

## Chapter 22 Modules: The Big Picture

**Python module**—the highest-level program organization unit, which packages program code and data for reuse, and provides selfcontained namespaces that minimize variable name clashes across your programs.

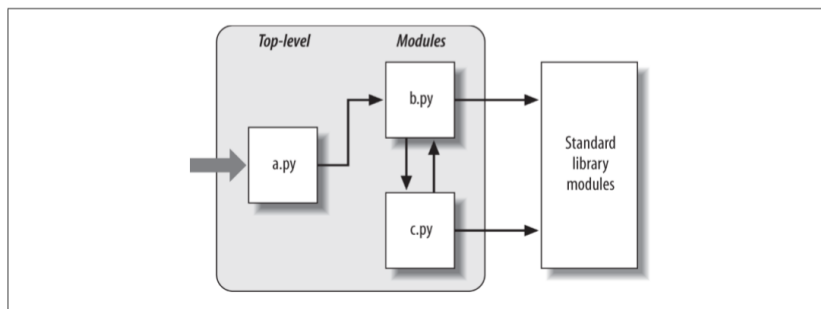
Modules are processed with two statements and one important function:

**import** Lets a client (importer) fetch a module as a whole

**from** Allows clients to fetch particular names from a module

**imp.reload** (reload in 2.X) Provides a way to reload a module's code without stopping Python

### Imports and Attributes:



For instance, suppose the file `b.py` defines a function called `spam`, for external use.

```
def spam(text):                # File b.py
    print(text, 'spam')
```

Now, suppose `a.py` wants to use `spam`.

```
import b                       # File a.py
b.spam('gumby')                # Prints "gumby spam"
```

**import** (and, as you'll see later, **from**) statements execute and load other files on request.

### Standard Library Modules

<https://docs.python.org/3/library/>

### How Imports Work

1. Find the module's file.
2. Compile it to byte code (if needed).
3. Run the module's code to build the objects it defines.

### Byte Code Files:

Byte code is stored in files in a subdirectory named `__pycache__`, which is located in the directory containing the corresponding source files.

```
c:\code\py3x> dir
10/31/2012  10:58 AM                39 script0.py
10/31/2012  11:00 AM    <DIR>          __pycache__
c:\code\py3x> dir __pycache__
10/31/2012  11:00 AM                184 script0.cpython-33.pyc
```

For Python 3.7:

```
C:\Users\wangp\Anaconda3\Scripts>dir __pycache__
Volume in drive C is Windows
Volume Serial Number is E225-D42D

Directory of C:\Users\wangp\Anaconda3\Scripts\__pycache__

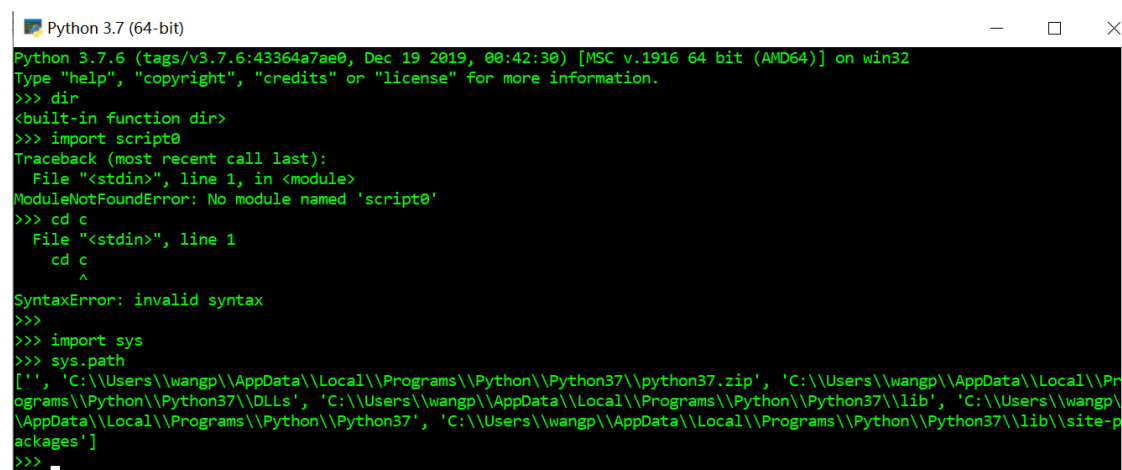
2020/02/10  15:50    <DIR>          .
2020/02/10  15:50    <DIR>          ..
2019/08/16  08:24             574 rst2html.cpython-37.pyc
2019/08/16  08:24             696 rst2html4.cpython-37.pyc
2019/08/16  08:24             700 rst2html5.cpython-37.pyc
2019/08/16  08:24             709 rst2latex.cpython-37.pyc
2019/08/16  08:24             664 rst2man.cpython-37.pyc
2019/08/16  08:24             733 rst2odt.cpython-37.pyc
2019/08/16  08:24        1,615 rst2odt_prepstyles.cpython-37.pyc
2019/08/16  08:24             580 rst2pseudoxml.cpython-37.pyc
2019/08/16  08:24             619 rst2s5.cpython-37.pyc
2019/08/16  08:24             795 rst2xetex.cpython-37.pyc
2019/08/16  08:24             582 rst2xml.cpython-37.pyc
2019/08/16  08:24             640 rstpep2html.cpython-37.pyc
2018/12/17  18:07        11,780 runxlr.cpython-37.pyc
                13 File(s)          20,687 bytes
                2 Dir(s)  115,150,872,576 bytes free
```

## The Module Search Path

1. The home directory of the program
2. PYTHONPATH directories (if set)
3. Standard library directories
4. The contents of any .pth files (if present)
5. The site-packages home of third-party extensions

## The sys.path List

**sys.path** is the module search path.



```
Python 3.7 (64-bit)
Python 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019, 00:42:30) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir
<built-in function dir>
>>> import script0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'script0'
>>> cd c
File "<stdin>", line 1
  cd c
  ^
SyntaxError: invalid syntax
>>>
>>> import sys
>>> sys.path
['', 'C:\\Users\\wangp\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip', 'C:\\Users\\wangp\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\wangp\\AppData\\Local\\Programs\\Python\\Python37\\lib', 'C:\\Users\\wangp\\AppData\\Local\\Programs\\Python\\Python37', 'C:\\Users\\wangp\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages']
>>>
```

## Module File Selection

### Module sources

For example, an import statement of the form `import b` might today load or resolve to:

- A source code file named `b.py`
- A byte code file named `b.pyc`
- An optimized byte code file named `b.pyo` (a less common format)
- A directory named `b`, for package imports (described in Chapter 24)
- A compiled extension module, coded in C, C++, or another language, and dynamically linked when imported (e.g., `b.so` on Linux, or `b.dll` or `b.pyd` on Cygwin and Windows)
- A compiled built-in module coded in C and statically linked into Python
- A ZIP file component that is automatically extracted when imported
- An in-memory image, for frozen executables
- A Java class, in the Jython version of Python
- A .NET component, in the IronPython version of Python

Saying **import b** gets whatever module `b` is, according to your module search path, and **b.attr** fetches an item in the module, be it a Python variable or a linked-in C function.

### Selection priorities

Python will always load the one found in the first (leftmost) directory of your module search path during the left-to-right search of `sys.path`

## Chapter 23 Module Coding Basics

### Module Creation

For instance, if you type the following **`def`** into a file called **`module1.py`** and **`import it`**, you create a module object with one **attribute**—the name **`printer`**

```
def printer(x):                # Module attribute
    print(x)
```

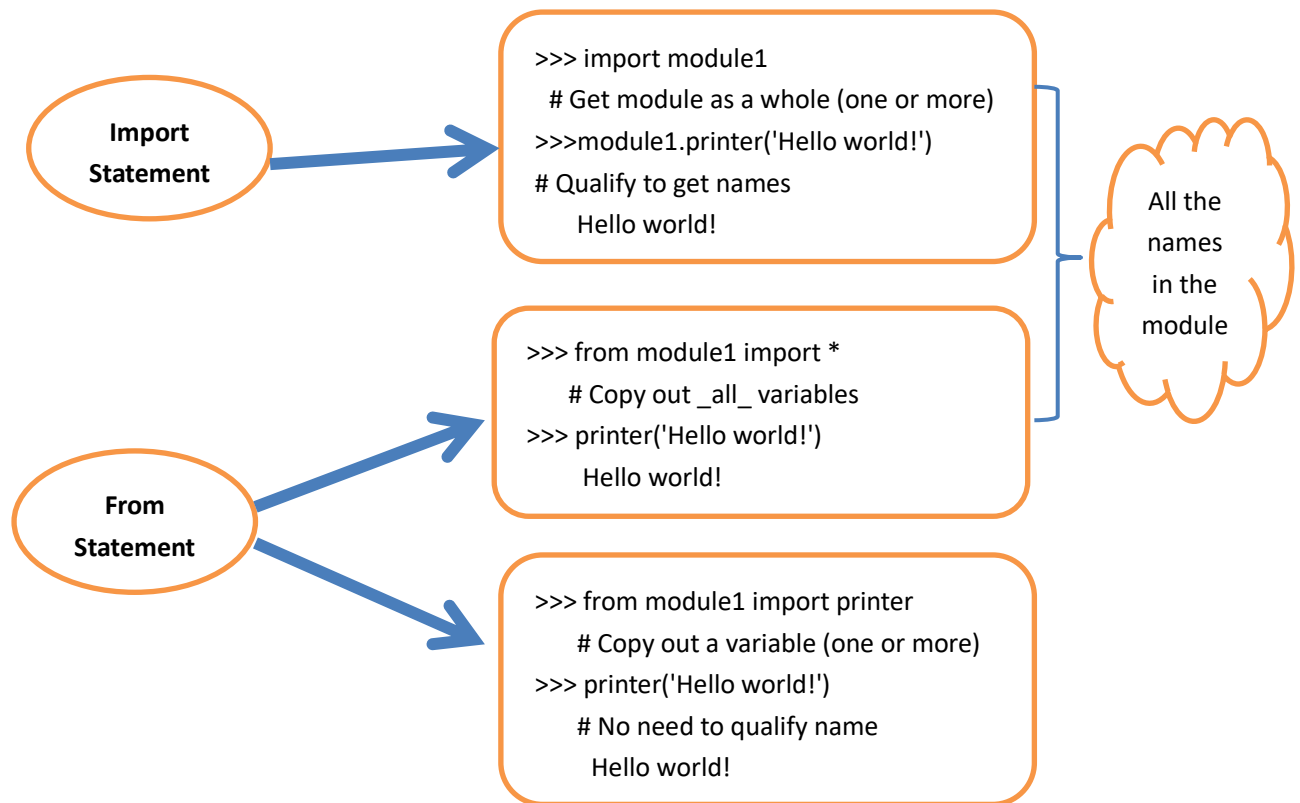
### Module Filenames

You can call modules just about anything you like, but module filenames should end in a **`.py`** suffix if you plan to import them.

They should also follow the normal variable name rules outlined in Chapter 11.

For example: `if.py`

### Module Usage



## Imports Happen Only Once

Modules are loaded and run on the first import or from, and only the first.

### Initialization code

For example: Consider the file simple.py,

```

print('hello')
spam = 1                                # Initialize variable
  
```

the print and = statements run the first time the module is imported, and the variable spam is initialized at import time:

```

% python
>>> import simple                        # First import: loads and runs file's code
hello
>>> simple.spam                          # Assignment makes an attribute
1
  
```

Second and later imports don't rerun the module's code; they just fetch the already created module object from Python's internal modules table. Thus, the variable spam is not reinitialized:

```

>>> simple.spam = 2                     # Change attribute in module
>>> import simple                       # Just fetches already loaded module
>>> simple.spam                         # Code wasn't rerun: attribute unchanged
2
  
```

## Import and from Are Assignments

Imported modules and names are not available until their associated import or from statements run.

### **Changing mutables in modules**

- **import** assigns an entire module object to a single name.
- **from** assigns one or more names to objects of the same names in another module.

consider the following file, small.py:

```
x = 1
y = [1, 2]
```

```
% python
>>> from small import x, y      # Copy two names out
>>> x = 42                      # Changes local x only
>>> y[0] = 42                   # Changes shared mutable in place

>>> import small                # Get module name (from doesn't)
>>> small.x                    # Small's x is not my x
1
>>> small.y                    # But we share a changed mutable
[42, 2]
```

### **Cross-file name changes**

To really change a global name in another file, you must use import:

```
% python
>>> from small import x, y      # Copy two names out
>>> x = 42                     # Changes my x only
>>> import small               # Get module name
>>> small.x = 42               # Changes x in other module
```

## **Import and from Equivalence**

a from statement like this one: from module

```
import name1, name2      # Copy these two names out (only)
```

is equivalent to this statement sequence:

```
import module             # Fetch the module object
name1 = module.name1      # Copy names out by assignment
name2 = module.name2
del module                # Get rid of the module name
```

## **Potential Pitfalls of the from Statement**

### **When import is required**

The only time you really must use import instead of from is when you must use the same name defined in two different modules. For example, if two files define the same name differently:

```
# M.py
```

```

def func():    ...do something...
# N.py
def func():    ...do something else...

# O.py
import M, N    # Get the whole modules, not their names
M.func()       # We can call both names now
N.func()       # The module names make them unique

```

## Module Namespaces

In simple terms, modules are just namespaces (places where names are created), and the names that live in a module are called its attributes.

### Files Generate Namespaces

The notion of module loading and scopes:

- Module statements run on the first import.
- Top-level assignments create module attributes.
- Module namespaces can be accessed via the attribute `__dict__` or `dir(M)`.
- Modules are a single scope (local is global).

Suppose we create the following module file in a text editor and call it `module2.py`:

```

print('starting to load...')
import sys
name = 42
def func(): pass
class klass: pass
print('done loading.')

```

Two print statements in this file execute at import time:

```

>>> import module2
starting to load...
done loading.

```

Once the module is loaded, its scope becomes an **attribute namespace** in the module object we get back from import. We can then access attributes in this namespace by qualifying them with the name of the enclosing module:

```

>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0x000000000222E7B8>
>>> module2.klass
<class 'module2.klass'>

```

## Namespace Dictionaries: `__dict__`

Module namespaces are stored as **dictionary objects**. Wrap this in a **list** call:

```
>>> list(module2.__dict__.keys())
['__loader__', 'func', 'klass', '__builtins__', '__doc__', '__file__', '__name__', 'name',
 '__package__', 'sys', '__initializing__', '__cached__']
```

## Attribute Name Qualification

In Python, you can access the attributes of any object that has attributes using the qualification (a.k.a. attribute fetch) syntax ***object.attribute***.

Simple variables X means search for the name X in the current scopes.

Qualification X.Y means find X in the current scopes, then search for the attribute Y in the object X (not in scopes).

Qualification paths X.Y.Z means look up the name Y in the object X, then look up Z in the object X.Y.

## Namespace Nesting

mod3.py defines a single global name and attribute by assignment:

```
X = 3
```

mod2.py in turn defines its own X, then imports mod3 and uses qualification to access the imported module's attribute:

```
X = 2
import mod3
print(X, end=' ')          # My global X
print(mod3.X)              # mod3's X
```

mod1.py also defines its own X, then imports mod2, and fetches attributes in both the first and second files:

```
X = 1
import mod2
print(X, end=' ')          # My global X
print(mod2.X, end=' ')     # mod2's X
print(mod2.mod3.X)         # Nested mod3's X
```

## Reloading Modules

- Imports (via both import and from statements) load and run a module's code only the first time the module is imported in a process.
- Later imports use the already loaded module object without reloading or rerunning the file's code.
- The reload function forces an already loaded module's code to be reloaded and rerun. Assignments in the file's new code change the existing module object in place.

## reload Basics

- reload is a function in Python, not a statement.

- reload is passed an existing module object, not a new name.
- reload lives in a module in Python 3.X and must be imported itself.

```
import module                                # Initial import
...use module.attributes...
...
...                                           # Now, go change the module file
...
from imp import reload                        # Get reload itself (in 3.X)
reload(module)                               # Get updated exports
...use module.attributes...
```

## reload Example

write a module file named changer.py with the following contents:

```
message = "First version"
def printer():
    print(message)
```

The function will print the value of the global message variable:

```
% python
>>> import changer
>>> changer.printer()
First version
```

Change the global message variable, as well as the printer function body:

```
message = "After editing"
def printer():
    print('reloaded:', message)
```

We have to call reload in order to get the new version: ...back to the Python interpreter...

```
>>> import changer
>>> changer.printer()                    # No effect: uses loaded module
First version
>>> from imp import reload
>>> reload(changer)                      # Forces new code to load/run
<module 'changer' from './changer.py'
> >>> changer.printer()                  # Runs the new version now reloaded:
After editing
```

## CHAPTER 24 Module Packages

### Package Import Basics

```
import dir1.dir2.mod
```



## from Versus import with Packages

The following three files are coded in a directory `dir1` and its subdirectory `dir2`—comments give the pathnames of these files:

```
# dir1\__init__.py
print('dir1 init')
x = 1
```

```
# dir1\dir2\__init__.py
print('dir2 init')
y = 2
```

```
# dir1\dir2\mod.py
print('in mod.py')
z = 3
```

```
C:\code> python
>>> from dir1.dir2 import mod          # Code path here only
dir1 init
dir2 init
in mod.py
>>> mod.z                              # Don't repeat path
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod        # Use shorter name (see Chapter 25)
>>> mod.z
3
>>> from dir1.dir2.mod import z as modz # Ditto if names clash (see Chapter 25)
>>> modz
3
```

## Relative Import Basics

```
from . import spam                    # Relative to this package
from .spam import name
# from a module named spam located in the same package as the file that contains this statement,
import the variable name.
import string                        # Skip this package's version
from . import string                 # Searches this package only
```

Within a module file located in a package directory named *mypkg*, the following alternative import forms work as described:

```
from .string import name1, name2
from . import string
from .. import string
```

```
# Imports names from mypkg.string
# Imports mypkg.string
# Imports string sibling of mypkg
```

## The Scope of Relative Imports

- Relative imports apply to imports within packages only.
- Relative imports apply to the from statement only.

## Relative Imports in Action

### Imports outside packages

the standard library string module:

```
C:\code> c:\Python33\python
>>> import string
>>> string
<module 'string' from 'C:\\Python33\\lib\\string.py'>
```

the current working directory (CWD):

```
# code\string.py
print('string' * 8)
C:\code> c:\Python33\python
>>> import string
stringstringstringstringstringstringstringstring
>>> string
<module 'string' from '.\\string.py'>
```

### Imports within packages

```
# code\pkg\spam.py
from . import eggs
print(eggs.X)
```

```
# code\pkg\eggs.py
X = 99999
import string
print(string)
```

```
C:\code> c:\Python33\python
>>> import pkg.spam
<module 'string' from 'C:\\Python33\\lib\\string.py'>
99999
```

### Imports are still relative to the CWD

```
# code\string.py
print('string' * 8)
```

```
# code\pkg\spam.py
```

```
from . import eggs
print(eggs.X)
```

```
# code\pkg\eggs.py
X = 99999
import string
print(string)
```

```
C:\code> c:\Python33\python      # Same result in 2.X
>>> import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from '.\string.py'>
99999
```

### **Relative imports search packages only**

```
# code\pkg\spam.py
from . import string      # <== Fails in both 2.X and 3.X if no string.py here!
```

```
C:\code> del pkg\string*
```

```
C:\code> C:\python33\python
>>> import pkg.spam
ImportError: cannot import name string
```

Modules referenced by relative imports must exist in the package directory.

### **Fix 1: Package subdirectories**

```
# code\pkg\main.py
import sub.spam          # <== Works if move modules to pkg below main file

# code\pkg\sub\spam.py
from . import eggs       # Package relative works now; in subdirectory

# code\pkg\sub\eggs.py
print('Eggs' * 4)

c:\code> python pkg\main.py  # From main script: same result in 2.X and 3.X
EggsEggsEggsEggs

c:\code> python             # From elsewhere: same result in 2.X and 3.X
>>> import pkg.sub.spam
EggsEggsEggsEggs
```

### **Fix 2: Full path absolute import**

```
# code\pkg\main.py
import spam
```

```
# code\pkg\spam.py
import pkg.eggs      # <== Full package paths work in all cases, 2.X+3.X
```

```
# code\pkg\eggs.py
print('Eggs' * 4)
```

```

c:\code> set PYTHONPATH=C:\code
c:\code> python pkg\main.py      # From main script: Same result in 2.X and 3.X
EggsEggsEggsEggs

c:\code> python                  # From elsewhere: Same result in 2.X and 3.X >>>
import pkg.spam
EggsEggsEggsEggs

```

## Python 3.3 Namespace Packages

- \* Basic module imports: `import mod`, `from mod import attr`
- \* Package imports: `import dir1.dir2.mod`, `from dir1.mod import attr`
- \* Package-relative imports: `from . import mod` (relative), `import mod` (absolute)
- \* Namespace packages: `import splitdir.mod`

## Namespace Package Semantics

### The import algorithm

1. If `directory\spam\__init__.py` is found, a regular package is imported and returned.
2. If `directory\spam.{py, pyc, or other module extension}` is found, a simple module is imported and returned.
3. If `directory\spam` is found and is a directory, it is recorded and the scan continues with the next directory in the search path.
4. If none of the above was found, the scan continues with the next directory in the search path.

### Namespace Packages in Action

the following two modules and nested directory structure—with two subdirectories named `sub` located in different parent directories, `dir1` and `dir2`:

```

C:\code\ns\dir1\sub\mod1.py
C:\code\ns\dir2\sub\mod2.py

```

composite name with normal imports:

```

c:\code> C:\Python33\python
>>> import sub
>>> sub                                     # Namespace packages: nested search paths
<module 'sub' (namespace)
> >>> sub.__path__
_NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])

>>> from sub import mod1
dir1\sub\mod1
>>> import sub.mod2                         # Content from two different directories
dir2\sub\mod2

```

```
>>> mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
```

This is also true if we import through the namespace package name immediately— because the namespace package is made when first reached, the timing of path extensions is irrelevant:

```
c:\code> C:\Python33\python
>>> import sub.mod1
dir1\sub\mod1
>>> import sub.mod2                                # One package spanning two directories
dir2\sub\mod2

>>> sub.mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'
>>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>

>>> sub
<module 'sub' (namespace)
>>>> sub.__path__
_NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])
```

## Namespace Package Nesting

Continuing the prior section's example:

```
c:\code> mkdir ns\dir2\sub\lower                    # Further nested components
c:\code> type ns\dir2\sub\lower\mod3.py
print(r'dir2\sub\lower\mod3')

c:\code> C:\Python33\python
>>> import sub.lower.mod3                            #Namespace pkg nested in namespace pkg
dir2\sub\lower\mod3

c:\code> C:\Python33\python
>>> import sub                                        # Same effect if accessed incrementally
>>> import sub.mod2
dir2\sub\mod2
>>> import sub.lower.mod3
dir2\sub\lower\mod3

>>> sub.lower                                        # A single-directory namespace pkg
<module 'sub.lower' (namespace)
>>>> sub.lower.__path__
_NamespacePath(['C:\\code\\ns\\dir2\\sub\\lower'])
```

namespace packages allow all three to be nested within them freely:

```
C:\code> mkdir ns\dir1\sub\pkg
C:\code> type ns\dir1\sub\pkg\__init__.py
print(r'dir1\sub\pkg\__init__.py')

C:\code> C:\Python33\python
>>> import sub.mod2                                # Nested module
dir2\sub\mod2
>>> import sub.pkg                                  # Nested regular package
dir1\sub\pkg\__init__.py
>>> import sub.lower.mod3                           # Nested namespace package
dir2\sub\lower\mod3

>>> sub                                              # Modules, packages, and namespaces
<module 'sub' (namespace)>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
>>> sub.pkg
<module 'sub.pkg' from 'C:\\code\\ns\\dir1\\sub\\pkg\\__init__.py'>
>>> sub.lower
<module 'sub.lower' (namespace)>
>>> sub.lower.mod3
<module 'sub.lower.mod3' from 'C:\\code\\ns\\dir2\\sub\\lower\\mod3.py'>
```

Assignment:

1. Name four file types that Python might load in response to an import operation.
2. How is the reload function related to imports?
3. What is the difference between `from mypkg import spam` and `from . import spam`?

Answer:

1. Python might load a source code (.py) file, a byte code (.pyc or .pyo) file, a C extension module (e.g., a .so file on Linux or a .dll or .pyd file on Windows), or a directory of the same name for package imports. Imports may also load more exotic things such as ZIP file components, Java classes under the Jython version of Python, .NET components under IronPython, and statically linked C extensions that have no files present at all. In fact, with import hooks, imports can load arbitrary items.
2. By default, a module is imported only once per process. The reload function forces a module to be imported again. It is mostly used to pick up new versions of a module's source code during development, and in dynamic customization scenarios.
3. In Python 3.X, `from mypkg import spam` is an absolute import—the search for mypkg skips the package directory and the module is located in an absolute directory in sys.path. A statement `from . import spam`, on the other hand, is a relative import—spam is looked up relative to the package in which this statement is contained only. In Python 2.X, the absolute import searches the package directory first before proceeding to sys.path; relative imports work as described.