Importing Python Modules

January 07, 1999 | Updated February 02, 2001 | Fredrik Lundh

Introduction

The *import* and *from-import* statements are a constant cause of serious confusion for newcomers to Python. Luckily, once you've figured out what they really do, you'll never have problems with them again.

This note tries to sort out some of the more common issues related to *import* and *from-import* and everything.

There are Many Ways to Import a Module

Python provides at least three different ways to import modules. You can use the *import* statement, the *from* statement, or the builtin *__import__* function. (There are more contrived ways to do this too, but that's outside the scope for this small note.)

Anyway, here's how these statements and functions work:

- **import X** imports the module X, and creates a reference to that module in the current namespace. Or in other words, after you've run this statement, you can use *X.name* to refer to things defined in module X.

- **from X import** * imports the module X, and creates references in the current namespace to all *public* objects defined by that module (that is, everything that doesn't have a name starting with "_"). Or in other words, after you've run this statement, you can simply use a plain *name* to refer to things defined in module X. But X itself is not defined, so *X.name* doesn't work. And if *name* was already defined, it is replaced by the new version. And if *name* in X is changed to point to some other object, your module won't notice.

- **from X import a, b, c** imports the module X, and creates references in the current namespace to the given objects. Or in other words, you can now use *a* and *b* and *c* in your program.

- Finally, **X = __import__('X')** works like **import X**, with the difference that you 1) pass the module name as a string, and 2) explicitly assign it to a variable in your current namespace.

Which Way Should I Use?

Short answer: always use **import**.

As usual, there are a number of exceptions to this rule:

- **The Module Documentation Tells You To Use from-import**. The most common example in this category is *Tkinter*, which is carefully designed to add only the widget classes and related constants to your current namespace. Using **import Tkinter** only makes your program harder to read; something that is generally a bad idea.

- **You're Importing a Package Component**. When you need a certain submodule from a package, it's often much more convenient to write **from io.drivers import zip** than **import io.drivers.zip**, since the former lets you refer to the module simply as **zip** instead of its full name. In this case, the *from-import* statement acts pretty much like a plain *import*, and there's not much risk for confusion.

- **You Don't Know the Module Name Before Execution**. In this case, use *__import__(module)* where *module* is a Python string. Also see the next item.

- **You Know Exactly What You're Doing**. If you think you do, just go ahead and use *from-import*. But think twice before you ask for help ;-)

What Does Python Do to Import a Module?

When Python imports a module, it first checks the module registry (*sys.modules*) to see if the module is already imported. If that's the case, Python uses the existing module object as is.

Otherwise, Python does something like this:

1. Create a new, empty module object (this is essentially a dictionary)
2. Insert that module object in the *sys.modules* dictionary
3. Load the module code object (if necessary, compile the module first)
4. Execute the module code object in the new module's namespace. All variables assigned by the code will be available via the module object.

This means that it's fairly cheap to import an already imported module; Python just has to look the module name up in a dictionary.

Import Gotchas

Using Modules as Scripts

If you run a module as a script (i.e. give its name to the interpreter, rather than importing it), it's loaded under the module name *__main__*.

If you then import the same module from your program, it's reloaded and reexecuted under its real name. If you're not careful, you may end up doing things twice.

Circular Imports

In Python, things like *def*, *class*, and *import* are statements too.

Modules are executed during import, and new functions and classes won't appear in the module's namespace until the *def* (or *class*) statement has been executed.

This has some interesting implications if you're doing recursive imports.

Consider a module *X* which imports module *Y* and then defines a function called *spam*:

```
# module X

import Y

def spam():
    print "function in module x"
```

If you import *X* from your main program, Python will load the code for *X* and execute it. When Python reaches the **import Y** statement, it loads the code for *Y*, and starts executing it instead.

At this time, Python has installed module objects for both *X* and *Y* in *sys.modules*. But *X* doesn't contain anything yet; the **def spam** statement hasn't been executed.

Now, if *Y* imports *X* (a recursive import), it'll get back a reference to an empty *X* module object. Any attempt to access the *X.spam* function on the module level will fail.

```
# module Y

from X import spam # doesn't work: spam isn't defined yet!
```

Note that you don't have to use from-import to get into trouble:

```
# module Y

import X

X.spam() # doesn't work either: spam isn't defined yet!
```

To fix this, either refactor your program to avoid circular imports (moving stuff to a separate module often helps), or move the imports to the end of the module (in this case, if you move **import Y** to the end of module X, everything will work just fine).