Core Python: Organizing Larger Programs

by Austin Bingham and Robert Smallshire

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

Hi everyone, my name is Austin Bingham, and welcome to my course, Core Python: Organizing Larger Programs. I'm a founder and principle consultant at Sixty North. As your Python programs grow, or if you need to contribute to existing larger Python-based systems, you'll need some techniques to help you manage the inevitable growth of complexity. Our experience is that successful Python projects pay close attention to the modular organization of their code. In this course, we are going to introduce you to features of the Python language which facilitate the structuring of your code base once your needs move beyond a few Python modules. Some of the major topics we will cover include packages, which allow us to nest Python modules into subpackages; namespace packages through which we can extend existing packages with additional code; executable packages, which provide a convenient means for delivering programs as a coherent yet modular whole; recommended file and directory layouts for new packages; and distributing your new packages to other users. By the end of this course, you'll know where to start when producing greenfield Python libraries or Python executables, and the patterns you should follow to grow your system from a vigorous seedling into a strong and substantial oak. Before beginning the course, you should be familiar with basic use of Python's module and import system. From here, you should feel comfortable diving into other core Python language courses on classes and object-orientation, functions and functional programming, and robust resource and error handling. I hope you'll join me on this journey to learn about the Python package system with the Core Python: Organizing Larger Programs course at Pluralsight.

Nesting Modules with Packages

Prerequisites

This is a course for people who already know the essentials of the Python programming language and are ready to dig deeper, to take the steps from novice to journeyman. In this module, we'll mention what we expect you to already know in order to benefit from this course. We'll show you how to import nested packages from the Python standard library, we'll see that packages are modules, and modules are objects with special attributes. Finally, we'll learn how Python locates modules when they're first imported, and the various ways in which we can control the search process. Modularization is perhaps the most important conceptual tool we have for managing systems as they grow from trivial to tremendous, and we're going to show you the features of the Python language which support that modularization. Before we really start, it's important that you are comfortable with the prerequisites for this course. We will assume that you know quite a bit already, and will spend very little time covering basic topics. For this course, you need to know how to work with basic single file modules in Python. We'll be covering packages in this course, but we won't spend any time covering basic module topics like creating modules or importing them. In particular, you will need to be familiar with the import x, from x import y, and from x import y as z forms of the import statement. You should already know how to create a module from a py file, and have an understanding of what happens when the module is imported, including the so-called main block concept to distinguish module execution from module import. If you find that you need to brush up on Python basics before you start this course, you can always refer to our Python Fundamentals course, which covers all of the prerequisites for this course. If you want to follow along with the examples yourself, you will need access to a working Python 3 system for this course. The material we present will work on all recent versions of Python 3. If you have a choice, you should probably get the most recent stable version. At a minimum, you need to be able to run a Python 3 REPL. You can, of course, use an IDE if you wish, but we won't require anything beyond what comes with the standard Python distribution. Finally, we need to make a quick note regarding terminology. In Python, many language features are implemented or controlled using special attributes or methods on objects. These special methods are named with two leading and two following underscores. This has the benefit of making them visually distinct, fairly easy to remember, and unlikely to collide with other names. This scheme has the disadvantage, however, of making these names difficult to pronounce, a problem we face when making courses like this. To resolve this issue, we have chosen to use the term dunder when referring to these special methods. Dunder is a portmanteau of the term double underscore, and we'll use it to refer to any method with leading and trailing double underscores. So, for example, when we talk about the method underscore, underscore, len, underscore, underscore, which, as you'll recall is invoked by the len function, we'll say dunder len. These kinds of attributes play a big role in this course, so we'll be using this convention frequently. If you'd like a book to support you as you work through the material in this course, you can check out The Python Journeyman, in which you'll find the same material in written form. By following the URL shown, you can obtain the book for a substantially discounted price. The Python Journeyman is the second book in our Python Craftsman trilogy, the first book being The Python Apprentice and the third book being The Python Master. Taken together, these contain material which correspond to our various Python courses here on Pluralsight. All three are available to Pluralsight viewers at reduced prices.

Introduction to Packages

As you'll recall, Python's basic tool for organizing code is the module. Module typically corresponds to a single source file, and you load modules into programs by using the import keyword. When you import a module, it is represented by an object of type module, and you can interact with it like any other object. A package in Python is just a special type of module. The defining characteristic of a package is that it can contain other modules, including other packages. So packages are a way to define hierarchies of modules in Python. This allows you to group modules with similar functionality together in ways that express their cohesiveness. Many parts of Python's standard library are implemented as packages. To see an example, open your REPL and import urllib and urllib.request. Now, if you check the types of both of these modules, you'll see that they're both of type module. The important point here is that urllib.request is nested inside urllib. In this case, urllib is a package and request is a normal module. Notice that only urllib is bound to a name in the local namespace, so it isn't possible to access the request sub-module directly. When imported this way, access to the sub-module must be via a fully qualified hierarchal module name. It is possible to import a sub-module directly using the from import syntax. Saying this will bind only request to a name in the local namespace. The parent urllib package will have been imported, but won't be directly accessible via the urllib name. Even when imported this way, the sub-module knows its own hierarchal module name. If you closely inspect each of these objects, you'll notice an important difference. The urllib package has a __path__ member that urllib.request does not have. This attribute is a list of file system paths indicating where urllib searches to find nested modules. This hints at the nature of the distinction between packages and modules. Packages are generally represented by directories in the file system, while modules are represented by single files. Note that in Python 3 versions prior to 3.3, __path__ was just a single string, not a list. In this course we're focusing on Python 3.7, but for most purposes the difference is not important.

Locating Modules

Before we get into the details of packages, it's important to understand how Python locates modules. Generally speaking, when you ask Python to import a module, Python looks on your file system for the corresponding Python source file and loads that code, but how does Python know where to look? The answer is that Python checks the path attribute of the standard sys module, commonly referred to as sys.path. The sys.path object is a list of directories. When you ask Python to import a module, it starts with the first directory in sys.path and checks for an appropriate file. If no match is found in the first directory, it checks subsequent entries in order until a match is found or Python runs out of entries in sys.path, in which case an import error is raised. Let's explore sys.path from the REPL. Go to your command line and start Python with no arguments. As you see, your sys.path can be quite large. Its precise entries depend on a number of factors, including how many third-party packages you've installed, and how you've installed them. For our purposes, a few of these entries are of particular importance. First, let's look at the very first entry. Remember that sys.path is just a normal list, so we can examine its contents with indexing and slicing. We see here that the first entry is an empty string. This happens when you start the Python interpreter with no arguments, and it instructs Python to search for modules first in the current directory. Let's also look at the tail of sys.path. These entries comprise Python's standard library and the site packages directory where you can install third-party modules. To really get a feel for sys.path, let's create a Python source file in a directory that Python would not normally search. In that directory, create a file called path_test.py with these contents. Now, start your REPL from the directory containing the not_searched directory, and try to import path_test. The path_test module, which remember is embodied in not_searched path_test.py, is not found because path_test.py is not in a directory contained in sys.path. To make path_test importable, we need to put the directory not_searched into sys.path. Since sys.path is just a normal list, we can add our new entry using the append method. Now when we try to import path_test in the same REPL session, we see that it works. Knowing how to manually manipulate sys.path can be useful, and sometimes it's the best way to make code available in Python. There is another way to add entries to sys.path, though, that doesn't require direct manipulation of that list. The PYTHONPATH environment variable is a list of paths that are added to sys.path when Python starts. The format of PYTHONPATH is the same as path on your platform. On Windows, PYTHONPATH is a semi-colon separated list of directories. On Linux and Mac, it's a colon separated list of directories. To see how PYTHONPATH works, let's add not_searched to it before starting Python again. On Windows, use the set command. On Linux or OS X, the syntax will depend on your shell, but for bash-like shells you can use export. Now start a new REPL and check that not_searched is indeed in sys.path. And, of course, we can now import path_test without manually editing sys.path. There are more details to sys.path and PYTHONPATH, but this is most of what you need to know. For further information, you can check these links to the Python documentation.

Summary

In this module, we have seen how to import nested packages from the Python standard library using dot syntax. We have seen that importing a nested package imports all modules and submodules with the parts of the hierarchal module name, yet only the first component of the hierarchal module name is bound to a name in the local namespace, therefore, submodules must be accessed via their fully qualified name, including parent modules. We have shown that packages store in __path__, the directory path, which stores the package and sub-module code in the file system. We have shown how sys.path controls the module loading search process, and how sys.path can be initialized from the PYTHONPATH environment variable.

Implementing Packages

Creating Packages

We've seen that packages are modules that can contain other modules, but how are packages implemented? In this module of Core Python: Organizing Larger Programs, you'll see how to define your own packages and submodules. You'll understand package initialization, you'll be introduced to the syntax for relative imports, and you'll learn how to control which attributes are imported by default from a package. To create a normal module, you simply create a Python source file in a directory contained in sys.path. The process for creating packages is not much different. To create a package, first you create the package's root directory. This root directory needs to be in some directory on sys.path. Remember, this is how Python finds modules and packages for importing. Then, in that root directory you create a file called __init__.py. This file, which we'll often call the package init file, is what makes the package a module. Dunder init.py can be, and often is, empty. Its presence alone suffices to establish the package. In a later section, we'll look at a somewhat more general form of packages which can span multiple directory trees. In that section we'll see that since PEP420 was introduced in Python 3.3, __init__.py files are not technically required for packages any more. So why do we still talk about them as if they are required? For one thing, they are still required for earlier versions of Python, in fact, many people writing code for 3.3 and later aren't aware that __init__.py files are optional. As a result, you'll still find them in the vast majority of packages, so it's good to be familiar with them. Furthermore, they provide a powerful tool for package initialization, so it's important to understand how they work. Perhaps most importantly, though, we recommend that you include __init__.py files when possible, because explicit is better than implicit. The existence of a package initialization file is an unambiguous signal that you intend for a directory to be a package, and it's something that many Python developers instinctively look for. As with many things in Python, an example is much more instructive than words. Go to your command prompt and create a new directory called demo_reader. Add an empty __init__.py file to this directory. On Linux and OS X, you can use the touch command. On Windows, you can use type. Now, if you start your REPL, you'll see that you can import demo_reader. If we ask for the type of the imported demo_reader package, we can see that it is a module object, which is the same Python type that a single file module would have. We can now start to examine the role that __init__.py plays in the functioning of a package. Check the __file__ attribute of the demo_reader package. We saw that demo_reader is a module, even though on our file system the name demo_reader refers to a directory. Furthermore, the source file that is executed when demo_reader is imported, is the package init file in the demo_reader directory, that is, demo_reader __init__.py. In other words, and to reiterate a critical point, a package is nothing more than a directory containing a file named __init__.py. To see that __init__.py is actually executed like any other module when you import demo_reader, let's add a small bit of code to it. We'll just print demo_reader is being imported. Restart your REPL, import demo_reader, and you'll see our little message printed out. Now that we've created a basic package, let's add some useful content to it. The goal of our package will be to create a class that can read data from three different file formats, uncompressed text files, text files compressed with gzip, and text files compressed with bz2. We'll call this class MultiReader since it can read multiple formats. We'll start by defining MultiReader. The initial version of this class will only know how to read uncompressed text files. We'll add support for gzip and bz2 later. Create the file demo_reader multireader.py with these contents. The MultiReader class contains three methods, the initializer, __init__, a close method, and a read method. The initializer accepts a filename parameter which is stored on an instance attribute of the same name. It then opens the file for read in text mode. The open file is bound to the f instance attribute. The close method closes the file, and the read method delegates to the read method of the file object, which returns the entire contents of the text file as a string. Start a new REPL and import your new module to try out the MultiReader class. We'll create an instance of MultiReader, passing the path to a text file as the only argument to the initializer. The text file we'll use is demo_reader __init__.py itself. We'll invoke the read method to return the file contents as a string, and neatly round off by closing the file. In a somewhat meta turn, our package is reading some of its own source.

Creating a Subpackage

To demonstrate how packages provide high levels of structure to your Python code, let's add more layers of packages to the demo_reader hierarchy. We're going to add a subpackage to demo_reader called compressed, which will contain the code for working with compressed files. First, let's create the new directory and its associated __init__.py. The name of the subpackage will be compressed, and so that's the name we must give to the directory. The __init__.py file can be empty at this juncture. Its presence is sufficient to mark the compressed directory as a package. Now, restart the REPL. We can import our subpackage using its dotted hierarchal name, demo_reader.compressed. If we inspect the __file__ attribute of our new subpackage with demo_reader.compressed .__file__, we get the location of the __init__.py file we created a few moments ago. Next, we'll create the file demo_reader compressed gsip.py, which will contain some code for working with the gzip compression format. As you can see, there's not much to this code. It simply defines the name opener, which is just an alias for gzip.open. This function behaves much like the normal open in that it returns a file-like object which can be read from. The main difference, of course, is that gzip open decompresses the contents of the file during reading, while open does not. Note the idiomatic main block here. It uses gzip to create a new compressed file and write data to it, the data in this case being the second and subsequent command line arguments, passed in when the program is invoked. These are joined into a single space-separated string. The zeroeth element of sys.argv always contains the script name in a non-portable operating system dependent format. We expect the first element of sys.argv to contain the path to the compressed file being created. We'll use that later to create some test files. For more information on __name__ and __main__, see module 4 of Python Fundamentals. Similarly, let's create another file for handling bz2 compression called demo_reader compressed bzipped.py. At this point, you should have a directory structure that looks like this. If you start a new REPL, you'll see that we can import all of our modules just as you might expect. We can import the top-level demo_reader package with import demo_reader, the multireader single file submodule with import demo_reader.multireader, the compressed subpackage with import demo_reader.compressed, and the two nested submodules we have just created with import demo_reader.compressed .gzipped and import demo_reader.compressed .bzipped. Let's glue all of this together into a more useful sort of program. We'll update MultiReader so that it can read from gzip files, bz2 files, and normal text files. It will determine which format to use based on file extensions. Change your MultiReader class in demo_reader multireader.py to use the compression handlers when necessary. First, we'll import the submodules bzipped and gzipped. This demonstrates the fundamental organizing power of packages. Related functionality can be grouped under a common name for easier identification. Next, we'll define a dictionary called extension_map, which associates the bz2 and gz file extensions with the corresponding opener functions from the bzipped and gzipped modules we imported above. Then, we'll modify the body of the MultiReader initializer to split the extension from the filename and retrieve the corresponding opener function from the extension_map. We use the get method of dict to fall back to the built-in open function if no matching extension is found. The opener callable we get from this lookup is used to open the file for read in text mode as before. To test this out, let's first create some compressed files using the utility code we built into our compression module. We'll execute our modules directly from an operating system shell. There's a lot going on in this incantation, so let's break it down. The first token is the Python interpreter, Python 3. The -m tells Python to run the module that we will specify in the next argument. The module name must be a fully-qualified dotted name resolvable from a directory in the default Python path, not a file system path with slashes. The next argument will be passed to the module as the sys.argv sub 1 element, and will be a path to the compressed file being created. Any subsequent arguments, here data compressed with bz2, will be passed as sys.argv sub 2, and so on, in this case up to sys.argv sub 5. We can do something equivalent by invoking the demo_reader.compressed .gzipped submodule. Finally, we'll check that the two compressed files have been created. Feel free to verify for yourself that the contents of test.bz2 and test.gz are actually compressed, or at least that they're not just plain text. Start a new REPL, and let's take our code for a spin. We'll import just the MultiReader class from its submodule and instantiate a MultiReader object called r, passing the path to the text.bz2 file we just created. We can successfully read the compressed file with read before closing the file. Two more uses of the MultiReader class show reading gzipped data and reading regular uncompressed data using MultiReader. If you've put all the right code in all the right places, you should see that your MultiReader can indeed decompress all these file types when it sees their associated file extensions.

Relative Imports

In this course, we've seen a number of uses of the import keyword, and if you've done any amount of Python programming then you should be familiar with it. All of the uses we've seen so far are what are called absolute imports, wherein you specify all of the ancestor modules of any module you want to import. For example, in order to import demo_reader.compressed .bzipped in the previous section, you had to mention both demo_reader and compressed in the import statement. There is an alternative form of imports called relative imports that lets you use shortened paths to modules and packages. The obvious difference between this form of import and the absolute imports we have seen so far is the dots before module_name. In short, each dot stands for an ancestor package of the module that is doing the import, starting with the package containing the module and moving towards the package root. Instead of specifying imports with absolute paths from the root of package tree, you can specify them relative to the importing module, hence, relative imports. Note that you can only use relative imports with the from module import names form of import. Trying to do something like import.module is a syntax error. Critically, relative imports can only be used within the current top-level package, never for importing modules outside of that package. So in our previous example, the demo_reader module could use a relative import for gzipped, but it must use absolute imports for anything outside of the top-level demo_reader package. Let's make some changes to our demo_reader package to illustrate relative imports. If we look at our main blocks in bzipped.py and gzipped.py, we see that they're almost identical. In fact, the only difference between the two is that one uses gzip.open and the other uses bz2.open. To remove this duplication, let's create a common implementation of main that they can both use. We'll put this in demo_reader util writer.py. We'll create the util directory as a sibling of the compressed directory. To signal that util is a package, we'll add a __init__.py file within it. The writer module will also be inside the util package directory. The main function defined here accepts the opener callable as a parameter, which neatly decouples this function from the specific opener implementation used when the function is called. With this change, we can rewrite bzipped.py to use main. We no longer need the sys module here, so we can remove that dependency. Then we import our new writer module from the new util subpackage of demo_reader. In the main block, we invoke the main function from the writer module, passing the opener implementation specific to bz2 as the only argument. We can apply an equivalent refactoring to gzipped.py, the only difference now being the specific opener callable, which is now gzipped.opener. This is important. The only difference in implementation of these two modules is now the only essential difference between what they do, namely the type of compression they deal in. In both cases, we used the import statement from demo_reader.util import writer. As an alternative, relative imports allow us to make the same import as from. .util import writer. The leading.. in the relative form means the parent of the package containing this module, or in other words, the demo_reader package. This table summarizes how the dots are interpreted when used to make relative imports from demo_reader.compressed .bzipped. In relative imports, it's also legal for the from section to consist purely of dots. In this case, the dots are still interpreted in exactly the same way as before. Going back to our example, suppose that bzipped.py wanted to import the demo_reader.util module itself. With absolute imports, we would use the form from demo_reader import util. With relative imports, on the other hand, we can use the form from.. import util. Here, just as with the first form of relative imports we looked at, the.. means the parent of the current package. It's easy to see how relative imports can be useful for reducing typing in deeply nested package structures. They also promote certain forms of modifiability since they allow you, principal, to rename top-level and subpackages in some cases. On the whole, though, the general consensus seems to be that relative imports are best avoided in most cases.

Using __all__

Another topic we want to look at is the optional __all__ attribute of modules. Dunder all lets you control which attributes are imported when someone uses the from module import * syntax. If __all__ is not specified, then from x import * imports all public names, those without leading underscores from the imported module. The __all__ module attribute must be a list of strings, and each string indicates a name which will be imported when the * syntax is used. For example, we can see what from demo_reader.compressed import * does. First, let's add some code to demo_reader.compressed __init__.py. From demo_reader.compressed .bzipped, with import opener as bz2_opener. And from demo_reader.compressed .gzipped, with import opener as gzip_opener. Next we'll start a REPL and display all the names currently in scope, using the pprint module to make them easier to read. We haven't imported anything yet, so we have various default attributes of the __main__ module used by the REPL. Now, we'll import all public names from compressed using from demo_reader.compressed import *, and then pretty print the contents of locals again. What we see is that from demo_reader.compressed import * imported the bzipped and gzipped submodules of the compressed package directly into our local namespace. We prefer that import * only imports the different opener functions from each of these modules, so let's update compressed. __init__.py to do that. To the global, that is module scope, __all__ variable, we assign a list containing the strings bz2_opener and gzip_opener. Note that this won't work if you use references to the actual objects rather than their stringified names. To understand why, consider that objects generally can't know the names to which they've been bound. With __all__ in place, let's repeat the previous REPL session to understand the difference. First, here's the baseline display of the local variables before we imported our subpackage. Here comes the import containing the *, from demo_reader.compressed import *. Now, let's examine the local variables again. Only the two functions mentioned in __all__ have been imported, and there's no sign of their containing modules. Just to check, we can verify that the two names newly bound in the local namespace refer to function objects, as expected. The __all__ module attribute can be a useful tool for limiting which names are exposed by your modules. We still don't recommend that you use the import * syntax outside of convenience in the REPL, but it is good to know about __all__ since you're likely to see it in the wild. We've covered a lot of information in this part of the course, so let's review. Packages are modules which can contain other modules. Packages are generally implemented as directories containing a special __init__.py file. Although __init__.py is technically optional, it's a clear signal to developers that a directory of Python modules is, in fact, package. The __init__.py file is executed when the package is imported. Packages can contain subpackages, which are themselves implemented as directories containing __init__.py files. The __all__ attribute of a package can be used to control which module attributes are imported by the from package import * syntax.

Namespace and Executable Packages

Namespace Packages

In this module on Namespace and Executable Packages, we'll introduce namespace packages. We'll demonstrate executable directories, we'll show that Python can execute code directly from zip to directories, and we'll make executable packages. Earlier, we said that packages are implemented as directories containing a __init__.py file. This is true for most cases, but there are situations where you want to be able to split a package across multiple directories. This is useful, for example, when a logical package needs to be delivered in multiple parts, as happens in some of the larger Python projects. Several approaches to addressing this need have been implemented, but it was in PEP 420 in 2012 that an official solution was built into the Python language. This solution is known as namespace packages. A namespace package is a package which is spread over several directories, with each directory tree contributing to a single logical package from the programmer's point of view. Namespace packages are different from normal packages in that they cannot have __init__.py files. This is important, because it means that namespace packages can't have package-level initialization code. Nothing will be executed by the package when it's imported. The primary reason for this limitation is that it avoids complex questions of initialization order when multiple directories contribute to a package. But if namespace packages don't have __init__.py files, how does Python find them during import? The answer is that Python follows a relatively simple algorithm to detect namespace packages. When asked to import a name like foo, Python scans each of the entries in sys.path in order. If, in any of these directories, it finds a directory named foo containing __init__.py, then a normal package is imported. If it doesn't find any normal packages, but it does find foo.py or any other file that can act as a module, then this module is imported instead. Otherwise, the import mechanism keeps track of any directories it finds which are named foo. If no normal packages or modules are found which satisfy the import, then all of the matching directory names act as parts of a namespace package. As a simple example, let's see how we might the demo_reader package into a namespace package. Instead of putting all of the code under a single directory, we would have two parts rooted at path1 and path2 like this. This separates the compressed subpackages from the rest of the package. Now to import demo_reader you need to make sure that both path1 and path2 are in your sys.path. We can do that in a REPL like this. We put path1 and path2 at the end of sys.path by literally extending sys.path with the two new paths. Remember that sys.path is just a regular list with an extend method. When we import demo_reader, we see that its __path__ includes portions from both path1 and path2, and when we import the util and compressed subpackages, we see that they are indeed coming from their respective directories. Demo_reader.util .__path__ contains the util subdirectory path, and demo_reader.compressed .__path__ contains the compressed subdirectory. There are more details to namespace packages, but this addresses most of the important details that you'll need to know. If you do want to learn more about them, though, you can start by reading PEP 420.

Executable Directories

Packages are often developed because they implement some program that you want to execute. There are a number of ways to construct such programs, but one of the simplest is through the use of executable directories. Executable directories let you specify a main entry point which is run when the directory is executed by Python. What do we mean when we say that Python executes a directory? To answer this, we'll work with the non-namespace package version of our demo_reader project. Put this package into a directory called multi-reader-program. You should have a directory structure like this. The top level, multi-reader-program directory, is just a directory. There's nothing special about it from Python's point of view, and it doesn't contain a __init__.py file. Within this regular directory we have our demo_reader package, signified as such by the __init__.py file it contains. That in turn contains the compressed subpackage, the multireader.py module, and the util subpackage. To attempt to execute the multi-reader-program directory, change to its parent directory and pass it to Python on the command line with python 3 multi-reader-program. Normally this doesn't work and Python will complain, saying that it can't find a __main__ module, however, as that error message suggests, you can put a special module named __main__.py in the directory and Python will execute it. This module can execute whatever code it wants, meaning that it can call into modules you created to provide, for example, a user interface to your modules. To illustrate this, let's add a __main__.py to our multi-reader-program directory. For now, we'll just have this print out a message to show that Python is executing it, executing multi-reader-program __main__.py. Now, if we pass this multi-reader-program directory to Python, we'll see our __main__.py executed. This is interesting, but used in this way __main__.py is not much more than a curiosity. As we'll soon see, however, the idea of an executable directory can be used to better organize code that might otherwise sprawl inside a single file. When Python executes a __main__.py, it first adds the directory containing __main__.py to its sys.path. This way, __main__.py can easily import any other modules with which it shares a directory. If you think of the directory containing __main__.py as a program, then you can see how this change to sys.path allows you to organize your code in better ways. You can use separate modules for the logically distinct parts of your program. In the case of our multi-reader-program, since the demo_reader package is in the multi-reader-program directory, we can import it into our __main__.py and use it. So let's update our __main__.py to use our MultiReader class. As you can see, it simply takes the first command line argument, constructs a MultiReader instance from it, and reads the contents of the file. If you take one of the bz2 compressed files from earlier, you can read it by executing your directory.

Executable Zip Files

We can take the executable directory idea one step further by zipping the directory. Python knows how to read zip files and treat them like directories, meaning that we can create executable zip files just like we created executable directories. Create a zip file from your multi-reader-program directory. The zip file should contain the contents of your executable directory, but not the executable directory itself. The zip file takes the place of the directory. Now we can tell Python to execute the zip file rather than the directory. Combining Python support for __main__.py with its ability to execute zip files, gives us a convenient way to distribute code in some cases. If you develop a program consisting of a directory containing some modules and a __main__.py, you can zip up the contents of the directory, share it with others, and they'll be able to run it with no need for installing any packages to their Python installation.

Executable Packages

Of course, sometimes you really do need to distribute proper packages rather than more ad hoc collections of modules, so we'll look at the role of __main__.py in packages next. In the previous section, we saw how to use __main__.py to make a directory directly executable. You can use a similar technique to create executable packages. If you put __main__.py in a package directory, then Python will execute it when you run the package using Python's -m flag. To demonstrate this, let's make our demo_reader package executable. First, copy the demo_reader package out of the multi-reader-program directory, then create a __main__.py in the demo_reader package. If we try to execute the demo_reader as a directory, we see that it doesn't work. Python complains that __main__.py can't import MultiReader with a relative import. This seems to be at odds with our design; __main__.py and multireader.py are clearly in the same package. The reason this fails is because of what we learned earlier about executable directories. When we ask Python to execute the demo_reader directory, it first adds the demo_reader directory to sys.path. It then executes __main__.py. The crucial detail is that sys.path contains demo_reader itself, not the directory containing demo_reader. As a result, Python doesn't recognize demo_reader as a package at all. We haven't told it to look in the right place. In order to execute our package, we need to tell Python to treat demo_reader as a module with the -m command line flag. Now, Python looks for the module demo_reader.__main__, that is our demo_reader __main__.py file, and executes it while treating demo_reader as a package. As a result, __main__.py is able to use a relative import to pull in MultiReader. As you'll recall, Python executes __init__.py the first time a package is imported, so you may be wondering why we need __main__.py at all. After all, can't we just execute the same code in __init__.py as we do in __main__.py? The short answer is no. You can, of course, put whatever code you want in __init__.py, but Python will not execute a package from the command line unless it contains __main__.py. To see this, first move demo_reader __main__.py out of demo_reader, and try running the package again with python -

m demo_reader test.gz. Python complains that demo_reader is a package and cannot be directly executed. Now move __main__.py back into place and edit demo_reader __init__.py to print a little message, print executing __init__.py. We can see that our package __init__.py is indeed executed when it's imported, but again, Python will not let us execute a package unless it contains __main__.py. In this part of the course, we've shown how a single logical package can be assembled for multiple physical directories containing Python modules using the namespace package feature. We've seen that namespace package directories cannot contain __init__.py. We've made directories of Python code directly executable by providing __main__.py. We've demonstrated that directories of Python code and packages can be zipped, and we've made packages which can be both usefully imported and executed. This is the package-level equivalent of the main block idiom used for single file modules.

Recommended Package Layout

Python Project Structure

In this module on recommended project layout, we'll look at a project structure that works for most projects. We'll learn about the basics of project description files, we'll look into the concept of plugins for extending packages, and we'll see two different methods for implementing plugins. Now that we know how to create packages, we need to look at how to structure the overall project that we use to develop packages. There are no hard and fast rules about how to layout your code, but some options are generally better than others. What we will present here is a good general-purpose structure that will work for almost any project you might work on. Here's the basic project layout. At the very top level, you have a directory with the project's name. This directory is not a package, but is a directory containing both your package, as well as supporting files like your setup.py, license details, and your tests directory. The file README.rst is there to give users a quick overview of what your project is for and how to use it. The rst suffix indicates that this file is written in restructured text, a text format that is very widely used in the Python community. While there are no requirements for what goes into a README, you should try to put yourself in the position of someone approaching your project for the first time when writing it. Your README should help the reader know if the project is right for them, and if so, how to get started with it. The first subdirectory in the project is the docs directory. Most projects of any substance should have some amount of documentation, and this is where it belongs. This documentation can come in many forms, so this project structure doesn't specify anything beyond a simple folder. We find that having a directory with an easily understood name like docs helps users find the documentation for your project. The README file mentioned earlier is, of course, part of the project documentation, but we suggest keeping it in the root directory because a lot of people and tools expect to find it there. In practice, a lot of README files give very brief introductions to a project, and then refer readers to contents in the docs directory. The next directory shown is the src, or source directory. This directory will contain your actual package directory, which will have the same name as your top-level directory. Your package contains all of the production code including any subpackages. Again, there is no rule that says that this structure is required, but this is a common pattern and makes it easy to recognize where you are when navigating your project. The src directory is there to help ensure that you know what version of your code you are using. Remember that when you run Python it will put your current directory at the front of sys.path, thereby allowing you to import any module from the current directory. As a result, without the source directory in this structure, you would be able to easily import your package directly from the uninstalled source code. While this may sound like a good thing, there are many situations where this can lead to confusion. Broadly speaking, the src, or source directory, ensures that when you import your package you're using an installed version just like your users would. This way you don't run the risk of experiencing different behavior during development versus standard deployment, and in the end this means that the code you ship behaves the way you expect it to. There are many situations where the source directory simplifies packaging and deployment, and you will discover these as you get into more advanced Python development. While the src directory may seem like an unnecessary complexity, we think it's a better overall approach than putting the package directory directly in the project root. The tests directory contains all of your tests. This may be as simple as a few Python files or as complex as multiple suites of unit integration and end-to-end tests. We recommend keeping your tests outside of your package for a number of reasons. Generally speaking, test and production code serve very different purposes, and shouldn't be coupled unnecessarily. Since you usually don't need your tests to be installed along with your package, this keeps packaging tools from bundling them together. Also, more exotically, this arrangement ensures that certain tools won't accidentally try to treat your tests as production code. As with all things, this test directory arrangement may not suit your needs. Certainly, you will find examples of packages that include their tests as subpackages. If you find that you need to include all or some of your tests as a subpackage, you absolutely should. There's not much more to it than that. This is a very simple structure, but it works well for most needs. It serves as a fine starting point for more complex project structures, and this is the structure we typically use when starting new projects.

A Concrete Example: demo_reader

To make this more concrete, and to pave the way for the next section on package distribution, let's set up the project structure for our demo_reader package. This project will include a setup.py, the package source, and a placeholder for tests. The overall structure will look like this. The src demo_reader subdirectory is simply the demo_reader package that we developed in the last section. The tests directory contains a single file, test_multireader.py. We'll leave this file largely empty in this example, but generally you would have many files in the tests directory containing tests for all the parts of your system. It's important to remember that the tests directory is not part of the demo_reader package, so you can't use relative imports in your test modules to access demo_reader. As such, test_multireader.py would start off something like this. Finally, we'll create a minimal setup.py for our project. Setup.py files generally need to use the setuptools package, so make sure you install that. We'll then import setuptools at the top of setup.py. While most setup.py scripts will define a lot more information, this is sufficient for installing your package into a Python environment and for building distributions.

Implementing Plugins with Namespace Packages

Plugins are a technique for adding new functionality to a package without modifying the package itself. The package will define various extension points that can be extended with code outside of the package. The package will then use various discovery techniques to find and load those extensions at runtime. There are various methods for implementing plugins in Python, and we'll look at two of them. The first approach uses namespace packages and the pkgutil module, while the second uses the entry points functionality of setuptools. In the namespace package-based technique, we have the notion of the core package that defines a subpackage for extension. The core package specifies that one or more of its subpackages is open for extension. The core package will scan that subpackage at runtime to see what plugins have been configured. To add a plugin, developers can create other additions to the namespace package that put new modules in the subpackage. The first step in this technique then is to create the core package which defines an extension point. We'll continue working with our demo_reader package, and we'll implement the various compression modules as plugins. Here's the directory structure for the core package. As you can see, the compressed directory doesn't have any content now. This is the extension point, and subsequent additions to the namespace package will add modules to that subpackage. The next place we need to modify is multireader.py. Before, it had a hard-coded dictionary called extension_map that mapped file extensions to opener functions. Now we want it to discover these openers dynamically at runtime. We'll use the pkgutil module from the standard library to do this. Here's the new implementation of multireader.py. As you can see, we've replaced the extension_map definition with something substantially more complex. Let's look at the new elements. First, the iter_namespace function accepts a package object as its ns_pkg argument. It passes this to pkgutil.iter_modules, which knows how to find all subpackages of its argument. The second argument iter_modules is a prefix that is put at the front of the names of the modules that it returns. In this case, we are prefixing them with the name of the ns_pkg argument so that they are absolute names instead of relative. Next we use a set comprehension to build the compression plugins set. This takes the module names yielded from iter_namespace and imports each of them, putting the imported module into the set. Finally, we build extension_map by iterating over compression plugins and looking for the module-level attributes extension and opener in each. Extension must be a string that indicates the file extensions for which this plugin is used. Opener must be a function that knows how to open files with the extension suffix for reading. The implementation of MultiReader can stay the same because the structure of extension_map has not changed. At this point, the demo_reader package can be used to open simple text files, but it doesn't have any support for opening gzip or bz2 files. We'll add support for those compression formats with plugins. To add the bz2 compression plugin, we need to create a new directory tree that looks like this. As you can see, this has the same directory structure as the core package, but it only has one Python file, bzipped.py. We're using the namespace package system to put the bzipped module into the demo_reader.compressed package. Here are the contents of bzipped.py. The only real difference between this and the original form of this file is the extension attribute. Remember that our extension point requires this so that the MultiReader knows which plugins to use for various compression formats. Finally, we need to create the gzip plugin. Similar to the bz2 plugin, we need a directory structure like this. The contents of gzip.py are, again, very similar to the original version. With all of this in place, we can pull it all together by executing the demo_reader package and using it to read files of different formats. Of course, since we're using namespace packages, we need to make sure that the directories containing the different parts of the package are in PYTHONPATH. Previously we did that by manipulating sys.path within a REPL, but this time we'll do it on the command line. Note that this example is using Linux Mac OS syntax for setting PYTHONPATH. On Windows, you'll need to use the set command. Now we can execute the demo_reader package and it will discover and run the code in the gzipped plugin when given a gzip file, and it will discover and execute the code in the bzipped plugin when given a bz2 file.

Implementing Plugins with setuptools

The other technique we'll investigate for implementing plugins uses the notion of entry points from the setuptools package. Instead of the namespace package technique we just looked at, setuptools allows you to define named extension points, and plugins are provided by adding them to this extension point explicitly in your setup.py. Then at runtime your extensible package can iterate over the plugins which have been added to the extension point. As with the previous section, the first thing we need to do is to create the core demo_reader package. This time it will be a standard non-namespace package with a __init__.py. Here's the directory structure we need. The only difference from this package and that of the last example is the empty __init__.py at the root and the implementation of multireader.py. Since we're using a new plugin technique, multireader.py needs to be updated to find its plugins in a new way. As you can see, the difference here is in how we discover plugins. In this case we use the pkg_resources.iter_entry_points function to iterate over the extension point named demo_reader.compression_plugins. The pkg_resources package is installed when you install setuptools. This function returns a sequence of entry points, and the load method on these entry points returns the plugin object, in our case, a module. Of course, now we need to implement plugins for our gzip and bz2 compression formats. Unlike the last section where we used namespace packages, each of these plugins will need to be its own installable package with a setup.py. It's in the setup.py where we register a plugin with the extension point. Here is the directory structure for the bz2 plugin. As you can see, this is a complete, though minimal project. The code for the extension itself is in the demo_reader_bz2.bzipped module. This is no different from the bzipped module from the namespace package plugin example. It still has the module-level extension and opener attributes, which we'll use in multireader.py. The most interesting change is in setup.py. For the most part, this should all be familiar. It's a very standard setup.py. The only new part is the entry_points argument to setuptools.setup. It's a mapping from entry point names to lists of extensions. In this case, the only entry point name is demo_reader.compression_plugins, and it only has one extension called bz2, which refers to the plugin module. If you install both the demo_reader and demo_reader_bz2 packages, the multireader will find the bz2 plugin during its discovery process, and because the demo_reader_bz2.bzipped module has the extension and opener attributes, we'll be able to open bz2 compressed files. Before we do that, though, let's create the gzip plugin. It has an almost identical structure. As you probably expected, the gzipped.py file is the same as the gzipped.py from the earlier example. The setup.py file is very similar to that of the bz2 plugin. With all of this in place, we can install each of these packages into a virtual environment and see that the plugins provide the expected functionality. Install the demo_reader package into the environment. Next, install the bz2 plugin package. And then install the gz plugin package. Finally, we can execute our demo_reader package and use the plugins to read compressed files that we created earlier. In this part of the course, we've looked at a structure for our project that supports everything from code to tests to documentation. We've learned why separation of tests and production code is a good practice, we've seen how to install our packages into a Python environment, we've learned how to use plugins to extend packages, and we've investigated two methods, namespace packages and setuptools entry points, for implementing plugins.

Package Distribution

Source Distributions

One of the most important things you can do with a Python project is to make it available for others to use. In this module, we'll look at creating package distributions. Specifically, we'll distinguish between source and built distributions. We'll show how to create source and built distributions, we'll explain how built distributions can be portable or specific to particular hardware and operating system environments. We'll upload our package to the Python package Index. Finally, we'll install our package from the Python Package Index into a fresh virtual environment. While there are any number of ways of sharing your Python projects, for example, by creating an executable zip file as we saw earlier, the standard method is to create what's called a distribution package. A distribution package is an archive of your package's contents that others can easily install into their own Python environments, and they come in various forms including zip files, tarballs, and wheels, which we will discuss later. There are two primary types of distribution packages, source and built. A built distribution is one that can be placed directly into a Python installation directory and used with no further steps. It's called built, because any build steps required for the package have already been performed and rendered into the distribution. Since some kinds of Python packages require platform-specific build steps and results, a built distribution can also be platform-specific, requiring different archives for different systems. A source distribution, on the other hand, contains all of the source and resources needed to build the package. Unlike a built distribution, a source distribution cannot simply be placed into a target file system, rather, you must run a command to build the package before installing it. The standard way to create a source distribution is by passing the sdist argument to your setup.py. Let's do that with the project we created in the last section. Go to the directory containing your setup.py and run python setup.py sdist. You'll see it print several lines of output, including some warnings about missing metadata in setup.py, and when it's done, you'll see that you have a new directory called dist. If you look in dist, you'll see the file demo_reader- 1.0 .0 .tar .gz. This is your source distribution package. You can install this into a Python environment using pip.

Built Distributions

As we mentioned earlier, the other category of distribution packages is called built. Unlike a source distribution, a built package doesn't require any processing before it can be installed. Any build steps the package requires have already been performed. There are a number of built distribution formats, but the most important of these is known as wheel. The wheel format was defined in PEP 427, and it is the recommended format for distributing Python packages. To create a wheel distribution, you need to first install the wheel package into your Python environment. Once you've installed wheel, you can create the distribution package by passing bdist_wheel to setup.py. Much like when you created the sdist above, this will print a number of lines to your console, and in the end you'll find a new file in your dist directory called demo_reader- 1.0 .0 - py3-none- any.whl. Also like the source distribution, you can install the package using pip, by passing the wheel file as an argument to the pip install command. Since built distributions like wheel might have platform or Python version-specific elements, it's important that we be able to tell when the distribution can be used on a particular system. Wheels like the one we just created include this information in their filename. The part that says py3-none-any, tells us when this particular package can be used. The first tag, py3, tells us what Python versions the package will work with. In this case, the package is only applicable for Python version 3. The next tag, none, indicates the application binary interface, or ABI, requirements for the package. In this case, since the package is pure Python code, we do not have a binary interface so we have no ABI constraints. But if the package included any compiled elements, for example, C++ or Rust code, then this tag would tell us whether the distribution would work on our system. Finally, the third tag, any, tells us what platforms or operating systems the distribution will work on. Since our package doesn't have any platform-specific elements, the any tag tells us that this package works on any platform.

Uploading Packages to a Package Server

When developing packages, you will normally want to create the built distributions that you think are necessary, along with the source distribution that anyone can use. Once you have those, though, you need to make sure that people can actually get them. The standard solution for making your distributions available to others is to upload them to the Python Package Index at PyPi.org. This is the central repository for publicly available Python packages, and anyone is free to upload packages there after creating an account. Since the package repository is a shared resource, you won't be able to upload the demo_reader package we've been developing, the name demo_reader is already taken in the repository, but we'll show you the steps you would take to upload your own package so that you can see how it's done. Before you can upload packages to the Python Package Index, you need to register an account there. You do this by going to PyPi.org and clicking the Register link. This will take you to a page where you can fill in your account details and create the account. Once you've created an account, you're ready to upload your package. The tool you use to upload packages to the Python Package Index is called twine. You tell twine the distribution you want to upload, and it will make sure it gets uploaded to the correct place in the index. Before you can use twine, you'll need to install it using pip, python -m pip install --user --upgrade twine. This form of the pip command ensures that you're using the version of pip associated with the version of Python you're currently using. Once this completes, you will be able to upload packages by using the twine upload command like this. As you can see, we passed the name of the distribution file as an argument to twine upload. Twine will ask you for your PyPi.org username and password, and if the login is successful it will upload your package. Once a package is uploaded, you can navigate to PyPi.org and find the package in the index. This means that people can now install your package directly from PyPi using pip. We can demonstrate this by creating a new virtual environment. We'll create a temporary directory called temp and navigate into it. Inside temp, we'll execute the Python standard library venv module to create our new virtual environment, which will also be called venv. Before installing, we must remember to activate our new environment using the activate script included in the environment. Once we're in our new environment, it's good practice to ask pip to upgrade itself before using it to install other packages, python3 -m pip install --upgrade pip. Now we're ready to install our package from PyPi with pip install demo_reader. We can see that pip is indeed downloading our package from PyPi and installing it into our virtual environment. We can verify this by starting Python, importing the demo_reader module, and checking its __file__ attribute, which we will see points to the site-packages directory of our freshly created virtual environment. And that's it. With the tools we've just covered you can create and share your Python packages with the world.

Summary

In this part of the course, we've seen how to make source distributions with python setup.py sdist, and built distributions with python setup.py bdist_wheel. We've seen that wheels are packaged built distributions which may contain platform-specific code. We've used twine to upload the package to the Python Package Index, and we've used pip to retrieve our package from the Python Package Index and install it into a new Python environment. Well done on completing Core Python: Organizing Larger Programs. We hope we have given you enough guidance to begin constructing your own modular systems in Python, and to navigate and maintain existing Python systems. Look out for other Core Python courses here on Pluralsight which build on the knowledge you've gained here, and which explain the many other tools and abstractions provided by Python for managing complexity. Remember to check out our Python Craftsman book series, which covers these topics in written form. Specifically, you'll find these topics covered in the Python Journeyman, the second book of the trilogy. We'll be back with more content for the ever-growing Python language and library. Please remember, though, that the most important characteristic of Python is that above all else it's great fun to write Python software, so enjoy yourselves.

Course authors

Austin Bingham

Robert Smallshire

Course info

Level	Intermediate
Rating	★★★★☆ (35)
My rating	★★★★★
Duration	1h 5m

Released 8 Mar 2019

Share course