31.1. `imp` — Access the `import` internals¶

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`imp.get_magic`()¶

Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

`imp.get_suffixes`()¶
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in `open()` function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and *type* is the file type, which has one of the values `PY_SOURCE`, `PY_COMPILED`, or `C_EXTENSION`, described below.

`imp.find_module`(*name*[, *path*])¶

Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (`C_BUILTIN`), then a frozen module (`PY_FROZEN`), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by `get_suffixes()` above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)`:

file is an open file object positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by `get_suffixes()` describing the kind of module found.

If the module does not live in a file, the returned *file* is `None`, *pathname* is the empty string, and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, `ImportError` is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is `None`, *pathname* is the package path and the last item in the *description* tuple is `PKG_DIRECTORY`.

This function does not handle hierarchical module names (names containing dots). In order to find *P*.*M*, that is, submodule *M* of package *P*, use `find_module()` and `load_module()` to find and load package *P*, and then use `find_module()` with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

`imp.load_module`(*name*, *file*, *pathname*, *description*)¶

Load a module that was previously found by `find_module()` (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it is equivalent to a `reload()`! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by `get_suffixes()`, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually `ImportError`) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try` ... `finally` statement.

`imp.new_module`(*name*)¶
Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

`imp.lock_held`()¶

Return `True` if the import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import holds an internal lock until the import is complete. This lock blocks other threads from doing an import until the original import completes, which in turn prevents other threads from seeing incomplete module objects constructed by the original thread while in the process of completing its import (and the imports, if any, triggered by that).

`imp.acquire_lock`()¶

Acquire the interpreter's import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules. On platforms without threads, this function does nothing.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

New in version 2.3.

`imp.release_lock`()¶

Release the interpreter's import lock. On platforms without threads, this function does nothing.

New in version 2.3.

The following constants with integer values, defined in this module, are used to indicate the search result of `find_module()`.

`imp.PY_SOURCE`¶

The module was found as a source file.

`imp.PY_COMPILED`¶

The module was found as a compiled code object file.

`imp.C_EXTENSION`¶

The module was found as dynamically loadable shared library.

`imp.PKG_DIRECTORY`¶

The module was found as a package directory.

`imp.C_BUILTIN`¶

The module was found as a built-in module.

`imp.PY_FROZEN`¶

The module was found as a frozen module (see `init_frozen()`).

The following constant and functions are obsolete; their functionality is available through `find_module()` or `load_module()`. They are kept around for backward compatibility:

`imp.SEARCH_ERROR`¶

Unused.

`imp.init_builtin`(*name*)¶

Initialize the built-in module called *name* and return its module object along with storing it in `sys.modules`. If the module was already initialized, it will be initialized *again*. Re-initialization involves the copying of the built-in module's `__dict__` from the cached module over the module's entry in `sys.modules`. If there is no built-in module called *name*, `None` is returned.

`imp.init_frozen`(*name*)¶

Initialize the frozen module called *name* and return its module object. If the module was already initialized, it will be initialized *again*. If there is no frozen module called *name*, `None` is returned. (Frozen modules are modules written in Python whose compiled byte-code object is incorporated into a custom-built Python interpreter by Python's **freeze** utility. See `Tools/freeze/` for now.)

`imp.is_builtin`(*name*)¶

Return `1` if there is a built-in module called *name* which can be initialized again. Return `-1` if there is a built-in module called *name* which cannot be initialized again (see `init_builtin()`). Return `0` if there is no built-in module called *name*.

`imp.is_frozen`(*name*)¶

Return `True` if there is a frozen module (see `init_frozen()`) called *name*, or `False` if there is no such module.

`imp.load_compiled`(*name*, *pathname*[, *file*])¶

Load and initialize a module implemented as a byte-compiled code file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the byte-compiled code file. The *file* argument is the byte-compiled code file, open for reading in binary mode, from the beginning. It must currently be a real file object, not a user-defined class emulating a file.

`imp.load_dynamic`(*name*, *pathname*[, *file*])¶

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized *again*. Re-initialization involves copying the `__dict__` attribute of the cached instance of the module over the value used in the module cached in `sys.modules`. The *pathname* argument must point to the shared library. The *name* argument is used to construct the name of the initialization function: an external C function called `initname()` in the shared library is called. The optional *file* argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

`imp.load_source`(*name*, *pathname*[, *file*])¶

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized *again*. The *name* argument is used to create or access a module object. The *pathname* argument points to the source file. The *file* argument is the source file, open for reading as text, from the beginning. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix `.pyc` or `.pyo`) exists, it will be used instead of parsing the given source file.

class `imp.NullImporter`(*path_string*)¶

The `NullImporter` type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises `ImportError`. Otherwise, a `NullImporter` instance is returned.

Python adds instances of this type to `sys.path_importer_cache` for any path entries that are not directories and are not handled by any other path hooks on `sys.path_hooks`. Instances have only one method:

`find_module`(*fullname*[, *path*])¶
This method always returns `None`, indicating that the requested module could not be found.

New in version 2.5.

31.1.1. Examples¶

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
   # Fast path: see if the module has already been imported.
   try:
       return sys.modules[name]
   except KeyError:
       pass

   # If any of the following calls raises an exception,
   # there's a problem we can't handle -- let the caller handle it.

   fp, pathname, description = imp.find_module(name)

   try:
       return imp.load_module(name, fp, pathname, description)
   finally:
       # Since we may exit via an exception, close fp explicitly.
       if fp:
           fp.close()
```

A more complete example that implements hierarchical module names and includes a `reload()` function can be found in the module `knee`. The `knee` module can be found in `Demo/imputil/` in the Python source distribution.

Last updated on Feb 26, 2010. Created using Sphinx 0.6.3.