:mod:`imp` --- Access the :ref:`import <importsystem>` internals

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 1**); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 1**); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 4**)

Unknown directive type "module".

```
.. module:: imp
   :synopsis: Access the implementation of the import statement.
   :deprecated:
```

Source code: :source:`Lib/imp.py`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 8**); *backlink*

Unknown interpreted text role "source".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 10**)

Unknown directive type "deprecated".

```
.. deprecated:: 3.4
   The :mod:`imp` module is deprecated in favor of :mod:`importlib`.
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 13**)

Unknown directive type "index".

```
.. index:: statement: import
```

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 17**); *backlink*

Unknown interpreted text role "keyword".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, **line 21**)

Unknown directive type "function".

```
.. function:: get_magic()

   .. index:: pair: file; byte-code

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

```
  .. deprecated:: 3.4
     Use :attr:`importlib.util.MAGIC_NUMBER` instead.
```

Unknown directive type "function".

```
.. function:: 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 :func:`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
   :const:`PY_SOURCE`, :const:`PY_COMPILED`, or :const:`C_EXTENSION`, described
   below.

   .. deprecated:: 3.3
      Use the constants defined on :mod:`importlib.machinery` instead.
```

Unknown directive type "function".

```
.. function:: 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 (:const:`C_BUILTIN`), then a frozen module (:const:`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 :func:`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 :term:`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 :func:`get_suffixes` describing the kind of
   module found.

   If the module is built-in or frozen then *file* and *pathname* are both ``None``
   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, :exc:`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 :const:`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
   :func:`find_module` and :func:`load_module` to find and load package *P*, and
   then use :func:`find_module` with the *path* argument set to ``P.__path__``.
   When *P* itself has a dotted name, apply this recipe recursively.

   .. deprecated:: 3.3
      Use :func:`importlib.util.find_spec` instead unless Python 3.3
      compatibility is required, in which case use
      :func:`importlib.find_loader`. For example usage of the former case,
      see the :ref:`importlib-examples` section of the :mod:`importlib`
      documentation.
```

Unknown directive type "function".

```
.. function:: load_module(name, file, pathname, description)

   Load a module that was previously found by :func:`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 will
   reload the module!  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 :func:`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 :exc:`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 :keyword:`try` ... :keyword:`finally` statement.

   .. deprecated:: 3.3
      If previously used in conjunction with :func:`imp.find_module` then
      consider using :func:`importlib.import_module`, otherwise use the loader
      returned by the replacement you chose for :func:`imp.find_module`. If you
      called :func:`imp.load_module` and related functions directly with file
      path arguments then use a combination of
      :func:`importlib.util.spec_from_file_location` and
      :func:`importlib.util.module_from_spec`. See the :ref:`importlib-examples`
      section of the :mod:`importlib` documentation for details of the various
      approaches.
```

Unknown directive type "function".

```
.. function:: new_module(name)

   Return a new empty module object called *name*.  This object is *not* inserted
   in ``sys.modules``.

   .. deprecated:: 3.4
      Use :func:`importlib.util.module_from_spec` instead.
```

Unknown directive type "function".

```
.. function:: 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 (the same as the *module* argument).

   When ``reload(module)`` is executed:

   * Python modules' code is recompiled and the module-level code reexecuted,
     defining a new set of objects which are bound to names in the module's
     dictionary.  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 :keyword:`try` statement it can test
    for the table's presence and skip its initialization if desired::

        try:
            cache
        except NameError:
            cache = {}

    It is legal though generally not very useful to reload built-in or dynamically
    loaded modules, except for :mod:`sys`, :mod:`__main__` and :mod:`builtins`.
    In many cases, however, 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 :keyword:`from` ...
    :keyword:`import` ..., calling :func:`reload` for the other module does not
    redefine the objects imported from it --- one way around this is to re-execute
    the :keyword:`!from` statement, another is to use :keyword:`!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.

    .. versionchanged:: 3.3
       Relies on both ``__name__`` and ``__loader__`` being defined on the module
       being reloaded instead of just ``__name__``.

    .. deprecated:: 3.4
       Use :func:`importlib.reload` instead.
```

The following functions are conveniences for handling PEP 3147 byte-compiled file paths.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, line 199)

Unknown directive type "versionadded".

```
    .. versionadded:: 3.2
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst`, line 201)

Unknown directive type "function".

```
    .. function:: cache_from_source(path, debug_override=None)

       Return the :pep:`3147` 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
       :func:`get_tag`; if :attr:`sys.implementation.cache_tag` is not defined then
       :exc:`NotImplementedError` will be raised). By passing in ``True`` or
       ``False`` for *debug_override* you can override the system's value for
       ``__debug__``, leading to optimized bytecode.

       *path* need not exist.

       .. versionchanged:: 3.3
          If :attr:`sys.implementation.cache_tag` is ``None``, then
          :exc:`NotImplementedError` is raised.

       .. deprecated:: 3.4
          Use :func:`importlib.util.cache_from_source` instead.

       .. versionchanged:: 3.5
```

```
                    The *debug_override* parameter no longer creates a ``.pyo`` file.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 225)

Unknown directive type "function".

```
    .. function:: 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` format, a :exc:`ValueError` is raised. If
       :attr:`sys.implementation.cache_tag` is not defined,
       :exc:`NotImplementedError` is raised.

       .. versionchanged:: 3.3
          Raise :exc:`NotImplementedError` when
          :attr:`sys.implementation.cache_tag` is not defined.

       .. deprecated:: 3.4
          Use :func:`importlib.util.source_from_cache` instead.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 243)

Unknown directive type "function".

```
    .. function:: get_tag()

       Return the :pep:`3147` magic tag string matching this version of Python's
       magic number, as returned by :func:`get_magic`.

       .. deprecated:: 3.4
          Use :attr:`sys.implementation.cache_tag` directly starting
          in Python 3.3.
```

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 259)

Unknown directive type "function".

```
    .. function:: lock_held()

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

       On platforms with threads, a thread executing an import first holds a
       global import lock, then sets up a per-module lock for the rest of the
       import.  This blocks other threads from importing the same module until
       the original import completes, preventing other threads from seeing
       incomplete module objects constructed by the original thread.  An
       exception is made for circular imports, which by construction have to
       expose an incomplete module object at some point.

       .. versionchanged:: 3.3
          The locking scheme has changed to per-module locks for
          the most part.  A global import lock is kept for some critical tasks,
          such as initializing the per-module locks.

       .. deprecated:: 3.4
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

```
.. function:: acquire_lock()

   Acquire the interpreter's global import lock for the current thread.
   This lock should be used by import hooks to ensure thread-safety when
   importing modules.

   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.

   .. versionchanged:: 3.3
      The locking scheme has changed to per-module locks for
      the most part.  A global import lock is kept for some critical tasks,
      such as initializing the per-module locks.

   .. deprecated:: 3.4
```

```
.. function:: release_lock()

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

   .. versionchanged:: 3.3
      The locking scheme has changed to per-module locks for
      the most part.  A global import lock is kept for some critical tasks,
      such as initializing the per-module locks.

   .. deprecated:: 3.4
```

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

```
.. data:: PY_SOURCE

   The module was found as a source file.

   .. deprecated:: 3.3
```

```
.. data:: PY_COMPILED

   The module was found as a compiled code object file.

   .. deprecated:: 3.3
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 331`)

Unknown directive type "data".

```
.. data:: C_EXTENSION

   The module was found as dynamically loadable shared library.

   .. deprecated:: 3.3
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 338`)

Unknown directive type "data".

```
.. data:: PKG_DIRECTORY

   The module was found as a package directory.

   .. deprecated:: 3.3
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 345`)

Unknown directive type "data".

```
.. data:: C_BUILTIN

   The module was found as a built-in module.

   .. deprecated:: 3.3
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 352`)

Unknown directive type "data".

```
.. data:: PY_FROZEN

   The module was found as a frozen module.

   .. deprecated:: 3.3
```

The :class:`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 :exc:`ImportError`. Otherwise, a :class:`NullImporter` instance is returned.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 361`); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 361`); *backlink*

Unknown interpreted text role "exc".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]imp.rst, line 361`); *backlink*

Unknown interpreted text role "class".

Instances have only one method:

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 :func:`find_module` has been extended and :func:`load_module` has been added in 1.4.)

```python
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()
```