Chapter 25. Module Odds and Ends

This chapter concludes this part of the book with an assortment of module-related topics—data hiding, the `__future__` module, the `__name__` variable, name-string imports, the `__gettattr__` hook, transitive reloads, and more—along with the usual set of gotchas and exercises related to what we've covered in this part of the book. Along the way, we'll build some useful tools that combine functions and modules. Like functions, modules are more effective when their interfaces are well-defined, so this chapter also briefly reviews module design concepts.

Though some coverage here might qualify as advanced and optional, this is mostly a miscellany of additional module subjects. Because some of the topics discussed here are very widely used—especially the `__name__` dual-mode trick—be sure to browse here before moving on to classes in the next part of the book.

Module Design Concepts

First up, some perspective. Like functions, modules present design trade-offs: you have to think about which functions go in which modules, module communication mechanisms, and so on. All of this will become clearer when you start writing bigger Python systems, but here are a few general ideas to keep in mind:

- **You're always in a module in Python.** There's no way to write code that doesn't live in some module. As mentioned briefly in Chapters 17 and 21, even code typed at the interactive prompt (a.k.a. REPL) really goes in a built-in module called `__main__`; the only unique things about the interactive prompt are that code runs and is discarded immediately, and expression results are printed automatically.
- **Minimize module coupling: global variables.** Like functions, modules work best if they're written to be mostly closed boxes. As a rule of thumb, they should be as independent of global variables used within other modules as possible, except for functions and classes imported from them.

The only things a module should share with the outside world are the tools it uses, and the tools it defines.

- **Maximize module cohesion: unified purpose.** Also like functions, you can minimize a module's couplings by maximizing its cohesion. If all the components of a module share a general purpose, they're less likely to depend on external names.
- **Modules should rarely change other modules' variables.** We illustrated this with code in Chapter 17, but it's worth repeating here: it's perfectly OK to use globals defined in another module (that's how clients import services, after all), but *changing* globals in another module is often a symptom of a design problem. There are exceptions, of course, but you should try to communicate results through devices such as function arguments and return values, not cross-module changes. Otherwise, your globals' values become dependent on the order of arbitrarily remote assignments in other files, and your modules become harder to understand and reuse.

As a summary, Figure 25-1 sketches the environment in which modules operate. Modules contain variables, functions, and classes, and import other modules for the tools they define. Functions have local variables of their own, as do classes—objects that live within modules and which we'll begin studying in the next chapter. As we saw in Part IV, functions can nest, too, but all are ultimately contained by modules at the top.

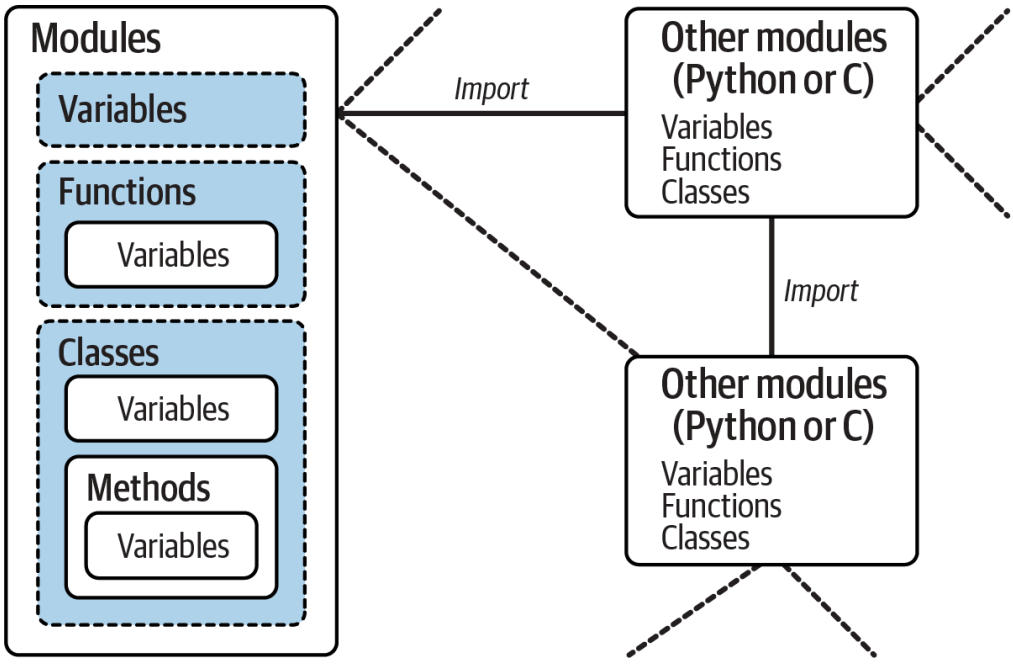

Figure 25-1. Module execution environment

Data Hiding in Modules

Next, we turn to private matters. As we've seen, a Python module exports all the names assigned at the top level of its file. There is no syntax for declaring which names should and shouldn't be visible outside the module. In fact, there's no way to prevent a client from changing names inside a module if it wants to.

In Python, data hiding in modules is a *convention*, not a syntactical constraint. If you want to break a module by trashing its names, you can, but most programmers don't count this as a life goal. Some purists object to this liberal attitude toward data hiding, claiming that it means Python can't implement encapsulation. However, encapsulation in Python is more about packaging than about restricting. We'll expand on this idea in the next part in relation to classes, which also have no privacy syntax but can often emulate its effect in code.

Minimizing from * Damage: _X and __all__

That being said, as a limited special case, you can prefix names with a single underscore (e.g., `_X`) to prevent them from being copied out when a client imports a module's names with a `from *` statement. This really is intended only to minimize namespace pollution; because `from *` copies out all names, the importer may get more than it's bargained for (including names that overwrite names in the importer). But underscores aren't "private" declarations: you can still see and change such names with other import forms. Example 25-1 demos the idea.

Example 25-1. unders.py

```
a, b, _c, _d = 1, 2, 3, 4                # Control from *
```

When names both with and without underscores are assigned this way, `from *` can't see the former, but `import` and normal `from` can:

```
$ python3
>>> from unders import *                 # Load non _X nc
>>> a, b
```

```
(1, 2)
>>> _c
NameError: name '_c' is not defined. Did you mean: '_'?

>>> from unders import _c              # But other impo
>>> _c
3
>>> import unders
>>> unders._d
4
```

Alternatively, you can achieve a hiding effect similar to the `_X` naming convention by assigning a list of variable name strings to the variable `__all__` at the top level of the module. When this feature is used, the `from *` statement will copy out *only* those names listed in the `__all__` list, though other imports work as before.

In effect, this is the converse of the `_X` convention: `__all__` identifies names to be *copied*, while `_X` identifies names *not* to be copied. Python looks for an `__all__` list in the module first, and copies its names irrespective of any underscores; if `__all__` is not found, `from *` copies all names without a single leading underscore. To demo, Example 25-2 uses both name-hiding tools.

Example 25-2. alls.py

```
__all__ = ['a', '_c']                  # Control from *
a, b, _c, _d = 1, 2, 3, 4              # __all__ has pr
```

On imports, `from *` gets everything in `__all__`, but no others; other importers again get everything:

```
$ python3
>>> from alls import *                 # Load __all__ r
>>> a, _c                              # Even if they h
(1, 3)
>>> b
NameError: name 'b' is not defined

>>> from alls import a, b, _c, _d      # But other impo
>>> a, b, _c, _d
```

```
(1, 2, 3, 4)

>>> import alls
>>> alls.a, alls.b, alls._c, alls._d
(1, 2, 3, 4)
```

Like the `_X` convention, the `__all__` list has meaning only to the `from *` statement form and does not amount to a privacy declaration: other import statements can still access all names, as the last two tests show. Still, module writers can use either technique to implement modules that are well-behaved when used with `from *`.

See also the discussion of `__all__` lists in package *__init__.py* files in Chapter 24. In this context, these lists declare nested submodules to be automatically loaded for a `from *` run on their container. The effect is similar to name hiding in module files, though packages extend it to apply to the content of a package folder in the filesystem.

Managing Attribute Access: __getattr__ and __dir__

On the subject of data hiding in modules, Python 3.7 added support for special functions at a module's top level that can be used to manage access to a module's attributes. If defined, a module's `__getattr__` function is automatically run when a module attribute is not found, and its `__dir__` overrides the normal attribute-list fetch run for the `dir` built-in. These can be used to implement both basic access constraints and arbitrarily dynamic interfaces.

These functions also shadow same-named tools in *classes* and are meant in part to obviate a long-standing and obscure trick that reset a module's object in the `sys.modules` table to an instance of a *class* with these same methods. This, of course, means that these functions may make more sense after we study classes in Part VI, but the artificial module in Example 25-3 demos the basics.

Example 25-3. gamod.py

```
var = 2                                     # Real attrib

def __getattr__(name):                      # Undefined a
    print(f'(virtual {name})', end=' ')
```

```
        match name:
            case 'test':
                return name * var
            case 'hack' | 'code':
                return name.upper()
            case _:
                raise AttributeError(f'{name} is undefined'

    def __dir__():
        return ['var', 'test', 'hack', 'code']
```

When imported, fetches of real attributes defined in the module work normally (subject to the `_X` and `__all__` of the prior section for `from *`), but missing names are routed to `__getattr__`, which can manage the request. It may also use a `raise` statement to flag an invalid request with an *exception*—a topic we'll study in full later in the book because it's also dependent on classes today:

```
>>> import gamod
>>> gamod.var                   # Real: __getattr__ not
2
>>> gamod.test                  # Virtual: computed wher
(virtual test) 'testtest'
>>> gamod.hack
(virtual hack) 'HACK'
>>> gamod.nonesuch
AttributeError: nonesuch is undefined

>>> dir(gamod)
['code', 'hack', 'test', 'var']
```

The `from` statement invokes `__getattr__` too, though `from *` requires names to be listed on `__all__` (you can largely ignore the spurious `__path__` fetch here, though a `__getattr__` must accommodate it):

```
>>> from gamod import code
(virtual __path__) (virtual code)
>>> code
'CODE'
>>> from gamod import *
(virtual __path__) (virtual __all__)
>>> var
```

```
2
>>> hack
NameError: name 'hack' is not defined
```

Importantly, `__getattr__` is *not* run for global-scope lookup within the module itself, so in-file undefined names remain undefined. It's really just for attribute fetches from *other* modules and does not catch *assignments* anywhere. For example, the first line of the following added at the bottom of Example 25-3 would fail, and the second line run in the REPL would make a new attribute in the module which bypasses `__getattr__` thereafter:

```
print(test)                # File: does NOT call __getc
gamod.hack = 'real'        # REPL: does NOT call __getc
```

Although this all works as advertised, it is a tool-builder's hook, and you'll have to unearth legitimate use cases. It may be useful in narrow roles, but it also *conflates* modules with classes and discounts the fact that module learners do not already understand these functions' origins in classes. This is a regrettably common theme in Python: additions often come with forward dependencies that seem to expect users to already know Python in order to use Python. Python is not just for Python experts, but that's a message baked into many a mod.

The good news here may be that a later proposal to add classes' `__setattr__` for module-attribute assignment was rejected by Python's steering committee—though only after allowing `__getattr__` and `__dir__` to sneak in. As usual, you should weigh the convolutions of this extension against its real-world applications.

Enabling Language Changes: __future__

Speaking of changes, Python mods that may break existing code are often introduced gradually. This is not always as "gradual" as it might be, and version 3.0 was a glaring exception (though 2.X was supported for 12 more years after 3.X's release). Sometimes, though, changes initially appear as optional extensions, which are disabled by default. To enable such an extension in Pythons that predate its official arrival, use a special `import` statement of this form:

```
from __future__ import featurename
```

When coded in a script, this statement must appear as the first executable statement in the file (possibly following a docstring or comment), because it enables special compilation of code on a per-module basis. It's also possible to submit this statement at the interactive prompt to experiment with upcoming language changes; the feature will then be available for the remainder of the interactive session.

For example, the prior edition of this book used this statement in Python 2.X to activate 3.X true division of Chapter 5, 3.X `print` calls of Chapter 11, and 3.X absolute imports for packages of Chapter 24. Earlier editions used this statement form to demonstrate generator functions, which require a `yield` that was not yet enabled by default.

This edition is boldly going forward with the present, but `__future__` can be used in older Pythons to enable Python 3.7's `StopIteration` "bubbling" behavior described at the end of Chapter 20:

```
from __future__ import generator_stop
```

For a list of futurisms you may import and turn on this way, see the Python library manual's entry for `__future__`. Per its documentation, none of its feature names will ever be removed, so it's safe to leave in a `__future__` import even in code run by a version of Python where the feature is enabled normally. The future does not erase the past.

Dual-Usage Modes: __name__ and __main__

Our next module-related trick lets you both import a file as a *module* and run it as a standalone *script*, a hook that is widely used in Python files. It's actually so simple that some learners miss the point at first. Each module has a built-in attribute called `__name__`, which Python creates and assigns automatically as follows:

- If the file is being run as a top-level script file, `__name__` is set to the string `'__main__'` when it starts.
- If the file is being imported instead, `__name__` is set to the module's name as known by its clients.

The upshot is that a module can test its own `__name__` to determine whether it's being run or imported. For example, suppose we create the code file named *dualmode.py* in Example 25-4, with a single function called `title`.

Example 25-4. dualmode.py

```
def title():
    print('Learning Python, 6E')

if __name__ == '__main__':           # Only when run
    title()                          # Not when importe
```

This module defines a function for clients to import and use as usual:

```
$ python3
>>> import dualmode
>>> dualmode.title()
Learning Python, 6E
```

But the module also includes code at the bottom that is set up to call the function automatically when this file is run as a program:

```
$ python3 dualmode.py
Learning Python, 6E
```

In effect, a module's `__name__` variable serves as a *usage mode flag*, allowing its code to be leveraged as *both* an importable library and a top-level script. Though simple, you'll see this hook used in many of the Python program files you are likely to encounter in the wild—both for testing and dual usage.

For instance, one of the most common ways you'll see the `__name__` test applied is for *self-test* code. In short, you can package code that tests a module's exports in the module itself by wrapping it in a `__name__` test at

the bottom of the file. This way, you can use the file in clients by *importing* it, but also test its logic by *running* it from the system shell or other launching scheme.

Coding self-test code at the bottom of a file under the `__name__` test is probably the most common and simplest unit-testing protocol in Python. It's much more convenient than retyping all your tests at the interactive prompt. (Preview: Chapter 36 will discuss other commonly used options for testing Python code—as you'll see, the `unittest` and `doctest` standard-library modules provide more advanced testing tools.)

In addition, the `__name__` trick is also commonly used when you're writing files that can be useful both as command-line utilities and as tool libraries. For instance, suppose you write a file-finder script in Python. You can get more mileage out of your code if you package it in functions, and add a `__name__` test in the file to automatically call those functions when the file is run standalone. That way, the script's code becomes reusable in other programs.

NOTE

What's in a __name__?: Don't confuse the `__main__` hook here with the *__main__.py* file discussed in the prior chapter. That file serves as a script when running an entire package *folder* as a program, but testing whether `__name__` is `'__main__'` is used to give two roles to a single *file*. Python often conflates the same names for similar but different purposes—see `__getattr__`!

Example: Unit Tests with __name__

In fact, we've already seen numerous cases in this book where the `__name__` check could be useful. As one example, we coded a script in Chapter 18 that computed the minimum value from the set of arguments sent—Example 18-3, whose code is repeated here for ease of reference:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res
```

```
def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))       # Self-t
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

This script includes self-test code at the bottom, so we can test it without having to retype test code in the REPL each time we run it. The problem with the way it is currently coded, however, is that the output of the self-test call will appear when this file is imported from another file to be used as a tool—not exactly a client-friendly feature! To do better, we can wrap up the self-test call in a `__name__` check so that it will be launched *only* when the file is run as a top-level script, not when it is imported. Example 25-5 lists this new-and-improved version of the module.

Example 25-5. minmax.py

```
print('I am:', __name__)

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))       # Se
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

We're also printing the value of `__name__` at the top here to trace its value (something you wouldn't do in a real library module). Python creates and assigns this usage-mode variable as soon as it starts loading a file. When we run this file as a top-level script, its name is set to the string `__main__`, so its self-test code kicks in automatically:

```
$ python3 minmax.py
I am: __main__
```

```
1
6
```

If we import the file, though, its name is not `__main__`, so we must explicitly call the function to make it run:

```
$ python3
>>> import minmax
I am: minmax
>>> minmax.minmax(minmax.lessthan, *'hack')
'a'
```

Again, regardless of whether this is used for testing, the net effect is that we get to use our code in *two different roles*—as a library module of tools, or as an executable program. It's buy-one-get-one code. You'll also see programs that route program-mode runs into a module called `main`:

```
def main():
    …
if __name__ == '__main__':
    main()
```

This works, but it's extra code, and there's nothing special about a function named `main` in Python—unlike some other languages, which may be part of the inspiration for this indirection's appearance in Python code.

NOTE

Fishing tutorial past: For another example of the `__name__ == '__main__'` test at work, see the dual-mode script/module *formats.py* in this book's examples package. It formats numbers with commas and currency conventions and demos how you can code your own flexible tools instead of relying on built-ins. It didn't add much here and was cut in this edition for space, but provides optional self-study code—and underscores that learning to fish generally beats being given one.

The as Extension for import and from

Next on the tour is a follow-up on a topic introduced in Chapter 23's name-collision coverage. As a minor but useful convenience, both the `import` and

`from` statements support an optional `as` clause, which simply *renames* a name imported by your script. For example, the following `import` statement using the `as` extension:

```
import modulename as name                    # And use
```

is equivalent to the following set of three statements, which renames the module in the importer's scope only (it's still known by its original name to other files), and drops the original name in the importer's scope altogether:

```
import modulename                            # Run a r
name = modulename                            # Rename
del modulename                               # Discard
```

After an `import` with `as` , you can—and in fact, must—use the name listed after the `as` to refer to the module. The longer equivalent works because modules are *first-class objects* just like functions, and can be passed around freely.

The `as` extension works in a `from` statement, too, to assign a name imported from a file to a different name in the importer's scope. As before, you get only the new name you provide, not its original:

```
from modulename import attrname as name      # And use
```

This in turn works the same as the following statements:

```
from modulename import attrname
name = attrname
del attrname
```

As noted in Chapter 23, this extension is commonly used both to provide *shorter synonyms* for longer names and to avoid *name clashes* when you are already using a name in your script that would otherwise be overwritten by a normal import:

```
import reallylongmodulename as name           # Use sho
name.func()                                   # Rename

from module1 import utility as util1          # Can hav
from module2 import utility as util2          # Rename
util1(); util2()
```

By way of review, the `as` clause also comes in handy for providing a short, simple name for an entire directory path and avoiding name collisions when using the *package import* feature described in Chapter 24:

```
import dir1.dir2.mod as mod                   # Only li
mod.func()                                    # Only or

from dir1.dir2.mod import func as modfunc     # Rename
modfunc()                                     # Allow f
```

Finally, the `as` clause is also something of a hedge against name *changes*: if a new release of a library renames a module or tool your code uses extensively, or provides a new alternative you'd rather use instead, you can simply rename it to its prior name on import to avoid breaking your code:

```
import newname as oldname
from library import newname as oldname
…and keep happily using oldname until you have time to
```

That said, if all software changes were just name changes, we'd have a lot less to fill our time!

Module Introspection

Next up is module plumbing. Because modules expose most of their interesting properties as built-in attributes, it's easy to write programs that manage other programs—tools we usually call *metaprograms*, because their subjects are other programs. This domain is also referred to as *introspection*, because programs can see and process object internals. Introspection is a

somewhat advanced feature, but it can be useful for building programming tools.

For instance, to fetch the value of a module's attribute, we can use attribute qualification or index the module's attribute dictionary, exposed in the built-in `__dict__` attribute we explored in Chapter 23. As we've also seen, Python's `vars` built-in is an alternative way to access `__dict__`, and its `sys.modules` dictionary records all loaded modules by import-name string. In addition, its `getattr` built-in lets us fetch attributes from their string names—it's like saying `object.attr`, but `attr` is an expression that produces a string at runtime.

Hence, all the following expressions reach the same attribute and object named `name` after importing `M` and `sys`:

```
M.name                                  # Qualify object by
M.__dict__['name']                      # Index namespace d
vars(M)['name']                         # Namespace dictior
sys.modules['M'].name                   # Index loaded-modu
getattr(M, 'name')                      # Call built-in fet
sys.modules['M'].__dict__['name']       # Module and attrib
```

Demoing with Example 25-5 (and chained comparisons that imply an `and` and a right-side repeat):

```
>>> import minmax, sys
>>> (minmax.lessthan
        is minmax.__dict__['lessthan']    is vars(minma
        is sys.modules['minmax'].lessthan is getattr(mi
        is sys.modules['minmax'].__dict__['lessthan'])
True
```

Of course, the first of these is much easier on the eyes (and keyboard), but the others support more generic access.

Example: Listing Modules with __dict__

By exposing module internals like this, Python helps you build programs about programs. As a demo, the module in Example 25-6, named *mydir.py*,

puts these ideas to work to implement a customized and expanded version of the built-in `dir` function. It defines and exports a function called `listing`, which takes a module object as an argument and prints a formatted display of the module's namespace sorted by attribute name.

Example 25-6. mydir.py

```
"""
mydir.py: a module that lists the namespaces of other m
Import this module's listing and pass an imported modul
run this file as a script to perform its self-test code
"""

sepchr = '-'
seplen = 60

def listing(module, verbose=True, unders=True):
    """
    List module: just attributes if verbose=False,
    hide built-in __X__ attributes if unders=False.
    """
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print(f'name: {module.__name__}\nfile: {module.
        print(sepline)

    # Scan namespace keys
    for (count, attr) in enumerate(sorted(module.__dict
        prefix = f'{count + 1:02d}) {attr}'
        if attr.startswith('__'):
            if unders:
               print(prefix, '<built-in name>')    # Sk
        else:
           print(prefix, getattr(module, attr))    # Or

    if verbose:
        print(sepline)
        print(f'{module.__name__} has {count + 1} names
        print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir)                                  # S
```

Notice the *docstrings* in this module; because we may want to use this as a general tool, the docstrings provide functional information accessible via `help` and the browser mode of PyDoc—tools that use similar introspection tools to do their jobs (see Chapter 15 for usage info). A *self-test* is also provided at the bottom of this module, which narcissistically imports and lists itself; here's the sort of output produced (with path edits for space):

```
$ python3 mydir.py
-------------------------------------------------------
name: mydir
file: /Users/me/…/LP6E/Chapter25/mydir.py
-------------------------------------------------------
01) __builtins__ <built-in name>
02) __cached__ <built-in name>
03) __doc__ <built-in name>
04) __file__ <built-in name>
05) __loader__ <built-in name>
06) __name__ <built-in name>
07) __package__ <built-in name>
08) __spec__ <built-in name>
09) listing <function listing at 0x1077758a0>
10) sepchr -
11) seplen 60
-------------------------------------------------------
mydir has 11 names
-------------------------------------------------------
```

To use this as a tool for listing other modules, simply pass the modules in as objects to this file's function. Here it is listing itself manually, as well as attributes in the `tkinter` GUI module in the Python standard library; it will technically work on any object with `__name__`, `__file__`, and `__dict__` attributes:

```
>>> from mydir import listing
>>> import mydir, tkinter

>>> listing(mydir, unders=False, verbose=False)
09) listing <function listing at 0x10800fba0>
10) sepchr -
11) seplen 60
```

```
>>> listing(tkinter, unders=False)
-------------------------------------------------------
name: tkinter
file: /…/lib/python3.12/tkinter/__init__.py
-------------------------------------------------------
01) ACTIVE active
02) ALL all
03) ANCHOR anchor
04) ARC arc
...more names omitted...
166) re <module 're' from '/…/lib/python3.12/re/__init_
167) sys <module 'sys' (built-in)>
168) types <module 'types' from '/…/lib/python3.12/type
169) wantobjects 1
-------------------------------------------------------
tkinter has 169 names
-------------------------------------------------------
```

You'll meet `getattr` and its relatives again later. The point to notice here is that `mydir` is a program that lets you browse other programs. Because Python exposes its internals, you can process objects generically.

NOTE

REPL startup tip: You can preload tools such as `mydir.listing` and the reloader we'll code in a moment into the interactive REPL by importing them in a file named by the `PYTHONSTARTUP` environment variable. Because code in the startup file runs in the interactive namespace, importing common tools in this file can save you some typing. See Appendix A for more info.

Importing Modules by Name String

Finally, it's time for something more dynamic. By now, you've probably noticed that the module name in an `import` or `from` statement is a hardcoded variable name. Sometimes, though, your program will get the name of a module to be imported as a string at runtime—from a user selection in a GUI, or a parse of an XML document, for instance. Unfortunately, you can't use import statements directly to load a module given its name as a string—Python expects a variable name that's taken literally and not evaluated, not a string or expression. For instance:

```
>>> import 'string'
SyntaxError: invalid syntax
```

It also won't work to simply assign the string to a variable name:

```
>>> x = 'string'
>>> import x
ModuleNotFoundError: No module named 'x'
```

Here, Python will try to import a file *x.py*, not the `string` module—the name in an `import` statement both becomes a variable assigned to the loaded module and identifies the external file literally.

Running Code Strings

To get around this, you need to use special tools to load a module dynamically from a string that is generated at runtime. The most general approach is to construct an `import` statement as a string of Python code and pass it to the `exec` built-in function to run:

```
>>> modname = 'string'
>>> exec('import ' + modname)       # Run a string of cc
>>> string                          # Imported in this r
<module 'string' from '/…/lib/python3.12/string.py'>
```

We met the `exec` function—and its cousin for expressions, `eval` —earlier, in Chapters 3, 5, 9, and 10. `exec` compiles a string of code and passes it to the Python interpreter to be executed. In Python, the bytecode compiler is available at runtime, so you can write programs that construct and run other programs like this. By default, `exec` runs the code in the current scope (as if pasted there), but you can get more specific by passing in optional namespace dictionaries. It also has security issues noted earlier in the book, which may be moot in a code string you are building yourself.

Direct Calls: Two Options

The only real drawback to `exec` here is that it must compile the `import` statement each time it runs, and compiling can be slow. Precompiling to

bytecode with the `compile` built-in may help for code strings run many times, but in most cases, it's probably simpler and may run quicker to use the built-in `__import__` function to import from a name string. The effect is similar, but `__import__` returns the module object—assign it to a name to keep it:

```
>>> modname = 'string'
>>> string = __import__(modname)
>>> string
<module 'string' from '/…/lib/python3.12/string.py'>
```

Because imports work by invoking `__import__`, it loads the named module normally. The newer standard-library call `importlib.import_module` does the same job; Python's docs describe it as a simplified wrapper around `__import__` for "everyday" use (though our code is growing longer as our tools are growing newer):

```
>>> import importlib
>>> modname = 'string'
>>> string = importlib.import_module(modname)
>>> string
<module 'string' from '/…/lib/python3.12/string.py'>
```

This call works the same as `__import__` in its basic roles, but see Python's manuals for more details on both calls' advanced usage and arguments. Python's docs also seem to prefer the newer `importlib` call for importing by name string, though this seems subjective, either call works, and the imports system has been a frequent morpher.

On callout here: both calls also work for the *package imports* of the prior chapter, but the first returns the *leftmost* component in a package path, and the second returns the *last*—in fact, this is their most prominent difference:

```
>>> import importlib
>>> __import__('email.message')
<module 'email' from '/…/lib/python3.12/email/__init__.
>>> importlib.import_module('email.message')
<module 'email.message' from '/…/lib/python3.12/email/m
```

The `importlib` call also works for a *package-relative* import string (with leading dots), if also passed the string name of a package path from which to resolve the import. See Chapter 24 for the story of package imports.

Example: Transitive Module Reloads

To tie together and apply some of the topics we've studied, this section develops a module tool that serves as a larger case study to close out this chapter and part. We explored module reloads in Chapter 23, as a way to pick up changes in code without stopping and restarting a program or REPL. When reloading a module, though, Python reloads only that particular module's file; it doesn't automatically reload modules that the file being reloaded happens to import.

For example, if you reload some module `A`, and `A` imports modules `B` and `C`, the reload applies only to `A`—not to `B` and `C`. The statements inside `A` that import `B` and `C` are rerun during the reload, but they just fetch the already loaded `B` and `C` module objects (assuming they've been imported before). In abstract code, here's the file *A.py*:

```
# A.py
import B                    # Not reloaded when A is!
import C                    # Just imports of already lo

$ python3
>>> …import and use A…
>>> from importlib import reload
>>> reload(A)
```

By default, this means that you cannot depend on reloads to pick up changes in all the modules in your program transitively. Instead, you must use multiple `reload` calls to update the subcomponents independently. This can require substantial work for large systems you're testing interactively. You can design your systems to reload their subcomponents automatically by adding `reload` calls in parent modules like `A`, but this complicates the code.

A recursive reloader

A better approach is to write a general tool to do transitive reloads automatically, by scanning a module's `__dict__` attributes dictionary and

checking the `type` of each attribute's value to find nested modules to reload. Such a utility function could call itself *recursively* to navigate arbitrarily shaped and deep import-dependency chains. The module `__dict__` was introduced in Chapter 23 and employed by *mydir.py* earlier, recursion was explored in Chapter 19, and the `type` call was presented in Chapter 9; we just need to combine these tools for this new role.

To this end, the module *reloadall.py* listed in Example 25-7 defines a `reload_all` function that automatically reloads a module, every module that the module imports, and so on, all the way to the bottom of each import chain. It uses a dictionary to keep track of already reloaded modules, recursion to walk the import chains, and the standard library's `types` module, which simply predefines `type` results for built-in types like modules. Its `visited` dictionary avoids repeats when imports are recursive or redundant (module objects are immutable, and so can be dictionary keys); as we saw in Chapters 5 and 8, a *set* could work similarly (and will in rewrites ahead).

Example 25-7. reloadall.py

```
"""
reloadall.py: transitively reload nested modules.
Call reload_all with one or more imported modules as ar
These modules, and all the modules they import, are rel
"""

import types
from importlib import reload

def status(module):
    print('reloading', module.__name__)

def tryreload(module):
    try:
        reload(module)
    except:
        print('FAILED:', module)

def transitive_reload(module, visited):
    if not module in visited:
        status(module)
        tryreload(module)
        visited[module] = True
        for attrobj in module.__dict__.values():
```

```
                if type(attrobj) == types.ModuleType:
                    transitive_reload(attrobj, visited)

    def reload_all(*args):
        visited = {}
        for arg in args:
            if type(arg) == types.ModuleType:
                transitive_reload(arg, visited)

    def tester(reloader, modname):
        import importlib, sys
        if len(sys.argv) > 1:
            modname = sys.argv[1]
        module = importlib.import_module(modname)
        reloader(module)

    if __name__ == '__main__':
        tester(reload_all, 'reloadall')
```

Besides namespace dictionaries, this script makes use of other tools we've studied before: it includes a `__name__` test to launch self-test code when run as a top-level script only, and its `tester` function uses `sys.argv` to inspect command-line arguments and `importlib` to import a module by name string passed in as a function or command-line argument. Review earlier coverage for more info if needed.

One curious bit: notice how this code's `tryreload` wraps the basic `reload` call in a `try` statement to catch exceptions. Reloads may fail for many reasons, and it's best to be defensive when using system interfaces. As you'll see in a moment, for example, an unreloadable `monitoring` module added to `sys` in Python 3.12 would otherwise crash the reloader. The `try` was previewed in Chapter 10 and will be covered in full in Part VII.

Testing recursive reloads

To use this module normally, import its `reload_all` function and pass it an already loaded module object—just as you would for the built-in `reload` function. Like `reload`, its module argument is usually obtained by a top-level `import`; as we've seen, `sys.modules` fetches work too, but modules accessed only by `from` don't apply.

To test first, run the module *standalone*. Its `tester` function runs a passed-in reloader on a module imported by *name string*—which is taken from a command-line argument if present, else a passed-in name. In this mode, the module's self-test code calls `tester` to run `reload_all` on its own imported module by default if no command-line arguments are used (its own name is not defined in the file without an import):

```
$ python3 reloadall.py
reloading reloadall
reloading types
```

With a command-line *argument*, the tester instead reloads the listed module by its name string—in the following, the Chapter 21 folder's benchmark module we coded in Example 21-8. To run this live, you need both the reloader module here and the module it reloads. One way to handle this is to add Chapter 21's code folder to your `PYTHONPATH` setting per Chapter 22. Copying files in either direction is subpar, and simply running in the *Chapter21* folder won't work because the reloader script's *Chapter25* folder is "home." Note that we give a *module* name in this mode, not a filename; because the script imports the module using the search path just like `import`, the *.py* extension is omitted:

```
$ pwd                                   # In Chapter
/Users/me/…/LP6E/Chapter25
$ export PYTHONPATH=../Chapter21        # Extend pat
$ python3 reloadall.py pybench          # Import+rel
reloading pybench
reloading sys
reloading sys.monitoring
FAILED: <module 'sys.monitoring'>
reloading os
reloading abc
reloading stat
reloading posixpath
reloading genericpath
reloading time
reloading timeit
reloading gc
reloading itertools
```

More usefully, we can also deploy this module at the *interactive* prompt. This works like the built-in `reload`, but adds recursive reloads for the module or modules passed—here, for standard-library modules:

```
$ python3
>>> from reloadall import reload_all       # Reload sta
>>> import os, tkinter
>>> reload_all(os)
reloading os
reloading abc
reloading sys
reloading sys.monitoring
FAILED: <module 'sys.monitoring'>
reloading stat
reloading posixpath
reloading genericpath

>>> reload_all(tkinter)
reloading tkinter
reloading collections
reloading collections.abc
…etc…
reloading _sre
reloading functools
reloading copyreg
```

In either mode, the reloader also works on module *packages*—here, for the standard library's `email` package:

```
$ python3 reloadall.py email.message       # Import+rel
reloading email.message
reloading binascii
reloading re
…etc…

$ python3
>>> from reloadall import reload_all       # Same, but
>>> import email.message
>>> reload_all(email.message)
reloading email.message
reloading binascii
```

```
reloading re
…etc…
```

The following runs the reloader on the `dir1.dir2.mod` path we coded in "Basic Package Structure". All items in the path are reloaded from a package root, and a `sys.path` mod in the REPL gives import access to another chapter's code folder—much like the `PYTHONPATH` setting used earlier (again, per Chapter 22):

```
>>> import sys
>>> sys.path.append('../Chapter24')          # Extend the
>>> import dir1.dir2.mod                     # A package
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod

>>> reload_all(dir1)                         # Reloads al
reloading dir1
Running dir1.__init__.py
reloading dir1.dir2
Running dir1.dir2.__init__.py
reloading dir1.dir2.mod
Loading dir1.dir2.mod
```

Finally, here is a simple session that demos the effect of normal versus *transitive* reloads—changes made to the two nested files are not picked up by reloads unless our transitive utility is used (files are listed inline here for brevity):

```
# File ra.py
import rb
X = 1

# File rb.py
import rc
Y = 2

# File rc.py
Z = 3

$ python3
>>> import ra
```

```
>>> ra.X, ra.rb.Y, ra.rb.rc.Z                  # Three-l
(1, 2, 3)
```

Now, without stopping Python, change all three files' assignment values, save the files, and reload back in the REPL:

```
>>> from importlib import reload
>>> reload(ra)                                 # Built-i
<module 'ra' from '/…/LP6E/Chapter25/ra.py>
>>> ra.X, ra.rb.Y, ra.rb.rc.Z
(111, 2, 3)

>>> from reloadall import reload_all
>>> reload_all(ra)                             # Normal
reloading ra
reloading rb
reloading rc
>>> ra.X, ra.rb.Y, ra.rb.rc.Z                  # Reloads
(111, 222, 333)
```

This is similar to the preceding package-path reload, but the imports here are explicit; module nesting in packages is implied. Study the reloader's code and results for more on its operation. The next section exercises its tools further.

Alternative codings

For all the recursion fans in the audience (and we know who we are), Example 25-8 lists an alternative *recursive* coding for the original function in Example 25-7. This new version uses a *set* instead of a dictionary to detect repeats and cycles, is marginally more *direct* because it eliminates a top-level loop, and serves to illustrate recursive coding in general. Compare with the original to see how this differs.

This version also gets some of its work for free from the original; in fact, this module essentially *extends* the original to replace just the parts that vary. Notice how it calls the original version's `tester`, passing in the `reload_all` defined here—which ensures that this module's reloader is run when this script is launched in standalone mode.

Example 25-8. reloadall2.py

```
"""
reloadall2.py: transitively reload nested modules.
Alternative coding: recursive, refactored.
"""

import types
from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    for obj in objects:
        if type(obj) == types.ModuleType and obj not ir
            status(obj)
            tryreload(obj)                          # F
            visited.add(obj)
            transitive_reload(obj.__dict__.values(), vi

def reload_all(*args):
    transitive_reload(args, set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall2')                # 1
```

As we saw in Chapter 19, there is usually an *explicit stack* or *queue* equivalent to recursive functions, which may be preferable in some contexts. Example 25-9 lists one such transitive reloader—it uses a stack instead of recursion, and a *set* to skip repeats and cycles. On each loop, all of a new module's attribute values are added to the end of the `objects` stack and filtered later, and this is repeated until the list of candidate objects becomes empty. A generator expression could filter out nonmodules in the `extend` call to avoid some pops, but this would be more complex.

Because it both pops and adds items at the *end* of its list, this version is stack-based, though the order of both pushes and dictionary values influences the order in which it reaches and reloads modules—it visits submodules in namespace dictionaries from *right to left*, unlike the left-to-right order of the recursive versions (trace through the code to see how). We could change this to match by reversing `values`, but reload order is unimportant.

Example 25-9. reloadall3.py

```
"""
reloadall3.py: transitively reload nested modules.
Alternative coding: nonrecursive, explicit stack.
"""

import types
from reloadall import status, tryreload, tester

def transitive_reload(objects, visited):
    while objects:
        next = objects.pop()                        # L
        if (type(next) == types.ModuleType          # I
            and next not in visited):               # N
            status(next)                            # F
            tryreload(next)
            visited.add(next)
            objects.extend(next.__dict__.values())

def reload_all(*args):
    transitive_reload(list(args), set())

if __name__ == '__main__':
    tester(reload_all, 'reloadall3')                # T
```

If the recursion and nonrecursion used in these examples is confusing, see the discussion of recursive functions in Chapter 19 for more background on the subject.

Testing reload variants

To prove that these two alternative reloaders work the same as the original, let's test all three of our reloader variants. Thanks to their common testing function, we can run all three from a command line both with no arguments to test the module reloading itself, and with the name of a module to be reloaded listed on the command line (in `sys.argv`):

```
$ python3 reloadall.py
reloading reloadall
reloading types
```

```
$ python3 reloadall2.py
reloading reloadall2
reloading types

$ python3 reloadall3.py
reloading reloadall3
reloading types
```

Though it's hard to see here, we really are testing the individual reloader alternatives—each of these tests shares a common `tester` function but passes it the `reload_all` from its own file. Here are the variants reloading the `tkinter` GUI module and all the modules its imports reach; again, the third's reloads order varies:

```
$ python3 reloadall.py tkinter
reloading tkinter
reloading collections
reloading collections.abc
…etc…
$ python3 reloadall2.py tkinter
reloading tkinter
reloading collections
reloading collections.abc
…etc…
$ python3 reloadall3.py tkinter
reloading tkinter
reloading re
reloading copyreg
…etc…
```

As usual, we can test interactively, too, by importing and calling either a module's main reload entry point with a module object, or the testing function with a reloader function and module name string:

```
$ python3
>>> import reloadall, reloadall2, reloadall3
>>> import tkinter
>>> reloadall.reload_all(tkinter)
reloading tkinter
reloading collections
reloading collections.abc
…etc…
>>> reloadall.tester(reloadall2.reload_all, 'tkinter')
reloading tkinter
```

```
reloading collections
reloading collections.abc
…etc…
>>> reloadall.tester(reloadall3.reload_all, 'reloadall3
reloading reloadall3
reloading types
```

Finally, as noted, the third reloader's results will generally vary by *order*; reload order in all reloaders depends on namespace dictionary ordering (which, as we've learned is deterministically ordered by key insertion time today), but the last also relies on the order in which items are added to its stack. To ensure that all three are reloading the same modules irrespective of the order in which they do so, we can use *sets* or *sorts* to test for order-neutral equality of their printed messages—obtained here by running shell commands with the `os.popen` utility we used in Chapter 21:

```
>>> import os
>>> res1 = os.popen('python3 reloadall.py tkinter').rea
>>> res2 = os.popen('python3 reloadall2.py tkinter').re
>>> res3 = os.popen('python3 reloadall3.py tkinter').re

>>> res1[:3]
['reloading tkinter\n', 'reloading collections\n', 'rel
>>> res3[:3]
['reloading tkinter\n', 'reloading re\n', 'reloading cc

>>> res1 == res2, res2 == res3
(True, False)
>>> set(res2) == set(res3)                  # Order-neu
True
>>> sorted(res2) == sorted(res3)            # Ditto
True
```

Run these scripts, study their code, and experiment on your own for more insight; these are the sort of importable tools you might want to add to your own source code library. Watch for a similar testing technique in the coverage of class tree listers in Chapter 31, where we'll apply it to passed *class* objects and extend it further.

Caveats: keep in mind that all the transitive reloaders, like the `reload` built-in that they use, rely on the fact that module reloads update module objects *in*

place, such that all references to those modules in any namespace will see the updated version automatically. Because `from` importers copy names out, they are not updated by reloads, transitive or not. Perhaps worse, modules imported only by `from` won't be reloaded, because they do not exist in any importer's namespace scanned. Doing better may require either source code analysis or import customization.

Tool impacts like this are perhaps another reason to prefer `import` to `from` —which brings us to the end of this chapter and part, and the standard set of warnings for this part's topic.

Module Gotchas

In this section, we'll explore the usual collection of boundary cases that can make life interesting for Python beginners. Some are review here, and a few are so obscure that coming up with representative examples can be a challenge, but most illustrate something important about the language.

Module Name Clashes: Package and Package-Relative Imports

If you have two modules of the same name, you may only be able to import one of them—by default, the one whose directory is leftmost in the `sys.path` module search path will always be chosen. This isn't an issue if the module you prefer is in your top-level script's directory; since that is always first in the module path, its contents will be located first automatically. For cross-directory imports, however, the linear nature of the module search path means that same-named files can clash.

To fix this, either avoid same-named files or use the package imports feature of Chapter 24. If you really need to get to two files of the same name, the latter is the solution: structure your source files in subdirectories, such that package-import directory names make the module references unique. As long as the enclosing package directory names are unique, you'll be able to access either or both of the same-named modules.

This issue can also crop up if you accidentally use a name for a module of your own that happens to be the same as a standard-library module you need

—your local module in the program's home directory (or another directory early in the module path) can hide and replace the library module.

To fix that, either avoid using the same name as another module you need or store your modules in a package directory and use the package-relative import model of Chapter 24. In this model, normal imports skip the package directory to access the library's version, but special dotted import statements can still select the local version of the module.

Statement Order Matters in Top-Level Code

As we've seen, when a module is first imported (or later reloaded), Python executes its statements one by one, from the top of the file to the bottom. This has a few subtle implications regarding *forward references* that are worth underscoring here:

- Code at the *top level* of a module file (not nested in a function) runs as soon as Python reaches it during an import; because of that, it cannot reference names assigned *lower* in the file.
- Code inside a *function* body doesn't run until the function is called; because names in a function aren't resolved until the function actually runs, they can usually reference names *anywhere* in the file.

In other words, forward references are usually only a concern in top-level module code that executes immediately; functions can reference names arbitrarily. Here's a file that illustrates forward reference dos and don'ts:

```
func1()                             # Error: func1 not ye

def func1():
    print(func2())                  # OK: func2 looked up

func1()                             # Error: func2 not ye

def func2():
    return "Hello"

func1()                             # OK: func1 and func2
```

When this file is imported (or run as a standalone program), Python executes its statements from top to bottom. The first call to `func1` fails because the `func1 def` hasn't run yet. The call to `func2` inside `func1` works as long as `func2`'s `def` has been reached by the time `func1` is called—and it hasn't when the second top-level `func1` call is run. The last call to `func1` at the bottom of the file works because `func1` and `func2` have both been assigned.

Mixing `def`s with top-level code is not only difficult to read, but it's also dependent on statement ordering. As a rule of thumb, if you need to mix immediate code with `def`s, put your `def`s at the top of the file and your top-level code at the bottom. That way, your functions are guaranteed to be defined and assigned by the time Python runs the code that uses them.

from Copies Names but Doesn't Link

Although it's commonly used, the `from` statement is the source of a variety of potential gotchas in Python. As we've seen, the `from` statement is really an assignment to names in the importer's scope—a *name-copy* operation, not a name aliasing. The implications of this are the same as for all assignments in Python, but they're especially subtle for names that live in different files. As a refresher, suppose we define the simple module *nested.py* in Example 25-10.

Example 25-10. nested.py

```
X = 99
def printer(): print(X)
```

If we import its two names using `from` in another module (or the REPL, which stands in for one), we get copies of those names, not links to them. Changing a name in the importer resets only the binding of the local version of that name, not the name in *nested1.py*:

```
>>> from nested import X, printer      # Copy names out
>>> X = 88                             # Changes my X or
>>> printer()                          # nested1's X is
99
```

If we instead use `import` to get the whole module and assign to a qualified name, we change the name in `nested1` (the module's loaded image, not its source code). Attribute qualification directs Python to a name in the module object, rather than a name in the importer:

```
>>> import nested                    # Get module as c
>>> nested.X = 88                    # Change nested1'
>>> nested.printer()
88
```

Takeaway: changes to names obtained with `from` don't impact any other module. As covered in Chapter 23, changes to mutable *objects* shared by names copied with `from` can impact other modules, but name changes cannot.

from * Can Obscure the Meaning of Variables

This was mentioned earlier but its demo was saved for here. Because you don't list the variables you want when using the `from *` statement form, it can accidentally overwrite names you're already using in your scope. Worse, it can make it difficult to determine where a variable comes from. This is especially true if the `from *` form is used on more than one imported file.

For example, if you use `from *` on three modules in the following, you'll have no way of knowing what a raw function call really means, short of searching all three external module files—all of which may be in other directories:

```
>>> from module1 import *            # May overwrite name
>>> from module2 import *            # No way to tell whc
>>> from module3 import *            # No way to see name

>>> func()                           # Huh?
```

The solution is simply not to do this: list the attributes you want in most `from` statements, and use at most one `from *` per file. That way, any undefined names must by deduction be in the module named in the single from *. You can avoid the issue altogether if you always use `import` instead

of `from` , but that advice is too harsh; like much else in programming, `from` is a convenient tool if used wisely. Even this example isn't an absolute evil—it's OK for a program to use this technique to collect names in a single module for convenience, as long as it's well known.

reload May Not Impact from Imports

Here's another `from` -related gotcha: as discussed previously, because `from` copies (assigns) names when run, there's no link back to the modules where the names came from. Names imported with `from` simply become references to objects, which happen to have been referenced by the same names in the importee when the `from` ran.

Because of this behavior, reloading the module of origin has no effect on clients that import its names using `from` . That is, the client's names will still reference the *original* objects fetched with `from` , even if the names in the original module are later reset. Here's the story in abstract code:

```
from module import X               # X may not reflect
…
from importlib import reload
reload(module)                     # Changes module, bu
X                                  # Still references o
```

To make reloads more effective, use `import` and name qualification instead of `from` . Because qualifications always go back to the module, they will find the new bindings of module names after reloading has updated the module's content in place:

```
import module                      # Get module, not na
…
from importlib import reload
reload(module)                     # Changes module in
module.X                           # Get current X: ref
```

This is why our transitive reloader earlier in this chapter doesn't apply to names fetched with `from` , only `import` ; again, if you're going to use reloads, you're probably better off with `import` .

reload, from, and Interactive Testing

In fact, the prior gotcha is even more nuanced than it appears. Chapter 3 warned that it's usually better not to launch programs with imports and reloads because of the complexities involved. Things get even worse when `from` is brought into the mix. Python beginners most often stumble onto its issues in scenarios like this—imagine that after opening a module file in a text edit window, you launch an interactive session to load and test your module with `from`:

```
from module import function
function(1, 2, 3)
```

Finding a bug, you jump back to the edit window, make a change, and try to reload the module this way:

```
from importlib import reload
reload(module)
```

This doesn't work, because the `from` statement assigned only the name `function`, not `module`. To refer to the module in a `reload`, you have to first bind its name with an `import` statement at least once:

```
from importlib import reload
import module
reload(module)
function(1, 2, 3)
```

But this doesn't quite work either— `reload` updates the module object in place, but as discussed in the preceding section, names like `function` that were copied out of the module in the past still refer to the *old objects*; in this instance, `function` is still the original version of the function. To really get the new function, you must refer to it as `module.function` after the `reload`, or rerun the `from`:

```
from importlib import reload
import module
reload(module)
```

```
from module import function        # Or give up and use
function(1, 2, 3)
```

Now, the new version of the function will finally run, but it seems an awful lot of work to get there.

As you can see, there are problems inherent in using `reload` with `from`: not only do you have to remember to reload after imports, but you also have to remember to rerun your `from` statements after reloads. This is complex enough to trip up even an expert once in a while. In fact, the situation grew even worse with Python 3.X, because you must also remember to import `reload` itself!

The short story is that you should not expect `reload` and `from` to play together nicely. Again, the best policy is not to combine them at all—use `reload` with `import`, or launch your programs other ways, as suggested in Chapter 3: using menu options in IDLE, file icon clicks, system command lines, the `exec` built-in function, or other.

Recursive from Imports May Not Work

The most bizarre (and, thankfully, obscure) gotcha has been saved for last. Because imports execute a file's statements from top to bottom, you need to be careful when using modules that import each other. This is often called *recursive* imports, but the recursion doesn't really occur (in fact, *circular* may be a better term here)—such imports won't get stuck in infinite importing loops. Still, because the statements in a module may not all have been run when it imports another module, some of its names may not yet exist.

If you use `import` to fetch the module as a whole, this probably doesn't matter; the module's names won't be accessed until you later use qualification to fetch their values, and by that time the module is likely complete. But if you use `from` to fetch specific names, you must bear in mind that you will only have access to names in that module that have already been assigned when a recursive import is kicked off.

As a demo of this phenomenon, consider the modules `recur1` and `recur2`, in Examples 25-11 and 25-12.

Example 25-11. recur1.py

```
X = 1
import recur2            # Run recur2 now if it doesn't
Y = 2
```

Example 25-12. recur2.py

```
from recur1 import X     # OK: X already assigned
from recur1 import Y     # Error: Y not yet assigned
```

Module `recur1` assigns a name `X` and then imports `recur2` before assigning the name `Y`. At this point, `recur2` can fetch `recur1` as a whole with an `import`—it already exists in Python's internal modules table, which makes it importable, and also prevents the imports from looping. But if `recur2` uses `from`, it will be able to see only the name `X`; the name `Y`, which is assigned below the `import` in `recur1`, doesn't yet exist, so you get an error:

```
$ python3
>>> import recur1
ImportError: cannot import name 'Y' from partially init
(most likely due to a circular import) (/…/LP6E/Chapter
```

Python avoids rerunning `recur1`'s statements when they are imported recursively from `recur2` (otherwise the imports would send the script into an infinite loop that might require a Ctrl+C solution or worse), but `recur1`'s namespace is incomplete when it's imported by `recur2`.

The solution? Don't use `from` in recursive imports (no, really!). Python won't get stuck in a cycle if you do, but your programs will once again be dependent on the order of the statements in the modules. In fact, there are two ways out of this gotcha:

- You can usually eliminate import cycles like this by careful design—maximizing cohesion and minimizing coupling per the start of this chapter are good first steps.

- If you can't break the cycles completely, postpone module name accesses by using `import` and attribute qualification (instead of `from` and direct names), or by running your `from`s either inside functions (instead of at the top level of the module) or near the bottom of your file to defer their execution.

There is additional perspective on this issue in the exercises at the end of this chapter—which we've officially reached.

Chapter Summary

This chapter surveyed module topics, some of which qualify as advanced. We studied data-hiding techniques, enabling new language features with the `__future__` module, the `__name__` usage-mode variable, transitive reloads, importing by name strings, and more. We also explored module design issues, wrote some substantial programs, and looked at common mistakes related to modules to help you avoid them in your code.

The next chapter begins our exploration of Python's *class*—its object-oriented programming tool. Much of what we've covered in the last few chapters will apply there, too: classes live in modules and are namespaces as well, but they add an extra component to attribute lookup called *inheritance search*. As this is the last chapter in this part of the book, however, before we dive into classes, be sure to work through this part's set of lab exercises. And before that, here is this chapter's quiz to review the topics covered here.

Test Your Knowledge: Quiz

1. What is significant about variables at the top level of a module whose names begin with a single underscore?
2. What does it mean when a module's `__name__` variable is the string `'__main__'`?
3. How might you step through all the attributes in a module with a loop?
4. If the user interactively types the name of a module to test, how can your code import it?
5. If the module `__future__` allows us to import from the future, can we also import from the past?

Test Your Knowledge: Answers

1. Variables at the top level of a module whose names begin with a single underscore are *not* copied out to the importing scope when the `from *` statement form is used. They can still be accessed by an `import` or the normal `from` statement form, though. The `__all__` list is similar, but the logical converse; its contents are the only names that *are* copied out for a `from *`.
2. If a module's `__name__` variable is the string `'__main__'`, it means that the file is being executed as a top-level script instead of being imported from another file in the program. That is, the file is being used as a program, not a library. This usage mode variable supports dual-mode code and tests.
3. By using the module's built-in `__dict__` attribute. This is a normal dictionary that holds all of the module's attributes, so code can iterate over its keys, values, or key/value pairs. When needed, attribute values can also be fetched for string names by indexing `__dict__`, or calling the `getattr` built-in function.
4. User input usually comes into a script as a string; to import the referenced module given its string name, you can build and run an `import` statement with `exec`, or pass the string name in a call to the `__import__` or `importlib.import_module` functions.
5. No, we can't import from the past in Python. We can install (or stubbornly use) an older version of the language, but the latest Python is generally the best Python (with apologies to 2.X fans in the audience).

Test Your Knowledge: Part V Exercises

See "Part V, Modules and Packages" in Appendix B for the solutions.

1. *Import basics*: Write a program that counts the lines and characters in a file (similar in spirit to part of what `wc` does on Unix). With your text editor, code a Python module called *mymod.py* that exports three top-level names:
   - A `countLines(`*`name`*`)` function that reads an input file specified by string *`name`*, and counts the number of lines in it (hint: *`file`*`.readlines` does most of the work for you, and `len` does the

rest, though you could count with `for` and file iterators to support massive files too).

– A `countChars(`*`name`*`)` function that reads an input file and counts the number of characters in it (hint: *`file`*`.read` returns a single string, which may be used in similar ways).

– A `test(`*`name`*`)` function that calls both counting functions with a given input filename. Such a filename generally might be passed in, hardcoded, input from a user like you with the `input` built-in function, or pulled from a command line via the `sys.argv` list demoed in this chapter's *reloadall.py* example; for now, you can assume it's a passed-in function argument.

All three `mymod` functions should expect a filename string to be passed in. If you type more than two or three lines per function, you're working much too hard—use the hints given!

Next, test your module interactively, using `import` and attribute references to fetch your exports. Does your `PYTHONPATH` need to include the directory where you created *mymod.py*? Try running your module on itself: for example, `test('mymod.py')`. Note that `test` opens the file twice; if you're feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: *`file`*`.seek(0)` is a file rewind).

2. `from`/`from *`: Test your `mymod` module from exercise 1 interactively by using `from` to load the exports directly, first by name, then using the `from *` variant to fetch everything.
3. `__main__`: Add a line in your `mymod` module that calls the `test` function automatically only when the module is run as a script, not when it is imported. The line you add will probably test the value of `__name__` for the string `'__main__'`, as shown in this chapter. Try running your module from the system command line or other program-launch scheme; then, import the module and test its functions interactively. Does it still work in both modes?
4. *Nested imports*: Write a second module, *myclient.py*, that imports `mymod` and tests its functions; then run `myclient` from the system command line or other scheme. If `myclient` uses `from` to fetch from `mymod`, will `mymod`'s functions be accessible from the top level of `myclient`? What if it imports with `import` instead? Try coding both variations in `myclient` and test interactively by importing `myclient` and inspecting its `__dict__` attribute.

5. *Package imports*: Import your *mymod.py* file from a package. Create a subdirectory called *mypkg* nested in a directory on your module import search path, copy or move the *mymod.py* module file you created in exercise 1 or 3 into the new directory, and try to import it with a package import of the form `import mypkg.mymod` and call its functions. Try to fetch your counter functions with a `from` too.

   This works on all Python platforms (that's part of the reason Python uses "." as a path separator). The package directory you create can be simply a subdirectory of the one you're working in; if it is, it will be found via the home directory component of the search path, and you won't have to configure your path.

   You also don't need an *__init__.py* file in the package directory your module was moved into to make this go, but make one with some basic prints in it and see if they run on each import or reload of the package folder. Finally, also copy *mymod.py* to the package folder's *__main__.py* and invoke it by running the folder itself; does it make sense to do that here? Can you still run the nested *mymod.py* module itself?
6. *Reloads*: Experiment with module reloads: if you haven't already, perform the tests in Chapter 23's *changer.py* (Example 23-10), changing the called function's message or behavior repeatedly, without stopping the Python REPL. Depending on your device, you might edit `changer` in another window, or suspend the Python interpreter and edit in the same window (on Unix, a Ctrl+Z key combination usually suspends the current process, and an `fg` command later resumes it, though a separate text-editor window can work just as well).
7. *Circular imports*: In the section on recursive (a.k.a. circular) import gotchas, importing `recur1` raised an error. But if you restart Python and import `recur2` interactively, the error doesn't occur—test this and see for yourself. Why do you think it works to import `recur2`, but not `recur1`? (Hint: Python records new modules before running their code, and later imports fetch the module first, whether the module is "complete" yet or not.)

   Now, run `recur1` as a top-level script file: `python3 recur1.py`. Do you get the same error that occurs when `recur1` is imported interactively? Why? (Hint: when modules are run as programs, they aren't imported, so this case has the same effect as importing `recur2` interactively; `recur2` is the first module imported.) What happens when you run `recur2` as a script? Circular imports are uncommon in practice.

On the other hand, if you can understand why they are a potential problem, you know a lot about Python's import semantics.