pkgutil — Package Utilities

Purpose: Add to the module search path for a specific package and work with resources included in a package.

The pkgutil module includes functions for changing the import rules for Python packages and for loading non-code resources from files distributed within a package.

Package Import Paths

The extend_path() function is used to modify the search path and change the way sub-modules are imported from within a package so that several different directories can be combined as though they are one. This can be used to override installed versions of packages with development versions, or to combine platform-specific and shared modules into a single package namespace.

The most common way to call extend_path() is by adding two lines to the __init__.py inside the package.

```
import pkgutil
__path__ = pkgutil.extend_path(__path__, __name__)
```

extend_path() scans sys.path for directories that include a subdirectory named for the package given as the second argument. The list of directories is combined with the path value passed as the first argument and returned as a single list, suitable for use as the package import path.

An example package called demopkg includes two files, __init__.py and shared.py. The __init__.py file in demopkg1 contains print statements to show the search path before and after it is modified, to highlight the difference.

```
# demopkg1/__init__.py

import pkgutil
import pprint

print('demopkg1.__path__ before:')
pprint.pprint(__path__)
print()

__path__ = pkgutil.extend_path(__path__, __name__)

print('demopkg1.__path__ after:')
pprint.pprint(__path__)
print()
```

The extension directory, with add-on features for demopkg, contains three more source files. There is an __init__.py at each directory level, and a not_shared.py.

```
$ find extension -name '*.py'

extension/__init__.py
extension/demopkg1/__init__.py
extension/demopkg1/not_shared.py
```

This simple test program imports the demopkg1 package.

```
# pkgutil_extend_path.py

import demopkg1
print('demopkg1           :', demopkg1.__file__)

try:
    import demopkg1.shared
except Exception as err:
    print('demopkg1.shared    : Not found ({})'.format(err))
else:
    print('demopkg1.shared     :', demopkg1.shared.__file__)

try:
    import demopkg1.not_shared
except Exception as err:
    print('demopkg1.not_shared: Not found ({})'.format(err))
else:
    print('demopkg1.not_shared:', demopkg1.not_shared.__file__)
```

When this test program is run directly from the command line, the `not_shared` module is not found.

Note

The full file system paths in these examples have been shortened to emphasize the parts that change.

```
$ python3 pkgutil_extend_path.py

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1']

demopkg1           : .../demopkg1/__init__.py
demopkg1.shared    : .../demopkg1/shared.py
demopkg1.not_shared: Not found (No module named 'demopkg1.not_sh
ared')
```

However, if the `extension` directory is added to the `PYTHONPATH` and the program is run again, different results are produced.

```
$ PYTHONPATH=extension python3 pkgutil_extend_path.py

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1',
 '.../extension/demopkg1']

demopkg1           : .../demopkg1/__init__.py
demopkg1.shared    : .../demopkg1/shared.py
demopkg1.not_shared: .../extension/demopkg1/not_shared.py
```

The version of demopkg1 inside the `extension` directory has been added to the search path, so the `not_shared` module is found there.

Extending the path in this manner is useful for combining platform-specific versions of packages with common packages, especially if the platform-specific versions include C extension modules.

Development Versions of Packages

While developing enhancements to a project, it is common to need to test changes to an installed package. Replacing the installed copy with a development version may be a bad idea, since it is not necessarily correct and other tools on the system are likely to depend on the installed package.

A completely separate copy of the package could be configured in a development environment using `virtualenv` or venv, but for small modifications the overhead of setting up a virtual environment with all of the dependencies may be excessive.

Another option is to use `pkgutil` to modify the module search path for modules that belong to the package under development. In this case, however, the path must be reversed so development version overrides the installed version.

Given a package demopkg2 containing an `__init__.py` and `overloaded.py`, with the function under development located in `demopkg2/overloaded.py`. The installed version contains

```
# demopkg2/overloaded.py


def func():
    print('This is the installed version of func().')
    print('This is the installed version of func().')
```

and demopkg2/__init__.py contains

```
# demopkg2/__init__.py

import pkgutil

__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()
```

`reverse()` is used to ensure that any directories added to the search path by `pkgutil` are scanned for imports *before* the default location.

This program imports demopkg2.overloaded and calls `func()`.

```
# pkgutil_devel.py

import demopkg2
print('demopkg2           :', demopkg2.__file__)

import demopkg2.overloaded
print('demopkg2.overloaded:', demopkg2.overloaded.__file__)

print()
demopkg2.overloaded.func()
```

Running it without any special path treatment produces output from the installed version of `func()`.

```
$ python3 pkgutil_devel.py

demopkg2           : .../demopkg2/__init__.py
demopkg2.overloaded: .../demopkg2/overloaded.py

This is the installed version of func().
```

A development directory containing

```
$ find develop/demopkg2 -name '*.py'

develop/demopkg2/__init__.py
develop/demopkg2/overloaded.py
```

and a modified version of `overloaded`

```
# develop/demopkg2/overloaded.py


def func():
    print('This is the development version of func().')
```

will be loaded when the test program is run with the `develop` directory in the search path.

```
$ PYTHONPATH=develop python3 pkgutil_devel.py

demopkg2           : .../demopkg2/__init__.py
demopkg2.overloaded: .../develop/demopkg2/overloaded.py

This is the development version of func().
```

Managing Paths with PKG Files

The first example illustrated how to extend the search path using extra directories included in the `PYTHONPATH`. It is also possible to add to the search path using `*.pkg` files containing directory names. PKG files are similar to the PTH files used by the site module. They can contain directory names, one per line, to be added to the search path for the package.

Another way to structure the platform-specific portions of the application from the first example is to use a separate directory for each operating system, and include a .pkg file to extend the search path.

This example uses the same demopkg1 files, and also includes the following files.

```
$ find os_* -type f

os_one/demopkg1/__init__.py
os_one/demopkg1/not_shared.py
os_one/demopkg1.pkg
os_two/demopkg1/__init__.py
os_two/demopkg1/not_shared.py
os_two/demopkg1.pkg
```

The PKG files are named demopkg1.pkg to match the package being extended. They both contain one line.

```
demopkg
```

This demo program shows the version of the module being imported.

```
# pkgutil_os_specific.py

import demopkg1
print('demopkg1:', demopkg1.__file__)

import demopkg1.shared
print('demopkg1.shared:', demopkg1.shared.__file__)

import demopkg1.not_shared
print('demopkg1.not_shared:', demopkg1.not_shared.__file__)
```

A simple wrapper script can be used to switch between the two packages.

```
# with_os.sh

#!/bin/sh

export PYTHONPATH=os_${1}
echo "PYTHONPATH=$PYTHONPATH"
echo

python3 pkgutil_os_specific.py
```

When run with "one" or "two" as the arguments, the path is adjusted.

```
$ ./with_os.sh one

PYTHONPATH=os_one

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1',
 '.../os_one/demopkg1',
 'demopkg']

demopkg1: .../demopkg1/__init__.py
demopkg1.shared: .../demopkg1/shared.py
demopkg1.not_shared: .../os_one/demopkg1/not_shared.py

$ ./with_os.sh two

PYTHONPATH=os_two

demopkg1.__path__ before:
['.../demopkg1']

demopkg1.__path__ after:
['.../demopkg1',
 '.../os_two/demopkg1',
 'demopkg']

demopkg1: .../demopkg1/__init__.py
demopkg1.shared: .../demopkg1/shared.py
demopkg1.not_shared: .../os_two/demopkg1/not_shared.py
```

PKG files can appear anywhere in the normal search path, so a single PKG file in the current working directory could also be used to include a development tree.

Nested Packages

For nested packages, it is only necessary to modify the path of the top-level package. For example, with this directory structure

```
$ find nested -name '*.py'

nested/__init__.py
nested/second/__init__.py
nested/second/deep.py
nested/shallow.py
```

Where nested/__init__.py contains

```
# nested/__init__.py

import pkgutil

__path__ = pkgutil.extend_path(__path__, __name__)
__path__.reverse()
```

and a development tree like

```
$ find develop/nested -name '*.py'

develop/nested/__init__.py
develop/nested/second/__init__.py
develop/nested/second/deep.py
develop/nested/shallow.py
```

Both the shallow and deep modules contain a simple function to print out a message indicating whether or not they come from the installed or development version.

This test program exercises the new packages.

```
# pkgutil_nested.py

import nested

import nested.shallow
print('nested.shallow:', nested.shallow.__file__)
nested.shallow.func()

print()
import nested.second.deep
print('nested.second.deep:', nested.second.deep.__file__)
nested.second.deep.func()
```

When pkgutil_nested.py is run without any path manipulation, the installed version of both modules are used.

```
$ python3 pkgutil_nested.py

nested.shallow: .../nested/shallow.py
This func() comes from the installed version of nested.shallow

nested.second.deep: .../nested/second/deep.py
This func() comes from the installed version of nested.second.de
ep
```

When the develop directory is added to the path, the development version of both functions override the installed versions.

```
$ PYTHONPATH=develop python3 pkgutil_nested.py

nested.shallow: .../develop/nested/shallow.py
This func() comes from the development version of nested.shallow

nested.second.deep: .../develop/nested/second/deep.py
This func() comes from the development version of nested.second.
deep
```

Package Data

In addition to code, Python packages can contain data files such as templates, default configuration files, images, and other supporting files used by the code in the package. The `get_data()` function gives access to the data in the files in a format-agnostic way, so it does not matter if the package is distributed as an EGG, part of a frozen binary, or regular files on the file system.

With a package `pkgwithdata` containing a `templates` directory

```
$ find pkgwithdata -type f

pkgwithdata/__init__.py
pkgwithdata/templates/base.html
```

The file `pkgwithdata/templates/base.html` contains a simple HTML template.

```
# pkgwithdata/templates/base.html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>

</body>
</html>
```

This program uses `get_data()` to retrieve the template contents and print them out.

```
# pkgutil_get_data.py

import pkgutil

template = pkgutil.get_data('pkgwithdata', 'templates/base.html')
print(template.decode('utf-8'))
```

The arguments to `get_data()` are the dotted name of the package, and a filename relative to the top of the package. The return value is a byte sequence, so it is decoded from UTF-8 before being printed.

```
$ python3 pkgutil_get_data.py

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>

</body>
```

```
</html>
```

get_data() is distribution format-agnostic because it uses the import hooks defined in PEP 302 to access the package contents. Any loader that provides the hooks can be used, including the ZIP archive importer in zipfile.

```
# pkgutil_get_data_zip.py

import pkgutil
import zipfile
import sys

# Create a ZIP file with code from the current directory
# and the template using a name that does not appear on the
# local filesystem.
with zipfile.PyZipFile('pkgwithdatainzip.zip', mode='w') as zf:
    zf.writepy('.')
    zf.write('pkgwithdata/templates/base.html',
             'pkgwithdata/templates/fromzip.html',
             )

# Add the ZIP file to the import path.
sys.path.insert(0, 'pkgwithdatainzip.zip')

# Import pkgwithdata to show that it comes from the ZIP archive.
import pkgwithdata
print('Loading pkgwithdata from', pkgwithdata.__file__)

# Print the template body
print('\nTemplate:')
data = pkgutil.get_data('pkgwithdata', 'templates/fromzip.html')
print(data.decode('utf-8'))
```

This example uses PyZipFile.writepy() to create a ZIP archive containing a copy of the pkgwithdata package, including a renamed version of the template file. It then adds the ZIP archive to the import path before using pkgutil to load the template and print it. Refer to the discussion of zipfile for more details about using writepy().

```
$ python3 pkgutil_get_data_zip.py

Loading pkgwithdata from
pkgwithdatainzip.zip/pkgwithdata/__init__.pyc

Template:
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>PyMOTW Template</title>
</head>

<body>
<h1>Example Template</h1>

<p>This is a sample data file.</p>

</body>
</html>
```

See also

- Standard library documentation for pkgutil
- virtualenv – Ian Bicking's virtual environment script.
- distutils – Packaging tools from Python standard library.
- setuptools – Next-generation packaging tools.
- **PEP 302** – Import Hooks
- zipfile – Create importable ZIP archives.
- zipimport – Importer for packages in ZIP archives.

importlib — Python's Import Mechanism   zipimport — Load Python Code from ZIP Archives

This page was last updated 2016-12-31.

Navigation

importlib — Python’s Import Mechanism
zipimport — Load Python Code from ZIP Archives