Example
```

```
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

Discussion

This recipe relies on the import hooks that were discussed in "Loading Modules from a Remote Machine Using Import Hooks", with a slight twist.

First, the role of the `@when_imported` decorator is to register handler functions that get triggered on import. The decorator checks `sys.modules` to see if a module was already loaded. If so, the handler is invoked immediately. Otherwise, the handler is added to a list in the `_post_import_hooks` dictionary. The purpose of `_post_import_hooks` is simply to collect all handler objects that have been registered for each module. In principle, more than one handler could be registered for a given module.

To trigger the pending actions in `_post_import_hooks` after module import, the `PostImportFinder` class is installed as the first item in `sys.meta_path`. If you recall from "Loading Modules from a Remote Machine Using Import Hooks", `sys.meta_path` contains a list of finder objects that are consulted in order to locate modules. By installing `PostImportFinder` as the first item, it captures all module imports.

In this recipe, however, the role of `PostImportFinder` is not to load modules, but to trigger actions upon the completion of an import. To do this, the actual import is delegated to the other finders on `sys.meta_path`. Rather than trying to do this directly, the function `imp.import_module()` is called recursively in the `PostImportLoader` class. To avoid getting stuck in an infinite loop, `PostImportFinder` keeps a set of all the modules that are currently in the process of being loaded. If a module name is part of this set, it is simply ignored by `PostImportFinder`. This is what causes the import request to pass to the other finders on `sys.meta_path`.

After a module has been loaded with `imp.import_module()`, all handlers currently registered in `_post_import_hooks` are called with the newly loaded module as an argument. From this point forward, the handlers are free to do what they want with the module.

A major feature of the approach shown in this recipe is that the patching of a module occurs in a seamless fashion, regardless of where or how a module of interest is actually loaded. You simply write a handler function that's decorated with `@when_imported()` and it all just magically works from that point forward.

One caution about this recipe is that it does not work for modules that have been explicitly reloaded using `imp.reload()`. That is, if you reload a previously loaded module, the post import handler function doesn't get triggered again (all the more reason to not use `reload()` in production code). On the other hand, if you delete the module from `sys.modules` and redo the import, you'll see the handler trigger again.

More information about post-import hooks can be found in PEP 369 . As of this writing, the PEP has been withdrawn by the author due to it being out of date with the current implementation of the `importlib` module. However, it is easy enough to implement your own solution using this recipe.

Installing Packages Just for Yourself

Problem

You want to install a third-party package, but you don't have permission to install packages into the

system Python. Alternatively, perhaps you just want to install a package for your own use, not all users on the system.

Solution

Python has a per-user installation directory that's typically located in a directory such as *~/.local/lib/python3.3/site-packages*. To force packages to install in this directory, give the `--user` option to the installation command. For example:

```
python3 setup.py install --user
```

or

```
pip install --user packagename
```

The user *site-packages* directory normally appears before the system *site-packages* directory on `sys.path`. Thus, packages you install using this technique take priority over the packages already installed on the system (although this is not always the case depending on the behavior of third-party package managers, such as `distribute` or `pip`).

Discussion

Normally, packages get installed into the system-wide *site-packages* directory, which is found in a location such as */usr/local/lib/python3.3/site-packages*. However, doing so typically requires administrator permissions and use of the `sudo` command. Even if you have permission to execute such a command, using `sudo` to install a new, possibly unproven, package might give you some pause.

Installing packages into the per-user directory is often an effective workaround that allows you to create a custom installation.

As an alternative, you can also create a virtual environment, which is discussed in the next recipe.

Creating a New Python Environment

Problem

You want to create a new Python environment in which you can install modules and packages. However, you want to do this without installing a new copy of Python or making changes that might affect the system Python installation.

Solution

You can make a new "virtual" environment using the `pyvenv` command. This command is installed in the same directory as the Python interpreter or possibly in the *Scripts* directory on Windows. Here is an example:

```
bash % pyvenv Spam
bash %
```

The name supplied to `pyvenv` is the name of a directory that will be created. Upon creation, the *Spam*

directory will look something like this:

```
bash % cd Spam
bash % ls
bin               include             lib           pyvenv.cfg
bash %
```

In the *bin* directory, you'll find a Python interpreter that you can use. For example:

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct  6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python33.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/Users/beazley/Spam/lib/python3.3/site-packages']
>>>
```

A key feature of this interpreter is that its *site-packages* directory has been set to the newly created environment. Should you decide to install third-party packages, they will be installed here, not in the normal system *site-packages* directory.

Discussion

The creation of a virtual environment mostly pertains to the installation and management of third-party packages. As you can see in the example, the `sys.path` variable contains directories from the normal system Python, but the *site-packages* directory has been relocated to a new directory.

With a new virtual environment, the next step is often to install a package manager, such as `distribute` or `pip`. When installing such tools and subsequent packages, you just need to make sure you use the interpreter that's part of the virtual environment. This should install the packages into the newly created *site-packages* directory.

Although a virtual environment might look like a copy of the Python installation, it really only consists of a few files and symbolic links. All of the standard library files and interpreter executables come from the original Python installation. Thus, creating such environments is easy, and takes almost no machine resources.

By default, virtual environments are completely clean and contain no third-party add-ons. If you would like to include already installed packages as part of a virtual environment, create the environment using the `--system-site-packages` option. For example:

```
bash % pyvenv --system-site-packages Spam
bash %
```

More information about `pyvenv` and virtual environments can be found in PEP 405.

Distributing Packages

Problem

You've written a useful library, and you want to be able to give it away to others.

Solution

If you're going to start giving code away, the first thing to do is to give it a unique name and clean up its directory structure. For example, a typical library package might look something like this:

```
    projectname/
         README.txt
         Doc/
             documentation.txt
         projectname/
            __init__.py
            foo.py
            bar.py
            utils/
                 __init__.py
                 spam.py
                 grok.py
         examples/
            helloworld.py
            ...
```

To make the package something that you can distribute, first write a *setup.py* file that looks like this:

```
# setup.py
from distutils.core import setup

setup(name='projectname',
      version='1.0',
      author='Your Name',
      author_email='you@youraddress.com',
      url='http://www.you.com/projectname',
      packages=['projectname', 'projectname.utils'],
)
```

Next, make a file *MANIFEST.in* that lists various nonsource files that you want to include in your package:

```
    # MANIFEST.in
    include *.txt
    recursive-include examples *
    recursive-include Doc *
```

Make sure the *setup.py* and *MANIFEST.in* files appear in the top-level directory of your package. Once you have done this, you should be able to make a source distribution by typing a command such as this:

```
% bash python3 setup.py sdist
```

This will create a file such as *projectname-1.0.zip* or *projectname-1.0.tar.gz*, depending on the platform. If it all works, this file is suitable for giving to others or uploading to the Python Package Index.

Discussion

For pure Python code, writing a plain *setup.py* file is usually straightforward. One potential gotcha is that you have to manually list every subdirectory that makes up the packages source code. A common mistake is to only list the top-level directory of a package and to forget to include package subcomponents. This is why the specification for `packages` in *setup.py* includes the list `packages=['projectname', 'projectname.utils']`.

As most Python programmers know, there are many third-party packaging options, including setuptools, distribute, and so forth. Some of these are replacements for the `distutils` library found in the standard library. Be aware that if you rely on these packages, users may not be able to install your software unless they also install the required package manager first. Because of this, you can almost never go wrong by keeping things as simple as possible. At a bare minimum, make sure your code can be installed using a standard Python 3 installation. Additional features can be supported as an option if additional packages are available.

Packaging and distribution of code involving C extensions can get considerably more complicated. Chapter 15 on C extensions has a few details on this. In particular, see "Writing a Simple C Extension Module".

Prev
Chapter 9. Metaprogramming
Home
Next
Chapter 11. Network and Web Programming