Chapter 23. Module Coding Basics

Now that we've studied the larger ideas behind modules, let's turn to some examples of modules in action. Although some of the early topics in this chapter will be review for linear readers who have already applied them in previous chapters' demos, even simple modules can quickly lead us to further details that we haven't yet encountered in full, such as nesting, reloads, scopes, and more, which we'll pick up here.

In general, Python modules are easy to *create*; they're just files of Python program code created with a text editor, and require no special syntax. Because Python does all the work of finding and loading modules, they are also easy to *use*; simply import a module or its names, and use the objects they reference. Let's explore both sides of this fence.

Creating Modules

To define a module, simply use your text editor to type Python code into a text file, and save it with a *.py* extension; any such file is automatically considered a Python module. As we've seen, all the names assigned at the top level of the module become its *attributes* (names associated with the module object) and are exported for clients to use—they morph from variable to module object attribute automatically.

For instance, if you type the code in Example 23-1 into a file called *module1.py* and import it, you create a module object with one attribute—the name `printer`, which happens to be a reference to a function object.

Example 23-1. module1.py

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

Again, this code may seem simplistic if you read the content of this book that precedes this chapter, but our goal here is to strip out the extraneous bits so we

can study modules in isolation.

Module Filenames

Before we go on, let's get more formal about 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 as modules. The *.py* is technically optional for top-level files that will be run but not imported (i.e., for *scripts*), but adding it in all cases makes your files' types more obvious, may enable Python-specific features in some text editors and file explorers, and allows you to import any of your files in the future (recall that this is one way to run a file).

Because module names become variable names inside a Python program (without the *.py*), they should also follow the normal variable name rules outlined in Chapter 11. For instance, you can create a module file named *if.py*, but you cannot import it because `if` is a reserved word—when you try to run `import if`, you'll get a syntax error. In fact, both the names of module *files* and the names of *directories* used in package imports (discussed in the next chapter) must conform to the rules for variable names presented in Chapter 11; they may, for example, contain only letters, digits, and underscores. Package directories also cannot contain platform-specific syntax such as spaces in their names.

When a module is imported, Python maps the module name to an external filename by adding a directory path from the module search path of the last chapter to the front, and a *.py* or other extension at the end. For instance, a module named `M` ultimately maps to an external file `directory/M.extension` that contains the module's code.

Other Kinds of Modules

As mentioned in the preceding chapter, it is also possible to create a Python module by writing code in an external language such as C, C++, and others (e.g., Java, in the Jython implementation of the language). Such modules are called *extension modules*, and they are generally used to wrap up external libraries for use in Python scripts or optimize parts of a program. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with import

statements, and they provide functions and objects as module attributes. They're also beyond the scope of this book; see Python's standard manuals for more details.

Using Modules

On the other side of the fence, clients can *use* the simple module file we just wrote by running an `import` or `from` statement. Both statements were introduced in Chapter 3, and have been used in earlier examples. They both find, compile, and run a module file's code if it hasn't yet been loaded, per the process covered in the prior chapter. The chief difference is that `import` fetches the module as a *whole*, so you must qualify to fetch its names, whereas `from` fetches (really, copies) specific *names* out of the module.

Let's see what this means in terms of code. All of the following examples wind up calling the `printer` function defined in module file *module1.py* of Example 23-1 but in different ways.

The import Statement

In the first example that follows, the name `module1` serves two different purposes—it identifies an external file to be loaded, and it becomes a variable in the script, which references the module object after the import. In a REPL:

```
>>> import module1                      # Get module
>>> module1.printer('Hello world!')     # Qualify to
Hello world!
```

The `import` statement simply lists one or more names of modules to load, separated by commas. Because it gives a name that refers to the *whole module* object, we must go through the module name to fetch its attributes (e.g., `module1.printer`).

The from Statement

By contrast, because `from` copies *specific names* from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., `printer`):

```
>>> from module1 import printer            # Copy out c
>>> printer('Hello world!')                 # No need to
Hello world!
```

This form of `from` allows us to list one or more names to be copied out, separated by commas. Here, it has the same effect as the prior example, but because the imported name is copied into the scope where the `from` statement appears, using that name in the script requires less typing—we can use it directly instead of naming the enclosing module. In fact, we must; `from` doesn't assign the name of the module itself, so `module1` is undefined.

As you'll see in more detail later, the `from` statement is really just a minor extension to the `import` statement—it imports the module file as usual (running the full three-step procedure of the preceding chapter), but adds an extra step that copies one or more *names* (not objects) out of the file. The entire file is loaded, but you're given names for more direct access to its parts.

The from * Statement

Finally, the next example uses a special form of `from` : when we use a `*` instead of specific names, we get copies of *all names* assigned at the top level of the referenced module. Here again, we can then use the copied name `printer` in our script without going through the module name:

```
>>> from module1 import *                  # Copy out
>>> printer('Hello world!')                # Use names
Hello world!
```

Technically, both `import` and `from` statements invoke the same import operation; the `from *` form simply adds an extra step that copies all the names in the module into the importing scope. It essentially *merges* one module's namespace into another, which again means less typing for us, albeit at the expense of name segregation.

Note that only a single `*` works in this context; you can't use arbitrary pattern matching to select a subset of names (you could in principle by looping through a module's `__dict__` discussed ahead, but it's substantially

more work). Also, note the `from *` may not really get "all" names in the module if that module uses special tricks like `_X` naming or an `__all__` list to hide some of its names from this statement, but we'll defer to Chapter 25 for more on such tools.

And that's it—apart from the search-path configurations of the prior chapter, modules really are simple to use. To give you a better understanding of what really happens when you define and use modules, though, let's move on to look at some of their properties in more detail.

NOTE

OK, there is a special case: The `*` form of the `from` statement form described here can be used *only* at the top level of a module file, not within a function (and generates a syntax error there), because this would make it impossible for Python to detect local variables before the function runs. It's rare to see either `import` or `from` inside a function anyhow, and best practice recommends listing all your imports at the top of a module file; it's not required, but makes them easier to spot, and avoids a speed penalty for re-importing modules on every call to a function. The `from *` has other issues enumerated ahead, and may be best limited to one per file, or interactive REPLs.

Imports Happen Only Once

One of the most common questions people seem to ask when they start using modules is, "Why won't my imports keep working?" They often report that the first import works fine, but later imports during an interactive session (or program run) seem to have no effect. In fact, this is by design—modules are loaded and run on the first `import` or `from`, and only the first. Because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

Initialization code

As one consequence, because top-level code in a module file is usually executed only once, you can use it to initialize names once, but allow their state to change. As a demo, consider the file *init.py* in Example 23-2.

Example 23-2. init.py

```
print('hello')
flag = 1                    # Initialize variable - just
```

In this example, the `print` and `=` statements run the first time the module is imported, and the variable `flag` is initialized at import time (recall from Chapter 17 that the REPL works like another module file here):

```
$ python3
>>> import init             # First import: loads and ru
hello
>>> init.flag               # Assignment makes an attrib
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 `flag` is not reinitialized:

```
>>> init.flag = 2           # Change attribute in module
>>> import init             # Just fetches already loade
>>> init.flag               # Code wasn't rerun: attribu
2
```

Of course, sometimes you really *want* a module's code to be rerun on a subsequent import. You'll see how to do this with Python's `reload` function later in this chapter.

Imports Are Runtime Assignments

Just like `def`, `import` and `from` are *executable statements*, not compile-time declarations. They may be nested in `if` tests, to select among module options; appear in function `def` s, to be loaded only on calls (subject to the preceding note); be used in `try` statements, to provide defaults on errors; and so on. As an abstract example:

```
if sometest:
    from moduleA import name
else:
    from moduleB import name
```

Wherever they appear, imports are not resolved or run until Python reaches them while executing your program. As one upshot, imported modules and names are not available until their associated `import` or `from` statements run.

Changing mutables in modules

Also like `def`, `import` and `from` are *implicit assignments*:

- `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.

Hence, everything you've already learned about assignment applies to module access, too. For instance, names copied with a `from` become references to shared objects; much like function arguments, reassigning a copied name has no effect on the module from which it was copied, but changing a shared *mutable object* through a copied name can also change it in the module from which it was imported. Consider the following file, *share.py*, in Example 23-3.

Example 23-3. share.py

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

When importing with `from`, we copy *names* to the importer's scope that initially share *objects* referenced by the module's names (again, the REPL serves as the importing module here):

```
$ python3
>>> from share import x, y          # Copy two names out
>>> x = 'hack'                      # Changes local x or
>>> y[0] = 'hack'                   # Changes shared mut
```

```
>>> x, y                          # This module's x ar
('hack', ['hack', 2])
```

Here, `x` is not a shared mutable object, but `y` is. The names `y` in the importer and the imported modules both reference the same list object, so changing it from one place changes it in the other; continuing the REPL session:

```
>>> import share                  # Get module name (j
>>> share.x                       # share's x is not m
1
>>> share.y                       # But we share a cha
['hack', 2]
```

For more background on this, see Chapter 6. And for a graphical picture of what `from` assignments do with references, flip back to Figure 18-1 (function argument passing), and mentally replace "caller" and "function" with "imported" and "importer." The effect is the same, except that here we're dealing with names in modules, not functions. Assignment works the same everywhere in Python.

Cross-file name changes

In the preceding example, the assignment to `x` in the interactive session changed the name `x` in that scope only, not the `x` in the file—there is no link from a name copied with `from` back to the file it came from. To really change a global name in another file, you must use `import`:

```
$ python3
>>> from share import x, y        # Copy two names out
>>> x = 23                        # Changes my x only

>>> import share                  # Get module name
>>> share.x = 23                  # Changes x in other
```

This phenomenon was introduced in Chapter 17. Because changing variables in other modules like this is a common source of confusion (and often a subpar design choice), we'll revisit this technique again later in this part of the book. Subtly, the change to `y[0]` in the prior session is different; it changes

an *object*, not a name, and the *name* in both modules references the same, changed object. This can be similarly subpar unless all module clients expect it.

import and from Equivalence

Notice in the prior example that we have to execute an `import` statement after the `from` to access the `share` module name at all. `from` only copies names from one module to another; it does not assign the module name itself. It may help to remember that, at least conceptually, a `from` statement like this one:

```
from module import name1, name2     # Copy these two nc
```

is equivalent to this statement sequence:

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

Like all assignments, the `from` statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the *names* are copied out, though, not the objects they reference, and not the name of the module itself. When we use the `from *` form of this statement (`from module import *`), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Importantly, the first step of the `from` runs a normal `import` operation, with all the semantics outlined in the preceding chapter. As a result, the `from` always imports the *entire* module into memory if it has not yet been imported, regardless of how many names it copies out of the file. There is no way to load just part of a module file (e.g., just one function), but because modules are bytecode in standard Python instead of machine code, the performance implications are generally negligible.

Potential Pitfalls of the from Statement

One downside of the `from` statement is that it makes the meaning of a variable more obscure: `name` is less useful to the reader than `module.name`, and may require a search for the `from` that loaded it. Because of this, some Python users recommend coding `import` instead of `from` most of the time. Like most advice, though, this doesn't always make sense. `from` is commonly and widely used, without dire consequences. Moreover, it's convenient not to have to type a module's name every time you wish to use one of its tools, especially when working in the REPL.

It is true that the `from` statement has the potential to corrupt namespaces, at least in principle—if you use it to import variables that happen to have the same names as existing variables in your scope, your variables will be silently overwritten. This problem doesn't occur with the `import` statement because you must always go through a module's name to get to its contents: `module.attr` will not clash with a variable named `attr` in your scope. As long as you understand that this can happen when using `from`, though, this isn't a concern in practice: assigning a variable with `from` has the same effect as any other assignment in your code.

On the other hand, the `from` statement has legitimate issues when used in conjunction with the `reload` call, as imported names might reference prior versions of objects. Moreover, the `from *` form really *can* trash namespaces and make code difficult to understand, especially when applied to more than one file—in this case, there is no way to tell which module a name came from, short of searching external files. In effect, the `from *` form collapses one namespace into another, and so defeats the namespace partitioning purpose of modules. We'll save demos of these issues for "Module Gotchas" (at the end of this part of the book), and meet tools that can minimize the `from *` damage with data hiding in Chapter 25.

Probably the best real-world advice here is to generally prefer `import` to `from` for simple modules; to explicitly list the variables you want in most `from` statements; and to limit the `from *` form to just one per file, or the REPL's throw-away code. That way, any names not called out by attribute qualification or `from` lists can be assumed to live in the sole module imported by the `from *`. Some care is required when using the `from` statement, but armed with a little knowledge, most programmers find it to be a convenient way to access modules.

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, imagine that two files define the same name differently, as in this abstract snippet:

```
# M.py
def func():
    …do something…

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

If you must use *both* versions of this name in your program, the `from` statement will fail—you can have only one assignment to the name in your scope:

```
# O.py
from M import func
from N import func              # This overwrites the or
func()                          # Calls N.func only!
```

An `import` will work here, though, because including the name of the enclosing module makes the two names unique:

```
# O.py
import M, N                     # Get the whole modules,
M.func()                        # We can call both names
N.func()                        # The module names make
```

This case is unusual enough that you're unlikely to encounter it very often in practice. If you do, though, `import` allows you to avoid the name collision. Another way out of this dilemma is using the `as` extension, a renaming tool that we'll cover fully in Chapter 25 but is simple enough to introduce here:

```
# O.py
from M import func as mfunc     # Rename uniquely with '
```

```
from N import func as nfunc
mfunc(); nfunc()                # Call one or the other
```

The `as` extension works in both `import` and `from` as a simple renaming tool (it can also be used to give a shorter synonym for a long module name in `import`); more on this form in Chapters 24 and 25.

Module Namespaces

Modules are probably best understood as packages of names—i.e., places to define names you want to make visible to the rest of a system. Technically, modules usually correspond to files, and Python creates a module object to contain all the names assigned in a module file. But in simple terms, modules are just namespaces (places where names are created), and the names that live in a module are its attributes. This section expands on the details behind this model.

How Files Generate Namespaces

We've seen that files *morph* into namespaces, but how does this actually happen? The short answer is that every name that is assigned a value at the top level of a module file (i.e., not nested in a function or class body) becomes an attribute of that module.

For instance, given an assignment statement such as `X = 1` at the top level of a module file *M.py*, the name `X` becomes an attribute of `M`, which we can refer to from outside the module as `M.X`. The name `X` also becomes a global variable to other code inside *M.py*, but we need to firm up the relationship of module loading and scopes to understand why:

- **Module statements run on the first import.** The first time a module is imported anywhere in a system, Python creates an empty module object and executes the statements in the module file one after another, from the top of the file to the bottom.
- **Top-level assignments create module attributes.** During an import, statements at the top level of the file not nested in a `def` or `class` that assign names (e.g., `=`, `def`) create attributes of the module object. Names assigned by these statements are stored in the module's namespace.
- **Module namespaces are dictionaries.** Module namespaces created by imports may be accessed through the built-in `__dict__` dictionary

attribute of module objects, and may be inspected with the `dir` function. `dir` is the same as the sorted keys of `__dict__` for modules, but includes inherited names for classes.

- **Modules are a single scope (local is global).** As we saw in Chapter 17, names at the top level of a module follow the same scope rules as names in a function, but the local and global scopes are the same—or, more formally, they follow the *LEGB* scope rule presented in Chapter 17, but without the *L* and *E* lookup layers.
- **Module scopes outlive imports.** A module's global scope becomes an attribute dictionary of a module object after the module has been loaded. Unlike function scopes, where the local namespace exists only while a function call runs, a module file's scope lives on after the import, providing a source of tools to importers.

Here's a demonstration of these ideas. Suppose we create the module file in Example 23-4 with a text editor, and call it *spaces.py*.

Example 23-4. spaces.py

```
print('starting to load...')
import sys
var = 23

def func(): pass

class klass: pass

print('done loading.')
```

The first time this module is imported (or run as a program), Python executes its statements from top to bottom. Some statements create names in the module's namespace as a side effect, but others do actual work while the import is going on. For instance, the two `print` statements in this file execute at import time:

```
>>> import spaces
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:

```
>>> spaces.sys
<module 'sys' (built-in)>
>>> spaces.var
23
>>> spaces.func
<function func at 0x101ce7b00>
>>> spaces.klass
<class 'spaces.klass'>
```

Here, `sys`, `var`, `func`, and `klass` were all assigned while the module's statements were being run, so they are attributes after the import. We'll study classes in Part VI, but notice the `sys` attribute— `import` statements *assign* module objects to names, and any type of assignment to a name at the top level of a file generates a module attribute.

Namespace Dictionaries: __dict__

In fact, internally, module namespaces are stored as *dictionary* objects. These are just normal dictionaries with all the usual methods. When needed, we can access a module's namespace dictionary through the module's `__dict__` attribute—for instance, to write tools that list module content generically, as we will in Chapter 25. Continuing the prior section's example:

```
>>> spaces.__dict__.keys()
dict_keys(['__name__', '__doc__', '__package__', '__loa
'__file__', '__cached__', '__builtins__', 'sys', 'var',
```

The names we assigned in the module file become dictionary keys internally, so some of the keys here reflect top-level assignments in our file. The value of `var`, for example, can be had two ways (though the first is normal):

```
>>> spaces.var, spaces.__dict__['var']
(23, 23)
```

Python also adds some useful names in the module's namespace. For instance, `__file__` gives the name of the file from which the module was loaded (useful to find resources bundled with code), and `__name__` gives its name as known to importers (without the *.py* extension and directory path):

```
>>> spaces.__name__, spaces.__file__
('spaces', '/Users/me/…/LP6E/Chapter23/spaces.py')
```

To see just the names your code assigns, filter out double-underscore names as we did in Chapter 15's `dir` coverage and Chapter 17's built-in scope coverage, but this time with a generator expression. We can also omit `keys` because dictionaries generate keys automatically, and the `dir` built-in for modules is just the sorted keys list of `__dict__`:

```
>>> list(name for name in spaces.__dict__ if not name.s
['sys', 'var', 'func', 'klass']
>>> dir(spaces) == sorted(spaces.__dict__)
True
```

As another lesser-used alternative, Python's `vars` built-in function simply fetches the `__dict__` of a module passed to it (an option, perhaps, after you've written enough Python code to wear out your keyboard's underscore key?):

```
>>> vars(spaces) == spaces.__dict__
True
>>> dir(spaces) == sorted(vars(spaces))
True
```

See Chapter 7 for other `vars` roles. You'll see similar `__dict__` dictionaries on *class*-related objects in Part VI too. In all cases, attribute fetch is similar to dictionary indexing, though only the former kicks off *inheritance* in classes.

Attribute Name Qualification

Speaking of attribute fetch, now that you're becoming more familiar with modules, we should firm up the notion of name qualification more formally

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

Qualification is really an expression that returns the value assigned to an attribute name associated with an object. For example, the expression `spaces.sys` in the previous example fetches the value assigned to `sys` in `spaces`. Similarly, if we have a built-in list object `L`, `L.append` returns the `append` method object associated with that list.

It's important to keep in mind that attribute qualification has nothing to do with the *scope* rules we studied in Chapter 17; it's an independent concept. When you use qualification to access names, you give Python an explicit object from which to fetch the specified names. The LEGB scope rule applies only to bare, unqualified names; it may be used for the leftmost name in a qualification path, but later names after dots search specific objects instead.

This distinction may seem blurred by the fact that module scopes morph into attributes at the end of an import, but thereafter, names are part of a module *object*. For reference, here are the full rules for name resolution in Python:

Simple variables

`X` means search for the name `X` in the current scopes (following the LEGB rule of Chapter 17).

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`.

Generality

Qualification works on all objects with attributes: modules, classes, C extension types, etc.

As noted earlier, in Part VI you'll see that attribute qualification means a bit more for classes—it's also the place where inheritance happens. In general, though, the rules outlined here apply to all names in Python.

Imports Versus Scopes

As we've seen, it is never possible to access names defined in another module file without first importing that file. That is, you never automatically get to see names in another file, regardless of the structure of imports or function calls in your program. A variable's meaning is always determined by the locations of assignments in your *source code*, and attributes are always requested of an object explicitly.

For example, consider the following two simple modules. The first, *lex1.py* in Example 23-5, defines a variable `X` global to code in its file only, along with a function that changes the global `X` in this file.

Example 23-5. lex1.py

```
X = 88                          # My X: global to this fi

def f():
    global X                    # Change this file's X
    X = 99                      # Cannot see names in oth
```

The second module, *lex2.py* in Example 23-6, defines its own global variable `X` and imports and calls the function in the first module.

Example 23-6. lex2.py

```
X = 11                          # My X: global to this fi

import lex1                     # Gain access to names ir
lex1.f()                        # Sets lex1.X, not this f
print(X, lex1.X)
```

When run, `lex1.f` changes the `X` in `lex1`, not the `X` in `lex2`. The global scope for `lex1.f` is always the file *enclosing* it, regardless of which module it is ultimately called from:

```
$ python3 lex2.py
11 99
```

In other words, import operations never give upward visibility to code in imported files—an imported file cannot see names in the *importing* file. More formally:

- Functions can never see names in other functions unless they are physically enclosing.
- Module code can never see names in other modules unless they are explicitly imported.

Such behavior is part of the *lexical scoping* notion—in Python, the scopes surrounding a piece of code are completely determined by the code's physical position in your file. Scopes are never influenced by function calls or module imports. Some languages act differently and provide for *dynamic scoping*, where scopes really may depend on runtime calls. This tends to make code trickier, though, because the meaning of a variable can differ over time. In Python, scopes more simply correspond to the text of your program.

Namespace Nesting

Finally, although imports do not nest namespaces upward, they do in some sense nest downward. That is, although an imported module never has direct access to names in a file that imports it, using attribute qualification paths it is possible to descend into arbitrarily nested modules and access their attributes. For example, consider the next three files. First, *nest3.py* of Example 23-7 defines a single global name and attribute by assignment.

Example 23-7. nest3.py

```
X = 3
```

Next, *nest2.py* of Example 23-8 defines its own `X`, then imports `nest3` and uses name qualification to access the imported module's attribute.

Example 23-8. nest2.py

```
X = 2
import nest3

print(X, end=' ')              # My global X
print(nest3.X)                 # nest3's X
```

And at the top, *nest1.py* of Example 23-9 also defines its own `X`, then imports `nest2`, and fetches attributes in both the first and second files.

Example 23-9. nest1.py

```
X = 1
import nest2

print(X, end=' ')              # My global X
print(nest2.X, end=' ')        # nest2's X
print(nest2.nest3.X)           # Nested nest3's X
```

Really, when `nest1` imports `nest2` here, it sets up a two-level namespace nesting. By using the path of names `nest2.nest3.X`, it can descend into `nest3`, which is nested in the imported `nest2`. The net effect is that `nest1` can see the `X`s in all three files, and hence has access to all three global scopes:

```
$ python3 nest1.py
2 3
1 2 3
```

The reverse, however, is not true: `nest3` cannot see names in `nest2`, and `nest2` cannot see names in `nest1`. This example may be easier to grasp if you don't think in terms of namespaces and scopes, but instead focus on the objects involved. Within `nest1`, `nest2` is just a name that refers to an object with attributes, some of which may refer to other objects with attributes (`import` is an assignment). For paths like `nest2.nest3.X`, Python simply evaluates from left to right, fetching attributes from objects along the way.

Note that `nest1` can say `import nest2`, and then `nest2.nest3.X`, but it cannot say `import nest2.nest3`—this syntax invokes something called *package* (directory) imports, the topic we'll take up in the next chapter after we study reloads here. Package imports also create module namespace nesting, but their `import` statements are taken to reflect *directory* trees, not simple file *import* chains.

Reloading Modules

As we've seen, a module's code is run only once per process by default. To force a module's code to be reloaded and rerun, you need to ask Python to do so explicitly by calling the `reload` built-in function, introduced briefly in Chapter 3. In this section, we'll explore how to use reloads to make your systems more dynamic. In a nutshell:

- 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. This is true even if you resave the module's source code file during the program run.
- 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.

So why care about reloading modules? In short, REPL testing and customization. When testing code interactively, it may be easier to reload a module you've changed in another window than it is to restart the REPL.

The more grandiose rationale is *dynamic customization*: the `reload` function allows parts of a program to be changed without stopping the whole program. With `reload`, the effects of changes in components can be observed immediately. Reloading doesn't help in every situation, but where it does, it makes for a much shorter development cycle. For instance, imagine a database program that must connect to a server on startup; because program changes or customizations can be tested immediately after reloads, you need to connect only once while debugging. Long-running servers can update themselves this way, too.

Because Python is interpreted (more or less), it already gets rid of the compile/link steps you need to go through to get a C program to run: modules are loaded dynamically when imported by a running program. Reloading offers a further performance advantage by allowing you to also change parts of running programs without stopping.

One note here: our use of `reload` in this book is limited to modules written in Python. Compiled extension modules coded in a language such as C can be

dynamically loaded by imports at runtime, too, but they are out of scope here (though most users probably prefer to code customizations in Python anyhow!).

reload Basics

Unlike `import` and `from`:

- `reload` is a function in Python, not a statement.
- `reload` is passed an existing module object, not a string name.
- `reload` lives in a standard-library module and must be imported itself.

Because `reload` expects an object, a module must have been previously imported successfully before you can reload it (and if the import was unsuccessful due to a syntax or other error, you may need to repeat it before you can reload the module). Furthermore, the syntax of `import` statements and `reload` calls differs: as a function, reloads require parentheses, but import statements do not. Abstractly, reloading looks like this:

```
import module                       # Initial module impo
…use module.attributes…
…change the module file…

from importlib import reload        # Get reload itself
reload(module)                      # Get updated module
…use module.attributes…
```

The typical usage pattern is that you import a module, then change its source code in a text editor, and then reload it. This can occur when working interactively, but also in larger programs that reload periodically.

When you call `reload`, Python rereads the module's code and reruns its top-level statements. Perhaps the most important thing to know about `reload` is that it changes a module object *in place*; it does not delete and re-create the module object. Because of that, every reference to an entire module *object* anywhere in your program is automatically affected by a reload. Here are the details:

- **`reload` runs a module file's new code in the module's current namespace.** Rerunning a module file's code overwrites its existing

namespace, rather than deleting and re-creating it.

- **Top-level assignments in the file replace names with new values.** For instance, rerunning a `def` statement replaces the prior version of the function in the module's namespace by reassigning the function name.
- **Reloads impact all clients that use `import` to fetch modules.** Because clients that use `import` qualify to fetch attributes, they'll find new values in the module object after a reload.
- **Reloads impact future `from` clients only.** Clients that used `from` to fetch attributes in the past won't be affected by a reload; they'll still have references to the old objects fetched before the reload.
- **Reloads apply to a single module only.** You must run them on each module you wish to update unless you use code or tools that apply reloads transitively.

reload Example

To demonstrate, here's a more concrete example of `reload` in action. In the following, we'll change and reload a module file without stopping the interactive Python session. Reloads are useful in other scenarios, too, but we'll keep things simple for illustration here. First, in the text editor of your choice, write a module file named *changer.py* with the contents given in Example 23-10.

Example 23-10. changer.py (start)

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

This module creates and exports two names—one bound to a string, and another to a function. Now, start the Python interpreter (i.e., your local REPL), import the module, and call the function it exports. The function will print the value of the global `message` variable as you probably expect:

```
$ python3
>>> import changer
>>> changer.printer()
First version
```

Keeping the interpreter active, now edit and save the module file in another window. Change the global `message` variable, as well as the `printer` function body:

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

Then, return to the Python window and reload the module to fetch the new code. Notice in the following interaction that importing the module again has no effect; we get the original message, even though the file's been changed. We have to call `reload` in order to get the new version:

```
>>> import changer
>>> changer.printer()                    # No effect: uses
First version

>>> from importlib import reload
>>> reload(changer)                      # Forces new code
<module 'changer' from '/…/LP6E/Chapter23/changer.py'>

>>> changer.printer()                    # Runs the new ve
reloaded: After editing
```

Notice that `reload` actually *returns* the module object for us—its result is usually ignored, but because expression results are printed at the interactive prompt, Python shows a default `<module …>` representation.

You can also reload a module by string name using the `sys.modules` dictionary demoed in the prior chapter if you don't have a variable assigned to it by `import`:

```
>>> import sys
>>> reload(sys.modules['changer'])
```

In this case, it's probably easier to run `import changer` to get a handle on the module, but the `sys.modules` scheme may be useful in programs that need to reload modules more generically. It's also possible to *delete* modules

from `sys.modules` to force a reload, but this is generally discouraged for reasons we'll skip here; use `reload`.

reload Odds and Ends

Finally, four brief module-reload notes in closing:

- If you use `reload`, you'll probably want to pair it with `import` instead of `from`, as names fetched with the latter are not updated by reload operations—leaving your names in a state that's strange enough to warrant postponing elaboration until this part's "gotchas" at the end of Chapter 25.
- By itself, `reload` updates only a *single* module, but it's straightforward to code a function that applies it transitively to related modules—an extension we'll save for a case study near the end of Chapter 25.
- Some development tools (e.g. Jupyter notebooks) have REPLs that offer an *auto-reload* mode that may obviate manual `reload` calls. This works only in specific tools, though, and doesn't address customization roles.
- And last, `reload` has had a long and checkered past—morphing from built-in function, to `imp` module attribute in this book's prior edition, to its current `importlib` host. While this may be a symptom of Python's constant morph, which is apt to relocate `reload` again, it must be asked: does this function have trouble working with others?!

Chapter Summary

This chapter drilled down into the essentials of module coding tools—the `import` and `from` statements, and the `reload` call. It showed how the `from` statement simply adds an extra step that copies names out of a file after it has been imported, and how `reload` forces a file to be imported again without stopping and restarting Python. This chapter also detailed what happens when imports are nested, explored the way files become module namespaces, and covered some potential pitfalls of the `from` statement.

Although you've already learned enough to handle module files in most programs, the next chapter extends the coverage of the import model by presenting *package imports*—a way for `import` statements to specify part of the directory path leading to the desired module. As you'll find, package imports give us a hierarchy that is useful in larger systems and allow us to

break conflicts between same-named modules. Before we move on, though, here's a quick quiz on the concepts presented here.

Test Your Knowledge: Quiz

1. How do you make a module?
2. How is the `from` statement related to the `import` statement?
3. How is the `reload` function related to imports?
4. When must you use `import` instead of `from`?
5. Name three potential pitfalls of the `from` statement.

Test Your Knowledge: Answers

1. To create a module, you simply write a text file containing Python statements; every source code file is automatically a module, and there is no syntax for declaring one. Import operations load module files into module objects in memory. You can also make a module by writing code in an external language like C or Java, but such extension modules are beyond the scope of this book (and most of its readers).
2. The `from` statement imports an entire module, like the `import` statement, but as an extra step, it also copies one or more variables from the imported module into the scope where the `from` appears. This enables you to use the imported names directly (`name`) instead of having to go through the module (`module.name`).
3. 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 where users change part of a system without restarting it.
4. You must use `import` instead of `from` only when you need to access the same name in two different modules. To make the two names unique, qualify with the names of their enclosing modules obtained with `import`. The `as` extension can render `from` usable in this context as well, by renaming imports uniquely.
5. The `from` statement can obscure the meaning of a variable (which module it is defined in), can have problems with the `reload` call (names may reference prior versions of objects), and can corrupt namespaces (it

might silently overwrite names you are using in your scope). The `from *` form is worse in most regards—it can corrupt namespaces arbitrarily and obscure the meaning of variables, so it is probably best used sparingly.