Chapter 24. Module Packages

So far, when we've imported modules, we've been loading *files*. This represents typical module usage, and it's probably the technique you'll use for most imports you'll code, especially early in your Python career. The module import story, though, is richer than implied up to this point. This chapter extends it to present module *packages*—collections of module files that normally correspond to *folders* (a.k.a. *directories*) on your device. It covers four topics:

- Package imports, which give part of a folder path leading to a file
- Packages themselves, which organize modules into folder bundles
- Package-relative imports, which use dots within a package to limit search
- Namespace packages, which build a package that may span multiple folders

As you'll find, a package import turns a folder on your computer into another Python namespace, with attributes corresponding to the module files and subfolders that the folder contains. As you'll also learn, package imports are sometimes required to resolve ambiguity when multiple program files of the same name are installed on a device.

Packages are a somewhat advanced topic, which many readers can defer until they gain experience with file-based modules. That said, packages provide an easy way to organize code files that avoids same-name conflicts and is employed by many standard-library and third-party tools that you'll be using. While packages, like much in Python, have grown convoluted over time, a basic knowledge of their usage can be beneficial to most Python learners.

Using Packages

To load items in a package folder, you'll use the normal import statements and tools we've already met, but provide a path of names that reflects a path of nested folders. Let's get started with this user-guide side of the packages story.

Package Imports

Coding package imports is straightforward. In all the places where you have been naming a simple *file* in your import operations, you can instead list a *path* of names separated by periods. For instance, this works in an `import` statement:

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

The same goes for `from` statements:

```
from dir1.dir2.mod import var
```

And this also applies to already-imported items in `reload` calls:

```
from importlib import reload
reload(dir1.dir2.mod)
```

The "dotted path" in these contexts is assumed to correspond to a path through the *folder hierarchy* on your device, leading to the module file *mod.py*, or other component with basename *mod* (as we've seen, it might also be a bytecode file, C extension module, or other). Most importantly here, the preceding paths indicate that on your device there is a directory *dir1*, which has a subdirectory *dir2*, which contains a module file *mod.py* (or similar).

Furthermore, these paths imply that *dir1* resides within some *container* directory *dir0*, which is a part of the normal module search path. In other words, these imports and reloads imply a directory structure that looks something like this on Unix and Windows, respectively:

```
dir0/dir1/dir2/mod.py            # Or mod.pyc, mod.so,
dir0\dir1\dir2\mod.py            # Ditto, on Windows
```

The container directory *dir0* needs to be added to your module search path unless it's an automatic part of that path, exactly as if *dir1* were a simple module file.

More formally, the leftmost component in a package import path is *relative to* (located in) a directory included in the `sys.path` module search-path list we explored in Chapter 22. From there down, though, the import statements in your script explicitly give the directory paths leading to modules in packages.

The dotted-path syntax of packages is platform neutral, but also reflects the fact that folder paths in import statements become nested objects: `dir1.dir2.mod` traverses three module *objects* after an import. This syntax also explains why Python complains about a file not being a package if you forget to omit the *.py* extension in import statements: it's taken to mean a package import!

Packages and the Module Search Path

If you use this feature, keep in mind that the directory paths in your import statements can be only *variables* separated by *periods*. You cannot use any platform-specific path syntax in your import statements, such as `C:\dir1`, `/Users/me/dir1`, or `../dir1`—these do not work syntactically. Instead, use any such platform-specific syntax in your module search path settings to name the directories *containing* your packages.

For instance, in the prior example, *dir0*—the directory name you add to your module search path—can be an arbitrarily long and platform-specific directory path leading up to *dir1*. You cannot use an invalid statement like this:

```
import C:\Users\me\mycode\dir1\dir2\mod      # Error: i
import /Users/me/mycode/dir1/dir2/mod        # Ditto, c
```

But you can add a path like *C:\Users\me\mycode* or */Users/me/mycode* to either your `PYTHONPATH` environment variable, a strategically placed *.pth* file, or `sys.path` itself in manual code, and then say this in your script:

```
import dir1.dir2.mod                         # OK: vari
```

In effect, entries on the module search path provide platform-specific directory *prefixes*, which lead to the leftmost names of package paths in `import` and `from` statements. These import statements themselves provide the remainder of the directory path in a platform-neutral fashion.

As for simple file imports, you do not need to add the *container* directory to your module search path if it's already there. Per Chapter 22, the search path automatically includes the "home" directory (where you're working in a REPL, or the container of a launched program's top-level file), along with the standard library's containers, and the *site-packages* third-party install root. While none of these components require search-path mods, your module search path must include all the directories containing *leftmost* components in your code's package-import statements.

Creating Packages

To make packages of your own, you'll bundle module files and nested folders in a package folder. A package folder can also optionally contain an *__init__.py* file run on first import, and a *__main__.py* file run when the entire package folder is run as a bundle. The following sections demo what this looks like in code, one step at a time.

Basic Package Structure

In their simplest form, packages are just folders containing normal module files, and perhaps nested subfolders of the same. Let's set one up as a demo. The following uses indentation to represent the folder nesting we'll be using in this and the following sections:

```
dir0/                         # Container listed on modu
    dir1/                     # Package root folder dir1
        mod.py                # Module file dir1.mod in
        dir2/                 # Nested package folder di
            mod.py            # Module dir1.dir2.mod in
```

These nested folders can be created in file explorers, or with the following commands on most platforms; use backslashes on Windows, or simply use this Chapter's folder in the examples package, which has prebuilt the structure:

```
$ mkdir dir1
$ mkdir dir1/dir2            # Use backward slashes on
```

Admin note: the names of package folders, like those of simple module files, must follow the rules for *variable* names because they become variables where imported. See Chapter 11 for a refresher on the constraints, but in short, use letters, digits, and underscores, and avoid reserved words.

Now, add the nested module files listed in Examples 24-1 and 24-2. Their top-level code runs on first import and creates attributes as usual, and their titles give their paths (using just Unix forward-slash separators for brevity).

Example 24-1. dir1/mod.py

```
var = 'hack'
print('Loading dir1.mod')
```

Example 24-2. dir1/dir2/mod.py

```
var = 'code'
print('Loading dir1.dir2.mod')
```

Using the basic package

To use our module package, import its modules from a REPL or another file just as you would for a simple *.py* file, but use dots to specify the path below the *dir0* folder in the search path—which is the current directory in a REPL:

```
$ python3
>>> import dir1.mod                      # Package path
Loading dir1.mod
>>> dir1.mod.var                         # Module code
'hack'

>>> import dir1.dir2.mod                 # A further-ne
Loading dir1.dir2.mod
>>> dir1.dir2.mod.var                    # Repeat path
'code'
```

As for simple top-level modules, the code of modules nested in a package is run only when first imported:

```
>>> import dir1.mod                              # No-op if alr
>>> import dir1.dir2.mod
```

And you generally must *import* a nested module in order to use its attributes—because modules corresponding to folders don't go back to the filesystem on attribute fetches, it's not enough to import just a root folder:

```
$ python3
>>> import dir1                                  # Gets dir1, r
>>> dir1.dir2.mod.var
AttributeError: module 'dir1' has no attribute 'dir2'

$ python3
>>> import dir1.dir2
>>> dir1.dir2.mod.var
AttributeError: module 'dir1.dir2' has no attribute 'mc

$ python3
>>> import dir1.dir2.mod                         # List full pc
Loading dir1.dir2.mod
>>> dir1.dir2.mod.var
'code'
```

All this works the same in the `from` statement—which, as we've seen, is really just `import` with an extra copy of names. Again, though, we must list a folder by its path in an import statement to be able to access its contents, and the paths listed in imports are taken literally, not fetched from variables:

```
$ python3
>>> from dir1.mod import var                     # Package path
Loading dir1.mod
>>> var                                          # Don't repeat
'hack'

>>> from dir1.dir2.mod import var
Loading dir1.dir2.mod
>>> var
'code'
```

```
>>> from dir1 import dir2                # Paths taken
>>> from dir2 import mod                 # Not from var
ModuleNotFoundError: No module named 'dir2'
```

One potential advantage of `from` here is that it avoids *repeating* a package path every time an item in a package is used. With `import`, you generally must repeat the full path each time, but `from` lets you code the path just once, and use the simple name fetched from it everywhere; this is especially useful if the package's directory structure later changes (as software is wont to do):

```
>>> import dir1.dir2.mod                 # import requi
>>> dir1.dir2.mod.var
'code'

>>> from dir1.dir2.mod import var        # from can sho
>>> var
'code'
>>> from dir1.dir2 import mod            # And avoids r
>>> mod.var
'code'

>>> import dir1.dir2.mod as mod          # Though "as"
>>> mod.var
'code'
```

As the last example shows, though, the `as` extension introduced in the preceding chapter can shorten up paths the same way as `from`—as can a simple manual reassignment, which the `as` sugarcoats (more on `as` in the next chapter).

Package __init__.py Files

If you need to run setup code when a package folder is imported, put it in a file named *__init__.py* and store it in the package's folder. Python will automatically run this file's code the first time the package is imported during a program run (or REPL session). Here's how this augments our package's structure:

```
dir0/
    dir1/
        __init__.py          # Run on first import of d
        mod.py
        dir2/
            __init__.py      # Run on first import of d
            mod.py
```

The new *__init__.py* files are listed in Examples 24-3 and 24-4.

Example 24-3. dir1/__init__.py

```
var = 'Python'
print('Running dir1.__init__.py')
```

Example 24-4. dir1/dir2/__init__.py

```
var = 3.12
print('Running dir1.dir2.__init__.py')
```

Using the updated package

These special *__init__.py* files are optional in each folder of a package. Because they are run on the first import of or through a folder level, though, they provide a natural hook for kicking off package-specific initializations (hence their abbreviated names). In fact, their assignments serve to initialize the *namespace* that corresponds to a folder on your device—as for files, they create attributes of the package's module object:

```
$ python3
>>> import dir1.dir2.mod                    # Runs all __i
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod
>>> dir1.var                                # Name in __in
'Python'
>>> dir1.dir2.var
3.12
```

```
>>> dir1.dir2.mod.var                          # Name in nest
'code'
```

Technically speaking, packages with *__init__.py* files are called “regular” packages, and those without them are called “namespace” packages, and the demo was a single-folder “namespace” package until adding *__init__.py* made it “regular.” We’ll get into the details behind this distinction later, but apart from the fact that “regular” packages have precedence during module search, the difference is mostly a historical artifact today, and won’t matter in most roles. Of course, *__init__.py* files can also do more than print beacons; we’ll also explore their roles later in this chapter.

As noted earlier, `reload` works with package paths too, if you’ve already imported the paths in question. Continuing the prior REPL session:

```
>>> from importlib import reload
>>> reload(dir1)
Running dir1.__init__.py
<module 'dir1' from '/…/LP6E/Chapter24/dir1/__init__.py

>>> reload(dir1.dir2)
Running dir1.dir2.__init__.py
<module 'dir1.dir2' from '/…/LP6E/Chapter24/dir1/dir2/_

>>> reload(dir1.dir2.mod)
Loading dir1.dir2.mod
<module 'dir1.dir2.mod' from '/…/LP6E/Chapter24/dir1/di
```

Notice from the initialization prints that this call reloads just the *single* module on the right end of the path. To do better, we’ll code a reloader tool in the next chapter that, given a package root, can load all the items on a package path like this, one at a time; See “Example: Transitive Module Reloads”.

Package __main__.py Files

Finally, if you want a package’s users to be able to run the package as though it were a program or script, add *__main__.py* files to each folder where you wish to support this mode; Python will automatically run these files each time their container folder is launched like a program. Here’s how this last bit augments our package’s structure:

```
dir0/
    dir1/
        __main__.py          # Run whenever dir1 bundle
        __init__.py
        mod.py
        dir2/
            __main__.py      # Run whenever dir1.dir2 l
            __init__.py
            mod.py
```

The added *__main__.py* files are listed in Examples 24-5 and 24-6; they're simple so we can focus on packages.

Example 24-5. dir1/__main__.py

```
print('Executing dir1.__main__.py')
```

Example 24-6. dir1/dir2/__main__.py

```
print('Executing dir1.dir2.__main__.py')
```

Using the updated package

When present, *__main__.py* files are automatically run for a variety of launch options, including direct console command lines that name the containing package folder—which need not be on the module search path in this mode:

```
$ python3 dir1                              # Runs __main_
Executing dir1.__main__.py
$ python3 dir1/dir2
Executing dir1.dir2.__main__.py
$ python3 dir1/dir2/mod.py                  # Runs nested
Loading dir1.dir2.mod
```

Package *__main__.py* files are also run for the Python `-m` mode, which, as we've seen, locates an item on the module search path and runs it as a top-level script. Unlike direct console command lines, this mode kicks off the full package-import machinery and runs any *__init__.py* files along the path, but requires the package to be on the search path:

```
$ python3 -m dir1                        # Runs both __
Running dir1.__init__.py
Executing dir1.__main__.py
$ python3 -m dir1.dir2                   # And package
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Executing dir1.dir2.__main__.py
$ python3 -m dir1.dir2.mod               # Runs mod.py,
Running dir1.__init__.py
Running dir1.dir2.__init__.py
Loading dir1.dir2.mod
```

This is similar in spirit to app “bundles” on macOS and smartphones, which are really folders but can be run as an executable item. The roles for *__main__.py* files are many, but they’re often used to provide a command-line interface to a package of tools. This way, you can import the package to use its tools in another program, but also launch it as a whole to use the tools in a standalone program. This file might also be used to host test code for the package.

A *__main__.py* file is also run if located in a *ZIP file* on the search path. As noted earlier, ZIP files are treated like a normal folder if encountered during a module search, but see Python’s docs for more on this option.

One subtlety here: intrapackage (same-package) imports in *__main__.py* may need to use the package-relative `from .` syntax we’ll study ahead when run by the `-m` argument only. This syntax fails for launches from direct command lines, so you may need to choose a launch mode to support when a *__main__.py* requires same-package imports. As you’ll find, code that wants to support both package and program usage may solve this dilemma with full-path imports.

Why Packages?

Now that you’ve seen how to use and create packages, you might be wondering why in the world anyone would go to all the bother. It’s a valid question. In truth, and despite what you may have heard, packages are completely optional: they are probably overkill early in your coding journey, and even later, you can write amazing programs without them.

In general, though, packages are useful to know about because they help make names unique in both larger programs and libraries published for others to use. By organizing your code as a package folder, you can be fairly sure that the names of its files won't clash with those of other software on hosting devices. This is the same namespace-segregation role that file-based modules and local scopes in functions play, but extended to the filesystem: because references to your package are qualified by the name of the package's folder, there's less risk of same-name collisions.

In addition, packages can make imports more descriptive, ease the task of tracking down the locations of variables in code files, and simplify your module search-path settings. The only time package imports are actually *required*, however, is to resolve ambiguities that may arise when multiple programs with same-named files are installed on the same machine. This is something of an install issue, but it can also become a concern in general practice—especially given the tendency of developers to use simple and similar names for module files.

To help you better understand why all these benefits matter, let's take a brief detour into the land of the abstract.

A Tale of Two Systems

To illustrate the roles of packages, suppose that a programmer develops a Python program that contains a file called *utilities.py* for common utility code, and a top-level file named *main.py* that users launch to start the program. All over this program, its files say `import utilities` to load and use the common code. When the program is installed, it unpacks all its files into a single directory named *system1* on the target machine that looks like this:

```
system1/
    utilities.py        # Common utility functions, cl
    main.py             # Launch this to start the pro
    other.py            # Import utilities to load my 
```

Now, suppose that a second programmer develops a different program with files also called *utilities.py* and *main.py*, and again uses `import utilities` throughout the program to load its own common code file. When this second system is fetched and installed on the same computer as the first system, its files will unpack into a new directory called *system2* on the host—ensuring that they do not overwrite same-named files from the first system:

```
system2/
    utilities.py        # Common utilities
    main.py             # Launch this to run
    other.py            # Imports utilities
```

So far, there's no problem: both systems can coexist and run on the same computer. In fact, you won't even need to configure the module search path to use these programs on your computer—because Python always searches the home directory first (that is, the directory containing the top-level file), imports in either system's files will automatically see all the files in that system's own directory. For instance, if you run *system1/main.py*, all imports will search *system1* first. Similarly, if you launch *system2/main.py*, *system2* will be searched first instead. Remember, module search path settings are only needed to import across directory boundaries.

However, suppose that after you've installed these two programs on your machine, you decide that you'd like to use some of the code in each of the *utilities.py* files in a system of your own. It's common utility code, after all, and Python code by nature "wants" to be reused. In this case, you'd like to be able to say the following from code that you're writing in a third directory to load one of the two files:

```
import utilities
utilities.func('hack')
```

Now the problem starts to materialize. To make this work at all, you'll have to set the module search path to include the directories containing the *utilities.py* files. But which directory do you put first in the path—*system1* or *system2*?

The issue here is the *linear* nature of the `sys.path` search path. It is always scanned from left to right, so no matter how long you ponder this dilemma, you will always get just one *utilities.py*—from the directory listed *first* (leftmost) on the search path. As is, you'll never be able to import this file from the other directory at all.

You could try changing `sys.path` within your script before each import operation, but that's both extra work and error-prone. And changing `PYTHONPATH` before each Python program run is too tedious, and won't allow you to use *both* versions in a single file in an event. By default, you're stuck.

This is the issue that packages actually fix. Rather than installing programs in independent directories listed on the module search path directly and individually, you can package and install them as *subdirectories* under a common root. For instance, you might organize all the code in this example as an install hierarchy that looks like this:

```
root/
    system1/
        utilities.py
        main.py
        other.py
    system2/
        utilities.py
        main.py
        other.py
    mycode/                         # Here or elsewhere
        myfile.py                   # Your new code here
```

Now, add just the common *root* directory to your search path. If your code's imports are all relative to this common root, you can import *either* system's utility file with a package import—the enclosing directory name makes the path (and hence, the module reference) unique. In fact, you can import *both* utility files in the same module, if you use an `import` statement and repeat the full path each time you reference the utility modules:

```
import system1.utilities                # Import from c
import system2.utilities                # Import from c

system1.utilities.function('hack')      # And use names
system2.utilities.function('code')      # Or both!
```

In short, the names of the enclosing directories here make the module references unique.

Note that you have to use `import` instead of `from` with packages only if you need to access the *same* attribute name in two or more paths. If the name of the called function here were different in each path, you could use `from` statements to avoid repeating the full package path whenever you call one of the functions, as described earlier; as also mentioned, the `as` extension in `import` and `from` can be used to provide unique synonyms too.

Technically, in this case, the *mycode* folder doesn't have to be under *root*—just the packages of code from which you will import. Because you never know when your own modules might be useful in other programs, though, placing them under the common root directory, too, may avoid similar name-collision problems in the future.

Importantly, *path configuration* also becomes simple if you're careful to unpack all your Python systems under a common root like this; you'll only need to add the common root directory once. Moreover, both of the two original systems' imports will keep working unchanged. Because their *home* directories are searched first, the addition of the common root on the search path is irrelevant to code in *system1* and *system2*; they can keep saying just `import utilities` and expect to find their own files when run as programs. As you'll see ahead, though, if they are *also* used as packages, they may need to use `from .` package-relative imports (and *main.py* could be *__main__.py*).

Finally, keep in mind that even if you never create packages of your own, you'll probably use standard-library and third-party tools that do. As a random sample of standard-library packages:

```
from email.message import Message                   #
from tkinter.filedialog import askopenfilename      #
from http.server import CGIHTTPRequestHandler       #
```

By bundling their code in a package, such tools become more self-contained, and avoid name conflicts. Whether you opt to do the same for your own code is up to you, but the next few sections dig deeper for if and when you opt-in.

The Roles of __init__.py Files

Now that we've seen the basics and addressed the why of packages, let's explore some of the details behind their usage. The *__init__.py* files in our opening demo were simple, but these files can contain arbitrary Python code, just like normal module files. Their names are special because their code is run automatically the first time a Python program imports a directory, and thus serves primarily as a hook for performing initialization steps required by the package. These files can also be completely empty, though, and sometimes have additional roles.

Specifically, the *__init__.py* file serves as a hook for package initialization-time actions, generates a module namespace for a directory, declares a directory as a "normal" Python package, and implements the behavior of `from *` statements when used for folders in package imports:

Package initialization

> The first time a Python program imports through a directory, it automatically runs all the code in the directory's *__init__.py* file. Because of that, these files are a natural place to put code to initialize the state required by files in a package. For instance, a package might use its initialization file to create required data files, open connections to databases, and so on. Typically, *__init__.py* files are not meant to be useful if executed directly (that's what *__main__.py* is for); instead, they are run automatically when a package is first imported for use elsewhere.

Module namespace initialization

> As noted earlier, in the package import model, the directory paths in your script become real nested object paths after an import. For instance, after an import in the preceding demo, the expression `dir1.dir2` works and returns a module object whose namespace contains all the names assigned by *dir2*'s *__init__.py* initialization file. The variable assignments in such files provide attribute namespaces for module objects created for folders, which have no real associated module file, and would otherwise have no place to define names.

Package indicator for search

> Package *__init__.py* files are also partly present to declare that a directory is a Python package. In this role, these files serve to prevent directories with common names from unintentionally hiding true

modules that appear later on the module search path. Without this safeguard, Python might pick a directory that has nothing to do with your code, just because it appears nested in an earlier directory on the search path. As you'll see later, the generalizations added for namespace packages subsume some of this role algorithmically, by scanning ahead on the search path to find later items before settling on simple folders. Package *__init__.py* files, though, still give a package higher priority than a same-name folder elsewhere on the search path.

`from *` statement behavior

As an advanced feature, a list of name strings named `__all__` at the top level of an *__init__.py* file defines what is exported for the `from *` statement form. Specifically, the `__all__` list in an *__init__.py* file is taken to be the names of nested submodules that should be automatically imported when `from *` is used on the package directory itself. If `__all__` is not set, the `from *` does not automatically load submodules nested in the package directory; instead, it loads just names defined by assignments in the directory's *__init__.py* file, including any submodules explicitly imported by code in this file. For instance, `from submodule import X` in a directory's *__init__.py* makes the name `X` available in that directory's namespace without an `__all__`.

You'll see an example of the `__all__` list later, and more coverage of it in Chapter 25 where you'll find that it also serves to declare `from *` exports of simple file-based modules, not just packages, and is part of the larger topic of *data hiding*. You can also simply leave *__init__.py* files empty to give your package precedence over *namespace* packages during an import search, but to understand why that matters, we need to move ahead.

NOTE

Classes conflation caution: As a preview, package *__init__.py* files are not the same as the class `__init__` constructor methods you'll meet in the next part of this book. The former are files of code run when imports first step through a package folder in a program run, while the latter are functions called whenever an instance is created. Both have initialization roles and are optional, but they are otherwise very different—despite their names.

Package-Relative Imports

The coverage of package imports so far has focused on importing package files from *outside* the package. Within the package itself, imports of same-package files can use the same full path syntax as imports from outside the package—and as you'll see, sometimes should. However, files within a package can also make use of *intrapackage* syntax that makes imports *relative* to the package itself, and can ensure that imports in a package load the package's own files.

Relative and Absolute Imports

The code behind this is straightforward. Imports run by files used as part of a package can use special syntax like the following, which works only in files used as package components, and only in `from` statements, not `import`:

```
from . import module            # Import a module in thi
from .module import name        # Import a name from a m
from .. import module           # Import a module siblir
from ..module import name       # Import a name from a p
```

This syntax wouldn't make sense in `import`, because that statement assigns modules to simple names, not paths. In a `from`, though, when the source of the import begins with (or is only) dots like this, the import is known as *relative*—it identifies an item relative to the folder of the enclosing package itself.

This name comes from the similarity to relative *filename* paths, which reference a file per the current working directory (CWD), as covered in Chapter 9. Much as in filenames, "." means the immediately enclosing package folder, and ".." means its parent folder another level up (and each additional dot means one level higher, though this is rarely used). Here, though, the effect names an item relative to an importer's package, not a content file relative to the CWD.

Importantly, relative imports within a package search *only* the referenced package: they never check the folders listed in `sys.path` as normal imports do (and skip the built-in and frozen modules precheck). Moreover, relative-import syntax can be used *only* when a file is being utilized as part of a package, and fails otherwise. This has consequences both good and bad that we'll explore in a moment.

Imports in files being used as part of a package can still be coded without dots as usual, in both `import` and `from`:

```
import module                    # Import a module from c
from module import name          # Import a name from the
```

Sans leading periods like this, an import is known as *absolute*—it identifies an item that's located in a folder listed in `sys.path`. This is the normal behavior of imports in Python, and there's really nothing "absolute" about it per the filename analogy (absolute filename paths give a complete filesystem path). In fact, these "absolute" imports are really *relative* to an entry in the module search path, but we're stuck with the terminology today.

Also importantly, absolute imports within a package do *not* automatically check the package itself: they skip the package and move right to a search of folders on `sys.path`. As we've seen, `sys.path` may include the package's folder anyhow, by virtue of the REPL's CWD or the home folder of the top-level script, and PYTHONPATH or other settings. By themselves, though, absolute imports don't check the package for imports run in package files.

Relative-Import Rationales and Trade-Offs

So why the dots? In short, relative imports allow a package to ensure that its imports will load its *own* modules. Because relative imports search *only* the package itself, they won't inadvertently load a same-named but unrelated module elsewhere on the host that happens to be accessible through `sys.path`. This in turn makes the package more self-contained: without relative imports, such an unrelated module might break the package's code; with them, packages are less reliant on client search-path settings that they cannot predict or control.

The downside is that this model is an *all-or-nothing* proposition. You essentially must choose a mode for your files—package or program. Relative imports give visibility to the package itself, but cannot be used in nonpackage mode, and absolute imports can be used in nonpackage mode, but do not give visibility to the package itself. This combination seems a catch-22 that limits your code's utility.

That being said, if you're coding a library of tools, this may be a nonissue: you can adopt relative imports throughout your package, and provide a *__main__.py* file to run the package as a program with the `-m` switch. If you're writing a more traditional folder of code that you want to use arbitrarily, though, your options appear to be limited.

One easy solution to this dilemma is to simply *avoid* relative imports altogether, and use full, explicit, and "absolute" package-import paths everywhere in your code. That is, use imports that list the path to the desired item from and including the package's root folder itself:

```
import package.module                # Import a module
from package import module           # Same as: from .
form package.module import name      # Same as: from .m
```

For this to work, the package's root folder must reside in a folder listed on `sys.path`, but this will be the *normal* case—there's no way for clients to import the package's code otherwise. This also doesn't directly support exotic imports like ".." parent siblings, but these seem likely to be rare (e.g., only one standard-library tool uses them today).

With this nonrelative equivalence, code folders can be used more flexibly, and both direct command lines and the `-m` module-mode switch can be used to launch files in the package as programs. The next section shows how.

Package-Relative Imports in Action

To demo all of the foregoing points, let's return to the simple package we coded at the start of this chapter. As you'll recall, its *__main__.py* file is run automatically when the folder is run as a bundle, but it will also play the role of any top-level file in the folder launched as a script. When we last met this file, it simply contained a print; here's a refresher of the last-known state of the files we'll be using here so you don't have to flip back:

```
# prior dir1/mod.py
var = 'hack'
print('Loading dir1.mod')

# prior dir1/__main__.py
print('Executing dir1.__main__.py')
```

The normal-import warm-up

This *__main__.py* worked as advertised, but things get more complicated if a file run as a top-level script tries to import another module in its own package folder—as in the rewrite of Example 24-7.

Example 24-7. dir1/__main__.py (modified)

```
import mod
print('Executing dir1.__main__.py:', mod.var.upper())
```

Coded this way, the file can be run with a direct command line (and both as a folder and a script), but *not* with Python's `-m` module-mode switch, which brings the package machinery online (notice the *__init__.py* output in this mode):

```
$ python3 dir1                                  # Launch wi
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 dir1/__main__.py                      # As folder
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 -m dir1                               # Launch wi
Running dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
$ python3 -m dir1.__main__                      # As packag
Running dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
```

The first two commands in this work because *__main__.py* is run as a normal, nonpackage program, and finds its imported sibling per the "home" entry on `sys.path`, the top-level file's own folder. The problem with the last two commands is that they use the file as part of a package: the `import mod` in *__main__.py* is then interpreted as an *absolute* import—which *skips* the enclosing package. Hence the fails.

The relative-import adventure

Our first reaction might be to add *relative* imports to appease the package system—as in Example 24-8.

Example 24-8. dir1/__main__.py (modified)

```
from . import mod
print('Executing dir1.__main__.py:', mod.var.upper())
```

Coded this way, we've fixed `-m` usage, but *broken* direct command lines:

```
$ python3 -m dir1
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 -m dir1.__main__
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 dir1
ImportError: attempted relative import with no known pa
$ python3 dir1/__main__.py
ImportError: attempted relative import with no known pa
```

The first two commands in this work because `-m` invokes package behavior, which enables the relative "." import syntax that searches the package and finds the target module. The problem with the last two commands is that Python doesn't allow relative imports to be used in nonpackage mode—which dooms these runs from the start, regardless of search-path settings. Hence the fails.

Under this regime, then, it would seem we must *choose* package or nonpackage roles for our code.

The absolute-import solution

As noted, though, an easy way out of this constraint is to use normal package imports that explicitly spell out absolute import paths (which, again, are actually relative to `sys.path`) from the package root—as in Example 24-9.

Example 24-9. dir1/__main__.py (modified)

```
import dir1.mod
print('Executing dir1.__main__.py:', dir1.mod.var.upper
```

As also noted, the package root, `dir1` , will normally be on `sys.path` , because that's the only way to use its code from outside the package (clients must import with paths that start at the `dir1` package root too).

To demo here, we'll add the package root explicitly using a relative—in *filesystem* terms—path of "." for the current directory (in typical use, this might instead be an absolute path, or Python's automatic *site-packages* root of installed packages). We must set this here because the top-level file's "home" on the search path in direct-command mode is one level *below* the package root, and both direct-command and `-m` modes need access to the package root in general:

```
$ export PYTHONPATH=.                    # Or similar ou
$ python3 dir1
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 dir1/__main__.py
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK

$ python3 -m dir1                        # Both usage mc
Running dir1.__init__.py
Loading dir1.mod
Executing dir1.__main__.py: HACK
$ python3 -m dir1.__main__
Running dir1.__init__.py
```

```
Loading dir1.mod
Executing dir1.__main__.py: HACK
```

Now launches by *both* direct command lines and the `-m` switch work, because we're not using a tool that limits the utility of our code to just one mode. The end result is *dual-mode* code—it can be used as both program and package.

If you'd rather not repeat import paths at each item reference, both launch modes also work if we code *__main__.py* in any of these ways (test on your own to verify; this is just import norms at work):

```
import dir1.mod as mod
print('Executing dir1.__main__.py:', mod.var.upper())

from dir1 import mod
print('Executing dir1.__main__.py:', mod.var.upper())

from dir1.mod import var
print('Executing dir1.__main__.py:', var.upper())
```

For more fun, you can also use a `from *` in this scheme, if you add an `__all__` to the package root's initialization file (subject to all the standard disclaimers about the evils of `from *` outlined in Chapter 23):

```
# dir1/__init__.py
__all__ = ['mod']

# dir1/__main__.py
from dir1 import *
print('Executing dir1.__main__.py:', mod.var.upper())
```

Again, if you're sure that your code will only ever be used as part of a package, you could use relative imports in all of its files to access same-package modules. Moreover, the absolute-path solution arrived at here could be applied *only* for files meant to be lunched as top-level *scripts*; other files only imported can use package-relative imports.

In both cases, relative imports come with the advantage that a same-named module earlier on `sys.path` won't accidentally override a crucial module in the package. But they also limit a file to package-only roles; if you want package files to support *both* launches and imports, relative imports are not your best option.

It should also be noted that we're skipping some details here for space. For one, `-m` mode unusually also checks the CWD for absolute imports, making the demo's `PYTHONPATH` setting optional for this mode and demo only. For another, some other resources suggest that relative-import failures in scripts stem from their name `'__main__'`, but files run with `-m` have the same name and employ other protocols that are too obscure to cover here.

Because package-relative imports are both prone to change and mostly meant for programmers building large-scale libraries of code for others to use, this Python-learners text will defer to Python's docs for further details. Modules are feature-rich tools, to say the least, but the good news is that namespace packages—the topic of the next and final section of this chapter—do not add any new syntax, but simply allow a module to span folders. To see how, let's move on.

Namespace Packages

Now that you've learned all about package and package-relative imports, there's one last package-related subject to cover. The evolutionary path of modules in Python eventually led to what are known as *namespace packages* —packages that may be composed of one or more folders located on different parts of the module search path.

While packages split across folders are probably atypical in the wild, the generalizations added to support namespace packages in the import-search *algorithm* (procedure) also allow for packages without *__init__.py* files in general. Since this revised algorithm is now just *the* import search, namespace packages are something of an incidental topic.

To understand namespace packages' place in the broader modules picture, though, as well as the changes they ushered in, we need a quick history lesson.

Python Import Models

All told, Python has four distinct import schemes. The following enumerates them from original to newest, with representative but incomplete examples (as we've seen, `from` also allows a `*` wildcard and `reload` reloads any module passed to it, but these are just add-on topics):

Basic modules

The original model, used to import files and their contents, relative to the `sys.path` module search path:

```
import module
from module import name
```

Basic packages

The original package model used to import from folder paths relative to the `sys.path` search path, where each package is contained in a single directory that has an *__init__.py* file:

```
import folder.folder.module
from folder.module import name
```

Package-relative imports

The model used for intrapackage (same-package) imports of the prior section, with its relative and absolute lookup rules for imports with and without leading dots, respectively:

```
from . import module
from .module import name
```

Namespace packages

The newest package model, which is still relative to `sys.path`, but allows packages to span multiple directories, and removes the requirement that packages must define *__init__.py* files:

```
import anyfolder.anyfolder.module
from anyfolder import name
```

The first two of these models are self-contained, but the third tightens up the search order and extends syntax for same-package imports, and the fourth upends some of the notions and requirements of the prior package model. In fact, Python now formally defines two flavors of packages:

- The original model, now known as *regular packages*
- The alternative model, known as *namespace packages*

The original and alternative package models are not mutually exclusive and can be used simultaneously in the same program. In fact, the namespace package model works as something of a *fallback option*, recognized only if basic modules and regular packages of the same name are not present on the module search path. Despite their showy title, namespace packages are really just a mutation of import search, as the following sections explain.

Namespace-Package Rationales

First off, it's important to know that a namespace package is not fundamentally different from a regular package: it is just a different way of creating packages. Moreover, they are still relative to `sys.path` at the top level: the leftmost component of a dotted namespace-package path must still be located in an entry on the normal module search path.

In terms of physical structure, though, regular and namespace packages have notable differences. Regular packages have an *__init__.py* file that is run automatically, and they reside in a single directory. Namespace packages, by contrast, cannot contain an *__init__.py*, and may or may not span multiple directories collected at import time. Because the presence of *__init__.py* differentiates package type, *none* of the directories that make up a namespace package can have this file, but the content nested within each of them is treated as a single composite package.

The *rationale* for namespace packages is rooted in package *installation* goals that may seem obscure unless you are responsible for such tasks. In short, though, they resolve a potential for collision of multiple *__init__.py* files when package parts are merged, by removing this file completely. Moreover, by providing standard support for packages that can be split across multiple directories and located in multiple `sys.path` entries, namespace packages both enhance install flexibility and replace multiple incompatible solutions that had arisen to address this goal.

Though split-folder packages have some narrow roles, average Python users will probably find namespace packages' biggest benefit to be that they remove the requirement for package *__init__.py* initialization files, and thus allow any directory of code to be used as an importable package. To see how, let's move on to the nitty-gritty of import search.

The Module Search Algorithm

To understand the way that namespace packages change and extend import search, we have to look under the hood to see how the import operation works.

As we've seen, Python fulfills imports by searching for a name among a set of candidate folders. For the *leftmost* components of imports, the set of candidates is all the directories listed on the `sys.path` module search path. For components of packages either nested in *package* imports or named in *relative* imports, the set of candidates is just the package itself. While package folders have just one instance of a given name, search paths may have many.

The way the import search selects items from these candidates is subtler than implied so far. For each *`directory`* in an import search's set of candidates, Python tests for a variety of matches to an imported *`name`*, in the following order (using Unix / for folder separators and {…} to mean a choice):

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

If this search's candidate scan completes without returning a regular package or module by steps 1 or 2, and at least one directory was recorded by step 3, then a *namespace package* is created and returned. Else an error is reported.

The creation of the namespace package happens immediately and is not deferred until a lower-level import occurs. The new namespace package is a module object that does not have a `__file__` attribute, but has a `__path__` set to an iterable of the directory path strings that were found and recorded during the candidates scan by step 3.

This `__path__` attribute is then used as the set of candidates for later and deeper accesses, to search the one or more component folders of the namespace package. That is, each recorded entry on a namespace package's `__path__` is searched whenever further-nested items are requested, instead of the sole directory of a regular package.

Viewed another way, the `__path__` attribute of a namespace package serves the same role for lower-level components that `sys.path` does at the top for the leftmost component of package import paths; it becomes the "parent" search path for accessing lower items using the same four-step procedure just sketched.

The net result is that a namespace package is a sort of *virtual concatenation* of directories located on one or more search candidates. Once a namespace package is created, though, there is no functional difference between it and a regular package; it supports everything we've learned for regular packages, including package-relative import syntax.

Importantly, because a *single directory* lacking an *__init__.py* but nested in a search-candidate folder is classified as a namespace package by this algorithm's step 3, any such directory qualifies as a package. The only operational difference between such a single folder and a regular package folder is that the former has lower *search precedence*: both same-named folders with an *__init__.py* and simple modules anywhere in a search path are chosen first by steps 1 and 2.

In other words, although the mods made to support namespace packages make *__init__.py* files optional in package folders, adding one, even if empty, ensures that a package will be selected instead of a same-named folder later on a search path (it may also boost import speed by ending search-path scans at step 1, but this is a one-time event). Whether this, or the other roles of *__init__.py* we met earlier, warrants including this file will naturally depend on your goals.

Namespace Packages in Action

Technically, we've already seen namespace packages at work: the opening step of the demo at the start of this chapter made *single-folder* namespace packages because its folders didn't yet have *__init__.py* files. Again, any folder nested in a `sys.path` search-path folder qualifies as a package, as long as it's not hidden by a same-named "regular" package with *__init__.py* or a simple module elsewhere on the path.

To see the grander folder-concatenation effect of "virtual" namespace packages work, let's work through a quick demo. To begin, make two modules in a nested directory structure that has subdirectories named `sub` located in different parent directories, `part1` and `part2`—like this in indentation notation meant to represent nesting:

```
ns/
    part1/
        sub/
            mod1.py
    part2/
        sub/
            mod2.py
```

Note that there are no *__init__.py* files here—as noted earlier, these files cannot be used in namespace packages, as this is their chief differentiation from regular, single-folder packages. We can create this demo's folders with console commands like the following; translate `/` to `\` and omit the `-p` on Windows, or use a file explorer or the prebuilt folders in the examples package if you prefer:

```
$ mkdir -p ns/part1/sub       # Two subdirs of same nan
$ mkdir -p ns/part2/sub       # And similar on Windows
```

The modules at the end of these paths are coded in Examples 24-10 and 24-11 to simply print on imports as a trace.

Example 24-10. ns/part1/sub/mod1.py

```
print('Loading ns/part1/sub/mod1')
```

Example 24-11. ns/part2/sub/mod2.py

```
print('Loading ns/part2/sub/mod2')
```

Now, if we add *both* `part1` and `part2` to the module search path, `sub` becomes a namespace package spanning both folders, with the two module files available under that name even though they live in separate physical directories. Here's the path-setting command on Unix (for Windows, use `set` and `:`, and see Chapter 22 for more tips); this uses paths relative to the CWD, but in practice would more likely list two full, absolute paths instead:

```
$ export PYTHONPATH=ns/part1:ns/part2
```

When imported directly, the namespace package is the *virtual concatenation* of its individual directory components, and allows further nested parts to be accessed through its single, composite name with normal imports (as usual, paths have been shortened here for space with "…" and line breaks were added for fit):

```
$ python3
>>> import sub
>>> sub                                        # Namespace pac
<module 'sub' (namespace) from
['/…/LP6E/Chapter24/ns/part1/sub', '/…/LP6E/Chapter24/r

>>> from sub import mod1
Loading ns/part1/sub/mod1
>>> import sub.mod2                            # Content from
Loading ns/part2/sub/mod2

>>> mod1
<module 'sub.mod1' from '/…/LP6E/Chapter24/ns/part1/sub
>>> sub.mod2
<module 'sub.mod2' from '/…/LP6E/Chapter24/ns/part2/sub
```

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

```
$ python3
>>> import sub.mod1
Loading ns/part1/sub/mod1
>>> import sub.mod2                    # One package s
Loading ns/part2/sub/mod2

>>> sub.mod1
<module 'sub.mod1' from '/…/LP6E/Chapter24/ns/part1/sub
>>> sub.mod2
<module 'sub.mod2' from '/…/LP6E/Chapter24/ns/part2/sub

>>> sub
<module 'sub' (namespace) from
['/…/LP6E/Chapter24/ns/part1/sub', '/…/LP6E/Chapter24/n
>>> sub.__path__
_NamespacePath(
['/…/LP6E/Chapter24/ns/part1/sub', '/…/LP6E/Chapter24/n
```

Interestingly, *relative imports* work in namespace packages too, if we add one as in Example 24-12.

Example 24-12. ns/part1/sub/mod1.py (modified)

```
from . import mod2
print('Loading ns/part1/sub/mod1')
```

The added package-relative import statement references a file in the package —even though the referenced file resides in a *different directory*:

```
$ python3
>>> import sub.mod1                    # Relative impo
Loading ns/part2/sub/mod2
Loading ns/part1/sub/mod1
>>> import sub.mod2                    # Already impor
>>> sub.mod2
<module 'sub.mod2' from '/…/LP6E/Chapter24/ns/part2/sub
```

As you can see, namespace packages are like ordinary single-directory packages in every way, except for having a split physical storage. By extension, this is why code folders without *__init__.py* files are exactly like regular packages, but with no initialization logic to be run; they're just an instance of namespace packages with a single directory.

Like package-relative imports, this book is also going to defer to Python's manuals for more details on this subject. Namespace packages are a potentially useful tool, but most Python learners are probably better served by first mastering packages that map to real folders, before tackling those that are spread virtually across a host's directories.

Chapter Summary

This chapter introduced Python's *package import* model—an optional but useful way to explicitly list part of the directory path leading up to modules. Package imports are still relative to a directory on your module search path, but your script gives the rest of the path to the module explicitly. Packages may be built with a simple folder and can make imports more meaningful, simplify import search-path settings, and resolve ambiguities when there is more than one module of the same name—the name of the enclosing directory makes them unique.

Because it's relevant only to code in packages, we also explored *relative imports*: a way for imports in package files to select modules in the same package explicitly using leading dots in `from`, instead of relying on a search in the host. Finally, we surveyed *namespace packages*: a tool that allows a package to span multiple directories as a fallback option of import searches, and make initialization files optional in single-folder packages.

The next chapter surveys a handful of both common and advanced module-related topics, such as the `__name__` usage mode variable, the `__getattr__` attribute hook, and name-string imports, and codes useful module tools along the way. As usual, though, let's close out this chapter first with a short quiz to review what you've learned here.

Test Your Knowledge: Quiz

1. What is the purpose of an *__init__.py* file in a module package directory?
2. How can you avoid repeating the full package path every time you reference a package's content?

3. Which directories require *__init__.py* files?
4. When must you use `import` instead of `from` with packages?
5. What is the difference between `from pkg import name` and `from . import name`?
6. What is a namespace package?

Test Your Knowledge: Answers

1. The *__init__.py* file serves to declare and initialize a regular module package; Python automatically runs its code the first time you import through a directory in a process. Its assigned variables become the attributes of the module object created in memory to correspond to that directory. It's optional for package folders but gives a folder search precedence over other folders of the same name that don't have this file.
2. Use the `from` statement with a package to copy names out of the package directly, or use the `as` extension with the `import` statement to rename the path to a shorter synonym. In both cases, the path is listed in only one place, in the `from` or `import` statement.
3. Trick question! These files used to be required for packages in earlier Pythons but became optional with accommodations for namespace packages in the module search algorithm. As noted in answer 1, though, these folders still have valid, if optional, roles, including boosting a folder's import-search precedence.
4. You must use `import` instead of `from` with packages only if you need to access the *same name* defined in more than one path. With `import`, the path makes the references unique, but `from` allows only one version of any given name (unless you also use the `as` extension to rename uniquely).
5. The `from pkg import name` is an *absolute* import—the search for `pkg` skips an enclosing package and then tries the "absolute" directories in `sys.path`. A statement `from . import name`, on the other hand, is a *relative* import— `name` is looked up relative to the package in which this statement is contained, only.
6. A *namespace package* is an extension to the import model that corresponds to one or more directories that do not have *__init__.py* files. When Python finds these during an import search and does not find a simple module or regular package first, it creates a namespace package that is the virtual concatenation of all found directories having the

requested module name. Further nested components are looked up in all the namespace package’s directories. The effect is similar to a regular package, but content may be split across multiple directories.