Chapter 7. Modules and Packages

A typical Python program is made up of several source files. Each source file is a *module*, grouping code and data for reuse. Modules are normally independent of each other, so that other programs can reuse the specific modules they need. Sometimes, to manage complexity, developers group together related modules into a *package*—a hierarchical, tree-like structure of related modules and subpackages.

A module explicitly establishes dependencies upon other modules by using `import` or `from` statements. In some programming languages, global variables provide a hidden conduit for coupling between modules. In Python, global variables are not global to all modules, but rather are attributes of a single module object. Thus, Python modules always communicate in explicit and maintainable ways, clarifying the couplings between them by making them explicit.

Python also supports *extension modules*—modules coded in other languages such as C, C++, Java, C#, or Rust. For the Python code importing a module, it does not matter whether the module is pure Python or an extension. You can always start by coding a module in Python. Should you need more speed later, you can refactor and recode some parts of your module in lower-level languages, without changing the client code that uses the module. **Chapter 25** (available **online**) shows how to write extensions in C and Cython.

This chapter discusses module creation and loading. It also covers grouping modules into packages, using `setuptools` to install packages, and how to prepare packages for distribution; this latter subject is more thoroughly covered in **Chapter 24** (also available **online**). We close this chapter with a discussion of how best to manage your Python environment(s) .

Module Objects

In Python, a module is an object with arbitrarily named attributes that you can bind and reference. Modules in Python are handled like other objects. Thus, you can pass a module as an argument in a call to a function. Similarly, a function can return a module as the result of a call. A module, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. Modules can be keys or values in a dictionary, and can be members of a set. For example, the `sys.modules` dictionary, discussed in **"Module Loading"**, holds module objects as its values. The fact that modules can be treated like other values in Python is often expressed by saying that modules are *first-class* objects.

The import Statement

The Python code for a module named *aname* usually lives in a file named *aname.py*, as covered in **"Searching the Filesystem for a Module"**. You can use any Python source file[1] as a module by executing an **`import`** statement in another Python source file. **`import`** has the following syntax:

```
import modname [as varname][,...]
```

After the **`import`** keyword come one or more module specifiers separated by commas. In the simplest, most common case, a module specifier is just *modname*, an identifier—a variable that Python binds to the module object when the **`import`** statement finishes. In this case, Python looks for the module of the same name to satisfy the **`import`** request. For example, this statement:

```
import mymodule
```

looks for the module named `mymodule` and binds the variable named `mymodule` in the current scope to the module object. *modname* can also be a

sequence of identifiers separated by dots (`.`) to name a module contained in a package, as covered in **"Packages"**.

When **`as`** *`varname`* is part of a module specifier, Python looks for a module named *`modname`* and binds the module object to the variable *`varname`*. For example, this:

```
import mymodule as alias
```

looks for the module named `mymodule` and binds the module object to the variable *`alias`* in the current scope. *`varname`* must always be a simple identifier.

The module body

The *body* of a module is the sequence of statements in the module's source file. There is no special syntax required to indicate that a source file is a module; as mentioned previously, you can use any valid Python source file as a module. A module's body executes immediately the first time a given run of a program imports it. When the body starts executing, the module object has already been created, with an entry in `sys.modules` already bound to the module object. The module's (global) namespace is gradually populated as the module's body executes.

Attributes of module objects

An **`import`** statement creates a new namespace containing all the attributes of the module. To access an attribute in this namespace, use the name or alias of the module as a prefix:

```
import mymodule
a = mymodule.f()
```

or:

```
import mymodule as alias
a = alias.f()
```

This reduces the time it takes to import the module and ensures that only those applications that use that module incur the overhead of creating it.

Normally, it is the statements in the module body that bind the attributes of a module object. When a statement in the module body binds a (global) variable, what gets bound is an attribute of the module object.

A MODULE BODY EXISTS TO BIND THE MODULE'S ATTRIBUTES

The normal purpose of a module body is to create the module's attributes: **`def`** statements create and bind functions, **`class`** statements create and bind classes, and assignment statements can bind attributes of any type. For clarity and cleanliness in your code, be wary about doing anything else in the top logical level of the module's body *except* binding the module's attributes.

A `__getattr__` function defined at module scope can dynamically create new module attributes. One possible reason for doing so would be to lazily define attributes that are time-consuming to create; defining them in a module-level `__getattr__` function defers the creation of the attributes until they are actually referenced, if ever. For instance, this code could be added to *mymodule.py* to defer the creation of a list containing the first million prime numbers, which can take some time to compute:

```
def __getattr__(name):
    if name == 'first_million_primes':
        def generate_n_primes(n):
            # ... code to generate 'n' prime numbers ...

        import sys
        # Look up __name__ in sys.modules to get current module
        this_module = sys.modules[__name__]
        this_module.first_million_primes = generate_n_primes(1_000_000)
```

```
            return this_module.first_million_primes

        raise AttributeError(f'module {__name__!r}
                             f' has no attribute {name!r}')
```

Using a module-level `__getattr__` function has only a small impact on the time to import *mymodule.py*, and only those applications that actually use `mymodule.first_million_primes` will incur the overhead of creating it.

You can also bind module attributes in code outside the body (i.e., in other modules); just assign a value to the attribute reference syntax *M.name* (where *M* is any expression whose value is the module, and the identifier *name* is the attribute name). For clarity, however, it's best to bind module attributes only in the module's own body.

The **`import`** statement binds some module attributes as soon as it creates the module object, before the module's body executes. The `__dict__` attribute is the `dict` object that the module uses as the namespace for its attributes. Unlike other attributes of the module, `__dict__` is not available to code in the module as a global variable. All other attributes in the module are items in `__dict__` and are available to code in the module as global variables. The `__name__` attribute is the module's name, and `__file__` is the filename from which the module was loaded; other dunder-named attributes hold other module metadata. (See also **"Special Attributes of Package Objects"** for details on the attribute `__path__`, in packages only.)

For any module object *M*, any object *x*, and any identifier string *S* (except `__dict__`), binding *M.S = x* is equivalent to binding *M*`.__dict__['`*S*`'] =` *x*. An attribute reference such as *M.S* is also substantially equivalent to *M*`.__dict__['`*S*`']`. The only difference is that, when *S* is not a key in *M*`.__dict__`, accessing *M*`.__dict__['`*S*`']` raises `KeyError`, while accessing *M.S* raises `AttributeError`. Module attributes are also available to all code in the module's body as global variables. In other words, within the module body, *S* used as a global variable is equivalent to *M.S* (i.e., *M*`.__dict__['`*S*`']`) for both binding and reference (when *S* is *not* a key in

M.__dict__, however, referring to *S* as a global variable raises NameError).

Python built-ins

Python supplies many built-in objects (covered in **Chapter 8**). All built-in objects are attributes of a preloaded module named builtins. When Python loads a module, the module automatically gets an extra attribute named __builtins__, which refers either to the module builtins or to its dictionary. Python may choose either, so don't rely on __builtins__. If you need to access the module builtins directly (a rare need), use an **import** builtins statement. When you access a variable found neither in the local namespace nor in the global namespace of the current module, Python looks for the identifier in the current module's __builtins__ before raising NameError.

The lookup is the only mechanism that Python uses to let your code access built-ins. Your own code can use the access mechanism directly (do so in moderation, however, or your program's clarity and simplicity will suffer). The built-ins' names are not reserved, nor are they hardwired in Python itself—you can add your own built-ins or substitute your functions for the normal built-in ones, in which case all modules see the added or replaced ones. Since Python accesses built-ins only when it cannot resolve a name in the local or module namespace, it is usually sufficient to define a replacement in one of those namespaces. The following toy example shows how you can wrap a built-in function with your own function, allowing abs to take a string argument (and return a rather arbitrary mangling of the string):

```
# abs takes a numeric argument; let's make it accept a string as well
import builtins
_abs = builtins.abs                        # save original built-in
def abs(str_or_num):
    if isinstance(str_or_num, str):        # if arg is a string
        return ''.join(sorted(set(str_or_num)))  # get this instead
```

```
    return _abs(str_or_num)                # call real built-in
builtins.abs = abs                         # override built-in w/wrapper
```

Module documentation strings

If the first statement in the module body is a string literal, Python binds that string as the module's documentation string attribute, named `__doc__`. For more information on documentation strings, see **"Docstrings"**.

Module-private variables

No variable of a module is truly private. However, by convention, every identifier starting with a single underscore (_), such as `_secret`, is *meant* to be private. In other words, the leading underscore communicates to client-code programmers that they should not access the identifier directly.

Development environments and other tools rely on the leading underscore naming convention to discern which attributes of a module are public (i.e., part of the module's interface) and which are private (i.e., to be used only within the module).

RESPECT THE "LEADING UNDERSCORE MEANS PRIVATE" CONVENTION

It's important to respect the convention that a leading underscore means private, particularly when you write client code that uses modules written by others. Avoid using any attributes in such modules whose names start with _. Future releases of the modules will strive to maintain their public interface, but are quite likely to change private implementation details: private attributes are meant exactly for such details.

The from Statement

Python's **`from`** statement lets you import specific attributes from a module into the current namespace. **`from`** has two syntax variants:

```
from modname import attrname [as varname][,...]
from modname import *
```

A **from** statement specifies a module name, followed by one or more attribute specifiers separated by commas. In the simplest and most common case, an attribute specifier is just an identifier *attrname*, which is a variable that Python binds to the attribute of the same name in the module named *modname*. For example:

```
from mymodule import f
```

modname can also be a sequence of identifiers separated by dots (.) to name a module within a package, as covered in **"Packages"**.

When **as** *varname* is part of an attribute specifier, Python gets the value of the attribute *attrname* from the module and binds it to the variable *varname*. For example:

```
from mymodule import f as foo
```

attrname and *varname* are always simple identifiers.

You may optionally enclose in parentheses all the attribute specifiers that follow the keyword **import** in a **from** statement. This can be useful when you have many attribute specifiers, in order to split the single logical line of the **from** statement into multiple logical lines more elegantly than by using backslashes (\):

```
from some_module_with_a_long_name import (
    another_name, and_another as x, one_more, and_yet_another as y)
```

from...import *

Code that is directly inside a module body (not in the body of a function or class) may use an asterisk (*) in a **from** statement:

```
from mymodule import *
```

The * requests that "all" attributes of module *modname* be bound as global variables in the importing module. When module *modname* has an attribute named `__all__`, the attribute's value is the list of the attribute names that this type of **from** statement binds. Otherwise, this type of **from** statement binds all attributes of *modname* except those beginning with underscores.

BEWARE USING "FROM M IMPORT *" IN YOUR CODE

Since **from** *M* **import** * may bind an arbitrary set of global variables, it can have unforeseen, undesired side effects, such as hiding built-ins and rebinding variables you still need. Use the * form of **from** very sparingly, if at all, and only to import modules that are explicitly documented as supporting such usage. Your code is most likely better off *never* using this form, which is meant mostly as a convenience for occasional use in interactive Python sessions.

---

from versus import

The **import** statement is often a better choice than the **from** statement. When you always access module *M* with the statement **import** *M*, and always access *M*'s attributes with the explicit syntax *M*.*A*, your code is slightly less concise but far clearer and more readable. One good use of **from** is to import specific modules from a package, as we discuss in **"Packages"**. In most other cases, **import** is better style than **from**.

Handling import failures

If you are importing a module that is not part of standard Python and wish to handle import failures, you can do so by catching the `ImportError` exception. For instance, if your code does optional output formatting using the third-party `rich` module, but falls back to regular output if that module has not been installed, you would import the module using:

```
try:
    import rich
except ImportError:
    rich = None
```

Then, in the output portion of your program, you would write:

```
if rich is not None:
        ... output using rich module features ...
else:
        ... output using normal print() statements ...
```

Module Loading

Module-loading operations rely on attributes of the built-in `sys` module (covered in **"The sys Module"**) and are implemented in the built-in function `__import__`. Your code could call `__import__` directly, but this is strongly discouraged in modern Python; rather, **`import`** `importlib` and call `importlib.import_module` with the module name string as the argument. `import_module` returns the module object or, should the import fail, raises `ImportError`. However, it's best to have a clear understanding of the semantics of `__import__`, because `import_module` and **`import`** statements both depend on it.

To import a module named *M*, `__import__` first checks the dictionary `sys.modules`, using the string *M* as the key. When the key *M* is in the dictionary, `__import__` returns the corresponding value as the requested module object. Otherwise, `__import__` binds `sys.modules[`*M*`]` to a new empty module object with a `__name__` of *M*, then looks for the right way to initialize (load) the module, as covered in the upcoming section on searching the filesystem for a module.

Thanks to this mechanism, the relatively slow loading operation takes place only the first time a module is imported in a given run of the program. When a module is imported again, the module is not reloaded, since `__import__` rapidly finds and returns the module's entry in `sys.modules`. Thus, all imports of a given module after the first one are very fast: they're just dictionary lookups. (To *force* a reload, see **"Reloading Modules"**.)

Built-in Modules

When a module is loaded, `__import__` first checks whether the module is a built-in. The tuple `sys.builtin_module_names` names all built-in modules, but rebinding that tuple does not affect module loading. When it loads a built-in module, as when it loads any other extension, Python calls the module's initialization function. The search for built-in modules also looks for modules in platform-specific locations, such as the Registry in Windows.

Searching the Filesystem for a Module

If module *M* is not a built-in, `__import__` looks for *M*'s code as a file on the filesystem. `__import__` looks at the items of the list `sys.path`, which are strings, in order. Each item is the path of a directory, or the path of an archive file in the popular **ZIP format**. `sys.path` is initialized at program startup, using the environment variable `PYTHONPATH` (covered in **"Environment Variables"**), if present. The first item in `sys.path` is always the directory from which the main program is loaded. An empty string in `sys.path` indicates the current directory.

Your code can mutate or rebind `sys.path`, and such changes affect which directories and ZIP archives `__import__` searches to load modules. Changing `sys.path` does *not* affect modules that are already loaded (and thus already recorded in `sys.modules`).

If there is a text file with the extension *.pth* in the `PYTHONHOME` directory at startup, Python adds the file's contents to `sys.path`, one item per line. *.pth* files can contain blank lines and comment lines starting with the character `#`; Python ignores any such lines. *.pth* files can also contain **`import`** statements (which Python executes before your program starts to execute), but no other kinds of statements.

When looking for the file for module *`M`* in each directory and ZIP archive along `sys.path`, Python considers the following extensions in this order:

1. *.pyd* and *.dll* (Windows) or *.so* (most Unix-like platforms), which indicate Python extension modules. (Some Unix dialects use different extensions; e.g., *.sl* on HP-UX.) On most platforms, extensions cannot be loaded from a ZIP archive—only source or bytecode-compiled Python modules can.
2. *.py*, which indicates Python source modules.
3. *.pyc*, which indicates bytecode-compiled Python modules.
4. When it finds a *.py* file, Python also looks for a directory called *__pycache__*. If it finds such a directory, Python looks in that directory for the extension .<*tag*>*.pyc*, where <*tag*> is a string specific to the version of Python that is looking for the module.

One last path in which Python looks for the file for module *`M`* is *M/__init__.py*: a file named *__init__.py* in a directory named *M*, as covered in **"Packages"**.

Upon finding the source file *M.py*, Python compiles it to *M.*<*tag*>*.pyc*, unless the bytecode file is already present, is newer than *M.py*, and was compiled by the same version of Python. If *M.py* is compiled from a writable directory, Python creates a *__pycache__* subdirectory if necessary and saves the bytecode file to the filesystem in that subdirectory so that future runs won't needlessly recompile it. When the bytecode file is

newer than the source file (based on an internal timestamp in the byte-code file, not on trusting the date as recorded in the filesystem), Python does not recompile the module.

Once Python has the bytecode, whether built anew by compilation or read from the filesystem, Python executes the module body to initialize the module object. If the module is an extension, Python calls the module's initialization function.

BE CAREFUL ABOUT NAMING YOUR PROJECT'S .PY FILES

A common problem for beginners occurs when programmers writing their first few projects accidentally name one of their *.py* files with the same name as an imported package, or a module in the standard library (stdlib). For example, an easy mistake when learning the `turtle` module is to name your program *turtle.py*. When Python then tries to import the `turtle` module from the stdlib, it will load the local module instead, and usually raise some unexpected `AttributeErrors` shortly thereafter (since the local module does not include all the classes, functions, and variables defined in the stdlib module). Do not name your project *.py* files the same as imported or stdlib modules!

You can check whether a module name already exists using a command of the form **`python -m`** ***`testname`***. If the message `'no module` *`testname`*`'` is displayed, then you should be safe to name your module *testname.py*.

In general, as you become familiar with the modules in the stdlib and common package names, you will come to know what names to avoid.

The Main Program

Execution of a Python application starts with a top-level script (known as the *main program*), as explained in **"The python Program"**. The main program executes like any other module being loaded, except that Python keeps the bytecode in memory, not saving it to disk. The module name for the main program is `'__main__'`, both as the `__name__` variable (module attribute) and as the key in `sys.modules`.

DON'T IMPORT THE .PY FILE YOU'RE USING AS THE MAIN PROGRAM

You should not import the same *.py* file that is the main program. If you do, Python loads the module again, and the body executes again in a separate module object with a different `__name__`.

Code in a Python module can test if the module is being used as the main program by checking if the global variable `__name__` has the value `'__main__'`. The idiom:

```
if __name__ == '__main__':
```

is often used to guard some code so that it executes only when the module runs as the main program. If a module is meant only to be imported, it should normally execute unit tests when run as the main program, as covered in **"Unit Testing and System Testing"**.

Reloading Modules

Python loads a module only the first time you import the module during a program run. When you develop interactively, you need to *reload* your modules after editing them (some development environments provide automatic reloading).

To reload a module, pass the module object (*not* the module name) as the only argument to the function `reload` from the `importlib` module. `importlib.reload(M)` ensures the reloaded version of *M* is used by client code that relies on `import` *M* and accesses attributes with the syntax *M*`.`*A*. However, `importlib.reload(M)` has no effect on other existing references bound to previous values of *M*'s attributes (e.g., with a **`from`** statement). In other words, already-bound variables remain bound as they were, unaffected by `reload`. `reload`'s inability to rebind such variables is a further incentive to use **`import`** rather than **`from`**.

`reload` is not recursive: when you reload module *`M`*, this does not imply that other modules imported by *`M`* get reloaded in turn. You must reload, by explicit calls to `reload`, every module you have modified. Be sure to take into account any module reference dependencies, so that reloads are done in the proper order.

Circular Imports

Python lets you specify circular imports. For example, you can write a module *a.py* that contains **`import`** `b`, while module *b.py* contains **`import`** `a`.

If you decide to use a circular import for some reason, you need to understand how circular imports work in order to avoid errors in your code.

AVOID CIRCULAR IMPORTS

In practice, you are nearly always better off avoiding circular imports, since circular dependencies are fragile and hard to manage.

Say that the main script executes **`import`** `a`. As discussed earlier, this **`import`** statement creates a new empty module object as `sys.modules['a']`, then the body of module `a` starts executing. When `a` executes **`import`** `b`, this creates a new empty module object as `sys.modules['b']`, and then the body of module `b` starts executing. `a`'s module body cannot proceed until `b`'s module body finishes.

Now, when `b` executes **`import`** `a`, the **`import`** statement finds `sys.modules['a']` already bound, and therefore binds global variable `a` in module `b` to the module object for module `a`. Since the execution of `a`'s module body is currently blocked, module `a` is usually only partly populated at this time. Should the code in `b`'s module body try to access some attribute of module `a` that is not yet bound, an error results.

If you keep a circular import, you must carefully manage the order in which each module binds its own globals, imports other modules, and accesses globals of other modules. You get greater control over the sequence

in which things happen by grouping your statements into functions, and calling those functions in a controlled order, rather than just relying on sequential execution of top-level statements in module bodies. Removing circular dependencies (for example, by moving an import away from module scope and into a referencing function) is easier than ensuring bombproof ordering to deal with circular dependencies.

SYS.MODULES ENTRIES

`__import__` never binds anything other than a module object as a value in `sys.modules`. However, if `__import__` finds an entry already in `sys.modules`, it returns that value, whatever type it may be. **`import`** and **`from`** statements rely on `__import__`, so they too can use objects that are not modules.

Custom Importers

Another advanced, rarely needed functionality that Python offers is the ability to change the semantics of some or all **`import`** and **`from`** statements.

Rebinding __import__

You can rebind the `__import__` attribute of the `builtin` module to your own custom importer function—for example, one using the generic built-in-wrapping technique shown in **"Python built-ins"**. Such a rebinding affects all **`import`** and **`from`** statements that execute after the rebinding and thus can have an undesired global impact. A custom importer built by rebinding `__import__` must implement the same interface and semantics as the built-in `__import__`, and, in particular, it is responsible for supporting the correct use of `sys.modules`.

AVOID REBINDING THE BUILT-IN __IMPORT__

While rebinding `__import__` may initially look like an attractive approach, in most cases where custom importers are necessary, you're better off implementing them via *import hooks* (discussed next).

Import hooks

Python offers rich support for selectively changing the details of imports' behavior. Custom importers are an advanced and rarely called for technique, yet some applications may need them for purposes such as importing code from archives other than ZIP files, databases, network servers, and so on.

The most suitable approach for such highly advanced needs is to record *importer factory* callables as items in the `meta_path` and/or `path_hooks` attributes of the module `sys`, as detailed in **PEP 451**. This is how Python hooks up the standard library module `zipimport` to allow seamless importing of modules from ZIP files, as previously mentioned. A full study of the details of PEP 451 is indispensable for any substantial use of `sys.path_hooks` and friends, but here's a toy-level example to help understand the possibilities, should you ever need them.

Suppose that, while developing the first outline of some program, we want to be able to use **`import`** statements for modules that we haven't written yet, getting just messages (and empty modules) as a consequence. We can obtain such functionality (leaving aside the complexities connected with packages, and dealing with simple modules only) by coding a custom importer module as follows:

```
import sys, types
class ImporterAndLoader:
    """importer and loader can be a single class"""
    fake_path = '!dummy!'
    def __init__(self, path):
        # only handle our own fake-path marker
        if path != self.fake_path:
            raise ImportError
    def find_module(self, fullname):
        # don't even try to handle any qualified module name
        if '.' in fullname:
            return None
        return self
    def create_module(self, spec):
```

```
            # returning None will have Python fall back and
            # create the module "the default way"
            return None
        def exec_module(self, mod):
            # populate the already initialized module
            # just print a message in this toy example
            print(f'NOTE: module {mod!r} not yet written')
sys.path_hooks.append(ImporterAndLoader)
sys.path.append(ImporterAndLoader.fake_path)
if __name__ == '__main__':      # self-test when run as main script
    import missing_module       # importing a simple *missing* module
    print(missing_module)       # ...should succeed
    print(sys.modules.get('missing_module'))  # ...should also succeed
```

We just wrote trivial versions of `create_module` (which in this case just returns `None`, asking the system to create the module object in the "default way") and `exec_module` (which receives the module object already initialized with dunder attributes, and whose task would normally be to populate it appropriately).

We could, alternatively, have used the powerful new *module spec* concept, as detailed in PEP 451. However, that requires the standard library module `importlib`; for this toy example, we don't need all that extra power. Therefore, we chose instead to implement the method `find_module`, which, although now deprecated, still works fine for backward compatibility.

Packages

As mentioned at the beginning of this chapter, a *package* is a module containing other modules. Some or all of the modules in a package may be *subpackages*, resulting in a hierarchical tree-like structure. A package named *P* typically resides in a subdirectory, also called *P*, of some directory in `sys.path`. Packages can also live in ZIP files; in this section we explain the case in which the package lives on the filesystem, but the case in which a package is in a ZIP file is similar, relying on the hierarchical filesystem-like structure within the ZIP file.

The module body of *P* is in the file *P/__init__.py*. This file *must* exist (except in the case of namespace packages, described in **PEP 420**), even if it's empty (representing an empty module body), in order to tell Python that directory *P* is indeed a package. Python loads the module body of a package when you first import the package (or any of the package's modules), just like with any other Python module. The other *.py* files in the directory *P* are the modules of package *P*. Subdirectories of *P* containing *__init__.py* files are subpackages of *P*. Nesting can proceed to any depth.

You can import a module named *M* in package *P* as *P.M*. More dots let you navigate a hierarchical package structure. (A package's module body always loads *before* any module in the package.) If you use the syntax **`import`** *P.M*, the variable *P* is bound to the module object of package *P*, and the attribute *M* of object *P* is bound to the module *P.M*. If you use the syntax **`import`** *P.M* as *V*, the variable *V* is bound directly to the module *P.M*.

Using **`from`** *P* **`import`** *M* to import a specific module *M* from package *P* is a perfectly acceptable and indeed highly recommended practice: the **`from`** statement is specifically OK in this case. **`from`** *P* **`import`** *M* **`as`** *V* is also just fine, and exactly equivalent to **`import`** *P.M* **`as`** *V*. You can also use *relative* paths: that is, module *M* in package *P* can import its "sibling" module *X* (also in package *P*) with **`from . import`** X.

SHARING OBJECTS AMONG MODULES IN A PACKAGE

The simplest, cleanest way to share objects (e.g., functions or constants) among modules in a package *P* is to group the shared objects in a module conventionally named *common.py*. That way, you can use **`from . import`** `common` in every module in the package that needs to access some of the common objects, and then refer to the objects as `common.`*f*, `common.`*K*, and so on.

Special Attributes of Package Objects

A package *P*'s `__file__` attribute is the string that is the path of *P*'s module body—that is, the path of the file *P/__init__.py*. *P*'s `__package__` attribute is the name of *P*'s package.

A package *P*'s module body—that is, the Python source that is in the file *P/__init__.py*—can optionally set a global variable named `__all__` (just like any other module can) to control what happens if some other Python code executes the statement **`from`** *P* **`import`** `*`. In particular, if `__all__` is not set, **`from`** *P* **`import`** `*` does not import *P*'s modules, but only names that are set in *P*'s module body and lack a leading _. In any case, this is *not* recommended usage.

A package *P*'s `__path__` attribute is the list of strings that are the paths to the directories from which *P*'s modules and subpackages are loaded. Initially, Python sets `__path__` to a list with a single element: the path of the directory containing the file *__init__.py* that is the module body of the package. Your code can modify this list to affect future searches for modules and subpackages of this package. This advanced technique is rarely necessary, but can be useful when you want to place a package's modules in multiple directories (a namespace package is, however, the usual way to accomplish this goal).

Absolute Versus Relative Imports

As mentioned previously, an **`import`** statement normally expects to find its target somewhere on `sys.path`—a behavior known as an *absolute* import. Alternatively, you can explicitly use a *relative* import, meaning an import of an object from within the current package. Using relative imports can make it easier for you to refactor or restructure the subpackages within your package. Relative imports use module or package names beginning with one or more dots, and are only available within the **`from`** statement. **`from . import`** *X* looks for the module or object named *X* in the current package; **`from .`***X* **`import`** *y* looks in module or subpackage *X* within the current package for the module or object named *y*. If your package has subpackages, their code can access higher-up objects in the package by using multiple dots at the start of the module or subpackage name you place between **`from`** and **`import`**. Each additional dot ascends the directory hierarchy one level. Getting too fancy with this feature can easily damage your code's clarity, so use it with care, and only when necessary.

Distribution Utilities (distutils) and setuptools

Python modules, extensions, and applications can be packaged and distributed in several forms:

Compressed archive files
: Generally *.zip*, *.tar.gz* (aka *.tgz*), *.tar.bz2*, or *.tar.xz* files—all these forms are portable, and many other forms of compressed archives of trees of files and directories exist

Self-unpacking or self-installing executables
: Normally *.exe* for Windows

Self-contained, ready-to-run executables that require no installation
: For example, *.exe* for Windows, ZIP archives with a short script prefix on Unix, *.app* for the Mac, and so on

Platform-specific installers
: For example, *.rpm* and *.srpm* on many Linux distributions, *.deb* on Debian GNU/Linux and Ubuntu, *.pkg* on macOS

Python wheels
: Popular third-party extensions, covered in the following note

---

PYTHON WHEELS

A Python *wheel* is an archive file including structured metadata as well as Python code. Wheels offer an excellent way to package and distribute your Python packages, and `setuptools` (with the `wheel` extension, easily installed with **`pip install wheel`**) works seamlessly with them. Read all about them at **PythonWheels.com** and in **Chapter 24** (available **online**).

When you distribute a package as a self-installing executable or platform-specific installer, a user simply runs the installer. How to run such a program depends on the platform, but it doesn't matter which language the program was written in. We cover building self-contained, runnable executables for various platforms in **Chapter 24**.

When you distribute a package as an archive file or as an executable that unpacks but does not install itself, it *does* matter that the package was coded in Python. In this case, the user must first unpack the archive file into some appropriate directory, say *C:\Temp\MyPack* on a Windows machine or *~/MyPack* on a Unix-like machine. Among the extracted files there should be a script, conventionally named *setup.py*, that uses the Python facility known as the *distribution utilities* (the now deprecated, but still functioning, standard library package `distutils`[2]) or, better, the more popular, modern, and powerful third-party package **setuptools**. The distributed package is then almost as easy to install as a self-installing executable; the user simply opens a command prompt window, changes to the directory into which the archive is unpacked, then runs, for example:

```
C:\Temp\MyPack> python setup.py install
```

(Another, often preferable, option is to use `pip`; we'll describe that momentarily.) The *setup.py* script run with this **`install`** command installs the package as a part of the user's Python installation, according to the options specified by the package's author in the setup script. Of course, the user needs appropriate permissions to write into the directories of the Python installation, so permission-raising commands such as `sudo` may also be needed; or, better yet, you can install into a *virtual environment*, as described in the next section. `distutils` and `setuptools`, by default, print some information when the user runs *setup.py*. Including the option **`--quiet`** right before the **`install`** command hides most details (the user still sees error messages, if any). The following command gives detailed help on `distutils` or `setuptools`, depending on which toolset the package author used in their *setup.py*:

```
C:\Temp\MyPack> python setup.py --help
```

An alternative to this process, and the preferred way to install packages nowadays, is to use the excellent installer `pip` that comes with Python.

`pip`—a recursive acronym for "`pip` installs packages"—is copiously documented **online**, yet very simple to use in most cases. **`pip install`** ***`package`*** finds the online version of *`package`* (usually in the huge **PyPI** repository, hosting more than 400,000 packages at the time of this writing), downloads it, and installs it for you (in a virtual environment, if one is active; see the next section for details). This books' authors have been using that simple, powerful approach for well over 90% of their installs for quite a while now.

Even if you have downloaded the package locally (say to */tmp/mypack*), for whatever reason (maybe it's not on PyPI, or you're trying out an experimental version that is not yet there), `pip` can still install it for you: just run **`pip install --no-index --find-links=/tmp/mypack`** and `pip` does the rest.

Python Environments

A typical Python programmer works on several projects concurrently, each with its own list of dependencies (typically, third-party libraries and data files). When the dependencies for all projects are installed into the same Python interpreter, it is very difficult to determine which projects use which dependencies, and impossible to handle projects with conflicting versions of certain dependencies.

Early Python interpreters were built on the assumption that each computer system would have "a Python interpreter" installed on it, to be used to run any Python program on that system. Operating system distributions soon started to include Python in their base installations, but, because Python has always been under active development, users often complained that they would like to use a more up-to-date version of the language than the one their operating system provided.

Techniques arose to let multiple versions of the language be installed on a system, but installation of third-party software remained nonstandard and intrusive. This problem was eased by the introduction of the *site-packages* directory as the repository for modules added to a Python in-

stallation, but it was still not possible to maintain multiple projects with conflicting requirements using the same interpreter.

Programmers accustomed to command-line operations are familiar with the concept of a *shell environment*. A shell program running in a process has a current directory, variables that you can set with shell commands (very similar to a Python namespace), and various other pieces of process-specific state data. Python programs have access to the shell environment through `os.environ`.

Various aspects of the shell environment affect Python's operation, as mentioned in **"Environment Variables"**. For example, the `PATH` environment variable determines which program, exactly, executes in response to **`python`** and other commands. You can think of those aspects of your shell environment that affect Python's operation as your *Python environment*. By modifying it you can determine which Python interpreter runs in response to the **`python`** command, which packages and modules are available under certain names, and so on.

LEAVE THE SYSTEM'S PYTHON TO THE SYSTEM

We recommend taking control of your Python environment. In particular, do not build applications on top of a system-distributed Python. Instead, install another Python distribution independently, and adjust your shell environment so that the **`python`** command runs your locally installed Python rather than the system's Python.

Enter the Virtual Environment

The introduction of the `pip` utility created a simple way to install (and, for the first time, to uninstall) packages and modules in a Python environment. Modifying the system Python's *site-packages* still requires administrative privileges, and hence so does `pip` (although it can optionally install somewhere other than *site-packages*). Modules installed in the central *site-packages* are visible to all programs.

The missing piece is the ability to make controlled changes to the Python environment, to direct the use of a specific interpreter and a specific set of Python libraries. That functionality is just what *virtual environments* (*virtualenvs*) give you. Creating a virtualenv based on a specific Python interpreter copies or links to components from that interpreter's installation. Critically, though, each one has its own *site-packages* directory, into which you can install the Python resources of your choice.

Creating a virtualenv is *much* simpler than installing Python, and requires far less system resources (a typical newly created virtualenv takes up less than 20 MB). You can easily create and activate virtualenvs on demand, and deactivate and destroy them just as easily. You can activate and deactivate a virtualenv as many times as you like during its lifetime, and if necessary use `pip` to update the installed resources. When you are done with it, removing its directory tree reclaims all storage occupied by the virtualenv. A virtualenv's lifetime can span minutes or months.

What Is a Virtual Environment?

A virtualenv is essentially a self-contained subset of your Python environment that you can switch in or out on demand. For a Python 3.*x* interpreter it includes, among other things, a *bin* directory containing a Python 3.*x* interpreter, and a *lib/python3.x/site-packages* directory containing preinstalled versions of `easy-install`, `pip`, `pkg_resources`, and `setuptools`. Maintaining separate copies of these important distribution-related resources lets you update them as necessary rather than forcing you to rely on the base Python distribution.

A virtualenv has its own copies of (on Windows) or symbolic links to (on other platforms) Python distribution files. It adjusts the values of `sys.prefix` and `sys.exec_prefix`, from which the interpreter and various installation utilities determine the locations of some libraries. This means that `pip` can install dependencies in isolation from other environments, in the virtualenv's *site-packages* directory. In effect, the virtualenv redefines which interpreter runs when you run the **`python`** command and which libraries are available to it, but leaves most aspects of your Python environment (such as the `PYTHONPATH` and `PYTHONHOME` variables) alone.

Since its changes affect your shell environment, they also affect any subshells in which you run commands.

With separate virtualenvs you can, for example, test two different versions of the same library with a project, or test your project with multiple versions of Python. You can also add dependencies to your Python projects without needing any special privileges, since you normally create your virtualenvs somewhere you have write permission.

The modern way to deal with virtualenvs is with the `venv` module of the standard library: just run **`python -m venv`** *`envpath`*.

Creating and Deleting Virtual Environments

The command **`python -m venv`** *`envpath`* creates a virtual environment (in the *envpath* directory, which it also creates if necessary) based on the Python interpreter used to run the command. You can give multiple directory arguments to create, with a single command, several virtual environments (running the same Python interpreter); you can then install different sets of dependencies in each virtualenv. `venv` can take a number of options, as shown in **Table 7-1**.

Table 7-1. `venv` options

Option	Purpose
`--clear`	Removes any existing directory content before installing the virtual environment
`--copies`	Installs files by copying on the Unix-like platforms where using symbolic links is the default
`--h` or **`--help`**	Prints out a command-line summary and a list of available options
`--symlinks`	Installs files by using symbolic links on platforms where copying is the system default

Option	Purpose
`--system-site-packages`	Adds the standard system *site-packages* directory to the environment's search path, making modules already installed in the base Python available inside the environment
`--upgrade`	Installs the running Python in the virtual environment, replacing whichever version had originally created the environment
`--without-pip`	Inhibits the usual behavior of calling `ensurepip` to bootstrap the `pip` installer utility into the environment

---

KNOW WHICH PYTHON YOU'RE RUNNING

When you enter the command **`python`** at the command line, your shell has rules (which differ among Windows, Linux, and macOS) that determine which program you run. If you are clear on those rules, you always know which interpreter you are using.

Using **`python -m venv`** *`directory_path`* to create a virtual environment guarantees that it's based on the same Python version as the interpreter used to create it. Similarly, using **`python -m pip`** *`package_name`* will install the package for the interpreter associated with the **`python`** command. Activating a virtual environment changes the association with the **`python`** command: this is the simplest way to ensure packages are installed into the virtual environment.

The following Unix terminal session shows the creation of a virtualenv and the structure of the created directory tree. The listing of the *bin* subdirectory shows that this particular user, by default, uses an interpreter installed in */usr/local/bin*.[3]

```
$ python3 -m venv /tmp/tempenv
```

```
$ tree -dL 4 /tmp/tempenv
/tmp/tempenv
|--- bin
|--- include
|___ lib
     |___ python3.5
          |___ site-packages
               |--- __pycache__
               |--- pip
               |--- pip-8.1.1.dist-info
               |--- pkg_resources
               |--- setuptools
               |___ setuptools-20.10.1.dist-info

11 directories
$ ls -l /tmp/tempenv/bin/
total 80
-rw-r--r-- 1 sh wheel 2134 Oct 24 15:26 activate
-rw-r--r-- 1 sh wheel 1250 Oct 24 15:26 activate.csh
-rw-r--r-- 1 sh wheel 2388 Oct 24 15:26 activate.fish
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install-3.5
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3.5
lrwxr-xr-x 1 sh wheel    7 Oct 24 15:26 python->python3
lrwxr-xr-x 1 sh wheel   22 Oct 24 15:26 python3->/usr/local/bin/python3
```

Deleting a virtualenv is as simple as removing the directory in which it resides (and all subdirectories and files in the tree: **`rm -rf`** *envpath* in Unix-like systems). Ease of removal is a helpful aspect of using virtualenvs.

The `venv` module includes features to help the programmed creation of tailored environments (e.g., by preinstalling certain modules in the environment or performing other post-creation steps). It is comprehensively documented **online**; we do not cover the API further in this book.

Working with Virtual Environments

To use a virtualenv, you *activate* it from your normal shell environment. Only one virtualenv can be active at a time—activations don't "stack" like function calls. Activation tells your Python environment to use the virtualenv's Python interpreter and *site-packages* (along with the interpreter's full standard library). When you want to stop using those dependencies, deactivate the virtualenv, and your standard Python environment is once again available. The virtualenv directory tree continues to exist until deleted, so you can activate and deactivate it at will.

Activating a virtualenv in Unix-based environments requires using the **`source`** shell command so that the commands in the activation script make changes to the current shell environment. Simply running the script would mean its commands were executed in a subshell, and the changes would be lost when the subshell terminated. For bash, `zsh`, and similar shells, you activate an environment located at path *`envpath`* with the command:

```
$ source envpath/bin/activate
```

or:

```
$ . envpath/bin/activate
```

Users of other shells are supported by the scripts *activate.csh* and *activate.fish*, located in the same directory. On Windows systems, use *activate.bat* (or, if using Powershell, *Activate.ps1*):

```
C:\> envpath/Scripts/activate.bat
```

Activation does many things. Most importantly, it:

- Adds the virtualenv's *bin* directory at the beginning of the shell's `PATH` environment variable, so its commands get run in preference to anything of the same name already on the `PATH`
- Defines a `deactivate` command to remove all effects of activation and return the Python environment to its former state
- Modifies the shell prompt to include the virtualenv's name at the start
- Defines a `VIRTUAL_ENV` environment variable as the path to the virtualenv's root directory (scripts can use this to introspect the virtualenv)

As a result of these actions, once a virtualenv is activated, the **`python`** command runs the interpreter associated with that virtualenv. The interpreter sees the libraries (modules and packages) installed in that environment, and `pip`—now the one from the virtualenv, since installing the module also installed the command in the virtualenv's *bin* directory—by default installs new packages and modules in the environment's *site-packages* directory.

Those new to virtualenvs should understand that a virtualenv is not tied to any project directory. It's perfectly possible to work on several projects, each with its own source tree, using the same virtualenv. Activate it, then move around your filesystem as necessary to accomplish your programming tasks, with the same libraries available (because the virtualenv determines the Python environment).

When you want to disable the virtualenv and stop using that set of resources, simply issue the command **`deactivate`**. This undoes the changes made on activation, removing the virtualenv's *bin* directory from your `PATH`, so the **`python`** command once again runs your usual interpreter. As long as you don't delete it, the virtualenv remains available for future use: just repeat the command to activate it.

DO NOT USE PY –3.X IN A VIRTUALENV ON WINDOWS

The Windows py launcher provides mixed support for virtualenvs. It makes it very easy to define a virtualenv using a specific Python version, using a command like the following:

```
C:\> py -3.7 -m venv C:\path\to\new_virtualenv
```

This creates a new virtualenv, running the installed Python 3.7.

Once activated, you can run the Python interpreter in the virtualenv using either the **`python`** command or the bare **`py`** command with no version specified. However, if you specify the **`py`** command using a version option, even if it is the same version used to construct the virtualenv, you will *not* run the *virtualenv* Python. Instead, you will run the corresponding *system-installed* version of Python.

Managing Dependency Requirements

Since virtualenvs were designed to complement installation with `pip`, it should come as no surprise that `pip` is the preferred way to maintain dependencies in a virtualenv. Because `pip` is already extensively documented, we mention only enough here to demonstrate its advantages in virtual environments. Having created a virtualenv, activated it, and installed dependencies, you can use the **`pip freeze`** command to learn the exact versions of those dependencies:

```
(tempenv) $ pip freeze
appnope==0.1.0
decorator==4.0.10
ipython==5.1.0
ipython-genutils==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.8
ptyprocess==0.5.1
Pygments==2.1.3
```

```
requests==2.11.1
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.1
wcwidth==0.1.7
```

If you redirect the output of this command to a file called *filename*, you can re-create the same set of dependencies in a different virtualenv with the command **`pip install -r`** *`filename`*.

When distributing code for use by others, Python developers conventionally include a *requirements.txt* file listing the necessary dependencies. `pip` installs any indicated dependencies along with the packages you request when you install software from PyPI. While you're developing software it's also convenient to have a requirements file, as you can use it to add the necessary dependencies to the active virtualenv (unless they are already installed) with a simple **`pip install -r requirements.txt`**.

To maintain the same set of dependencies in several virtualenvs, use the same requirements file to add dependencies to each one. This is a convenient way to develop projects to run on multiple Python versions: create virtualenvs based on each of your required versions, then install from the same requirements file in each. While the preceding example uses exactly versioned dependency specifications as produced by **`pip freeze`**, in practice you can specify dependencies and version requirements in quite complex ways; see the **documentation** for details.

Other Environment Management Solutions

Python virtual environments are focused on providing an isolated Python interpreter, into which you can install dependencies for one or more Python applications. The **`virtualenv`** package was the original way to create and manage virtualenvs. It has extensive facilities, including the ability to create environments from any available Python interpreter. Now maintained by the Python Packaging Authority team, a subset of its functionality has been extracted as the standard library `venv` module covered earlier, but `virtualenv` is worth learning about if you need more control.

The **pipenv** package is another dependency manager for Python environments. It maintains virtual environments whose contents are recorded in a file named *Pipfile*. Much in the manner of similar JavaScript tools, it provides deterministic environments through the use of a *Pipfile.lock* file, allowing the exact same dependencies to be deployed as in the original installation.

conda, mentioned in **"Anaconda and Miniconda"**, has a rather broader scope and can provide package, environment, and dependency management for any language. conda is written in Python, and installs its own Python interpreter in the base environment. Whereas a standard Python virtualenv normally uses the Python interpreter with which it was created; in conda, Python itself (when it is included in the environment) is simply another dependency. This makes it practical to update the version of Python used in the environment, if necessary. You can also, if you wish, use pip to install packages in a Python-based conda environment. conda can dump an environment's contents as a YAML file, and you can use the file to replicate the environment elsewhere.

Because of its additional flexibility, coupled with comprehensive open source support led by its originator, Anaconda, Inc. (formerly Continuum), conda is widely used in academic environments, particularly in data science and engineering, artificial intelligence, and financial analytics. It installs software from what it calls *channels*. The default channel maintained by Anaconda contains a wide range of packages, and third parties maintain specialized channels (such as the *bioconda* channel for bioinformatics software). There is also a community-based ***conda-forge* channel**, open to anyone who wants to join up and add software. Signing up for an account on **Anaconda.org** lets you create your own channel and distribute software through the *conda-forge* channel.

Best Practices with Virtualenvs

There is remarkably little advice on how best to manage your work with virtualenvs, though there are several sound tutorials: any good search engine will give you access to the most current ones. We can, however, offer

a modest amount of advice that we hope will help you to get the most out of virtual environments.

When you are working with the same dependencies in multiple Python versions, it is useful to indicate the version in the environment name and use a common prefix. So, for the project *mutex* you might maintain environments called *mutex_39* and *mutex_310* for development under two different versions of Python. When it's obvious which Python is involved (remember, you see the environment name in your shell prompt), there's less risk of testing with the wrong version. You can maintain dependencies using common requirements to control resource installation in both.

Keep the requirements file(s) under source control, not the whole environment. Given the requirements file it's easy to re-create a virtualenv, which depends only on the Python release and the requirements. You distribute your project, and let your users decide which version(s) of Python to run it on and create the appropriate virtual environment(s).

Keep your virtualenvs outside your project directories. This avoids the need to explicitly force source code control systems to ignore them. It really doesn't matter where else you store them.

Your Python environment is independent of your projects' locations in the filesystem. You can activate a virtual environment and then switch branches and move around a change-controlled source tree to use it wherever is convenient.

To investigate a new module or package, create and activate a new virtualenv and then **`pip install`** the resources that interest you. You can play with this new environment to your heart's content, confident in the knowledge that you won't be installing unwanted dependencies into other projects.

You may find that experiments in a virtualenv require installation of resources that aren't currently project requirements. Rather than "pollute" your development environment, fork it: create a new virtualenv from the same requirements plus the testing functionality. Later, to make these

changes permanent, use change control to merge your source and requirements changes back in from the forked branch.

If you are so inclined, you can create virtual environments based on debug builds of Python, giving you access to a wealth of information about the performance of your Python code (and, of course, of the interpreter itself).

Developing a virtual environment also requires change control, and the ease of virtualenv creation helps here too. Suppose that you recently released version 4.3 of a module, and you want to test your code with new versions of two of its dependencies. You *could*, with sufficient skill, persuade `pip` to replace the existing copies of dependencies in your existing virtualenv. It's much easier, though, to branch your project using source control tools, update the requirements, and create an entirely new virtual environment based on the updated requirements. The original virtualenv remains intact, and you can switch between virtualenvs to investigate specific aspects of any migration issues that might arise. Once you have adjusted your code so that all tests pass with the updated dependencies, you can check in your code *and* requirement changes and merge into version 4.4 to complete the update, advising your colleagues that your code is now ready for the updated versions of the dependencies.

Virtual environments won't solve all of a Python programmer's problems: tools can always be made more sophisticated, or more general. But, by golly, virtualenvs work, and we should take all the advantage of them that we can.

1 One of our tech reviewers reports that *.pyw* files on Windows are an exception to this.

2 `distutils` is scheduled for deletion in Python 3.12.

3 When running these commands on reduced-footprint Linux distributions, you may need to separately install `venv` or other supporting packages first.