CHAPTER

# THIRTYTWO

# IMPORTING MODULES

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

## 32.1 zipimport — Import modules from Zip archives

This module adds the ability to import Python modules (`*.py`, `*.pyc`) and packages from ZIP-format archives. It is usually not needed to use the *zipimport* module explicitly; it is automatically used by the built-in `import` mechanism for *sys.path* items that are paths to ZIP archives.

Typically, *sys.path* is a list of directory names as strings. This module also allows an item of *sys.path* to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but only files `.py` and `.pyc` are available for import. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

ZIP archives with an archive comment are currently not supported.

**See also:**

**PKZIP Application Note** Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

**PEP 273 - Import Modules from Zip Archives** Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

**PEP 302 - New Import Hooks** The PEP to add the import hooks that help this module work.

This module defines an exception:

`exception zipimport.ZipImportError`
Exception raised by zipimporter objects. It's a subclass of *ImportError*, so it can be caught as *ImportError*, too.

### 32.1.1 zipimporter Objects

*zipimporter* is the class for importing ZIP files.

**class zipimport.zipimporter**(*archivepath*)

Create a new zipimporter instance. *archivepath* must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an *archivepath* of `foo/bar.zip/lib` will look for modules in the `lib` directory inside the ZIP file `foo/bar.zip` (provided that it exists).

*ZipImportError* is raised if *archivepath* doesn't point to a valid ZIP archive.

**find_module**(*fullname*[, *path*])

Search for a module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or *None* if it wasn't. The optional *path* argument is ignored—it's there for compatibility with the importer protocol.

**get_code**(*fullname*)

Return the code object for the specified module. Raise *ZipImportError* if the module couldn't be found.

**get_data**(*pathname*)

Return the data associated with *pathname*. Raise *OSError* if the file wasn't found.

Changed in version 3.3: *IOError* used to be raised instead of *OSError*.

**get_filename**(*fullname*)

Return the value `__file__` would be set to if the specified module was imported. Raise *ZipImportError* if the module couldn't be found.

New in version 3.1.

**get_source**(*fullname*)

Return the source code for the specified module. Raise *ZipImportError* if the module couldn't be found, return *None* if the archive does contain the module, but has no source for it.

**is_package**(*fullname*)

Return `True` if the module specified by *fullname* is a package. Raise *ZipImportError* if the module couldn't be found.

**load_module**(*fullname*)

Load the module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. It returns the imported module, or raises *ZipImportError* if it wasn't found.

**archive**

The file name of the importer's associated ZIP file, without a possible subpath.

**prefix**

The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The *archive* and *prefix* attributes, when combined with a slash, equal the original *archivepath* argument given to the *zipimporter* constructor.

### 32.1.2 Examples

Here is an example that imports a module from a ZIP archive - note that the *zipimport* module is not explicitly used.

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
```

(continues on next page)

(continued from previous page)

```
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

## 32.2 pkgutil — Package extension utility

**Source code:** Lib/pkgutil.py

---

This module provides utilities for the import system, in particular package support.

**class** pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)
A namedtuple that holds a brief summary of a module's info.

New in version 3.6.

pkgutil.**extend_path**(*path*, *name*)
Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's `__init__.py`:

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

This will add to the package's `__path__` all subdirectories of directories on `sys.path` named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the *site* module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value: apart from checking for duplicates, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

**class** pkgutil.**ImpImporter**(*dirname=None*)
**PEP 302** Finder that wraps Python's "classic" import algorithm.

If *dirname* is a string, a **PEP 302** finder is created that searches that directory. If *dirname* is `None`, a **PEP 302** finder is created that searches the current *sys.path*, plus any modules that are frozen or built-in.

Note that *ImpImporter* does not currently support being used by placement on *sys.meta_path*.

Deprecated since version 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

**class** pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
*Loader* that wraps Python's "classic" import algorithm.

Deprecated since version 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

pkgutil.**find_loader**(*fullname*)
Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around *importlib.util.find_spec()* that converts most failures to *ImportError* and only returns the loader rather than the full `ModuleSpec`.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

Changed in version 3.4: Updated to be based on **PEP 451**

pkgutil.**get_importer**(*path_item*)
Retrieve a *finder* for the given *path_item*.

The returned finder is cached in *sys.path_importer_cache* if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**get_loader**(*module_or_name*)
Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns `None` if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package `__path__`.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

Changed in version 3.4: Updated to be based on **PEP 451**

pkgutil.**iter_importers**(*fullname=''*)
Yield *finder* objects for the given module name.

If fullname contains a '.', the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on `sys.path`.

*path* should be either `None` or a list of paths to look for modules in.

*prefix* is a string to output on the front of every module name on output.

---

**Note:** Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and

*zipimport.zipimporter*.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)
Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is `None`, all accessible modules.

*path* should be either `None` or a list of paths to look for modules in.

*prefix* is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

*onerror* is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

**Note:** Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Changed in version 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**get_data**(*package*, *resource*)
Get a resource from a package.

This is a wrapper for the *loader* *get_data* API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then `None` is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

## 32.3 `modulefinder` — Find modules used by a script

**Source code:** Lib/modulefinder.py

This module provides a *ModuleFinder* class that can be used to determine the set of modules imported by a script. `modulefinder.py` can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

`modulefinder.AddPackagePath(pkg_name, path)`
Record that the package named *pkg_name* can be found in the specified *path*.

`modulefinder.ReplacePackage(oldname, newname)`
Allows specifying that the module named *oldname* is in fact the package named *newname*.

`class modulefinder.ModuleFinder(path=None, debug=0, excludes=[], replace_paths=[])`
This class provides *run_script()* and *report()* methods to determine the set of modules imported by a script. *path* can be a list of directories to search for modules; if not specified, `sys.path` is used. *debug* sets the debugging level; higher values make the class print debugging messages about what it's doing. *excludes* is a list of module names to exclude from the analysis. *replace_paths* is a list of `(oldpath, newpath)` tuples that will be replaced in module paths.

`report()`
Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

`run_script(pathname)`
Analyze the contents of the *pathname* file, which must contain Python code.

`modules`
A dictionary mapping module names to modules. See *Example usage of ModuleFinder*.

### 32.3.1 Example usage of `ModuleFinder`

The script that is going to get analyzed later on (bacon.py):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))
```

(continues on next page)

(continued from previous page)

```
print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Sample output (may vary depending on the architecture):

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
--------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

## 32.4 runpy — Locating and executing Python modules

**Source code:** Lib/runpy.py

---

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a *runpy* function has returned. If that limitation is not acceptable for a given use case, *importlib* is likely to be a more suitable choice than this module.

The *runpy* module provides two functions:

**runpy.run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the

special global variables below are defined in the supplied dictionary, those definitions are overridden by *run_module()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to *True*, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of *sys* is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the *sys* module be left alone when invoking this function from threaded code.

**See also:**

The `-m` option offering equivalent functionality from the command line.

Changed in version 3.1: Added ability to execute packages by looking for a `__main__` submodule.

Changed in version 3.2: Added `__cached__` global variable (see **PEP 3147**).

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

`runpy.run_path`(*file_path*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a *__main__* module using the updated path. Note that there is no special protection against invoking an existing *__main__* entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by *run_path()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not *None* and to `'<run_path>'` otherwise.

If the supplied path directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to the supplied path, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to *None*.

If the supplied path is a reference to a valid sys.path entry, then `__spec__` will be set appropriately for the imported `__main__` module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the *sys* module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Note that, unlike *run_module()*, the alterations made to *sys* are not optional in this function as these adjustments are essential to allowing the execution of sys.path entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

**See also:**

using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

New in version 3.2.

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid sys.path entry rather than being executed directly.

**See also:**

**PEP 338** – **Executing modules as scripts** PEP written and implemented by Nick Coghlan.

**PEP 366** – **Main module explicit relative imports** PEP written and implemented by Nick Coghlan.

**PEP 451** – **A ModuleSpec Type for the Import System** PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The *importlib.import_module()* function

## 32.5 `importlib` — The implementation of `import`

New in version 3.1.

**Source code:** Lib/importlib/__init__.py

---

### 32.5.1 Introduction

The purpose of the *importlib* package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement `import` are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

**See also:**

**import** The language reference for the `import` statement.

**Packages specification** Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *sys.modules*).

**The *__import__()* function** The `import` statement is syntactic sugar for this function.

**PEP 235** Import on Case-Insensitive Platforms

**PEP 263** Defining Python Source Code Encodings

**PEP 302** New Import Hooks

**PEP 328** Imports: Multi-Line and Absolute/Relative

**PEP 366** Main module explicit relative imports

**PEP 420** Implicit namespace packages

**PEP 451** A ModuleSpec Type for the Import System

**PEP 488** Elimination of PYO files

**PEP 489** Multi-phase extension module initialization

**PEP 552** Deterministic pycs

**PEP 3120** Using UTF-8 as the Default Source Encoding

**PEP 3147** PYC Repository Directories

### 32.5.2 Functions

`importlib.__import__`(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
An implementation of the built-in *__import__()* function.

> **Note:** Programmatic importing of modules should use *import_module()* instead of this function.

`importlib.import_module`(*name*, *package=None*)
Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The *import_module()* function acts as a simplifying wrapper around *importlib.__import__()*. This means all semantics of the function are derived from *importlib.__import__()*. The most important difference between these two functions is that *import_module()* returns the specified package or module (e.g. `pkg.mod`), while *__import__()* returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call *invalidate_caches()* in order for the new module to be noticed by the import system.

Changed in version 3.3: Parent packages are automatically imported.

`importlib.find_loader`(*name*, *path=None*)
Find the loader for a module, optionally within the specified *path*. If the module is in *sys.modules*, then `sys.modules[name].__loader__` is returned (unless the loader would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to *path*.

New in version 3.3.

Changed in version 3.4: If `__loader__` is not set, raise *ValueError*, just like when the attribute is set to `None`.

Deprecated since version 3.4: Use *importlib.util.find_spec()* instead.

`importlib.invalidate_caches()`
Invalidate the internal caches of finders stored at *sys.meta_path*. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

New in version 3.3.

`importlib.reload(`*module*`)`
Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in *sys.modules*).

When *reload()* is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, *builtins* and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

New in version 3.4.

Changed in version 3.7: *ModuleNotFoundError* is raised when the module being reloaded lacks a `ModuleSpec`.

### 32.5.3 `importlib.abc` – Abstract base classes related to import

**Source code:** Lib/importlib/abc.py

---

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

**class** importlib.abc.**Finder**

An abstract base class representing a *finder*.

Deprecated since version 3.3: Use *MetaPathFinder* or *PathEntryFinder* instead.

**abstractmethod find_module**(*fullname*, *path=None*)

An abstract method for finding a *loader* for the specified module. Originally specified in **PEP 302**, this method was meant for use in *sys.meta_path* and in the path-based import subsystem.

Changed in version 3.4: Returns `None` when called instead of raising *NotImplementedError*.

**class** importlib.abc.**MetaPathFinder**

An abstract base class representing a *meta path finder*. For compatibility, this is a subclass of *Finder*.

New in version 3.3.

**find_spec**(*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return.

New in version 3.4.

**find_module**(*fullname*, *path*)

A legacy method for finding a *loader* for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

If *find_spec()* is defined, backwards-compatible functionality is provided.

Changed in version 3.4: Returns `None` when called instead of raising *NotImplementedError*. Can use *find_spec()* to provide functionality.

Deprecated since version 3.4: Use *find_spec()* instead.

**invalidate_caches()**
An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.invalidate_caches()* when invalidating the caches of all finders on *sys.meta_path*.

Changed in version 3.4: Returns `None` when called instead of `NotImplemented`.

**class importlib.abc.PathEntryFinder**
An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of *Finder* for compatibility reasons only.

New in version 3.3.

**find_spec**(*fullname*, *target=None*)
An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return.

New in version 3.4.

**find_loader**(*fullname*)
A legacy method for finding a *loader* for the specified module. Returns a 2-tuple of `(loader, portion)` where `portion` is a sequence of file system locations contributing to part of a namespace package. The loader may be `None` while specifying `portion` to signify the contribution of the file system locations to a namespace package. An empty list can be used for `portion` to signify the loader is not part of a namespace package. If `loader` is `None` and `portion` is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If *find_spec()* is defined then backwards-compatible functionality is provided.

Changed in version 3.4: Returns `(None, []) ` instead of raising *NotImplementedError*. Uses *find_spec()* when available to provide functionality.

Deprecated since version 3.4: Use *find_spec()* instead.

**find_module**(*fullname*)
A concrete implementation of *Finder.find_module()* which is equivalent to `self.find_loader(fullname)[0]`.

Deprecated since version 3.4: Use *find_spec()* instead.

**invalidate_caches()**
An optional method which, when called, should invalidate any internal cache used by the finder. Used by `PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

**class importlib.abc.Loader**
An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.abc.ResourceReader*.

Changed in version 3.7: Introduced the optional `get_resource_reader()` method.

**create_module**(*spec*)
A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

New in version 3.4.

Changed in version 3.5: Starting in Python 3.6, this method will not be optional when *exec_module()* is defined.

**exec_module**(*module*)
An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, *create_module()* must be defined.

New in version 3.4.

Changed in version 3.6: *create_module()* must also be defined.

**load_module**(*fullname*)
A legacy method for loading a module. If the module cannot be loaded, *ImportError* is raised, otherwise the loaded module is returned.

If the requested module already exists in *sys.modules*, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into *sys.modules* before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from *sys.modules*; modules already in *sys.modules* before the loader began execution should be left alone (see *importlib.util.module_for_loader()*).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded):

- **__name__** The name of the module.
- **__file__** The path to where the module data is stored (not set for built-in modules).
- **__cached__** The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- **__path__** A list of strings specifying the search path within a package. This attribute is not set on modules.
- **__package__** The parent package for the module/package. If the module is top-level then it has a value of the empty string. The *importlib.util.module_for_loader()* decorator can handle the details for `__package__`.
- **__loader__** The loader used to load the module. The *importlib.util.module_for_loader()* decorator can handle the details for `__package__`.

When *exec_module()* is available then backwards-compatible functionality is provided.

Changed in version 3.4: Raise *ImportError* when called instead of *NotImplementedError*. Functionality provided when *exec_module()* is available.

Deprecated since version 3.4: The recommended API for loading a module is *exec_module()* (and *create_module()*). Loaders should implement it instead of load_module(). The import machinery takes care of all the other responsibilities of load_module() when exec_module() is implemented.

**module_repr**(*module*)
A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default repr() will use the result of this method as appropriate.

New in version 3.3.

Changed in version 3.4: Made optional instead of an abstractmethod.

Deprecated since version 3.4: The import machinery now takes care of this automatically.

**class** importlib.abc.**ResourceReader**
An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored in a e.g. zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_loader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return *None*. An object compatible with this ABC should only be returned when the specified module is a package.

New in version 3.7.

**abstractmethod open_resource**(*resource*)
Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, *FileNotFoundError* is raised.

**abstractmethod resource_path**(*resource*)
Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise *FileNotFoundError*.

**abstractmethod is_resource**(*name*)
Returns `True` if the named *name* is considered a resource. *FileNotFoundError* is raised if *name* does not exist.

**abstractmethod contents**()
Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which *is_resource()* would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

**class importlib.abc.ResourceLoader**
An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

Deprecated since version 3.7: This ABC is deprecated in favour of supporting resource loading through *importlib.abc.ResourceReader*.

**abstractmethod get_data**(*path*)
An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Changed in version 3.4: Raises *OSError* instead of *NotImplementedError*.

**class importlib.abc.InspectLoader**
An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

**get_code**(*fullname*)
Return the code object for a module, or `None` if the module does not have a code object (as would

be the case, for example, for a built-in module). Raise an *ImportError* if loader cannot find the requested module.

---

**Note:** While the method has a default implementation, it is suggested that it be overridden if possible for performance.

---

Changed in version 3.4: No longer abstract and a concrete implementation is provided.

**abstractmethod get_source**(*fullname*)
An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises *ImportError* if the loader cannot find the module specified.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

**is_package**(*fullname*)
An abstract method to return a true value if the module is a package, a false value otherwise. *ImportError* is raised if the *loader* cannot find the module.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

**static source_to_code**(*data*, *path='<string>'*)
Create a code object from Python source.

The *data* argument can be whatever the *compile()* function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

New in version 3.4.

Changed in version 3.5: Made the method static.

**exec_module**(*module*)
Implementation of *Loader.exec_module()*.

New in version 3.4.

**load_module**(*fullname*)
Implementation of *Loader.load_module()*.

Deprecated since version 3.4: use *exec_module()* instead.

**class importlib.abc.ExecutionLoader**
An abstract base class which inherits from *InspectLoader* that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

**abstractmethod get_filename**(*fullname*)
An abstract method that is to return the value of `__file__` for the specified module. If no path is available, *ImportError* is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Changed in version 3.4: Raises *ImportError* instead of *NotImplementedError*.

**class importlib.abc.FileLoader**(*fullname*, *path*)
An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

New in version 3.3.

**name**
The name of the module the loader can handle.

**path**
Path to the file of the module.

**load_module**(*fullname*)
Calls super's `load_module()`.

Deprecated since version 3.4: Use *Loader.exec_module()* instead.

**abstractmethod get_filename**(*fullname*)
Returns *path*.

**abstractmethod get_data**(*path*)
Reads *path* as a binary file and returns the bytes from it.

**class** importlib.abc.**SourceLoader**
An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- *ExecutionLoader.get_filename()* Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

**path_stats**(*path*)
Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

New in version 3.3.

Changed in version 3.4: Raise *OSError* instead of *NotImplementedError*.

**path_mtime**(*path*)
Optional abstract method which returns the modification time for the specified path.

Deprecated since version 3.3: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

Changed in version 3.4: Raise *OSError* instead of *NotImplementedError*.

**set_data**(*path*, *data*)
Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES*/*PermissionError*), do not propagate the exception.

Changed in version 3.4: No longer raises *NotImplementedError* when called.

`get_code`(*fullname*)
Concrete implementation of *InspectLoader.get_code()*.

`exec_module`(*module*)
Concrete implementation of *Loader.exec_module()*.

New in version 3.4.

`load_module`(*fullname*)
Concrete implementation of *Loader.load_module()*.

Deprecated since version 3.4: Use *exec_module()* instead.

`get_source`(*fullname*)
Concrete implementation of *InspectLoader.get_source()*.

`is_package`(*fullname*)
Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

### 32.5.4 `importlib.resources` – Resources

**Source code:** Lib/importlib/resources.py

---

New in version 3.7.

This module leverages Python's import system to provide access to *resources* within *packages*. If you can import a package, you can access resources within that package. Resources can be opened or read, in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system.

---

**Note:** This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

---

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.abc.ResourceReader*.

The following types are defined.

`importlib.resources.Package`
The `Package` type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a `Package`, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not `None`.

`importlib.resources.Resource`
This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

The following functions are available.

importlib.resources.**open_binary**(*package*, *resource*)
Open for binary reading the *resource* within *package*.

*package* is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns a `typing.BinaryIO` instance, a binary I/O stream open for reading.

importlib.resources.**open_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Open for text reading the *resource* within *package*. By default, the resource is opened for reading as UTF-8.

*package* is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*.

This function returns a `typing.TextIO` instance, a text I/O stream open for reading.

importlib.resources.**read_binary**(*package*, *resource*)
Read and return the contents of the *resource* within *package* as `bytes`.

*package* is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns the contents of the resource as *bytes*.

importlib.resources.**read_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Read and return the contents of *resource* within *package* as a `str`. By default, the contents are read as strict UTF-8.

*package* is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*. This function returns the contents of the resource as *str*.

importlib.resources.**path**(*package*, *resource*)
Return the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary file created when the resource needs to be extracted from e.g. a zip file.

*package* is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory).

importlib.resources.**is_resource**(*package*, *name*)
Return `True` if there is a resource named *name* in the package, otherwise `False`. Remember that directories are *not* resources! *package* is either a name or a module object which conforms to the `Package` requirements.

importlib.resources.**contents**(*package*)
Return an iterable over the named items within the package. The iterable returns *str* resources (e.g. files) and non-resources (e.g. directories). The iterable does not recurse into subdirectories.

*package* is either a name or a module object which conforms to the `Package` requirements.

### 32.5.5 importlib.machinery – Importers and path hooks

**Source code:** Lib/importlib/machinery.py

---

This module contains the various objects that help `import` find and load modules.

`importlib.machinery.SOURCE_SUFFIXES`
A list of strings representing the recognized file suffixes for source modules.

New in version 3.3.

`importlib.machinery.DEBUG_BYTECODE_SUFFIXES`
A list of strings representing the file suffixes for non-optimized bytecode modules.

New in version 3.3.

Deprecated since version 3.5: Use *BYTECODE_SUFFIXES* instead.

`importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES`
A list of strings representing the file suffixes for optimized bytecode modules.

New in version 3.3.

Deprecated since version 3.5: Use *BYTECODE_SUFFIXES* instead.

`importlib.machinery.BYTECODE_SUFFIXES`
A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

New in version 3.3.

Changed in version 3.5: The value is no longer dependent on `__debug__`.

`importlib.machinery.EXTENSION_SUFFIXES`
A list of strings representing the recognized file suffixes for extension modules.

New in version 3.3.

`importlib.machinery.all_suffixes()`
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

New in version 3.3.

`class importlib.machinery.BuiltinImporter`
An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.5: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

`class importlib.machinery.FrozenImporter`
An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

`class importlib.machinery.WindowsRegistryFinder`
*Finder* for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

New in version 3.3.

Deprecated since version 3.6: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

**class** importlib.machinery.**PathFinder**

A *Finder* for *sys.path* and package **__path__** attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

**classmethod find_spec**(*fullname*, *path=None*, *target=None*)

Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then **None** is both stored in the cache and returned.

New in version 3.4.

Changed in version 3.5: If the current working directory – represented by an empty string – is no longer valid then **None** is returned but no value is cached in *sys.path_importer_cache*.

**classmethod find_module**(*fullname*, *path=None*)

A legacy wrapper around *find_spec()*.

Deprecated since version 3.4: Use *find_spec()* instead.

**classmethod invalidate_caches**()

Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to **None** are deleted.

Changed in version 3.7: Entries of **None** in *sys.path_importer_cache* are deleted.

Changed in version 3.4: Calls objects in *sys.path_hooks* with the current working directory for **''** (i.e. the empty string).

**class** importlib.machinery.**FileFinder**(*path*, **loader_details*)

A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call *importlib.invalidate_caches()*.

New in version 3.3.

**path**

The path the finder will search in.

**find_spec**(*fullname*, *target=None*)
Attempt to find the spec to handle *fullname* within *path*.

New in version 3.4.

**find_loader**(*fullname*)
Attempt to find the loader to handle *fullname* within *path*.

**invalidate_caches**()
Clear out the internal cache.

**classmethod path_hook**(**loader_details*)
A class method which returns a closure for use on *sys.path_hooks*. An instance of *FileFinder* is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, *ImportError* is raised.

**class** importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.SourceLoader* by subclassing *importlib.abc.FileLoader* and providing some concrete implementations of other methods.

New in version 3.3.

**name**
The name of the module that this loader will handle.

**path**
The path to the source file.

**is_package**(*fullname*)
Return true if *path* appears to be for a package.

**path_stats**(*path*)
Concrete implementation of *importlib.abc.SourceLoader.path_stats()*.

**set_data**(*path*, *data*)
Concrete implementation of *importlib.abc.SourceLoader.set_data()*.

**load_module**(*name=None*)
Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6: Use *importlib.abc.Loader.exec_module()* instead.

**class** importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.FileLoader* which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

New in version 3.3.

**name**
The name of the module the loader will handle.

**path**
The path to the bytecode file.

**is_package**(*fullname*)
Determines if the module is a package based on *path*.

**get_code**(*fullname*)
Returns the code object for *name* created from *path*.

`get_source`(*fullname*)
Returns `None` as bytecode files have no source when this loader is used.

`load_module`(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Deprecated since version 3.6: Use *importlib.abc.Loader.exec_module()* instead.

`class importlib.machinery.ExtensionFileLoader`(*fullname*, *path*)
A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

New in version 3.3.

`name`
Name of the module the loader supports.

`path`
Path to the extension module.

`create_module`(*spec*)
Creates the module object from the given specification in accordance with **PEP 489**.

New in version 3.5.

`exec_module`(*module*)
Initializes the given module object in accordance with **PEP 489**.

New in version 3.5.

`is_package`(*fullname*)
Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

`get_code`(*fullname*)
Returns `None` as extension modules lack a code object.

`get_source`(*fullname*)
Returns `None` as extension modules do not have source code.

`get_filename`(*fullname*)
Returns *path*.

New in version 3.4.

`class importlib.machinery.ModuleSpec`(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)
A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute available directly on the module object. E.g. `module.__spec__.origin == module.__file__`. Note however that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

New in version 3.4.

`name`

(`__name__`)

A string for the fully-qualified name of the module.

`loader`

(__loader__)

The loader to use for loading. For namespace packages this should be set to None.

**origin**

(__file__)

Name of the place from which the module is loaded, e.g. "builtin" for built-in modules and the filename for modules loaded from source. Normally "origin" should be set, but it may be None (the default) which indicates it is unspecified (e.g. for namespace packages).

**submodule_search_locations**

(__path__)

List of strings for where to find submodules, if a package (None otherwise).

**loader_state**

Container of extra module-specific data for use during loading (or None).

**cached**

(__cached__)

String for where the compiled module should be stored (or None).

**parent**

(__package__)

(Read-only) Fully-qualified name of the package to which the module belongs as a submodule (or None).

**has_location**

Boolean indicating whether or not the module's "origin" attribute refers to a loadable location.

### 32.5.6 importlib.util – Utility code for importers

**Source code:** Lib/importlib/util.py

---

This module contains the various objects that help in the construction of an *importer*.

importlib.util.**MAGIC_NUMBER**
The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *importlib.abc.SourceLoader*.

New in version 3.4.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, ***, *optimization=None*)
Return the **PEP 3147**/**PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is /foo/bar/baz.py the return value would be /foo/bar/__pycache__/baz.cpython-32.pyc for Python 3.2. The cpython-32 string comes from the current magic tag (see get_tag(); if sys.implementation.cache_tag is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so /foo/bar/baz.py with an *optimization* of '' will result in a bytecode path of /foo/bar/__pycache__/baz.cpython-32.pyc. None causes the interpter's optimization level to be used. Any other value's string representation being used, so /foo/bar/baz.py with an *optimization* of 2 will lead to the bytecode path of /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *TypeError* is raised.

New in version 3.4.

Changed in version 3.5: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Changed in version 3.6: Accepts a *path-like object*.

`importlib.util.source_from_cache(path)`
Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

New in version 3.4.

Changed in version 3.6: Accepts a *path-like object*.

`importlib.util.decode_source(source_bytes)`
Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

New in version 3.4.

`importlib.util.resolve_name(name, package)`
Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the **package** argument is needed.

*ValueError* is raised if **name** is a relative module name but package is a false value (e.g. `None` or the empty string). *ValueError* is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

New in version 3.3.

`importlib.util.find_spec(name, package=None)`
Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

**name** and **package** work the same as for `import_module()`.

New in version 3.4.

Changed in version 3.7: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

`importlib.util.module_from_spec(spec)`
Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return `None`, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

New in version 3.5.

**@importlib.util.module_for_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. `load_module(self, module)`) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in *sys.modules* then a new one is constructed. Regardless of where the module came from, `__loader__` set to **self** and `__package__` is set based on what *importlib.abc.InspectLoader.is_package()* returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to *sys.modules*, then the module will be removed to prevent a partially initialized module from being in left in *sys.modules*. If the module was already in *sys.modules* then it is left alone.

Changed in version 3.3: `__loader__` and `__package__` are automatically set (when possible).

Changed in version 3.4: Set `__name__`, `__loader__` `__package__` unconditionally to support reloading.

Deprecated since version 3.4: The import machinery now directly performs all the functionality provided by this function.

**@importlib.util.set_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to set the `__loader__` attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. `self`) is what `__loader__` should be set to.

Changed in version 3.4: Set `__loader__` if set to `None`, as if the attribute does not exist.

Deprecated since version 3.4: The import machinery takes care of this automatically.

**@importlib.util.set_package**

A *decorator* for *importlib.abc.Loader.load_module()* to set the `__package__` attribute on the returned module. If `__package__` is set and has a value other than `None` it will not be changed.

Deprecated since version 3.4: The import machinery takes care of this automatically.

**importlib.util.spec_from_loader**(*name*, *loader*, ***, *origin=None*, *is_package=None*)

A factory function for creating a `ModuleSpec` instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as `InspectLoader.is_package()`, to fill in any missing information on the spec.

New in version 3.4.

**importlib.util.spec_from_file_location**(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a `ModuleSpec` instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

New in version 3.4.

Changed in version 3.6: Accepts a *path-like object*.

**importlib.util.source_hash**(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the *source_hash()* of the corresponding source file's contents in its header.

New in version 3.7.

**class** importlib.util.**LazyLoader**(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return `None` or a type for which its `__class__` attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

---

**Note:** For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

---

New in version 3.5.

Changed in version 3.6: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

**classmethod** **factory**(*loader*)

A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

### 32.5.7 Examples

#### Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

#### Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
```

(continues on next page)

(continued from previous page)

```
if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```

### Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module
```

### Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on *sys.meta_path* while the latter is what you create using a *path entry hook* on *sys.path_hooks* which works with *sys.path* entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
```

(continues on next page)

(continued from previous page)

```
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

### Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *`importlib.import_module()`* (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        raise ImportError(f'No module named {absolute_name!r}')
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```