# Modules and Imports

Most Python programs end up as a combination of several modules with a main application importing them. Whether using the features of the standard library or organizing custom code in separate files to make it easier to maintain, understanding and managing the dependencies for a program is an important aspect of development. sys includes information about the modules available to an application, either as built-ins or after being imported. It also defines hooks for overriding the standard import behavior for special cases.

## Imported Modules

`sys.modules` is a dictionary mapping the names of imported modules to the module object holding the code.

```
# sys_modules.py

import sys
import textwrap

names = sorted(sys.modules.keys())
name_text = ', '.join(names)

print(textwrap.fill(name_text, width=64))
```

The contents of `sys.modules` change as new modules are imported.

```
$ python3 sys_modules.py

__main__, _abc, _bootlocale, _codecs, _collections,
_collections_abc, _frozen_importlib, _frozen_importlib_external,
_functools, _heapq, _imp, _io, _locale, _operator, _signal,
_sre, _stat, _thread, _warnings, _weakref, abc, builtins,
codecs, collections, contextlib, copyreg, encodings,
encodings.aliases, encodings.latin_1, encodings.utf_8, enum,
functools, genericpath, heapq, importlib, importlib._bootstrap,
importlib._bootstrap_external, importlib.abc,
importlib.machinery, importlib.util, io, itertools, keyword,
marshal, operator, os, os.path, posix, posixpath, re, reprlib,
site, sphinxcontrib, sre_compile, sre_constants, sre_parse,
stat, sys, textwrap, types, warnings, zipimport
```

## Built-in Modules

The Python interpreter can be compiled with some C modules built right in, so they do not need to be distributed as separate shared libraries. These modules do not appear in the list of imported modules managed in `sys.modules` because they were not technically imported. The only way to find the available built-in modules is through `sys.builtin_module_names`.

```
# sys_builtins.py

import sys
import textwrap

name_text = ', '.join(sorted(sys.builtin_module_names))

print(textwrap.fill(name_text, width=64))
```

The output of this script will vary, especially if run with a custom-built version of the interpreter. This output was created using a copy of the interpreter installed from the standard python.org installer for OS X.

```
$ python3 sys_builtins.py

_abc, _ast, _codecs, _collections, _functools, _imp, _io,
```

```
_abc, _ast, _codecs, _collections, _functools, _imp, _io,
_locale, _operator, _signal, _sre, _stat, _string, _symtable,
_thread, _tracemalloc, _warnings, _weakref, atexit, builtins,
errno, faulthandler, gc, itertools, marshal, posix, pwd, sys,
time, xxsubtype, zipimport
```

**See also**

- Build Instructions – Instructions for building Python, from the README distributed with the source.

## Import Path

The search path for modules is managed as a Python list saved in `sys.path`. The default contents of the path include the directory of the script used to start the application and the current working directory.

```
# sys_path_show.py

import sys

for d in sys.path:
    print(d)
```

The first directory in the search path is the home for the sample script itself. That is followed by a series of platform-specific paths where compiled extension modules (written in C) might be installed, and then the global `site-packages` directory is listed last.

```
$ python3 sys_path_show.py

/Users/dhellmann/Documents/PyMOTW/pymotw-3/source/sys
.../python35.zip
.../lib/python3.5
.../lib/python3.5/plat-darwin
.../python3.5/lib-dynload
.../lib/python3.5/site-packages
```

The import search-path list can be modified before starting the interpreter by setting the shell variable `PYTHONPATH` to a colon-separated list of directories.

```
$ PYTHONPATH=/my/private/site-packages:/my/shared/site-packages \
> python3 sys_path_show.py

/Users/dhellmann/Documents/PyMOTW/pymotw-3/source/sys
/my/private/site-packages
/my/shared/site-packages
.../python35.zip
.../lib/python3.5
.../lib/python3.5/plat-darwin
.../python3.5/lib-dynload
.../lib/python3.5/site-packages
```

A program can also modify its path by adding elements to `sys.path` directly.

```
# sys_path_modify.py

import importlib
import os
import sys

base_dir = os.path.dirname(__file__) or '.'
print('Base directory:', base_dir)

# Insert the package_dir_a directory at the front of the path.
package_dir_a = os.path.join(base_dir, 'package_dir_a')
sys.path.insert(0, package_dir_a)

# Import the example module
import example
```

```
print('Imported example from:', example.__file__)
print('  ', example.DATA)

# Make package_dir_b the first directory in the search path
package_dir_b = os.path.join(base_dir, 'package_dir_b')
sys.path.insert(0, package_dir_b)

# Reload the module to get the other version
importlib.reload(example)
print('Reloaded example from:', example.__file__)
print('  ', example.DATA)
```

Reloading an imported module re-imports the file, and uses the same `module` object to hold the results. Changing the path between the initial import and the call to `reload()` means a different module may be loaded the second time.

```
$ python3 sys_path_modify.py

Base directory: .
Imported example from: ./package_dir_a/example.py
   This is example A
Reloaded example from: ./package_dir_b/example.py
   This is example B
```

## Custom Importers

Modifying the search path lets a programmer control how standard Python modules are found. But, what if a program needs to import code from somewhere other than the usual `.py` or `.pyc` files on the file system? **PEP 302** solves this problem by introducing the idea of *import hooks*, which can trap an attempt to find a module on the search path and take alternative measures to load the code from somewhere else or apply pre-processing to it.

Custom importers are implemented in two separate phases. The *finder* is responsible for locating a module and providing a *loader* to manage the actual import. Custom module finders are added by appending a factory to the `sys.path_hooks` list. On import, each part of the path is given to a finder until one claims support (by not raising `ImportError`). That finder is then responsible for searching data storage represented by its path entry for named modules.

```
# sys_path_hooks_noisy.py

import sys


class NoisyImportFinder:

    PATH_TRIGGER = 'NoisyImportFinder_PATH_TRIGGER'

    def __init__(self, path_entry):
        print('Checking {}:'.format(path_entry), end=' ')
        if path_entry != self.PATH_TRIGGER:
            print('wrong finder')
            raise ImportError()
        else:
            print('works')
        return

    def find_module(self, fullname, path=None):
        print('Looking for {!r}'.format(fullname))
        return None


sys.path_hooks.append(NoisyImportFinder)

for hook in sys.path_hooks:
    print('Path hook: {}'.format(hook))

sys.path.insert(0, NoisyImportFinder.PATH_TRIGGER)

try:
    print('importing target_module')
    import target_module
except Exception as e:
    print('Import failed:', e)
```

This example illustrates how the finders are instantiated and queried. The `NoisyImportFinder` raises `ImportError` when instantiated with a path entry that does not match its special trigger value, which is obviously not a real path on the file system. This test prevents the `NoisyImportFinder` from breaking imports of real modules.

```
$ python3 sys_path_hooks_noisy.py

Path hook: <class 'zipimport.zipimporter'>
Path hook: <function
FileFinder.path_hook.<locals>.path_hook_for_FileFinder at
0x101afb6a8>
Path hook: <class '__main__.NoisyImportFinder'>
importing target_module
Checking NoisyImportFinder_PATH_TRIGGER: works
Looking for 'target_module'
Import failed: No module named 'target_module'
```

## Importing from a Shelve

When the finder locates a module, it is responsible for returning a *loader* capable of importing that module. This example illustrates a custom importer that saves its module contents in a database created by `shelve`.

First, a script is used to populate the shelf with a package containing a sub-module and sub-package.

```
# sys_shelve_importer_create.py

import shelve
import os

filename = '/tmp/pymotw_import_example.shelve'
if os.path.exists(filename + '.db'):
    os.unlink(filename + '.db')
with shelve.open(filename) as db:
    db['data:README'] = b"""
==============
package README
==============

This is the README for ``package``.
"""
    db['package.__init__'] = b"""
print('package imported')
message = 'This message is in package.__init__'
"""
    db['package.module1'] = b"""
print('package.module1 imported')
message = 'This message is in package.module1'
"""
    db['package.subpackage.__init__'] = b"""
print('package.subpackage imported')
message = 'This message is in package.subpackage.__init__'
"""
    db['package.subpackage.module2'] = b"""
print('package.subpackage.module2 imported')
message = 'This message is in package.subpackage.module2'
"""
    db['package.with_error'] = b"""
print('package.with_error being imported')
raise ValueError('raising exception to break import')
"""
    print('Created {} with:'.format(filename))
    for key in sorted(db.keys()):
        print('  ', key)
```

A real packaging script would read the contents from the file system, but using hard-coded values is sufficient for a simple example like this.

```
$ python3 sys_shelve_importer_create.py
```

```
Created /tmp/pymotw_import_example.shelve with:
   data:README
   package.__init__
   package.module1
   package.subpackage.__init__
   package.subpackage.module2
   package.with_error
```

The custom importer needs to provide finder and loader classes that know how to look in a shelf for the source of a module or package.

```
# sys_shelve_importer.py

import imp
import os
import shelve
import sys


def _mk_init_name(fullname):
    """Return the name of the __init__ module
    for a given package name.
    """
    if fullname.endswith('.__init__'):
        return fullname
    return fullname + '.__init__'


def _get_key_name(fullname, db):
    """Look in an open shelf for fullname or
    fullname.__init__, return the name found.
    """
    if fullname in db:
        return fullname
    init_name = _mk_init_name(fullname)
    if init_name in db:
        return init_name
    return None


class ShelveFinder:
    """Find modules collected in a shelve archive."""

    _maybe_recursing = False

    def __init__(self, path_entry):
        # Loading shelve causes an import recursive loop when it
        # imports dbm, and we know we are not going to load the
        # module # being imported, so when we seem to be
        # recursing just ignore the request so another finder
        # will be used.
        if ShelveFinder._maybe_recursing:
            raise ImportError
        try:
            # Test the path_entry to see if it is a valid shelf
            try:
                ShelveFinder._maybe_recursing = True
                with shelve.open(path_entry, 'r'):
                    pass
            finally:
                ShelveFinder._maybe_recursing = False
        except Exception as e:
            print('shelf could not import from {}: {}'.format(
                path_entry, e))
            raise
        else:
            print('shelf added to import path:', path_entry)
            self.path_entry = path_entry
        return

    def __str__(self):
```

```
        return '<{} for {!r}>'.format(self.__class__.__name__,
                                      self.path_entry)

    def find_module(self, fullname, path=None):
        path = path or self.path_entry
        print('\nlooking for {!r}\n  in {}'.format(
            fullname, path))
        with shelve.open(self.path_entry, 'r') as db:
            key_name = _get_key_name(fullname, db)
            if key_name:
                print('  found it as {}'.format(key_name))
                return ShelveLoader(path)
        print('  not found')
        return None


class ShelveLoader:
    """Load source for modules from shelve databases."""

    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def _get_filename(self, fullname):
        # Make up a fake filename that starts with the path entry
        # so pkgutil.get_data() works correctly.
        return os.path.join(self.path_entry, fullname)

    def get_source(self, fullname):
        print('loading source for {!r} from shelf'.format(
            fullname))
        try:
            with shelve.open(self.path_entry, 'r') as db:
                key_name = _get_key_name(fullname, db)
                if key_name:
                    return db[key_name]
                raise ImportError(
                    'could not find source for {}'.format(
                        fullname)
                )
        except Exception as e:
            print('could not load source:', e)
            raise ImportError(str(e))

    def get_code(self, fullname):
        source = self.get_source(fullname)
        print('compiling code for {!r}'.format(fullname))
        return compile(source, self._get_filename(fullname),
                       'exec', dont_inherit=True)

    def get_data(self, path):
        print('looking for data\n  in {}\n  for {!r}'.format(
            self.path_entry, path))
        if not path.startswith(self.path_entry):
            raise IOError
        path = path[len(self.path_entry) + 1:]
        key_name = 'data:' + path
        try:
            with shelve.open(self.path_entry, 'r') as db:
                return db[key_name]
        except Exception:
            # Convert all errors to IOError
            raise IOError()

    def is_package(self, fullname):
        init_name = _mk_init_name(fullname)
        with shelve.open(self.path_entry, 'r') as db:
            return init_name in db

    def load_module(self, fullname):
        source = self.get_source(fullname)
```

```
        if fullname in sys.modules:
            print('reusing module from import of {!r}'.format(
                fullname))
            mod = sys.modules[fullname]
        else:
            print('creating a new module object for {!r}'.format(
                fullname))
            mod = sys.modules.setdefault(
                fullname,
                imp.new_module(fullname)
            )

        # Set a few properties required by PEP 302
        mod.__file__ = self._get_filename(fullname)
        mod.__name__ = fullname
        mod.__path__ = self.path_entry
        mod.__loader__ = self
        # PEP-366 specifies that package's set __package__ to
        # their name, and modules have it set to their parent
        # package (if any).
        if self.is_package(fullname):
            mod.__package__ = fullname
        else:
            mod.__package__ = '.'.join(fullname.split('.')[:-1])

        if self.is_package(fullname):
            print('adding path for package')
            # Set __path__ for packages
            # so we can find the sub-modules.
            mod.__path__ = [self.path_entry]
        else:
            print('imported as regular module')

        print('execing source...')
        exec(source, mod.__dict__)
        print('done')
        return mod
```

Now ShelveFinder and ShelveLoader can be used to import code from a shelf. For example, importing the package just created:

```
# sys_shelve_importer_package.py

import sys
import sys_shelve_importer


def show_module_details(module):
    print('  message    :', module.message)
    print('  __name__   :', module.__name__)
    print('  __package__:', module.__package__)
    print('  __file__   :', module.__file__)
    print('  __path__   :', module.__path__)
    print('  __loader__ :', module.__loader__)


filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('Import of "package":')
import package

print()
print('Examine package details:')
show_module_details(package)

print()
print('Global settings:')
print('sys.modules entry:')
```

```
print(sys.modules['package'])
```

The shelf is added to the import path the first time an import occurs after the path is modified. The finder recognizes the shelf and returns a loader, which is used for all imports from that shelf. The initial package-level import creates a new module object and then uses exec to run the source loaded from the shelf. It uses the new module as the namespace so that names defined in the source are preserved as module-level attributes.

```
$ python3 sys_shelve_importer_package.py

Import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

Examine package details:
  message    : This message is in package.__init__
  __name__   : package
  __package__: package
  __file__   : /tmp/pymotw_import_example.shelve/package
  __path__   : ['/tmp/pymotw_import_example.shelve']
  __loader__ : <sys_shelve_importer.ShelveLoader object at
0x104589b70>

Global settings:
sys.modules entry:
<module 'package' (<sys_shelve_importer.ShelveLoader object at
0x104589b70>)>
```

## Custom Package Importing

Loading other modules and sub-packages proceeds in the same way.

```
# sys_shelve_importer_module.py

import sys
import sys_shelve_importer


def show_module_details(module):
    print('  message    :', module.message)
    print('  __name__   :', module.__name__)
    print('  __package__:', module.__package__)
    print('  __file__   :', module.__file__)
    print('  __path__   :', module.__path__)
    print('  __loader__ :', module.__loader__)


filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('Import of "package.module1":')
import package.module1

print()
print('Examine package.module1 details:')
show_module_details(package.module1)

print()
print('Import of "package.subpackage.module2":')
import package.subpackage.module2
```

```
print()
print('Examine package.subpackage.module2 details:')
show_module_details(package.subpackage.module2)
```

The finder receives the entire dotted name of the module to load, and returns a `ShelveLoader` configured to load modules from the path entry pointing to the shelf file. The fully qualified module name is passed to the loader's `load_module()` method, which constructs and returns a `module` instance.

```
$ python3 sys_shelve_importer_module.py

Import of "package.module1":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

looking for 'package.module1'
  in /tmp/pymotw_import_example.shelve
  found it as package.module1
loading source for 'package.module1' from shelf
creating a new module object for 'package.module1'
imported as regular module
execing source...
package.module1 imported
done

Examine package.module1 details:
  message    : This message is in package.module1
  __name__   : package.module1
  __package__: package
  __file__   : /tmp/pymotw_import_example.shelve/package.module1
  __path__   : /tmp/pymotw_import_example.shelve
  __loader__ : <sys_shelve_importer.ShelveLoader object at
0x10457dc18>

Import of "package.subpackage.module2":

looking for 'package.subpackage'
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.__init__
loading source for 'package.subpackage' from shelf
creating a new module object for 'package.subpackage'
adding path for package
execing source...
package.subpackage imported
done

looking for 'package.subpackage.module2'
  in /tmp/pymotw_import_example.shelve
  found it as package.subpackage.module2
loading source for 'package.subpackage.module2' from shelf
creating a new module object for 'package.subpackage.module2'
imported as regular module
execing source...
package.subpackage.module2 imported
done

Examine package.subpackage.module2 details:
  message    : This message is in package.subpackage.module2
  __name__   : package.subpackage.module2
  __package__: package.subpackage
  __file__   :
/tmp/pymotw_import_example.shelve/package.subpackage.module2
  __path__   : /tmp/pymotw_import_example.shelve
```

```
  __path__   : /tmp/pymotw_import_example.shelve
  __loader__ : <sys_shelve_importer.ShelveLoader object at
0x1045b5080>
```

## Reloading Modules in a Custom Importer

Reloading a module is handled slightly differently. Instead of creating a new module object, the existing object is re-used.

```
# sys_shelve_importer_reload.py
```

```
import importlib
import sys
import sys_shelve_importer

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

print('First import of "package":')
import package

print()
print('Reloading "package":')
importlib.reload(package)
```

By re-using the same object, existing references to the module are preserved even if class or function definitions are modified by the reload.

```
$ python3 sys_shelve_importer_reload.py

First import of "package":
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

Reloading "package":

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
reusing module from import of 'package'
adding path for package
execing source...
package imported
done
```

## Handling Import Errors

When a module cannot be located by any finder, `ImportError` is raised by the main import code.

```
# sys_shelve_importer_missing.py
```

```
import sys
import sys_shelve_importer

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

try:
```

```
try:
    import package.module3
except ImportError as e:
    print('Failed to import:', e)
```

Other errors during the import are propagated.

```
$ python3 sys_shelve_importer_missing.py

shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done

looking for 'package.module3'
  in /tmp/pymotw_import_example.shelve
  not found
Failed to import: No module named 'package.module3'
```

## Package Data

In addition to defining the API for loading executable Python code, PEP 302 defines an optional API for retrieving package data intended for distributing data files, documentation, and other non-code resources used by a package. By implementing `get_data()`, a loader can allow calling applications to support retrieval of data associated with the package without considering how the package is actually installed (especially without assuming that the package is stored as files on a file system).

```
# sys_shelve_importer_get_data.py
```

```
import sys
import sys_shelve_importer
import os
import pkgutil

filename = '/tmp/pymotw_import_example.shelve'
sys.path_hooks.append(sys_shelve_importer.ShelveFinder)
sys.path.insert(0, filename)

import package

readme_path = os.path.join(package.__path__[0], 'README')

readme = pkgutil.get_data('package', 'README')
# Equivalent to:
#  readme = package.__loader__.get_data(readme_path)
print(readme.decode('utf-8'))

foo_path = os.path.join(package.__path__[0], 'foo')
try:
    foo = pkgutil.get_data('package', 'foo')
    # Equivalent to:
    #  foo = package.__loader__.get_data(foo_path)
except IOError as err:
    print('ERROR: Could not load "foo"', err)
else:
    print(foo)
```

`get_data()` takes a path based on the module or package that owns the data, and returns the contents of the resource “file” as a byte string, or raises `IOError` if the resource does not exist.

```
$ python3 sys_shelve_importer_get_data.py

shelf added to import path: /tmp/pymotw_import_example.shelve
```

```
shelf added to import path: /tmp/pymotw_import_example.shelve

looking for 'package'
  in /tmp/pymotw_import_example.shelve
  found it as package.__init__
loading source for 'package' from shelf
creating a new module object for 'package'
adding path for package
execing source...
package imported
done
looking for data
  in /tmp/pymotw_import_example.shelve
  for '/tmp/pymotw_import_example.shelve/README'

==============
package README
==============

This is the README for ``package``.

looking for data
  in /tmp/pymotw_import_example.shelve
  for '/tmp/pymotw_import_example.shelve/foo'
ERROR: Could not load "foo"
```

### See also

- pkgutil – Includes `get_data()` for retrieving data from a package.

## Importer Cache

Searching through all of the hooks each time a module is imported can become expensive. To save time, `sys.path_importer_cache` is maintained as a mapping between a path entry and the loader that can use the value to find modules.

```
# sys_path_importer_cache.py

import os
import sys

prefix = os.path.abspath(sys.prefix)

print('PATH:')
for name in sys.path:
    name = name.replace(prefix, '...')
    print(' ', name)

print()
print('IMPORTERS:')
for name, cache_value in sys.path_importer_cache.items():
    if '..' in name:
        name = os.path.abspath(name)
    name = name.replace(prefix, '...')
    print('  {}: {!r}'.format(name, cache_value))
```

A `FileFinder` is used for path locations found on the file system. Locations on the path not supported by any finder are associated with a `None`, since they cannot be used to import modules. The output below has been truncated due to formatting constraints.

```
$ python3 sys_path_importer_cache.py

PATH:
  /Users/dhellmann/Documents/PyMOTW/Python3/pymotw-3/source/sys
  .../lib/python35.zip
  .../lib/python3.5
  .../lib/python3.5/plat-darwin
  .../lib/python3.5/lib-dynload
  .../lib/python3.5/site-packages

IMPORTERS:
  /Users/dhellmann/Documents/PyMOTW/Python3/pymotw-3/source/sys: None
  .../lib/python35.zip: None
  .../lib/python3.5: <FileFinder('.../lib/python3.5')>
  .../lib/python3.5/plat-darwin: <FileFinder('.../lib/python3.5/plat-darwin')>
  .../lib/python3.5/lib-dynload: <FileFinder('.../lib/python3.5/lib-dynload')>
  .../lib/python3.5/site-packages: <FileFinder('.../lib/python3.5/site-packages')>
```

```
  .../lib/python3.5/site-packages

IMPORTERS:
  sys_path_importer_cache.py: None
  .../lib/python3.5/encodings: FileFinder(
  '.../lib/python3.5/encodings')
  .../lib/python3.5/lib-dynload: FileFinder(
  '.../lib/python3.5/lib-dynload')
  .../lib/python3.5/lib-dynload: FileFinder(
  '.../lib/python3.5/lib-dynload')
  .../lib/python3.5/site-packages: FileFinder(
  '.../lib/python3.5/site-packages')
  .../lib/python3.5: FileFinder(
  '.../lib/python3.5/')
  .../lib/python3.5/plat-darwin: FileFinder(
  '.../lib/python3.5/plat-darwin')
  .../lib/python3.5: FileFinder(
  '.../lib/python3.5')
  .../lib/python35.zip: None
  .../lib/python3.5/plat-darwin: FileFinder(
  '.../lib/python3.5/plat-darwin')
```

## Meta Path

The `sys.meta_path` further extends the sources of potential imports by allowing a finder to be searched *before* the regular `sys.path` is scanned. The API for a finder on the meta-path is the same as for a regular path. The difference is that the metafinder is not limited to a single entry in `sys.path` – it can search anywhere at all.

```
# sys_meta_path.py

import sys
import types


class NoisyMetaImportFinder:

    def __init__(self, prefix):
        print('Creating NoisyMetaImportFinder for {}'.format(
            prefix))
        self.prefix = prefix
        return

    def find_module(self, fullname, path=None):
        print('looking for {!r} with path {!r}'.format(
            fullname, path))
        name_parts = fullname.split('.')
        if name_parts and name_parts[0] == self.prefix:
            print(' ... found prefix, returning loader')
            return NoisyMetaImportLoader(path)
        else:
            print(' ... not the right prefix, cannot load')
        return None


class NoisyMetaImportLoader:

    def __init__(self, path_entry):
        self.path_entry = path_entry
        return

    def load_module(self, fullname):
        print('loading {}'.format(fullname))
        if fullname in sys.modules:
            mod = sys.modules[fullname]
        else:
            mod = sys.modules.setdefault(
                fullname,
                types.ModuleType(fullname))

        # Set a few properties required by PEP 302
        mod.__file__ = fullname
```

```
        mod.__file__ = fullname
        mod.__name__ = fullname
        # always looks like a package
        mod.__path__ = ['path-entry-goes-here']
        mod.__loader__ = self
        mod.__package__ = '.'.join(fullname.split('.')[:-1])

        return mod


# Install the meta-path finder
sys.meta_path.append(NoisyMetaImportFinder('foo'))

# Import some modules that are "found" by the meta-path finder
print()
import foo

print()
import foo.bar

# Import a module that is not found
print()
try:
    import bar
except ImportError as e:
    pass
```

Each finder on the meta-path is interrogated before `sys.path` is searched, so there is always an opportunity to have a central importer load modules without explicitly modifying `sys.path`. Once the module is "found," the loader API works in the same way as for regular loaders (although this example is truncated for simplicity).

```
$ python3 sys_meta_path.py

Creating NoisyMetaImportFinder for foo

looking for 'foo' with path None
 ... found prefix, returning loader
loading foo

looking for 'foo.bar' with path ['path-entry-goes-here']
 ... found prefix, returning loader
loading foo.bar

looking for 'bar' with path None
 ... not the right prefix, cannot load
```

## See also

- `importlib` – Base classes and other tools for creating custom importers.
- `zipimport` – Implements importing Python modules from inside ZIP archives.
- The Internal Structure of Python Eggs – setuptools documentation for the egg format
- Wheel – Documentation for `wheel` archive format for installable Python code.
- **PEP 302** – Import Hooks
- **PEP 366** – Main module explicit relative imports
- **PEP 427** – The Wheel Binary Package Format 1.0
- Import this, that, and the other thing: custom importers – Brett Cannon's PyCon 2010 presentation.

Low-level Thread Support

Tracing a Program As It Runs

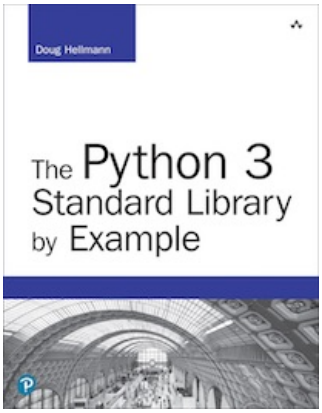
**Quick Links**



*This page was last updated 2018-12-09.*

**Navigation**

Low-level Thread Support
Tracing a Program As It Runs




Get the book

*The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.*

*Looking for examples for Python 2?*

**This Site**

Module Index
Index



**Other Writing**

Blog
The Python Standard Library By Example