Prerequisites

Prerequisites

Hello. My name is Austin Bingham, and welcome to Python: Beyond the Basics. This is a course for people who already know the essentials of the Python programing language and are ready to dig deeper, to take the steps from novice to journeyman. In this course, Robert Smallshire and I will cover topics to help you prepare to produce useful, high-quality, Python programs in professional commercial settings. Python is a large language, and even after this course there will be plenty left for you to learn, but we will teach you the tools, techniques, and idioms you need to be a productive member of any Python development team. 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. If you find that you need to brush up on Python basics before you start this course, you can always watch the Python Fundamentals course first. Python: Beyond the Basics was specifically designed as a follow on course to Python Fundamentals, and Python Fundamentals will get you up to speed on all of the topics you need for this course. First and foremost you will need access to a working Python 3 system for this course. Any version of Python 3 will suffice, and we have tried to avoid any dependencies on Python 3 minor versions. With that said, more recent Python 3 versions have lots of exciting new features and standard library functionality, so if you have a choice you should probably get the most recent staple 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. You will of course need to know how to define functions, and you need to be comfortable with concepts like keyword arguments, default argument values, and returning values from functions. Likewise, for this course you need to know how to work with basic, single file module 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 this course we'll make extensive use of Python's basic built-in types, so you need to make sure that you are fluent in their syntax and application. In particular, you need to make sure that you know the following types well: int, float, str, list, dict, and set. Many of our examples in this course use these types liberally and without undue explanation, so review these before proceeding if necessary. Like the basic types we just mentioned, this course assumes that you are familiar with the basic Python object model. Python: Beyond the Basics goes into greater depth on some advanced object model topics, so make sure you understand concepts like single inheritance, instance attributes, and other topics covered in Python Fundamentals. In Python exceptions are fundamental to how programs are built. We'll assume that you're familiar with the basic concept of exceptions, as well as the specifics of how to work with them in Python. This includes raising exceptions, catching them, finally blocks, and defining your own exceptions. In this course you will learn how to define iterable objects and iterators, so we expect you to already know how to use them. This includes syntax like the for loop, as well as how to use the next and iter functions to manually iterate over sequences. Like functions, which we mentioned earlier, classes are a basic part of Python, and we'll expect you to be very comfortable with them in this course. You will need to know how to define classes and give them methods, as well as create and work with instances of them.

In Python, as with many languages, you can treat the data in files in one of two basic ways, text and binary. In this course we'll work with both kinds of files, so you need to make sure that you understand the distinction and the ramifications of the two different modes. And of course you need to know how to work with files in general including opening, closing, reading from, and writing to them.

Before you start this course, make sure that you are familiar with unit testing, debugging, and basic deployment of Python programs. Some of these topics will be used directly in the course. Perhaps more importantly, you may want to apply these skills to the code you write as a part of this course. Some of the topics in this course can be complex and a bit tricky, so knowing how to test and debug your code as you learn might be very useful.

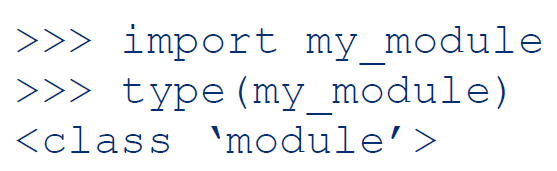
Finally, we need to make a quick note regarding terminology. In Python many language features are implemented or controlled using special methods on objects. These special methods are generally named with two leading and two following underscores. This has the benefit of making them visually distinct, 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 \_\_len\_\_, which as you'll recall is invoked by the len function, we'll say "dunder-len. " These kinds of methods play a big role in this course, so we'll be using this convention frequently.

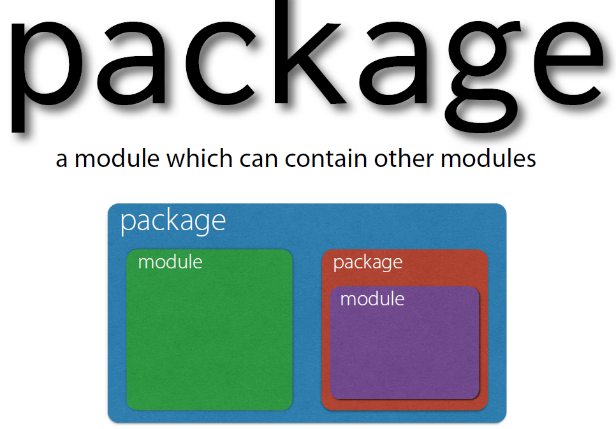
Python is becoming more popular every day, and it's being applied in all sorts of domains and applications. One of Python's strengths is that it's approachable and easy to learn so that almost anyone can learn to write a basic Python program. As the title says though, this course will take you beyond that, beyond the basics. We want to teach you some of the deeper aspect of Python to give you the skills you need to write great Python programs.

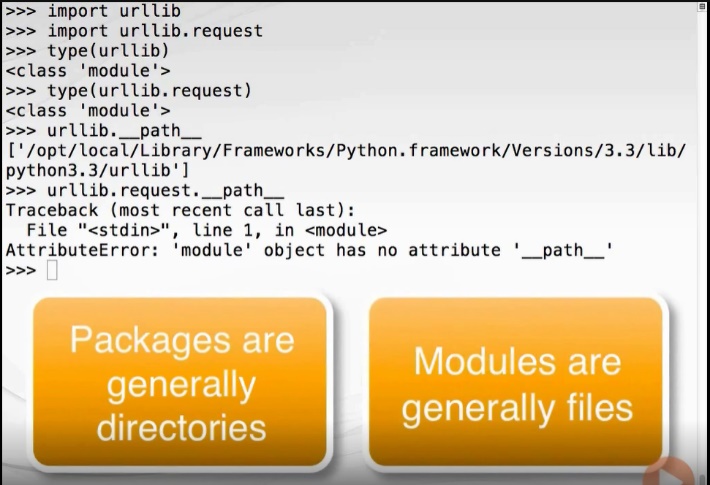
# Organizing Larger Programs

## Packages

Hello. My name is Austin Bingham, and welcome to the first module of Python: Beyond the Basics. In this module we'll be covering more of Python's techniques for organizing programs. Specifically, we'll be looking at Python's concept of packages and how these can be used to add structure to your program as it grows beyond simple modules. As you'll recall, Python's basic tool for organizing code is the module. A 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 communicate 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 are both of type module. Urllib. request is nested inside urllib. In this case, urllib is a package and request is a normal module. If you look closely at each of these objects, you'll notice an important difference. The urllib package has a dunder-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.

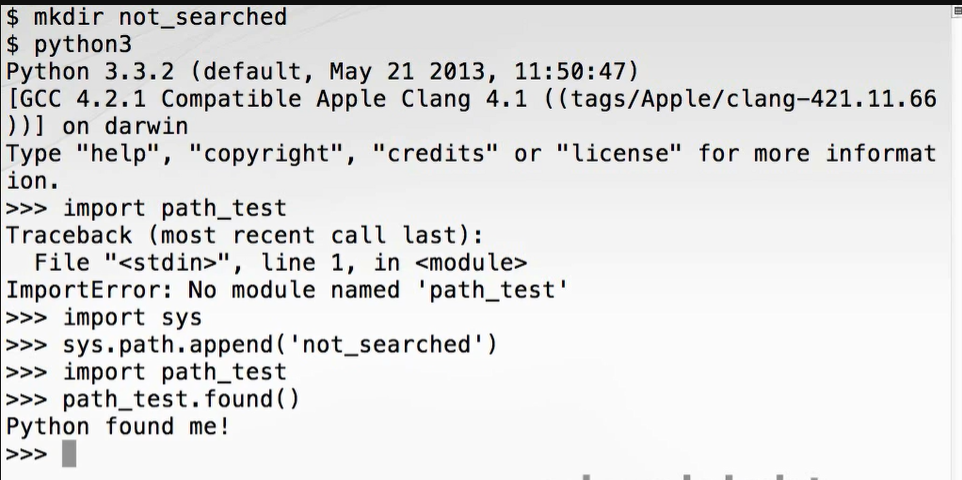
Note that in Python prior to version 3. 3 dunder-path was just a single string, not a list. In this course, we're focusing on Python 3. 3, but for most purposes the difference is not important. Before we get into the details of packages, it's important to understand how Python locates modules. Generally 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?

## Imports from sys.path

The answer is that Python checks the path attribute of the standard sys module commonly referred to as sys. path. Sys.path is nothing more than 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 the next and so forth until a match is found or Python runs out of entries in sys. path in which case an import error is raised.

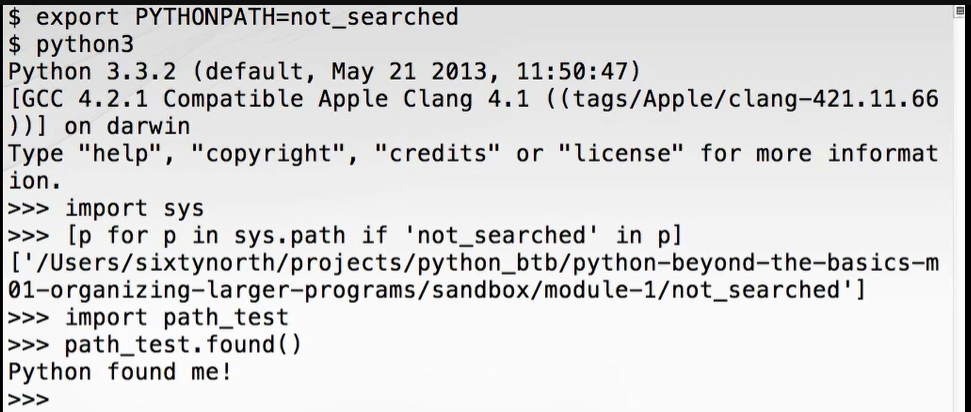
Let's explore sys. path in the REPL. Go to your command line and start Python with no arguments. As you see, your sys. path can be quite large. The entries depend on a number of factors including how many third-party packages you've installed and how you've installed them. For our purposes a few of these 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 the 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 the following function definition. (Typing) Now start your REPL from the directory containing the not\_searched directory, and try to import path\_test. The path\_test module is not found because it's not in a path in sys. path. To make it importable, let's add not\_searched to sys.path by appending it. (Typing) And now when we try to import path\_test we see that it works.

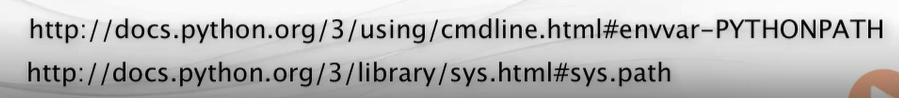


Knowing how to manually manipulate sys. path can be useful, and sometimes it's the best way to make code available to Python.

There is another way to add entries to sys. path though that doesn't require direct manipulation of the 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. That is for Windows it's a semicolon separated list of directories while on Linux and Mac OS X it's a colon separated list of directories. To see how PYTHONPATH works, let's add not\_searched to it before staring Python again. On Windows use the set command. On Linux or Mac OS X the syntax will depend on your shell, but for Bash-like shells you can use export. Now start your REPL and check that not\_searched is indeed in sys. path. (Typing) And of course we can now import path\_test without manually editing sys. path.

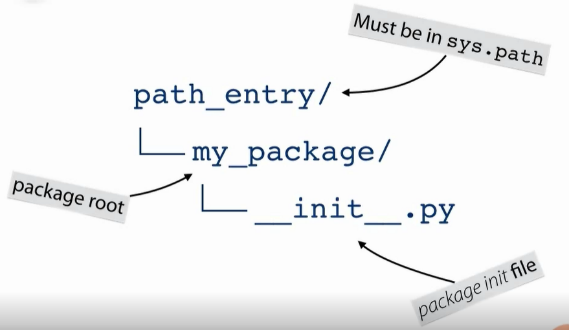


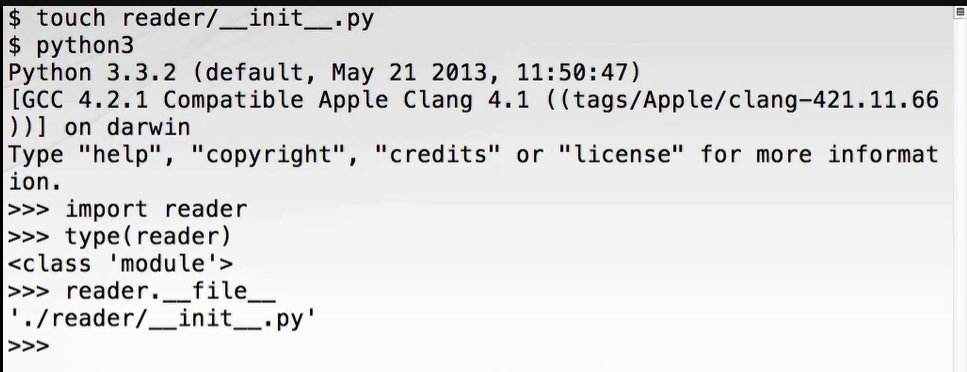
There are more details to sys. path and PYTHONPATH, but this is most of what you need to know. For more information, you can check these links to the Python documentation.



Implementing Packages

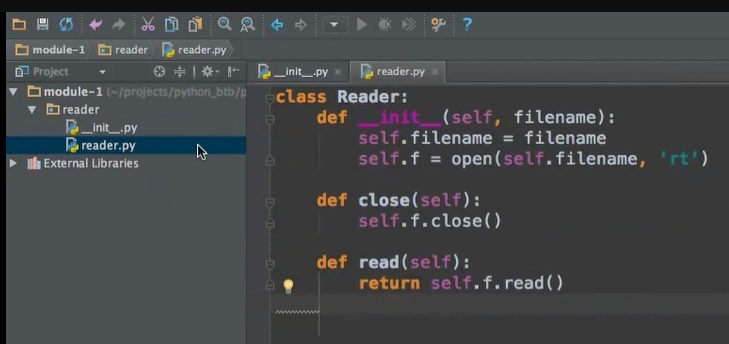
So, how are packages implemented? To create a normal module, you simply created a Python source file and made sure it was in sys.path. The processor creating packages is not much different. To create a package, first you create the package's root directory somewhere in sys. path. Then in that directory 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.

 As with many things in Python, an example is much more instructive than words. First, go to your command prompt and create a new directory called reader. To turn that directory into a package, we need to create the package init file. Create a new empty file in the reader directory called \_\_init\_\_. py. On Linux or Mac OS X you can use the touch command or on Windows use type. Now, if you start your REPL, you'll see that you can import reader, and to see the role that \_\_init\_\_. py plays in the functioning of a package check the \_\_file\_\_ attribute of the reader package. So we see that reader is a module, and the source file that is imported when reader is imported is the package init file in the reader directory. In other words, a package is just a directory containing a file named \_\_init\_\_. py.

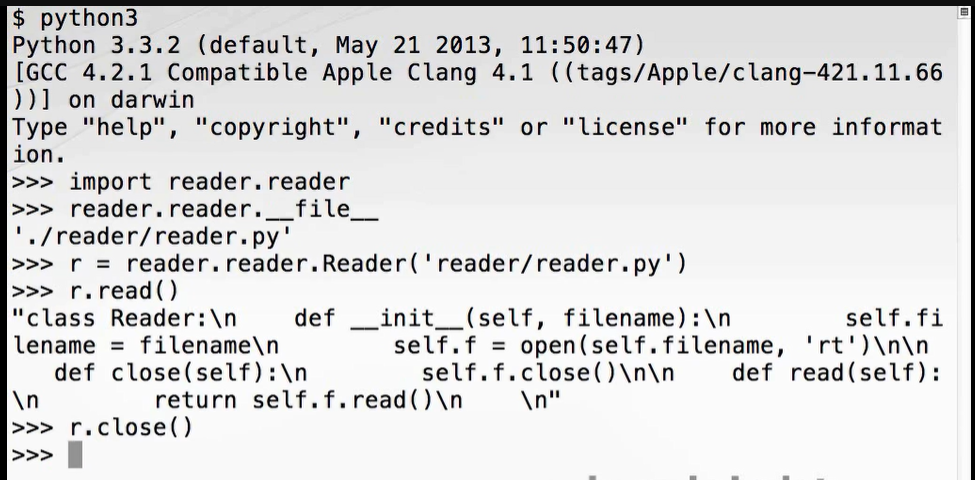


To see that the \_\_init\_\_. py is actually executed like any other module when you import reader, let's add a small bit of code to it. (Typing) Now, restart your REPL and import reader, and you'll see our little message printed out.

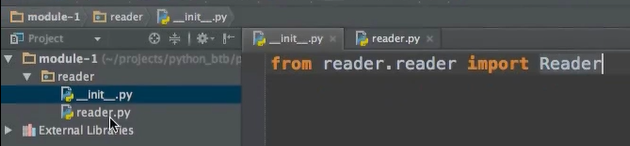
Now that we've created a basic package, let's add some useful content to it. For reader we want to implement a very simple class called reader, which knows how to read text from both normal files, as well as a number of compressed formats. You'll start by defining the reader class, which just knows how to read from normal files. First, let's get rid of the print statement from the reader package init file. Now create a new file in the reader directory called reader. py. (Typing)



Now if you start a new REPL you can import your new module and try out the Reader class. (Typing) Let's use it to read the contents of reader. py. (Typing) That's a bit loopy.



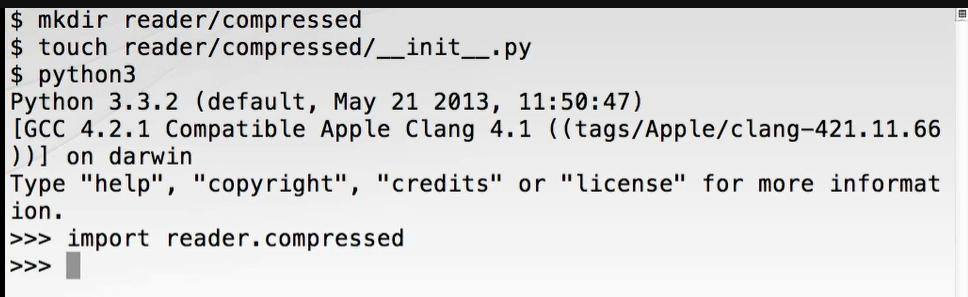
One thing that's a bit ugly about the code is the fact that we need to type reader.reader. It would be much cleaner if the reader class was in the top level reader package. Let's edit reader \_\_init\_\_.py to hoist the reader class up into the top level reader package. Remember that the \_\_init\_\_.py file is executed when the reader package is imported and is perfectly normal for package init files to run all sorts of code.



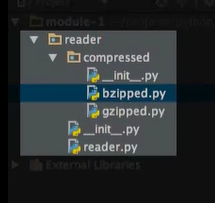
Now if you restart your REPL you'll see that the reader class is in the top level reader namespace. That's much cleaner. (Typing)

## Subpackages

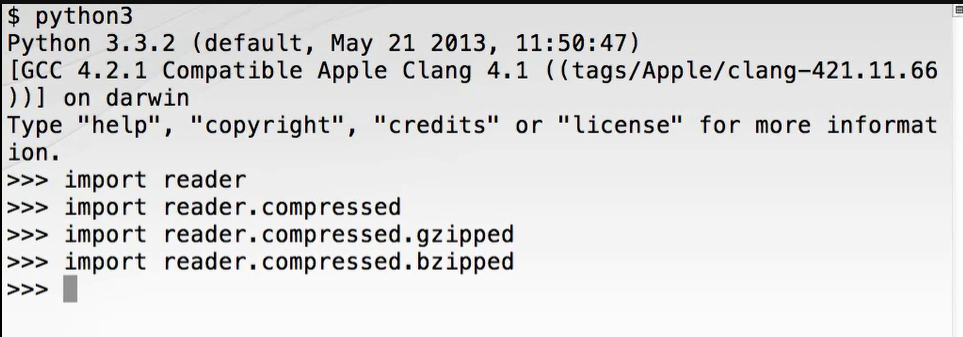
To really see how packages provide high levels of structure to your Python code, let's see how we can add more layers of packages to the reader hierarchy. We're going to add a sub package to reader called compressed, which will contain some code for working with compressed files. So, first let's create the new directory and package init file. (Typing) If you restart your REPL, you'll see that you can import reader.compressed.



In the compressed directory, create a file called gzipped.py. This will contain some code for working with the gzip compression format. Similarly, create another file in that directory called bzipped. py. At this point you should have a directory structure that looks like this.

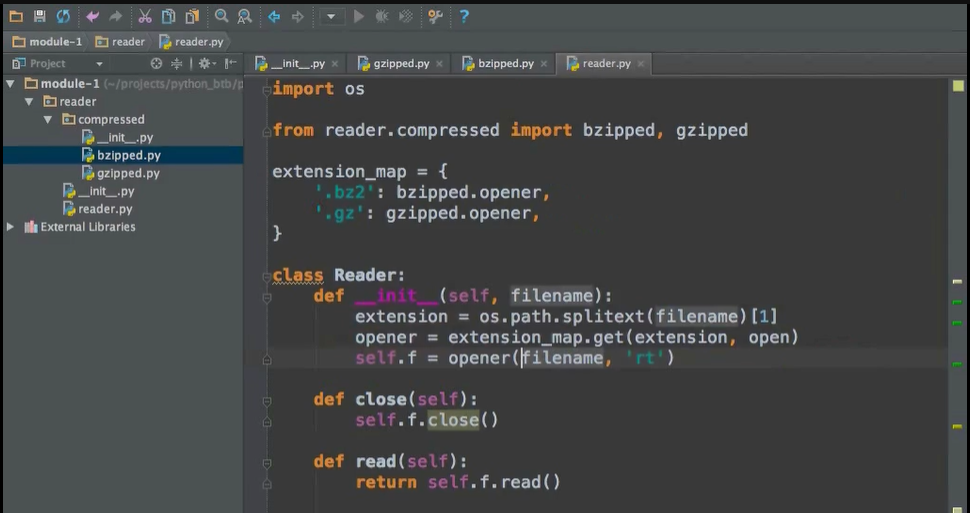


If you start a new REPL, you'll see that we can import all of our modules just as you might expect. (Typing)



## Example: A Full Program

So, let's glue all of this together into a more useful sort of program. We'll update reader to know how to read from gzipped files, bz2 files, and normal text files all based on file extensions. First, update your reader class to use the compression handlers when necessary. (Typing) Now reader will check the extension of the filename it's handed. If that extension is in the extension map, a specialized file opener will be used, in this case either the bz2 or gzip openers. Otherwise, the normal file open function will be used.



To test this out, let's first create some compressed files using the utility functions we built into our compression modules. Execute the modules directly from the command line. Now start a new REPL, and let's take our code for a spin. (Typing)

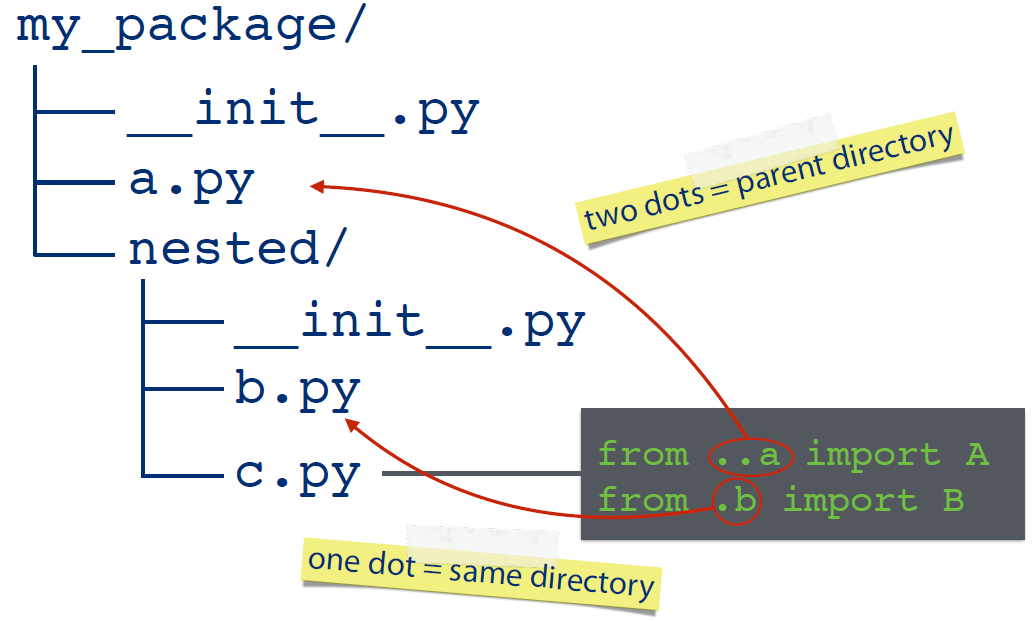


* So, we've covered a lot of information. Let's review what we've learned so far.
* Packages are modules, which can contain other modules.
* Packages are generally implemented as directories containing a special \_\_init\_\_. py file.
* A \_\_init\_\_. py file is executed when the package is imported.
* Packages can contain sub packages, which themselves are implemented with \_\_init\_\_. py files in directories.
* The module objects or packages have a \_\_path\_\_ attribute.

## Relative Imports

Up to now we've always imported modules in packages using their full names. For example, when we imported the reader class into the reader top level package, we used from reader. reader import Reader. This is called an absolute import because we specify the full package and sub module's path to the reader sub module. There's nothing wrong with this approach, but sometimes it can be tedious to write fully specified absolute paths when importing within a package. There are a few more topics related to packages that we need to cover in this module. The first topic is the concept of relative imports.

Relative imports allow us to use shortened paths to modules and packages. Critically though, this only applies when you're importing something inside the same top level package. That is in our reader example you could use relative imports to import any module under the top level reader package from any modules under the top level reader package, but imports from other top level packages to reader could not be relative, and modules in reader can't make relative imports of modules outside of reader. To perform a relative import, you have to use the “from module import names” form of import, and you start the module portion of the statement with one or more dots. Each of the dots stands for a package in the hierarchy containing the current module. One dot means the package containing the current module. Two dots mean the package containing the package containing the current module and so forth. Everything after the leading dots are simply module and package names relative to the package indicated by the dots.



Let's illustrate this with some simple examples. Suppose we had this package structure, and further suppose you needed to import the ruminate function from farm.bovine.common into farm.bovine.cow. In cow.py you could use an absolute import or you could use a relative import. The leading dot in the relative form means the package containing this file or in other words the farm.bovine package. So, the relative form is really just saying from farm. bovine. common import ruminate. You can also use just dots with no module in the from section of a relative import. In this case, the dots are still interpreted in exactly the same way, and the import section must specify a module name. So, going back to our example where we import ruminate into cow.py you could instead use this in which case the ruminate function would need to be qualified with common when you call it.

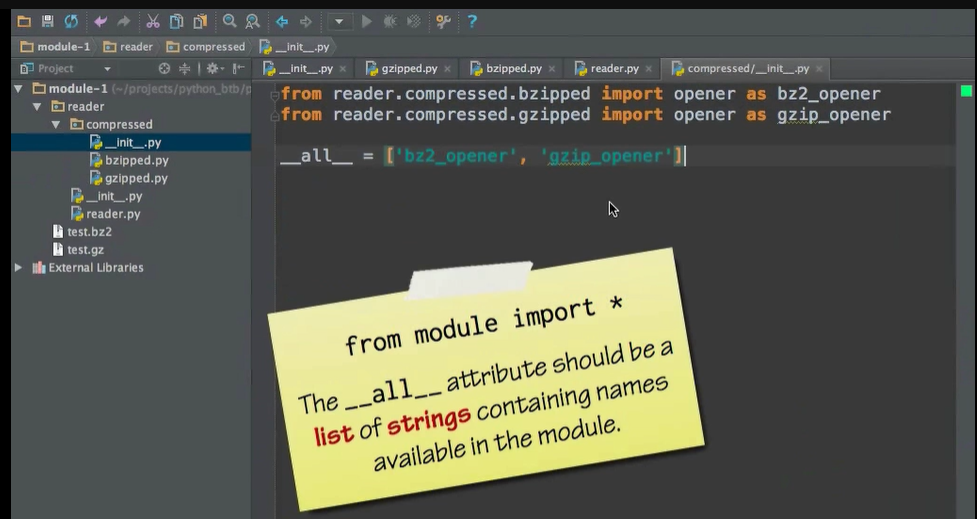
It's easy to see how relative imports can be useful for reducing typing in deeply nested package structures. They also promote certain forms of modifiability since they allow you in principle to rename top level and sub packages in some cases. On the whole though, the general consensus seems to be that relative imports are best avoided in most cases.

## Controlling Imports with \_\_all\_\_

Another topic we want to look at is the optional dunder-all attribute of modules. Dunder-all lets you control which attributes are important when someone uses the from module import \* syntax. If dunder-all is not specified, then from X import \* imports all public names from the imported module. Dunder-all must be a list of strings, and each string indicates a name, which will be imported when the star syntax is used. For example, let's see what import \* does if we import our compressed modules. What we see is that import \* imported the bzipped and gzipped sub modules of the compressed package directly into our local namespace.



What we might prefer is that import \* imports the different opener functions from each of these modules. Let's update compressed \_\_init\_\_. py to do that.



Now, if we use import \* on reader. compressed, we get the two opener functions with new names. So, dunder-all can be a useful tool for limiting what names are exposed from your modules. We still don't recommend that you use the import \* syntax as a rule, but it's good to know about dunder-all since you're likely to see it in the wild.

## Namespace Packages

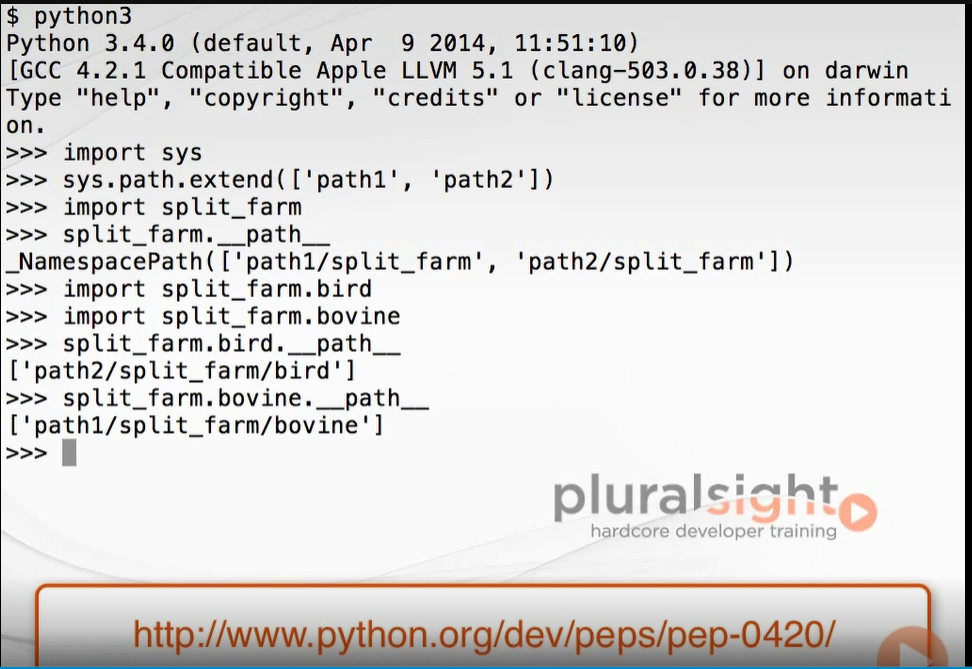
Earlier we said that packages were implemented as directories containing a \_\_init\_\_. py file. This is true for most cases, but there are certain cases where you want to be able to split packages across multiple directories. This is useful, for example, when a logical package needs to be delivered in multiple parts. This happens in some of the larger Python packages. Several approaches to addressing this need have been implemented,

but it was in PEP420 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 don't 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 reason for this limitation is primarily that it avoids complex questions of initialization order when multiple directories contribute to a package.

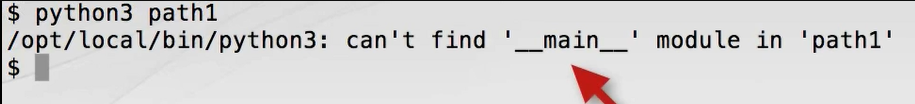
But if namespace packages don't have \_\_init\_\_. py files, how does Python find them during import? The answer is that Python follows a relatively simple algorithm to detect namespace packages. When asked to import the name foo, Python scans each of the entries in sys. path in order. If in any of these 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 turn the farm package into a namespace package called split\_farm. Instead of putting all of the code under a single directory, we would have two independent parts rooted at path1 and path2 like this. This splits the distribution of the farm package between birds and bovine, which seems pretty natural. Now, to import farm 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. Now when we import split\_farm we see that its dunder-path includes portions from both path1 and path2. (Typing) And when we import the bird and bovine sub packages, we see that they are indeed coming from their respective directories. There are more details to namespace packages, but this addresses most of the important details that you'll need to know. In fact, for the most part it's not likely that you'll need to develop your own namespace packages at all. If you do want to learn more about them though, you can start by reading PEP420 at www. python. org/dev/peps/pep-0420/.



## Executable Directories

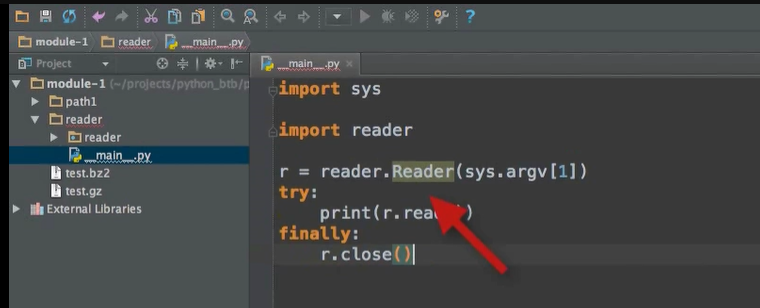
Packages are often developed because they implement some program that you want to execute. There are a number of ways to construct such programs, but one of the simplest is through the use of executable directories. Executable directories let you specify a main entry point, which is run when the package is executed by Python. First, what do we mean when we say that Python executes a directory? What we mean is when you pass a directory name to Python on the command line like this.



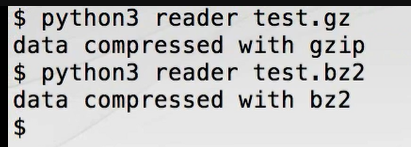
Normally this doesn't work, and Python will complain saying that it can't find a dunder-main module. However, as that error message suggests, you can put a special module named \_\_main\_\_. py in a directory, and Python will execute it.

This module can of course execute whatever code it wants meaning that it can call into a package you've created to provide a user level interface.

To illustrate this, let's create a program out of our reader package. First, let's move our existing reader package into a new directory also called reader like this. In this new directory, you can see that we've created a new file called \_\_main\_\_. py. Let's add a little code to that file to see how it works. (Typing) Now, if we pass this top level reader directory to Python, we will see our \_\_main\_\_. py executed. We see that \_\_main\_\_. py is named \_\_main\_\_ much as we might expect. What's also interesting to note is that the top level reader directory is automatically placed into our import path. This means that \_\_main\_\_. py can import our reader module. Let's leverage that to turn \_\_main\_\_. py into a simple driver for our reader package. Edit \_\_main\_\_. py to look like this. (Typing)

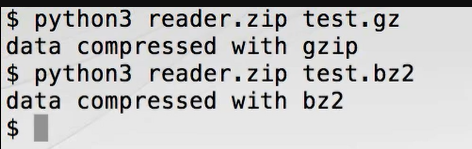


Here we create a reader object passing our first command line argument to its initializer. We then read the contents of the argument, print them, and close the reader. Here's how that looks on the command line. (Typing)



This is a really convenient way to bundle up small programs, and it makes it easy to distribute them to people.

We can take this executable directory idea one step further by zipping the directory. Python knows how to read zip files and treats them like directories meaning that we can create executable zip files just like we created executable directories. First, let's create a zip file from our top level reader directory. (Typing) Now we can pass that zip file to Python and get that exact same results at the previous example.



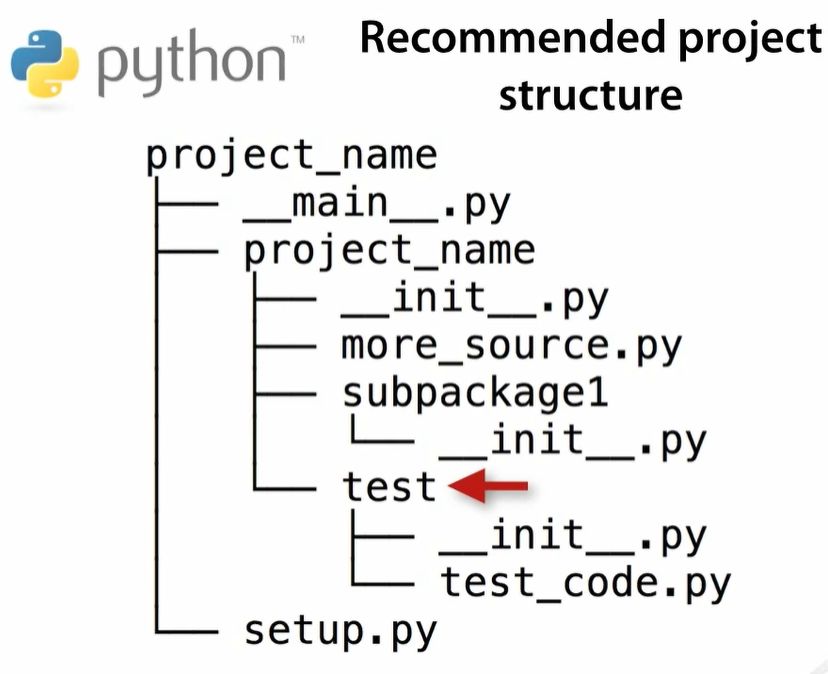
## Recommended Layout

As we close out this module, let's look at how you can best structure your projects. There are no hard and fast rules about how you lay out your code, but some options are generally better than others. What we'll 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 and things like licenses. If you intend to make your package executable, you can also include \_\_main\_\_. py in the top level directory.

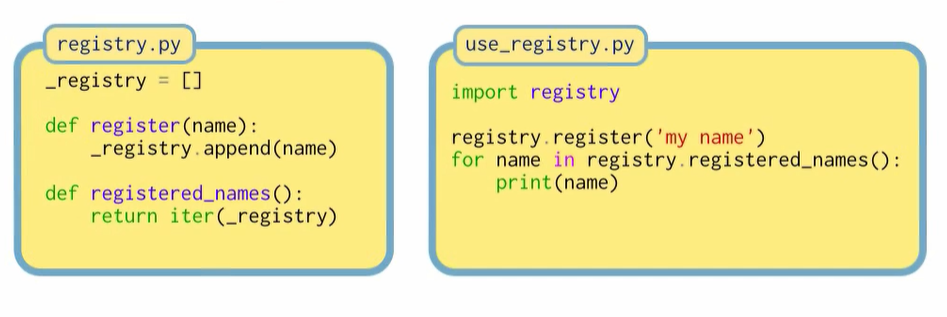
The next directory down is your actual package directory. This has the same name as your top level directory. Again, there's no rule that says that this must be so, but it's a common pattern and makes it easy to recognize when you're navigating your project.

Your package contains all of the code as you might expect including any sub packages. Importantly, you'll see it includes a test sub package. This sub package is the root of all of your tests. There's not much more to it than that. This is a very simple structure, but it works well for most needs, and it serves as a fine starting point for more complex project structures.



## Duck Tails: Modules Are Singletons

The singleton pattern is one of the most widely known patterns in software development in large part because it's very simple and in some ways provides a superior option to the dreaded global variable. If you find that you need a singleton in Python, one simple and effective way to implement it is as module level attribute. Since modules are only ever executed once, this guarantees that your singleton will only be initialized once. And since modules are initialized in a well-defined user control order, you can have strong guarantees about when your singleton will be initialized. This forms a strong basis for implementing practical singletons. Consider a simple singleton registry implemented in registry. py where callers could leave their names. Callers would use it like this. The first time registry is imported the \_registry list would be initialized. Then every call to register and registered\_names would access that list with complete assurance that it had been properly initialized. You will recall that the leading underscore on \_registry is a Python idiom indicating that \_registry is an implementation detail that should not be accessed directly. This simple pattern leverages Python's robust module semantics and is a useful way to implement singletons in a safe, reliable way.



## Summary

Packages are an important concept in Python, and in this module we've covered most of the major topics related to implementing and working with them. Let's review the topics we looked at.

* Packages are a special type of module. Unlike normal modules, packages can contain other modules including other packages. Package hierarchies are a powerful way to organize related code.
* Packages have a \_\_path\_\_ member, which is a sequence specifying the directories from which a package is loaded.(Multiple directories in case of namespace packages)
* Sys.path is a list of directories where Python searches for modules.
* Sys. path is a normal list and can be modified and queried like any other list.
* If you start Python with no arguments, an empty string is put at the front of sys.path. This instructs Python to import modules from the current directory.
* Appending directories to sys.path at runtime allows modules to be imported from those directories.
* PYTHONPATH is an environment variable containing a list of directories. The format of PYTHONPATH is the same as for PATH on your system. It's a semicolon separated list on Windows and a colon separated list on Linux or Mac OS X. The contents of PYTHONPATH are added as entries to sys.path.
* Normal packages are implemented by putting a file named \_\_init\_\_.py into a directory. A \_\_init\_\_. py file for a package is executed when the package is imported.
* \_\_init\_\_. py files can hoist attributes from sub modules into higher namespaces for convenience.
* Modules can be executed by passing them to Python with the -m argument.
* Relative imports allow you to import modules within a package without specifying the full module path. Relative imports must use the from module import name form of import. The from portion of a relative import starts with at least one dot. Each dot in a relative import represents a containing package. The first dot in a relative import means the package containing this module. Relative imports can be useful for reducing typing. Relative imports can improve modifiability in some cases. In general, it's best to avoid relative imports because they can make code harder to understand.
* The \_\_all\_\_ attribute of a module is a list of strings specifying the names to export when from module import \* is used.
* A namespace package is a package split across several directories. Namespace packages are described in PEP420.
* Namespace packages don't use \_\_init\_\_. py files. Namespace packages are created when one or more directories in the PYTHONPATH match an import request and no normal packages or modules match the request.
* Each directory that contributes to namespace package is listed in the package's \_\_path\_\_ attribute.
* Executable directories are created by putting a \_\_main\_\_.py file in a directory. You execute a directory with Python by passing it to the Python executable on the command line. When \_\_main\_\_. py is executed, its dunder-main attribute is set to dunder-main. When \_\_main\_\_. py is executed, its parent directory is automatically added to sys. Path.
* Executable directories can be compressed into zip files, which can be executed as well. Executable directories and zip files are convenient ways to distribute Python programs.
* A simple standard project structure includes a location for non-Python files, the project's package, and a dedicated test sub package.
* Module level attributes provide a good mechanism for implementing singletons. Modules have well-defined initialization semantics.

Thanks for watching, and we'll see you in the next module.

# **Beyond Basic Functions**

## **Function Review**

Hello. My name is Robert Smallshire. Welcome to the second module of the Python: Beyond the Basics course where we'll cover a generalization of functions known as callable objects and explore some other types of callable objects including callable instances and lambdas. Up to now we've encountered free functions, which are defined at module or global scope and methods, which are functions enclosed within a class definition. The first parameter to an instance method is the object on which the method is invoked. Methods can also be overridden in subclasses. We've seen that function arguments come in two flavors, positional and keyword. Positional arguments used in a call are associated with the formal arguments used in the definition in order. Keyword arguments use the name of the actual argument at the call site to associate with the name of the formal argument in the definition and can be provided in any order so long as they follow any positional arguments. The choice of whether a particular argument is a positional or keyword argument is made at the call site, not in the definition. A particular argument may be passed as a positional argument in one call, but is a keyword argument in another. Furthermore, in the function definition each argument may be given a default value. It's important to remember that the right hand side of these default value assignments is only evaluated once at the time the enclosing def statement is executed, which is typically when a module is first imported. As such, care must be taken when using mutable default values which can inadvertently retain modifications between calls. Lastly, we have seen that just like almost everything else in Python functions are first class, which is to say they are objects which can be passed around just like any other object. As we have seen, the def keyword is responsible for binding a function object, which contains a function definition to a function name. Here we create a simple function resolve, which is just a thin wrapper around a function from the Python standard library socket module. Inspecting the resolve binding shows that it refers to a function object, and to invoke the function we must use the postfix parentheses, which are the function called operator. In a very real sense then, function objects are callable objects in so far as we can call them.

Callable Instances

Here we'll introduce the Python special method call, which is call delimited by double underscores, which we'll call dunder-call for short. This method allows objects of our own design to become callable just like functions. To demonstrate, we'll make a caching version of our DNS resolver in a file resolver. py. We're using PyCharm in this example, but you can use any text editor, preferably one which supports Python syntax highlighting. Occasionally we would like to have a function which maintains some state between calls. In general, implementing functions for which the return value depends on the arguments to previous calls is frowned upon and rightly so because of the difficultly of reasoning about and therefore testing and debugging such functions. That said, there are legitimate cases for retaining information within a function between calls such as to implement a caching policy to improve performance. In such cases, the values returned by the function given particular arguments are not changed, but the time within which a result is produced may be reduced. Notice that just like any other method dunder-call accepts self as its first parameter, although we won't need to provide this when we call it. Now, let's test this from Python REPL. (Typing) We must call the constructor of our class to create an instance object, which because we implemented the dunder-call method can be called like a function. This is really just syntactic sugar for spelling the dunder-call method name out in full, but we would never do this in practice. Since resolve is an object of type resolver, we can retrieve its attributes to inspect the state of the cache. The \_cache attribute, which is a dictionary, contains just a single entry. Let's make another function call. Now we see that the cache has grown. In order to convince ourselves that the caching is working as designed, we can run some simple timing experiments using the Python standard library timeit module, which contains a handy timing function also called timeit. For reasons we won't go into now, the timeit function accepts two code snippets as strings, one of which is used to perform any set of operations and the other of which is the code for which the elapsed time will be reported. The function also accepts a number argument, which is a count of how many times the code in the test will be run. In our case of testing a cache, it's important that we set this to 1. Since our code to be timed will refer to names in the current namespace, that of the REPL, we must specifically import them from the REPL namespace, which is called dunder-main, into the namespace used by timeit. You can see here that the DNS lookup took 688 microseconds. Now execute the same line of the code again. This time the time taken is short enough that Python reports it in scientific notation. Let's ask Python to report it without scientific notation using the str. format method relying on the fact that the special underscore variable stores the previous REPL result. Now it's easier to see that the result is returned from the cache in around 8 microseconds, a factor of 100 faster. Our callable object has been instantiated from a regular class, so it's perfectly possible to define other methods within that class giving us the ability to create functions, which also have methods. For example, let's set a method called clear to empty the cache and another method called has\_host to query the cache for the presence of a particular host. Let's exercise our modified Resolver in a fresh REPL session. First we import and instantiate our Resolver callable as before. Then we can check whether a particular destination has been cached, which it is not in this case, although of course resolving that host by invoking our callable changes that result. Now we can test that cache clearing works as expected, which of course it does. So, we see how the special dunder-call method can be used to define classes, which when instantiated can be called using regular function call syntax. This is useful when we want a function which maintains state between calls and optionally needs to support attributes or methods to query and modify that state.

Classes Are Callable

It may not be immediately obvious, but the previous exercise also demonstrates a further type of callable object, and that is the class object. Remember that everything in Python is an object, and that includes classes. We must take great care when discussing Python programs because class objects and instance objects produced from those classes are quite different things. Let's start a new REPL session and give a practical demonstration using our Resolver class. First we'll import just the class from our module. When we ask the REPL to simply evaluate the imported name Resolver, the REPL displays a representation of the class object. This class object is itself callable, and of course that is what we've been doing all along whenever we've called a constructor to create new instances, so we see in Python constructor calls are made by calling the class object. As we have seen, any arguments passed when the class object is called in this way will in due cause be forwarded to the dunder-init method of the class if one has been defined. In essence, the class object callable is a factory function which when invoked produces new instances of that class. Knowing that classes are simply objects and that constructor calls are simply using class objects as callables, we can build interesting functions which exploit this fact. Let's write a function which returns a Python sequence type, which will either be a tuple if we request an immutable sequence or a list if we request a mutable sequence. In the function we'll just test a Boolean flag and bind either tuple or list, both of which are class objects, to our cls reference using the assignment operator, which we then return. Notice that we must take care when choosing variable names which refer to class objects not to use the class keyword as a variable name. Popular alternatives are the abbreviation cls and the deliberate misspelling klass with a K. Now we can use this function to produce a sequence\_class with the required characteristics. Here we set the immutable argument to true so that the function returns an immutable sequence\_class. We can then create an instance of the class by using the class object as a callable in effect calling the constructor.

Conditional Expressions

As an aside, we'd like to introduce you to a new language feature called conditional expressions which can evaluate one of two subexpressions depending on a Boolean value. They take the form result = true\_value if condition else false\_value. This syntax is perhaps surprising with the condition being placed between the two possible result values, but it reads nicely as English and emphasizes the true\_value, which is usually the most common result. We can use it to simplify our sequence\_class function down to a single expression obviating the need for the intermediate variable binding cls and retaining the single point of return. Conditional expressions can be used any place a Python expression is expected. Their full syntax and somewhat tortured history is covered in PEP308.

Lambdas

Sometimes we want to be able to create a simple callable object usually to pass directly to a function without the bureaucratic overhead of the def statement and the code block it introduces. Indeed in many cases it's not even necessary for the callable object to be bound to a name if we're passing it directly to a function. An anonymous function object would suffice. This is where the lambda construct comes into play, and used with care it can make for some expressive and concise code. However, as with comprehensions, excessive use of lambdas can serve to obfuscate rather than to clarify code when encountered to the Pythonic principles which value readability so highly, so take care to deploy them wisely. If you're wondering why a technique for making callable objects from Python expressions is named after the eleventh letter of the Greek alphabet, the origins go back to a foundational working computing science in 1936 by Alonzo Church predating electronic computers even. Church developed the lambda calculus, which forms the basis of the functional programming techniques used in languages such as Lisp. A good example of a Python function that expects a callable is the sorted built-in for sorting iterable series which accepts an optional key argument, which must be a callable. For example, if we have a list of names and strings and we wish to sort them by second name, we need to pass a callable as the key argument of sorted, which will extract the second name. To do this, we can use a lambda to produce such a function without the bother of needing to think up a name for it. Let's create a list of scientist names, Marie Curie, Albert Einstein, Niels Bohr, Isaac Newton, Dmitri Mendeleev, Antoine Lavoisier, Carl Linnaeus, Alfred Wegener, and Charles Darwin. Now we'll produce a new list of sorted names using the sorted built-in and passing a lambda function as the key argument. Here our lambda accepts a single argument called name in the body of the lambda after the colon calls the str. split method and returns the last element of the resulting sequence using negative indexing. Lambda is itself an expression which results in a callable object. We can see this by binding the result of the lambda expression to a named reference using assignment. We can see that the resulting object is a function, although it doesn't have a name, and that it is indeed callable like a function. Creating a callable function this way using lambda and binding to a name through assignment is equivalent to defining a regular function using def like this. Let's spend a moment to point out the differences between lambdas on the one hand and regular functions or methods on the other. Def is a statement, which defines a function and has the effect of binding it to a name. Lambda is an expression, which returns a function object. Regular functions must be given a name whereas lambdas are anonymous. The argument list for functions is delimited by parentheses and separated by commas. The argument list for lambdas is terminated by a colon and separated by commas. Lambda arguments are without enclosing parentheses. Versions of Python predating Python 3 have special handling of tuples using tuple unpacking in the argument list. This confusing feature has been removed from Python 3, so in Python 3 code there are never any parentheses between the lambda keyword and the colon after the argument list. Both regular functions and lambdas support 0 or more arguments, a zero argument function using empty parentheses for the argument list whereas a zero argument lambda places the colon immediately following the lambda keyword. The body of a regular function is a block containing statements whereas the body of a lambda is an expression to be evaluated, and the lambda body can contain only a single expression, no statements. Any return value from a regular function must be explicitly returned using the return statement. No return statement is needed or indeed allowed in the lambda body. The return value will be the value of the supplied expression. Unlike regular functions, there is no simple way to document a lambda with a docstring. Regular functions can easily be tested using external testing tools because they can be fetched by name. Most lambdas can't be tested in this way simply because they're anonymous and can't be retrieved. This points the way to a guideline to keep your lambdas simple, simple enough that they're obviously correct by inspection.

Detecting Callable Objects

To determine whether an object is callable, you can simply pass it to the built-in function callable, which returns true or false. So, as we have seen, regular functions are callable such as this is\_even function; lambda expressions are callable such as this expression, which determines whether a number is\_odd; class objects are callable because calling a class invokes the constructor; and methods are callable. We have also seen that instance objects can be made callable by defining the dunder-call method. Lest you get the impression that everything is callable, there are plenty of objects which aren't such as string instances.

Extended Formal Argument Syntax

We've already been using functions which support extended argument syntax, although you may not have realized it. For example, have you wondered how it's possible for print to accept zero, one, two, or in fact any number of arguments? Another example we've seen is the use of the str format method, which can accept arbitrary named arguments corresponding to the format placeholders in the string. In this section we'll learn how to define functions or more generally callables, which can accept arbitrary positional or keyword arguments. Let's start with positional arguments. Drawing an example from geometry, let's write a function which can return the area of a two-dimensional rectangle, the volume of a three-dimensional cuboid, or indeed the hypervolume of an n-dimensional hypercuboid. Such a function needs to accept an arbitrary number of numeric arguments and multiply them together. To do this we use a special argument syntax where the argument name is prefixed with a single asterisk. Before we implement the computation, we'll simply print out the value of args and its type using the built-in type function. Notice that the asterisk does not form part of the argument name, and the argument name we have chosen here, args, is widely used in this case by convention, although it's by no means necessary. Colloquially this form is called \*args. Now, let's call our function a few times. We can see that args is passed as a tuple, which contains the function arguments. Knowing this it's a simple matter to write code to multiply all the values in the tuple together to get the result. Redefining hypervolume and using a more documentary name for the argument gives us this. The function works by obtaining an iterator I over the tuple and using the next built-in to retrieve the first value, which is used to initialize a variable V in which the final volume will be accumulated. We then use a for loop to continue iteration with the same iterator to deal with the remainder of the values. We can use the function to compute the areas of rectangles, the volumes of cuboids, and the hypervolumes of hypercuboids. It also generalizes nicely down to the lower dimensions to give us the length of lines. However, if called with no arguments, the function raises stop iteration exposing an implementation detail about which clients of our function should be unaware. There are a couple of approaches to fixing this. One change could be to wrap the call to next in a try/accept construct and translate the exception into something more meaningful for the caller such as the type error that is usually raised when an insufficient number of arguments is passed to a function. We'll take a different approach of using a regular positional argument for the first length and the \*args to soak up any further length arguments. Using this design the function continues to work as expected when arguments are supplied and raises a predictable type error exception when insufficient arguments are given. At the same time, the revised implementation is even simpler and easier to understand than the previous version, which used an iterator. When you need to accept a variable number of arguments with a positive lower bound, you should consider this practice of using regular position arguments for the required parameters and \*args to deal with any extra arguments. The \*args syntax only collect positional arguments, and a complimentary syntax is provided for handling keyword arguments. Let's look at that now. Arbitrary keyword arguments can be accepted by callables that use an argument prefixed by a double asterisk. Conventionally this argument is called kwargs, although depending on your situation you may care to choose a more meaningful name. Let's make a function which returns a single HTML tag as a string. The first argument to the function will be a regular positional argument, which will accept the tag name. This will be followed by the arbitrary keyword args construct to which tag attributes can be passed. As before, we'll perform a simple experiment to determine how the keyword arguments are delivered by printing the keyword args and the type of the keyword args. When we call the function with some suitable attributes to create an HTML image tag like this, we can see that the arguments are transferred to our keyword arguments formal parameter as a register Python dictionary where each key is a string bearing the actual argument name. Note that as with any other Python dictionary the ordering of the arguments is not preserved. Now we'll go ahead and implement our tag function properly using a more descriptive name than kwargs. Here we iterate over the names in the attributes dictionary building up the result string as we go. It's worth pointing out at this stage that this str format method we call here also uses the arbitrary keyword args technique to allow us to pass arbitrary named arguments corresponding to our replacement fields in the format string. This example also shows that it's quite possible to combine positional arguments and keyword arguments. In fact, the overall syntax is very powerful so long as we respect the order of the arguments we define. First, \*args, if present, must always precede double star keyword args, so this isn't allowed. Second, any arguments preceding \*args are taken to be regular positional arguments as we saw in the hypervolume example earlier. Thirdly, any regular arguments after \*args must be passed as mandatory keyword arguments. (Typing) Failure to do so results in a type error. Fourthly and finally, we have the double star keyword args arbitrary keyword arguments, which if present must be last in the argument list. Any attempt to define an additional formal argument after double star keyword args results in a syntax error. You should take particular care when combining these language features with default arguments, which have their own ordering rules specifying that mandatory arguments must be specified before optional arguments at the call site. Before moving on we should point out that all of the features of the extended formal argument syntax apply equally to regular functions, lambdas, and other callables.

Extended Call Syntax

The compliment to extended formal argument syntax is extended call syntax, which allows us to use iterable series such as a tuple to populate positional arguments and any mapping type such as a dictionary that has string keys to populate keyword arguments. Let's go back to a simple version of print\_args that only deals with mandatory positional and \*args arguments. We'll now create an iterable series, in this case a tuple, although it could've been any other conforming type and apply it at the call site for print\_args using the asterisk prefix to instruct Python to unpack the series into the positional arguments. Notice that the use of the star syntax in the actual arguments does not necessarily need to correspond to the use of the star in the formal argument list in the definition. In this example the first two elements of our tuple have been unpacked into the mandatory positional arguments, and the last two have been transferred into the args tuple. Similarly, we can use the double asterisk prefix at the call site to unpack a mapping type such a dictionary into the keyword arguments, mandatory or optional. First we'll define a function color, which accepts three arguments, red, green, and blue. Each of these three arguments could be used either as a positional or keyword argument at the call site. At the end of the argument list we add double star keyword args to soak up any additional keyword arguments that are passed. Now we'll create a dictionary to serve as our mapping type of keyword arguments and apply it at the function call site using the double star prefix. Notice again how there's no necessary correspondence between the use of double star and the actual arguments versus the use of double star in the formal argument list. Items in the passed in dictionary are matched up with the formal arguments, and any remaining entries are bundled into the keyword args parameter. Before moving on we'll remind you that the dict constructor uses the double star keyword args technique to permit the creation of dictionaries directly from keyword arguments. We could've used that technique to construct the dictionary K in the previous example, but we didn't want to make the example more complex than necessary.

Forwarding Arguments

One of the most common uses of \*args and double star keyword args is to use them in combination to forward all arguments of a function to another function. For example, suppose we want to define a function for tracing the argument and return values of other functions. We pass the function whose execution is to be traced, but that function could take any arguments whatsoever. We can use extended argument syntax to accept any arguments to our tracing function and extended call syntax to pass those arguments to the traced function. In this example we trace a call to int with the string ff in base 16 to demonstrate that trace can work with any function without advanced knowledge of the signature of that function.

Duck Tail: Transposing Tables

Recall that the zip built-in function can be used to combine two iterable series element wise into one series of tuples where the two elements of each tuple are corresponding elements from the two series passed to zip. Consider these two tables of daytime temperatures for Sunday and Monday. We can zip them together into one series showing temperatures on Sunday and Monday at corresponding times of day. In fact, the zip built-in accepts any number of iterable series, which it achieves by accepting an argument of star iterables, that is any number of iterables as positional arguments. Let's add another data series for Tuesday and zip up Sunday, Monday, and Tuesday. Now consider what we would need to do if instead of three separate lists for Sunday, Monday, and Tuesday we had a single data structure in the form of a list of lists, and we can pretty print it using the Python standard library pprint function from the module of the same name. Now, to pass the individual daily records to zip we must index into the outer list, although this is clumsy and not extensible. Now we return to one of the main topics of the course module, extended call syntax, which allows us to apply any iterable series to function call arguments using the asterisk prefix, so-called \*args. Our list of lists is perfectly acceptable as an iterable series of iterable series, so we can use extended call syntax like this. Or instead of using the for loop, we can produce the result as a single data structure simply by passing the result of zip to the list constructor. Notice what is happening here. We've transformed this structure containing lists of daily temperatures into this structure containing a series of hourly temperatures effectively converting columns into rows and rows into columns, an operation known as transposition. This zip star idiom is an important technique to learn not least because if you're not familiar with the idea it may not be immediately obvious what the code is doing. It's fairly widely used in Python code and definitely one worth learning to recognize on site.

Summary

Let's sum up. In this module we showed that functions can be generalized into the notion of callables. We can make callable objects from instances by implementing the special dunder-call method. We can then invoke the object as if it were a function. We can use this technique to define functions, which maintain state such as caches between calls. This also allows us to give functions attributes and methods, which can be used to query and modify any hidden state. Whenever we create an object by invoking a constructor, we're actually calling a class object. Class objects are themselves callable. Class objects can be used just like any other callable object including being passed to and returned from functions and bound to names through assignment. A single expression can be used as a callable by creating a lambda, which is an anonymous callable. Lambdas are most frequently used in line and passed directly as arguments to other functions. Unlike regular functions, the lambda argument list isn't enclosed in parentheses, and the lambda body is restricted to being a single expression, the value of which will be returned. Callable objects can be detected using the built-in callable predicate function. Extended argument syntax allows arbitrary positional arguments to be accepted using the \*args syntax in the callable definition, which results in arguments being packaged into a tuple. Similarly, arbitrary keyword arguments can be accepted using the double star keyword args syntax, which results in keyword arguments being packaged into a dictionary. Extended call syntax allows us to unpack iterable series and mappings into the positional and keyword function arguments respectively. There is no requirement for the use of star and double star at the call site to correspond to the use of star and double star in the definition. Arguments will be unpacked and repacked as necessary. Star args and double start keyword args can be combined with mandatory positional and keyword arguments in a well-defined order. In passing, we discovered that the timeit module can be used to measure the performance of small code snippets. Python supports a syntax for conditional expressions of the form result = true\_value if condition else false\_value and that the zip built-in function uses extended argument syntax to accept an arbitrary number of iterable series as arguments. By combining zip with extended call syntax using star to unpack an iterable series of iterable series, we can transpose two-dimensional tables of data converting rows into columns and vice versa. The zip star idiom is widespread enough that you need to be able to recognize it on site. Next time on Python: Beyond the Basics we'll continue our journey of understanding functions looking at how they interact with scopes and local functions, which can be used to form closures, which in turn are useful in constructing decorator functions. Thanks for watching, and we'll see you in the next module.

Closures and Decorators

Local Functions

Hello. My name is Austin Bingham, and welcome to the third module of Python: Beyond the Basics. In this module we'll look at local functions, that is functions defined within the scope of other functions. We'll also look at the related concept of closures, which are key to really understanding local functions, and we'll close off the module with a look at Python's function decorators. As you'll recall, in Python the def keyword is used to define new functions. Def essentially binds the body of a function to a name in such a way that functions are simply objects like everything else in Python. It's important to remember that def is executed at runtime meaning that functions are defined at runtime. Up to now almost all of the functions we have looked at have been defined at module scope or inside classes in which case we refer to them as methods. However, Python doesn't restrict you to just defining functions in those two contexts. In fact, Python allows you to define functions inside other functions. Such functions are often referred to as local functions since they're defined local to a specific function's scope. Let's see a quick example. Here we define a function sort\_by\_last\_letter, which sorts a list of strings by their last letter. We do this by using the sorted function and passing last\_letter as the key function. Last\_letter is defined inside sort\_by\_last\_letter. It is a local function. Let's test it out. (Typing) Just like module level function definitions, the definition of a local function happens at runtime when the def keyword is executed. Interestingly, this means that each call to sort\_by\_last\_letter results in a new definition of the function last\_letter. That is just like any other name bound in a function body, last\_letter is bound to separately to a new function each time it's called. We can see this for ourselves by making a small modification to sort by last\_letter to print the last\_letter object. (Typing) If we run this a few times, we see that indeed each execution of sort\_by\_last\_letter results in a new last\_letter instance. (Typing) The main point here is that the def call in sort\_by\_last\_letter is no different from any other name binding in the function, and a new function is created each time def is executed. Local functions are subject to the same scoping rules as other functions. Remember the LEGB rule for name lookup. First the local scope is checked, then the enclosing scope, next the global scope, and finally the built-in scope. This means that name lookup in local functions starts with names defined in the function itself. It proceeds to the enclosing scope, which in this case is the containing function. This enclosing scope includes both the local names of the containing function, as well as its parameters. Finally, of course, the global scope includes any module-level name bindings. You can see this in a small example. (Typing) Here we define the function inner local to outer. Inner simply prints a global variable and a few bindings from outer. This example shows the essence of how the LEGB rule applies to local functions. If you don't fully understand what's going on, you should play with this code on your own until it's clear. It's important to note that local functions are not members of their containing function in any way. As we've mentioned, local functions are simply local name bindings in the function body. To see this, you can try to call a local function via member access syntax. The function object outer has no attribute named inner. Inner is only defined when outer is executed, and even then it's just a normal variable in the execution of the function's body. So, what are local functions useful for? As we've seen, they're useful for things like creating sorting key functions. It makes sense to define these close to the call site if they're one-off specialized functions, so local functions are a code organization in readability aid. In this way they're similar to lambdas, which as you'll recall are simple, unnamed function objects. Local functions are more general than lambdas though since they may contain multiple expressions and may contain statements such as import. Local functions are also useful for other more interesting purposes, but before we can look at those we'll need to investigate two more concepts, returning functions from functions and closures.

Returning Functions From Functions

As we've just seen, local functions are no different from any other object created inside a function's body. New instances are created for each execution of the enclosing function. They're not somehow specially bound to the enclosing function and so forth. Like other bindings in a function, local functions can also be returned from functions. Returning a local function does not look any different than returning any other object. Let's see an example. Here enclosing defines local\_func and returns it. Callers of enclosing combined its returned value to a name, in this case lf, and then call it like any other function. This ability to return functions is part of the broader notion of first-class functions where functions can be passed to and returned from other functions or more generally treated like any other piece of data. This concept can be very powerful, particularly when combined with closures, which we'll explore in the next section.

Closures and Nested Scopes

So far the local functions we've looked at have all been fairly boring. They're defined within another function scope, but they don't really interact with the enclosing scope; however, we did see that local functions can reference bindings in their enclosing scope via the LEGB rule. Furthermore, we saw that local functions can be returned from their defining scope and executed in another scope. This raises an interesting question. How does a local function use bindings to objects defined in a scope that no longer exists? That is once a local function is returned from its enclosing scope, that enclosing scope is gone along with any local objects it defined. How can the local function operate without that enclosing scope? The answer is that the local function forms what is known as a closure. A closure essentially remembers the objects from the enclosing scope that the local function needs. It then keeps them alive so that when the local function is executed they can still be used. One way to think of this is that the local function closes over the objects it needs preventing them from being garbage collected. Python implements closures with a special attribute named dunder-closure. If a function closes over any objects, the net function has a dunder-closure attribute, which maintains the necessary references to those objects. We can see that in a simple example. The dunder-closure attribute of lf indicates that lf is a closure, and we can see that the closure is referring to a single object. In this case, that object is the X variable defined in the function that defined lf. So, we can see that local functions can safely use objects from an inner enclosing scope, but how is this really useful?

Function Factories

A very common use for closures is in so-called function factories. These factories are functions that return other functions where the returned functions are specialized in some way based on arguments to the factory. In other words, the factory function takes some arguments. It then creates a local function, which takes its own arguments, but also uses the arguments passed to the factory. The combination of runtime function definition enclosures makes this possible. A typical example of this kind of factory creates a function, which raises numbers to a particular power. Here's how the factory looks. Raise\_to takes a single argument, exp, which is an exponent. It returns a function that raises its arguments to that exponent. You can see that the local function raise\_to\_exp refers to exp in its implementation, and this means that Python will create a closure to refer to that object. If we call raise\_to, we can verify that it creates this closure. We can also see that square does indeed behave as we expect, and we can create other functions the same way. (Typing)

The Nonlocal Keyword

The use of local functions raises some interesting questions regarding name lookup. We've looked in some detail at the LEGB rule, which determines how names are resolved in Python when we want the values to which those name refer. However, LEGB doesn't apply when we're making new name bindings. Consider this simple example. (Typing) When we assign to message in the function local, what precisely is happening? In this case we're creating a new name binding in that function's scope for the name message to the string local. Critically we are not rebinding either of the other message variables in the code. We can see this by instrumenting the code a bit. (Typing) Now we're actually calling the function's enclosing and local. Again, local is creating an entirely new name binding, which only applies in the context of that function. If we run this code, we'll see that neither the global nor enclosing bindings for message are affected by calling local. In earlier courses we discussed Python's global keyword. Global can be used to introduce a binding from the global scope into another scope. So, in our example if we wanted the function local to modify the global binding for message rather than creating a new one, we could use the global keyword to introduce the global message binding into local. Let's do that and see the effects. First let's use the global keyword to introduce the module level binding of message into the function local. If we run this, we can see that the module level binding of message is indeed changed when local is called. Again, the global keyword should be familiar to you already. If it's not, you can always review Module 4 of the Python Fundamentals course. If global allows you to insert module-level name bindings into a function in Python, how can you do the same for name bindings in enclosing scopes? Or in terms of our example, how can we make the function local modify the binding for message defined in the function enclosing? The answer to that is that Python also provides the keyword nonlocal. Nonlocal inserts a name binding from an enclosing namespace into the local namespace. More precisely, nonlocal searches the enclosing namespace from innermost to outermost for the name you give it. As soon as it finds a match, that binding is introduced into the scope where nonlocal was invoked. Let's modify our example again to show how the function local can be made to modify the binding of message created in the function enclosing by using nonlocal. Now when we run this code we see that local is indeed changing the binding in enclosing. It's important to remember that it's an error to use nonlocal when there's no matching enclosing binding. If you do this, Python will raise a SyntaxError. You can see this if you add a call to nonlocal in our function local, which refers to a nonexistent name. When you try to execute this code, Python will complain that no such name does not exist. Like global, nonlocal is not something you're likely to need to use a lot, but it's important to understand how to use it for those times when it's really necessary or for when you see it used in other people's code. To really drive it home, let's create a more practical example that uses nonlocal. In this example, the make\_timer function returns a new function. Each time you call this new function it returns the elapsed time since the last time you called it. Here's how it looks. (Typing) And here's how you can use it. (Typing) As you can see, the first time you invoke T it returns nothing. After that it returns the amount of time since the last invocation. How does this work? Every time you call make\_timer it creates a new local variable named last\_called. It then defines a local function called elapsed, which uses the nonlocal keyword to insert make\_timer's binding of last\_called into its local scope. Elapsed then uses the last\_called binding to keep track of the last time it was called. In other words, elapsed uses nonlocal to refer to a name binding, which will exist across multiple calls to elapsed. In this way elapsed is using nonlocal to create a form of persistent storage. It's worth noting that each call to make\_timer creates a new independent binding of last\_called, as well as a new definition of elapsed. This means that each call to make\_timer creates a new independent timer object, which you can verify by creating multiple timers. As you can see, calls to T1 have no affect on T2, and they are both keeping independent times.

Function Decorators

Now that we've looked at the concepts of local functions and closures, we have what we need to finally look at an interesting and useful Python feature called the decorators. At a high level decorators are a way to modify or enhance existing functions in a nonintrusive and maintainable way. In Python a decorator is a callable object that takes in a callable and returns a callable. If that sounds a bit abstract, it might be simpler for now to think of decorators as functions that take a function as an argument and return another function, but the concept is a bit more general than that as we'll see. Coupled with this definition is a special syntax that lets you decorate functions with decorators. The syntax looks like this. This example applies the decorator, in this case named @my\_decorator, to the function named in this case my\_function(). The at symbol is a special syntax for applying decorators to functions. So, what does this actually do? When Python sees decorator application like this, it first compiles the base function, which in this case is my\_function. As always, this produces a new function object. Python then passes this function object to the function my\_decorator. Remember that decorators by definition take callable objects as their only argument, and they're required to return a callable object as well. After calling the decorator with the original function object, Python takes the return value from the decorator and binds it to the name of the original function. The end result is that the name my\_function is bound to the result of calling my\_decorator with the function created by the def my\_function line. In other words, decorators allow you to replace, enhance, or modify existing functions without changing those functions. Callers of the original function don't have to change their code because the decorator mechanism ensures that the same name is used for both the decorated and undecorated function.

A First Decorator Example

As with so many things in Python, a simple example is much more instructive than words. Suppose that we had some functions which returned strings and we needed to ensure that these strings only contained ASCII characters. We can use the ascii function to convert all non-ascii characters to escape sequences, so one option would be to simply modify every function to use the ascii function. This would work, but it isn't particularly scalable or maintainable. Any change to the system would have to be made in many places including if we decided to remove it completely. A simpler solution is to create a decorator which does the work for us. This puts all of the logic in a single place. Here's the decorator. (Typing) As you can see, the decorator escape\_unicode is just a normal function. Its only argument F is the function to be decorated. The important part really is the local function wrap. Wrap uses the \*args and \*\*kwargs idiom to accept any number of arguments. It then calls F, the argument to escape\_unicode, with these arguments. Wrap takes F's return value, converts non-ascii characters to escape sequences, and returns the resulting string. In other words, wrap behaves just like F except that it escapes non-ascii characters, which is precisely what we want. It's important to notice how escape\_unicode returns wrap. Remember that a decorator takes a callable as its argument and returns a new callable. In this case the new callable is wrap. By using closures, wrap is able to use the parameter F even after escape\_unicode has returned. Now that we have a decorator, let's create a function that might benefit from it. Our extremely simple function returns the name of a particular northern\_city, and of course we can see that it works. (Typing) To add Unicode escaping to our function, we simply decorate northern\_city() with our @escape\_unicode decorator. Now when we call norther\_city we see that indeed non-ascii characters are converted to escape sequences. This is of course a very simple example, but it demonstrates the most important elements of decorators. If you understand what's going on in this example, then you understand 90% of what there is to know about decorators.

What Can Be a Decorator?

Now that we've seen how decorators work, let's look at how other kinds of callables can be used as decorators. We've just used a function as a decorator, and that's probably the most common form of decorator in general use; however, two other kinds of callables are also used fairly commonly. The first of these is class objects. Remember that class objects are callable, and calling them produces new instances of that class. So, by using a class object as a decorator you replace the decorated function with a new instance of the class because the function to be decorated will be passed to the constructor and thereby the initializer. Remember, however, that the object returned by the decorator must itself be a callable object, so the instance resulting from the constructor call must be callable, which means it must support the dunder-call method. That is we can use class objects as decorators so long as the instance objects we get when we call the constructor are themselves callable by virtue of implementing the dunder- call method. In this example we'll create a class CallCount, which keeps track of how many times it's called. (Typing) CallCount's initializer takes a single function F and keeps it as a member attribute. It also initializes a count attribute to 0. CallCount's dunder-call method then increments that count each time it's called and then calls F returning whatever value F produces. You use this decorator much as you might expect by using @CallCount to decorate a function. (Typing) Now, if we call Hello a few times, we can check its CallCount. (Typing)

Instances as Decorators

So, now we've seen how to use class objects as decorators. Another common kind of decorator is a class instance. As you might have guessed, when you call a class instance as a decorator, Python calls that instance's dunder-call method with the original function and uses dunder-call's return value as the new function. These kinds of decorators are useful for creating collections of decorated functions, which you can dynamically control in some way. For example, let's define a decorator which prints some information each time the decorated function is called. First, here's a class, but let's also make it possible to toggle this tracing feature by manipulating the decorator itself, which will implement as a class instance. (Typing) Remember that unlike in our previous example the class object itself is not the decorator. Rather, instances of trace can be used as decorators, so let's create an instance of trace and decorate a function with it. (Typing) Now, if we call rotate\_list a few times, we can see that tracer is doing its job. (Typing) We can now disable tracing simply by setting tracer. enabled to False, and we see that the decorated function no longer prints out tracing information. The ability to use functions, class objects, and class instances to create decorators gives you a lot of power and flexibility. Deciding which to use will depend a great deal upon exactly what you're trying to do. Experimentation in small examples are a great way to develop a better sense of how to design decorators.

Multiple Decorators

In all of the examples we've seen so far we've used a single decorator to decorate functions; however, it's entirely possible to use more than one decorator at a time. All you need to do is list each decorator on a separate line above the function, each with its own at symbol like this. When you use multiple decorators like this, they are processed in reverse order. So, in this example some\_function() is first passed to @decorator3. The callable returned by @decorator3 is then passed to @decorator2. That is @decorator2 is applied to the result of @decorator3 in precisely the same way that it would be applied to a normal function. Finally, @decorator1 is called with the result of @decorator2. The callable returned by @decorator1 is ultimately bound to the name some\_function(). There's no extra magic going on, and the decorators involved don't need to know that they're being used with other decorators. This is part of the beauty of the decorator abstraction. As an example, let's see how we can combine two decorators we've already seen, our tracer and our Unicode escaper. First, let's see the decorators again. (Typing) And now let's decorate a single function with both of them. Now when we use this to invent names for Norwegian islands, our non-ascii characters will be properly escaped, and tracer will record the call. (Typing) And of course we can disable the tracing without affecting the escaping. (Typing)

Decorating Methods

So far we've only seen decorators applied to functions, but it's entirely possible to decorate methods on classes as well. In general there's absolutely no difference in how you use decorators for methods. To see this, let's create a class function version of our IslandMaker function and use the tracer decorator on it. (Typing) We can also use this to cross the North Sea and make a more British version of our IslandMaker. (Typing) As you can see, tracer works perfectly well with methods.

functools.wraps()

Decorators replace a function with another callable object, and we've seen how this can be a powerful technique for adding functionality in a modular maintainable way. There's a subtle problem, however, with how we've used decorators so far. By naively replacing a function with another callable, we lose important metadata about the original function, and this can lead to confusing results in some cases. To see this, let's define an extremely simple function. (Typing) In the REPL let's look at some attributes of this function. First it has an attribute dunder-name, which is simply the name of the function as the user defined it. Similarly, it has an attribute dunder-doc, which is the docstring defined by the user. You may not interact with these attributes a lot directly, but they are used by tools like debuggers and IDEs to display useful information about your objects. In fact, Python's built-in help function uses these attributes. So far so good, but let's see what happens when we use a decorator on our function. First, let's define a simple noop decorator. (Typing) And now let's decorate our hello function. All of the sudden help is a whole lot less helpful. Instead of telling us that hello is named hello and reporting the expected docstring, we're seeing information about the wrapper function used by the noop decorator. If we look at hello's dunder-name and dunder-doc attributes, we can see why. Since we've replaced the original hello function with a new function, what dunder-name and dunder-doc attributes we get when we inspect hello are those of the replacement function. This is an obvious result in retrospect, but generally not what we want. Instead, we'd like the decorated function to have its original name and docstring. Fortunately it's very easy to get the behavior we want. We simply need to replace the dunder-name and dunder-doc attributes of our noop\_wrapper function with the same attributes from the wrapped function. Let's update our decorator to do this. Now when we examine our decorated function we get the results we want. This works, but it's a bit ugly, and it would be nice if there were a more concise way of creating wrapper functions which properly inherited the appropriate attributes from the functions they wrap. Well, we're in luck. The function wraps, and the functools package does precisely that. Functools. wraps() is a function decorator which you apply to your wrapper functions. The decorator takes the function to be decorated as its argument, and it does the hard work of updating the wrapper function with the wrapped function's attributes. Here's how that looks. (Typing) If we now look at our hello function in the REPL one more time, we can see that indeed everything is as we want. (Typing) As you start to develop your own decorators, it's probably best to use functools. wraps to ensure that your decorated functions continue to behave as your users expect. We've seen how to use and create decorators, and hopefully it's clear that decorators are a powerful tool for Python programming. They are being used widely in many popular Python packages, so it's very useful to be familiar with them. One word of warning though. Like many powerful features in many programming languages, it's possible to overuse decorators, so use decorators when they are the right tool when they improve maintainability, add clarity, and simplify your code. If you find that you're using decorators just for the sake of using decorators, take a step back and think about whether they're really the right solution.

Duck Tails: Validating Arguments

One interesting and practical use of decorators is for validating function arguments. In many situations you want to ensure that function arguments are within a certain range or meet some other constraints. Let's create a decorator which verifies that a given argument to a function is a non-negative number. This decorator is interesting in that it takes an argument. This might appear confusing at first, but you'll see how it works if you just follow the description of decorators, which we just finished covering. (Typing) Here's how you can use this decorator to ensure that the second argument to a function is non-negative. (Typing) We can see that it works as expected. (Typing) So, how does this decorator work? First we need to recognize that check\_non\_negative is not in fact a decorator at all. A decorator is a callable object that takes a callable object as an argument and returns a callable object. Check\_non\_negative takes an integer as an argument and returns a function, the nested validator function. What's going on here? You'll see that at the point where we decorate create\_list we actually call check\_non\_negative. In other words, the return value of check\_non\_negative is the actual decorator. Python takes check\_non\_negative's return value and passes our function create\_list to it. Indeed if you look at the validate function by itself you'll see that it looks exactly like the other decorators we've defined in this module. Interestingly, the wrap function returned by validator forms a closure over not just F, the decorated function, but also over index, the argument passed to check\_non\_negative. This can be a bit of a mind bender, and it's well worth spending a little extra time to make sure you really understand how this works. If you understand this example, you're well on your way to mastering Python decorators.

Summary

To close out this module, let's review what we've covered. Def is executed at runtime. Def defines a function in the scope in which it's called, and this can be inside other functions. Functions defined inside other functions are commonly called local functions. A new local\_function is created each time the containing function is executed. Local functions are no different from other local name bindings and can be treated like any other object. Local functions can access names in other scopes via the LEGB rule. The enclosing scope for a local\_function includes the parameters of its enclosing function. Local functions can be useful for code organization. Local functions are similar to lambdas, but are more general and powerful. Functions can return other functions including local functions defined in their body. Closures allow local functions to access objects from scopes which have terminated. Closures ensure that objects from terminated scopes are not garbage collected. Functions with closures have a special dunder-closure attribute. Local functions and closures are the keys to implementing function factories, which are functions that create other functions. Function decorators are used to modify the behavior of existing functions without having to change them directly. Decorators are callable objects, which accept a single callable object as an argument and return a new callable object. You use the at symbol to apply decorators to functions. Decorators can enhance maintainability, readability, and scalability of designs. Decorators can be any kind of callable object. We looked specifically at functions, class objects, and class instances. The dunder-name and dunder-doc attributes of decorated functions are actually those of their replacement function, which is not always what you want. You can manually update the dunder-name and dunder-doc attributes of your wrapper functions. The functools. wraps function can be used to create well-behaved wrappers in a simple and clear manner. Multiple decorators can be applied to a function. When there are multiple decorators, they are applied in reverse order. Decorators don't have to be specially designed to work with other decorators. Decorators are a powerful tool, but make sure you don't overuse them or use them unnecessarily. There is technically no such thing as a decorator that takes extra arguments. To parameterize decorators, you need a function that creates decorators. Local functions can create closures over objects in any number of enclosing scopes. Thanks for watching, and we'll see you in the next module.

Properties and Class Methods

Class Attributes

Hello. My name is Robert Smallshire. Welcome to the fourth module of the Python: Beyond the Basics course where we'll use some of the approaches from the previous module including decorators to improve the design of our classes. First though we need to look at class scope attributes, so let's start there. You should already be familiar with instance attributes. These are attributes which are assigned on a per object basis usually in the dunder-init method of a class. To illustrate, we'll start with an object that defines a simple shipping container with two instance attributes called owner\_code and contents. You can follow along whilst we develop this system in PyCharm. (Typing) We'll put the code in a Python module called shipping. py. First we introduce the class ShippingContainer, then we define the dunder-init method taking two arguments, owner\_code and contents, and then in body of the method we assign the two arguments to two new instance attributes. This is simple enough to use from the REPL. First we import everything from our shipping module, then we instantiate a ShippingContainer passing the YML owner code and books as contents. As you would expect, both of these attributes can be retrieved normally. If we create a second ShippingContainer instance, it has its own independent owner and contents attributes just as you would expect. Sometimes, however, we would like to have an attribute that is associated with the class and not with each instance of the class. In other words, we would like an attribute whose value is shared between all instances of that class. Such attributes are known as class attributes, and they can be created by assigning to their names within the scope of the class. Let's say we'd like to give each ShippingContainer instance we create a new serial number. We add a next\_serial attribute at class scope starting at the arbitrary value of 1337 we've chosen to make our example more interesting. We also modify the initializer method to assign the current value of the next\_serial class attribute to a new instance attribute self. serial. Finally, we increment the next\_serial class attribute. Let's try it in a new REPL session. We'll import everything from our module again and instantiate a new ShippingContainer. As we can see, this didn't work as planned. Python can't resolve the next\_serial name where we first refer to it in the dunder-init method. To understand why, we need to recall the Python rules for searching scopes local, enclosing function, global, and built-in or LEBG. Since next\_serial doesn't exist at any of these scopes because classes don't introduce new scopes, we need to locate an object that is in one of the scopes and drill down to next\_serial from there. In this case, the ShippingContainer class object is at global or module scope, so we must start from there by qualifying the next\_serial class attribute name as ShippingContainer. next\_serial. At first it might look odd to have to refer to the class by name from within the class definition, but it's really not that much different from having to qualify instance attributes with self. As with the self prefix, using the class name prefix for class attributes confers the same understandability advantage reducing the amount of detective work required to figure out which objects are being referred to. Remember the Zen of Python. Explicit is better than implicit, and readability counts. With these changes in place, our example works as expected. Restarting the REPL we can import everything from our module again and instantiate shipping containers and check that their serial numbers are incrementing from 1337 as we would expect. We can also retrieve the class attribute from outside the class by qualifying it with a class name or we can access the same class attribute through any of the instances. Returning to our code, this shows that we could've written our dunder-init function like this by prefixing the class attribute with self rather than the class name. Although this works, this style is best avoided since it makes it much less clearer within the function body which attributes are instance attributes and which are class attributes. There's another pitfall here of which you must be made aware. Although you can read a class attribute through the self instance reference, attempting to assign to a class attribute through the self reference won't have the desired effect. Look at the other instance attributes we assign to in the initializer, owner\_code, contents, and serial. Assigning to an instance attribute is exactly how we bring these attributes into being. If we attempt to assign to an existing class attribute through the self reference, we actually create a new instance attribute, which hides the class attribute, and the class attribute would remain unmodified. You might think that the use of the augmented assignment operators such as the += we use here would also be verboten, but they are not. The augmented assignment operators work by calling a special method on the referred to object and don't rebind the reference on the left hand side. All said, it's much better and safer to access class attributes as, well, attributes of the class object rather than via the instance.

Static Methods

Let's perform a small refactoring by extracting the logic for obtaining the next identifier into a method \_get\_next\_serial, which as you can see from the leading underscore is an implementation detail of this class. Notice that like all of the other methods we've encountered so far the first argument to \_get\_next\_serial is self, which is the instance on which the method will operate. Notice, however, that although we must accept the self instance argument, nowhere in the method do we actually refer to self, so it seems completely redundant. What we would like to do is associate \_get\_next\_serial with the class rather than with instances of the class. Python gives us two mechanisms to achieve this, the first of which is the @staticmethod decorator. To convert our method into a static method, we simply decorate it with the @staticmethod decorator and remove the unused self argument. Although not strictly necessary, we can also modify the call site to call through the class rather than through the instance by replacing self. \_get\_next\_serial with ShippingContainer. \_get\_next\_serial. When we test this at the REPL, we can see that the modified code has exactly the same behavior as before. Static methods in Python have no direct knowledge of the class within which they are defined. They simply allow us to group a function within the class because the function is conceptually related to the class. The name @staticmethod is something of an anachronism in Python. The static refers to a keyword used to indicate the equivalent concept in the C++ programming language, which itself was a reuse of a keyword from the C programming language.

Class Methods

As an alternative to @staticmethod, we can use a different decorator called @classmethod. Let's further modify our function to use the @classmethod decorator instead of the @staticmethod decorator. @classmethod accepts the class object as the first formal argument by convention using the abbreviated name cls since we can't use the fully spelled out keyword class as an argument name. The cls argument for class methods plays an analogous role to the self argument for instance methods. Now, when we call ShippingContainer. \_get\_next\_serial, the ShippingContainer class object is passed as the cls argument of the class method, which we then refer to within the body of the method to locate the next serial class attribute. The @staticmethod and @classmethod decorators are quite similar, and you may find it difficult to choose between them, and this may be even more confusing if you have a heritage in another object-oriented language such as C++, C#, or Java, which has a similar static method concept. The rule is simple though. If you need to refer to the class object within the method, for example to access a class attribute, prefer to use @classmethod. If you don't need to access the class object, use @staticmethod. In practice, most static methods will be internal implementation details of the class marked as such with a leading underscore since having no access to either the class object or the instance object they rarely form a useful part of the class interface. In principle, it would also be possible to implement any @staticmethod completely outside the class at module scope without any loss of functionality, so you may want to consider carefully whether a particular function should be a module scope function or a @staticmethod. The @staticmethod decorator merely facilitates a particular logical organization of the code allowing us to put within classes what would otherwise need to be free functions. Sometimes we would like a class to support named constructors also known as factory functions, which construct objects with certain configurations. For example, we could use a factory function to implement a method, which creates an empty ShippingContainer. We create a new method decorated with a @classmethod decorator called create\_empty. The method calls the constructor on the class type, cls, passing the owner\_code and contents set to None. Of course in this case cls will refer to ShippingContainer, so this is exactly the same as calling the ShippingContainer constructor. We can invoke our factory method directly on the ShippingContainer class object like this. (Typing) This technique allows us to support multiple functions which behave similarly to constructors, but with different behaviors without having to resort to contortions within the dunder-init method to interpret different forms of argument lists. Here we add a constructor for placing an iterable series of items in the container called create\_with\_items. Again, this is a function decorated with a @classmethod decorator, which internally calls the cls constructor where cls will be ShippingContainer. Here's a demonstration of the new named constructor. We pass a list of items to be placed within the container. Let's modify our example to make it slightly more realistic. We'll adjust ShippingContainer to use a string code rather than an integer serial number. In fact, we'll modify our class to use fully-fledged BIC codes where B-I-C is the Bureau International des Conteneur or the International Container Bureau. Each container has a unique BIC code, which follows a standard format defined in the ISO 6346 standard. We won't go into the details of the coding system here, but we have included a simple Python module called iso6364. py in the example code associated with this course. All you need to know for now is that the module can create a conforming BIC code given a three letter owner code and a six digit serial number together with an optional equipment category identifier. We'll retain the integer serial number generator and introduce a @staticmethod called \_make\_bic\_code with a leading underscore to combine the owner\_code and integer serial number into a single string BIC code. Within the \_make\_bic\_code @staticmethod we call the iso6346 create function to which we pass the owner\_code and the string version of the serial number. This is padded with leading zeros to ensure it's six characters long using the string zfill method. We'll also rework the initializer function to create and store the BIC code instead of the separate owner\_code and serial numbers. Within the dunder-init initializer method we call ShippingContainer. \_make\_bic\_code and assign the result to the new instance attribute bic. The numeric serial argument to this function is obtained by calling our existing \_get\_next\_serial class method. Now, let's try the modified code. We create an empty ShippingContainer using our factory @classmethod create\_empty passing the YML owner\_code. We can see that the BIC attribute now contains a legitimate BIC code.

Static Methods with Inheritance

We'll return to class inheritance in more depth later in this course, but for now we'll look at how class and static methods behave in the presence of inheritance. First, we'd like to point out that unlike static methods in many other languages, static methods in Python can be overridden in subclasses. Let's introduce a subclass of ShippingContainer called RefrigeratedShippingContainer. RefrigeratedShippingContainers use an equipment category code of R, which comprises the fourth character of the BIC code rather than the default U we saw previously. We do this in an overridden \_make\_bic\_code @staticmethod in the derived class. We must specify this when creating the BIC code by passing an additional category argument to the iso6346. create function. Let's try instantiating our new class at the REPL and checking its BIC code. (Typing) Hmm. This hasn't worked as we had hoped. The fourth character in the BIC code is still U. This is because in the init method we have called \_make\_bic\_code through a specific class. To get polymorphic override behavior, we need to call the @staticmethod on an instance. Let's experiment a little at the REPL so we understand what's going on. First we'll test the @staticmethod by calling it directly on the base class. Here we call it on ShippingContainer. Now we'll call it directly on the derived class, RefrigeratedShippingContainer. In the latter case we have an R for refrigeration in the fourth character. If you're wondering why the last digit also changes, it's because the last digit is a check digit computed by the ISO6346 implementation. In both cases we get exactly what we've asked for. The class-specific versions of the static methods are called. Now we'll create some instances, first off the base class. Notice that here we're directly calling an internal implementation detail method we wouldn't normally call on an existing instance. We've deliberately used the different owner\_code from the one we constructed the instance with to make that clear. Here the fourth character of the result is the default U, so we know the base version was called. Notice that although we've created an instance, we're ignoring any instance attribute data when we invoke the @staticmethod directly in this way. Now we'll instantiate the derived class, RefrigeratedShippingContainer. Again, we'll call \_make\_bic\_code through the instance, although we're ignoring the attributes of the instance R we've created. We can see from the R in the fourth place of the BIC code that the derived class implementation was called, so we can get polymorphic dispatcher static methods only when we call the method through an instance, not when we call the method to the class. To get the desired behavior, we must modify our dunder-init method in the base class to use polymorphic dispatch of the @staticmethod by calling through the instance self. With this change in place, we get polymorphic BIC generation from the single constructor implementation. Be aware then that by calling static methods through the class you effectively prevent them being overridden, at least from the point of view of the base class. If you need polymorphic dispatch of @staticmethod invocations, call through the self instance.

Class Methods with Inheritance

Now, let's look at class methods and how they interact with inheritance. The class methods we defined in the base class will be inherited by the subclass. And what is more, the cls argument of these methods will be set appropriately, so calling create\_empty on RefrigeratedShippingContainer will create an object of the appropriate subtype. For those of you coming to Python from other popular object-oriented languages, you should recognize this ability to have class methods behave polymorphically as a distinguishing feature of Python. The other factory method also works as expected. These invocations work because the base class dunder-init initializer method is inherited into the subclass. Let's move on by making our RefrigeratedShippingContainer more interesting by adding a PerContainerTemperatureSetting as an instance attribute. First we'll add a class attribute, which defines the maximum temperature of a refrigerator container. Being a class attribute, this will apply to all RefrigeratedContainer instances. Next we'll need to override the dunder-init method in the subclass. The overridden method does two things. First, it calls the base class version of dunder-init forwarding the owner\_code and contents arguments to the base class initializer. Unlike other object-oriented languages where constructors at every level in an inheritance hierarchy will be called automatically, the same cannot be said for initializers in Python. If we want a base class initializer to be called when we override that initializer, we must do so explicitly. Remember, explicit is better than implicit. To get a reference to the base class instance, we call the built-in super() function. We then call dunder-init on the returned reference and forward the constructor arguments. We'll be covering super() in a lot more detail later in the course, so don't concern yourself overly with it now. We're just using it so the subclass version of dunder-init can extend the base class version. With this done, we validate the celsius argument against the MAX\_CELSIUS class attribute and assign the celsius instance attribute. Let's try it at the REPL. We start by importing all classes and create an instance of RefrigeratedShippingContainer using our create\_with\_items factory method. Oops! There's no way the factory method in the base class can know or indeed should know the signature of the dunder-init function in derived classes. As such, it doesn't accommodate our extra celsius argument in the derived class. Fortunately we can use \*args and keyword args to work around this by having our factory functions accept both \*args and \*\*kwargs and forward them unmodified to the underlying constructors. We can have our base class factory functions accept arguments destined for derived class initializers. (Typing) When we try this at the REPL by creating another RefrigeratedShippingContainer by calling create\_with\_items on the class and remember to pass the additional celsius argument, which will be forwarded to the underlying constructor through keyword args, everything works as expected. So far so good. We can construct instances of our derived class using a factory function defined in the base class and can gain access to our new celsius attribute as expected. Unfortunately, our design also allows us to set the instance temperature outside the range indicated by the MAX\_CELSIUS class attribute. Doing so violates what should be an important class invariant, and we should find a way to prevent it.

Properties

Using the Python tools we already have at our disposal, one approach would be to rename the celsius attribute to \_celsius to discourage meddling and wrap the attribute with two methods called get\_celsius and set\_celsius with the setter performing validation against the MAX\_CELSIUS class attribute. Such an approach would work, but would be considered deeply un-Pythonic. Remember, Python is not Java. Furthermore, it would require all uses of the celsius attribute to be adjusted to use the method call syntax. Fortunately, Python provides an altogether superior alternative to getter and setter methods called properties, which allow getters and setters to be exposed to seemingly regular attributes performing a graceful upgrade in capabilities. As with static and class methods, decorators are the basis of the property system. Let's take a look. First we'll rename our celsius attribute to \_celsius to indicate that it should no longer be considered to be part of the public interface. Then we'll define a new method called celsius, now we've freed up that name, which will retrieve the attribute. The method will be decorated with a built-in @property decorator. Back in the REPL once we've reimported our module and instantiated a new RefrigeratedShippingContainer with suitable temperature, we can see that we can still get ahold of our attribute value using regular attribute access syntax without the function-called parentheses. What's happened here is that the @property decorator has converted our celsius method into something that when accessed behaves like an attribute. The details of exactly how this is achieved are beyond the scope of this course. For the time being, it's sufficient to understand that the @property decorator can be used to transform getter methods so they can be called as if they were attributes. Now, if we attempt to assign to the attribute, we receive an attribute error informing us that the attribute can't be set. To make attribute assignment work, we need to define a setter, which uses another decorator, but first we need to cover some background information. Recall that decorators are functions which accept one function as an argument and return another object, which is usually a wrapper around the original function which modifies its behavior. Here we show a regular function, which is bound by the def statement to the name F and then processed by a decorator, which creates a wrapper function object which refers back to the original function. Finally, application of the decorator rebinds the name F to the wrapper. Moving on to the specifics of properties, we'll start with an example class into which we place a getter function P. We then decorate this with the built-in @property decorator, which creates a special property object. This contains a reference back to the original getter function. Then P is rebound to the property object. This much we've already seen in action with our celsius property. If needed, we can then create a setter function, which can also be called simply P, although this would also need to be decorated. This time rather than the built-in property decorator, we use a decorator specific to this property, which is itself an attribute of the property object that was created when we defined the getter. This new decorator is always called setter and must be accessed via the property object, so in our case it's called @p. setter. Decorating our setter function with the @p. setter decorator causes the property object to be modified associating it with our setter method in addition to the getter method. This is all fairly mind bending, and we apologize if you've not yet consumed enough caffeine today for this to make sense. As usual, another example clarify matters somewhat. Let's define our celsius setter. So, when we decorate our celsius getter with @property, the returned object is also bound to the name celsius. It is this returned property object which has the setter attribute attached to it, which is another decorator and which is used to decorate our setter definition. The @celsius. setter method accepts a value, which is validated against the MAX\_CELSIUS class attribute, and then if that validation is okay we assign to the \_celsius instance attribute. (Typing) Back in the REPL we can now assign to the property using regular attribute syntax, which will call the setter method and execute our validation code. Shipping containers are moved around the world between cultures which prefer the Celsius measurement scale and those which prefer the Fahrenheit scale. Let's round off this section by adding support for Fahrenheit property access to the same underlying temperature data. Notice that we've added two new static methods, \_c\_to\_f and \_f\_to\_c to perform temperature conversions. These are good candidates for static methods since they don't depend on the instance or class objects, but don't really belong at global or module scope in a module of ShippingContainer classes either. The getter and setter methods for our new fahrenheit property are implemented in terms of our new temperature conversion static methods and significantly in terms of the existing celsius property rather than going directly to the stored \_celsius attribute. This is so we can reuse the validation logic in the existing property. Finally, notice that we can simplify our subclass initializer by leaning on the celsius property setter for validation here too. We simply assign through the property rather than directly to the underlying attribute and get validation for free. Let's test all these changes at the REPL. We create an instance of a RefrigeratedShippingContainer using our create\_empty factory method with a valid Celsius temperature of -20. 0. This is echoed back to us both in Celsius and in Fahrenheit when we access the appropriate attributes. We can also set the temperature through the fahrenheit attribute. Finally, we'll check that validation in the constructor is still working by attempting to recreate a RefrigeratedShippingContainer with a temperature that is too high.

Properties and Inheritance

Now let's look at how properties work in the presence of inheritance. We'll modify our ShippingContainer class to contain the width and height as class attributes since they are the same for all containers and the length as an instance attribute since that varies across individual containers. In this case our class attributes are constants, but we need to modify our init method and all the methods that call it, our two @classmethod factory functions, to accept the additional argument, length in feet. Now we'll add a read-only property, which reports the volume in cubic feet of a container instance making the simplifying assumption that the sides of a container have zero thickness. Notice that the height and width are qualified with a class object and the length of the instance object. Constructing an empty 20 foot container, we can now determine that it has a volume of 1360 cubic feet. We also modify the initializer of the RefrigeratedShippingContainer subclass to accept the new length in feet argument and forward it to its super() class implementation. With this done, the volume in cubic feet property is inherited into the RefrigeratedShippingContainer subclass without issue. We know, however, that the cooling machinery in a refrigerated shipping container occupies 100 cubic feet of space, so we should subtract that from the total. Let's add a class attribute for that volume and override the volume in cubic feet property with a modified formula. We can see that overriding property getters is straightforward. We just redefine the property in the subclass. (Typing) Testing at the REPL, this works well enough giving a volume of 1260 cubic feet. However, we've duplicated the bulk volume calculation between the overridden property and its base class implementation, so we'll address that by having the derived class version delegate to the base class. As before, this is done by retrieving the base class property using a call to the super() function. So, overriding property getters like volume and cubic feet is straightforward enough. We just need to redefine the property in the derived class's normal delegating to the base class if we need to. Unfortunately, overriding property setters is much more involved. Let's see why. To demonstrate, we need a property for which it makes sense to override the setter. We'll introduce a third class into our class hierarchy for a heated refrigerated shipping container, and I'm not making this up. Such things do exist, and their purpose is to maintain a temperature within a wide range of ambient conditions. For the purposes of this exercise, we'll assume that such containers should never fall below a fixed temperature of -20 degrees celsius, which we'll represent with another class attribute. We don't need to override the celsius getter here, but we do need to override the @celsius. setter. Let's have a go. (Typing) Unfortunately, this obvious approach doesn't work for the reason that the celsius object from which we retrieved the setter decorator is to visible in the scope of the derived class. You've probably noticed that PyCharm flagged this too telling us that there was an unresolved reference to celsius. We can solve this by fully qualifying the name of the celsius object with the base class name. (Typing) Now this works just fine. We can happily create instances of the new class through our existing named constructor, and any attempt to set a temperature below the minimum via the overridden property causes the ValueError to be raised. Validation also works when we attempt to construct an instance of this new class with an out of range temperature even though we haven't defined the dunder-init method for the new class. Recall that the initializer assigns the underlying \_celsius attribute through the celsius property, so our overridden property validator is invoked during construction too thanks to polymorphic dispatch. Our overridden property is also interesting because it highlights a useful ability of the Python language to chain the relational operators, so we can do a < b < c rather than (a < b) and (b < c). That said, our code is certainly violating the Don't Repeat Yourself or DRY principle by duplicating the comparison with MAX\_CELSIUS, which is already implemented in the parent class. We could try to eliminate the duplication by delegating this test to the super class via super() like this. (Typing) But surprisingly this doesn't work. We get a runtime error, AttributeError at super has no attribute celsius. With the combination of super and properties, there is much hidden machinery at play, which we won't get into in this intermediate-level course. This is solvable by retrieving the base class property setter function from the base class property and calling it directly. This is available through the fset attribute of the property remembering to explicitly pass self. A bonus of delegating to the base class in this way is that we get a slightly more informative error message, which now tells us whether the requested temperature is too hot or too cold rather than just out of range. Notice that we've been careful to root all access to the \_celsius attribute through the celsius property. As such, none of the other code which needs to respect the constraints needs to be modified. For example, the fahrenheit setter, although not itself overridden, now respects the lower temperature limit. For reference, -14 degrees Fahrenheit is a little below the limit of -25 degrees Celsius. Now, all of this works, but to be honest we think this implementation of the overridden celsius property setter is a bit of a mess containing as it does two direct references to the base class, perhaps not so bad in this case, but the class defining the original property could've been many levels up in the inheritance hierarchy. Knowing this technique is useful though for the time when you're not in a position to modify the base class implementation. Nonetheless, we'd like to find a more elegant or be it more intrusive solution, and that's what we'll pursue in this module's Duck Tails segment.

Duck Tail: The Template Method Pattern

We've seen that it's quite straightforward to override property getters, but somewhat more involved and quite syntactically messy to override property setters. In this Duck Tail we deploy a standard design pattern, the Template Method, to resolve these shortcomings and confer some additional benefits on our code. The Template Method is a very straightforward design pattern where we implement skeletal operations in base classes deferring some details to subclasses. We do this by calling methods in the base class, which are either not defined at all in the base class or have trivial implementations which raise an exception such as NotImplementedError and so must be overridden in order to be useful. An alternative is that we do supply useful details in the base class, but allow them to be specialized in derived classes, which is what we'll do here. As the old saying goes, there's no problem in computer science which can't be solved by an additional level of indirection. Let's start by using the Template Method pattern to implement a getter. We'll use the volume in cubic feet property to demonstrate because that's a property we override in a subclass. To do this, we extract the computation from the getter in the base class into a separate function \_calc\_volume. The volume in cubic feet property is now a Template Method. It doesn't do anything itself except delegate to a regular undecorated method, which can easily be supplied or overridden in a derived class. We'll now override this regular method in the derived class by converting the existing overridden property into a regular undecorated method and changing its name to \_calc\_volume. This implementation leans on the base class implementation by using the call to super(). (Tying) We can use the same technique to override a property setter without having to remember any funky syntax. In this case, we'll turn the @celsius. setter into a Template Method, which delegates to an undecorated \_set\_celsius method. We can now remove the horrible property override construct in the subclass and simply override the \_set\_celsius method. In this case we've decided to use super() to call the base class implementation.

Summary

Let's wrap up by summarizing what we've covered. In this module we've covered the distinction between class attributes and instance attributes. We've demonstrated how class attributes are shared between all instances of a class and shown how to refer to class attributes from within or without the class definition by fully qualifying with a class name. We've also warned you against the dangers of trying to assign to a class attribute through the self instance, which actually creates a new instance attribute. We've used the @staticmethod decorator to define methods within the class, which do not depend on either the class or instance objects. We've used the @classmethod decorator to define methods, which operate on the class object rather than instance object. We showed how to implement an idiom called named constructors using class methods, and we've shown how both static and class methods behave with respect to inheritance. Both static and class methods can support polymorphic method dispatch when invoked through an instance rather than through a class. We introduced the concept of properties to wrap attributes with getters and optional setter methods using the @property decorator. Finally, we demonstrated an easier way to override properties by having them defer to regular methods so they can be overridden, an example of the Template Method design pattern. We've covered a lot of ground including some complex interactions of Python features in this module, and it's important you understand them before moving on. In the next part of this course we'll change track and look at how to make your classes more user and developer friendly by controlling their string representations. Thanks for watching, and we'll see you in the next module.

Strings and Representations

Two String Representations

Hello. My name is Austin Bingham, and welcome to the fifth module of Python: Beyond the Basics. In this module we'll look at string representations of objects in Python, and in particular we'll cover the important, but often confused differences between repr() and str(). Understanding and properly using the various string representations in Python is important for writing maintainable, debuggable, and human-friendly programs. In this module we'll show you what you need to know to use them properly. As you already know, Python supports two primary ways of making string representations of objects, the function's repr() and str(). Each of these can take any object as an argument and produce a string representation of some form. These two functions rely on the special methods dunder-repr and dunder-str to generate the strings they produce, and it's possible for class designers to control these string representations by defining those functions. Here's a quick example. Here the class Point2D defines both dunder-str, which returns a simple format, and dunder-repr, which returns a more complete unambiguous format. If we print both string representations, we can see how the free functions str() and repr() use these methods. So, the big question is why are there two representations, and what are they used for?

repr()

First, let's look at repr(), an abbreviation of representation. Repr() is intended as much as is possible to be an unambiguous representation of an object. By unambiguous we mean that it should include the type of the object along with any identifying fields. In the case of Point2D, the repr() clearly indicates the type, and it shows the two attributes of the object which identify it. Anyone who sees the repr() of a Point2D will know for sure what kind of object it is and what values it holds. Some people will go so far as to suggest that the repr() of an object should be legitimate source code, that is that you should be able to take the repr(), enter it into the REPL or source file, and have it reconstruct the object. This isn't realistic for many classes, but it's not a bad guideline, so keep it in mind when designing reprs. Repr() is important for situations where exactness is more important than concision or readability. For example, repr() is well-suited for debugging because it tells you all of the important details of an object. If an object's repr() tells you its type and important details, you can spend less time inspecting objects and more time debugging logic. Likewise, repr() is generally the best option for logging purposes for many of the same reasons. Generally speaking, repr() should contain more information than the str() representation of an object, so repr() is best suited for situations where explicit information is needed. Another way of thinking about this is to say that repr() is intended for developers where str() is intended for clients. This isn't a hard and fast rule, but it's a useful starting point. The repr() of an object should tell a developer everything they need to know about an object to fully identify it and as much as is practical see where it came from and how it fits into the larger program context. More concretely, the repr() of an object is what the developer will see in a debugger, so when deciding what to put into a repr() think about what you'd want to know when debugging code that uses the class you're developing. This is a helpful guideline and will result in classes that are, as you might imagine, easier to debug and work with in general. It's a good idea to always implement dunder-repr for any class you write. It doesn't take much work to write a good repr(), and the work pays off when you find yourself debugging or scanning logs. All objects come with a default implementation of dunder-repr, but you'll almost always want to override this. The default representation tells you the class name and the ID of the object, but it tells you nothing about the important attributes. For example, here's what we get if we use the default dunder-repr implementation of our Point2D. (Typing) We can see that it's a Point2D, but not much else of any consequence.

str()

Where repr() is used to provide unambiguous, debugger-friendly output, str() is intended to provide readable, human-friendly output. The str() representation is used in situations where, for example, it might be integrated into normal text or where the programming level details such as class might be meaningless. Recall also that the str() function is actually the constructor for the str() type. For example, consider our Point2D class again. Its str() representation looks like this. That representation doesn't tell you anything about the type of the object being printed, but in the right context it tells a human reader everything they need to know. For example, this is perfectly meaningful to a person. The version using repr() is correct, but more than a user needs, and it's likely to be confusing to a lot of people.

When Are the Representations Used?

So, we have two possible string representations of objects produced by str() and repr(). When are they used by other parts of Python, and do they ever rely on each other? An obvious place to look is the print function. Print, since it's generally designed to provide console output to users, uses the human-friendly str() representation as you can see here. Interestingly, the default implementation of str() simply calls repr(). That is, if you don't define dunder-str for a class, then that class will use dunder-repr when str() is required. You can see this if you remove dunder-str from Point2D. (Typing) However, the reverse does not hold true. Implementing dunder-str does not do anything if you choose not to implement dunder-repr. Again, we can see this if we remove dunder-repr from Point2D. (Typing) Here dunder-repr just uses the default version rather than using str() as some might expect. How about when we print collections of objects? What representation is used? It turns out that Python uses the repr() of an object when it's printed as part of a list, dict, or any other built-in type. (Typing) As you can see, repr() is used for contained objects whether repr() or str() is used for the container itself.

Interaction with format()

The format method on strings is another place where string representations are called behind the scenes. When you run code like this, it seems pretty clear that str() is being used, but actually something a bit more complex is going on. When the format method replaces curly braces with an object's representation, it actually calls the special dunder-format method on that object. We can see that by adding a format method to Point2D. (Typing) Now when we print the point via format we get yet another representation. But what is that F argument to dunder-format? That contains any special formatting options specified in the original format string. If the caller puts a colon inside the curly braces of a formatting string, anything after the colon is sent verbatim as the argument to dunder-format. So, for example, we could implement format to reverse X and Y if R is passed in as the format string. (Typing) This gives us the behavior we'd expect. (Typing) In general, however, you don't need to implement dunder- format. Most classes can rely on the default behavior, which is to call dunder-str, which explains why string's format function seems at first to just call str(). You can force format to use a class's repr() instead of its format by putting! r in the formatting placeholder. Likewise, you can bypass dunder-format and use dunder-str directly by putting! s in the formatting placeholder. By and large though, you won't have to think about these details surrounding dunder-format. Almost all of the time you can simply implement dunder-repr and possibly dunder-str, and you will have well-behaved, fully-functioning Python objects.

reprlib

Since we're talking about repr, this is a good time to introduce the reprlib module. Reprlib provides an alternative implementation of the built-in repr() function. The primary feature of this replacement is that it places limits on how large a string can be. For example, if it's used to print a very large list, it will only print a limited number of the elements. This is useful when you're dealing with large data structures whose normal representation might go on for many, many thousands of lines. The basic usage of reprlib involves using the reprlib. repr() function. This function is a drop-in replacement for the built-in repr(). For example, we can use it to print a huge list of our Point2D objects. (Typing) Here we made a list of one million points, and if we had used in the built-in repr() to print it, we would have had to print all one million of the entries. Instead, reprlib. repr() just printed the first few elements with an ellipses to indicate that there were more elements. For many purposes this is a much more useful representation. In a debugger, for instance, seeing a string containing all one million entries would be worse than useless. It would often be extremely detrimental, so reprlib is useful for situations like that. The reprlib. repr() function is the main entry point into reprlib for most people, but there's significantly more to the module than that. Reprlib's functionality is built around a class reprlib. Repr with a capital R. This class implements all of the support for customizing representations. The repr() class is designed to be customized in subclass so you can create your own specialized repr() generators if you want to. The details of how to do that are beyond the scope of this course, but you can find all of the details in the Python Standard Library documentation. Reprlib instantiates a singleton instance of this repr() class for you. It's named rerlib. aRepr, and the reprlib. Repr function actually just calls the repr. repr function on this instance, so you can manipulate this premade instance if you want to control default reprlib behavior throughout your program. Reprlib is a good module to know about, and while you may never need to actually work with it in detail, using just its basic functionality can be very useful.

asciii(), ord(), and chr()

We'll finish up this module by looking at a few more functions that can be useful when dealing with string representations. These aren't required for implementing repr() or str() by any stretch, but since we're talking about strings so much in this module it's a good place to mention them. The first function we'll look at is ascii(). This function takes a string as an argument and converts all of the non-ASCII characters into escape sequences. We've actually seen this function in the module on decorators, though we didn't explain it then. Here's how it looks in action. You can see that ascii() takes in a Unicode string, replaces all of the non- ASCII characters with escape sequences, and then returns another Unicode string. This can be useful in situations where you need to serialize data as ascii() or if you can't communicate encoding information, but you don't want to lose Unicode data. Two other Unicode-related functions are ord() and chr() or chr. These are complimentary functions in that each reverses the other. Ord() takes a single character string as input and returns the integer Unicode codepoint for that character. For example, here we convert the glyph for three-quarters into the decimal codepoint 190. Likewise, chr() takes a Unicode codepoint and returns a single character string containing the character. For example, here we convert 190 back into the glyph for three-quarters. As I mentioned a few seconds ago, these clearly reverse one another, so the ord of the chr of X always equals X, and the chr of the ord of Y always equals Y.

Duck Tail: Bigger Isn't Always Better

As we've discussed in this module, the repr() of an object is intended to be used by developers for logging, debugging, and other activities where an unambiguous format is more important that a human-readable one. Very often this means that the repr() of an object is larger than the str() if only because the repr() contains extra identifying information; however, there are times when it makes sense for a repr() to be smaller than a str(). For example, consider a simple class for rendering tabular data. It consists of a list of header strings and a collection of lists of data for the table's columns. A natural str() representation for this class is a textual multi-line table showing the headers and all of the data. That would look something like this. There's quite a bit going on in this method, but most of it simply involves calculating column widths and then making sure that everything is printed with the correct widths. We won't cover this method in detail here, but it's well worth making sure you understand how it works. We'll leave that as an exercise for curious students. In the end, you can see its results with a simple example. (Typing) And of course you can imagine tables with much more data than that. But is this format really what you'd want for say debugging purposes? For the case of a table class like this, a good repr() mostly just needs to contain column headers. The actual data is not nearly as important. As a result, you can implement repr() something like this. (Typing) This is not only shorter to implement, but the string it produces is shorter as well. (Typing) So, while you might generally find that your reprs are longer than your strs, that won't always be the case. The important thing to remember is that each of these functions serves a distinct purpose, and addressing these purposes is your real goal.

Summary

String representations may seem like a small issue to be concerned about, but this module shows that there are good reasons to pay attention to them. Let's review what we've covered. Python has two primary string representations for objects, str() and repr(). The str() function is used to create str() representations and relies on the dunder-str method. The repr() function is used to create repr() representations, and it relies on the dunder-repr method. Dunder-repr should produce an unambiguous precise representation of the object. Dunder-repr should include the type of an any identifying information for the object. The repr() form is useful for contexts like debugging and logging where information is more important than human readability. You should always implement dunder-repr for your classes. The default dunder-repr implementation is not very useful. The str() form is intended for human consumption and doesn't need to be as precise as repr(). By default dunder-str uses dunder-repr. The print function uses the str() representation. The default dunder-repr does not use dunder-str. Built-in collections like list use repr() to print their elements even if the collection itself is printed with str(). Str. format uses an object's dunder-format method when inserting it into string templates. The default implementation of dunder-format is to call dunder-str. The arguments to the dunder-format method contain any special formatting instructions from the format string. These instructions must come after a colon between the curly braces for the object. In general, you do not need to implement dunder-format. Reprlib provides a drop-in replacement for repr() which limits output size. Reprlib is useful when printing large data structures. Reprlib provides the class Repr, which implements most of reprlib's functionality. Repr class is designed to be extended and customized via inheritance. Function ascii() replaces non-ASCII characters in a Unicode string with escape sequences. Ascii() takes in a Unicode string and returns a Unicode string. The ord() function takes a single character Unicode string and returns the integer codepoint of that character. The chr() function takes an integer codepoint and returns a single character string containing that character. Ord() and chr() are inverses of one another. Good dunder-repr implementations are easy to write and can improve debugging. When reporting errors, the repr() of an object is generally more helpful than the str(). Thanks for watching, and we'll see you in the next module.

Numeric and Scalar Types

Reviewing int and float

Hello. My name is Robert Smallshire. Welcome to the sixth module of the Python: Beyond the Basics course where we'll dig deeper into some of the fundamentals of numerical computing, take a look at some of the numeric types included in the Python language and the Python Standard Library including dates and times. Let's start though by reviewing and looking in a little more detail at some of the scaler types we've already encountered. Throughout this course and the preceding Python Fundamentals course, we've extensively used two built-in numeric types, int and float. We've seen that Python 3 int objects can represent integers, that is whole numbers, of arbitrary magnitude limited only by practical constraints of available memory and the time required to manipulate large numbers. This sets Python apart from many other programming languages where the standard integer types are fixed size storing only 16-, 32-, or 64-bits of precision. Python handles large integers with consummate ease. For example, if we import the factorial function from the math module, we can easily compute the factorial of 1000, which is a vast number. Python's float type, an abbreviation of floating point number, is specifically a 64-bit floating point number using a binary internal representation officially known as binary64 in the IEEE-754 standard. For those of you with a background in C-derived languages, this is commonly known as a double, although that terminology is inappropriate in the context of Python, and we do not use it here. Of the 64-bits within a Python float, one is allocated to representing the sign of the number; 11 are used to represent the exponent, the value to which the fraction is raised; and the remaining 52 are dedicated to representing he fraction, also known as the mantissa or significant, although owing to the way the encoding works in conjunction with a sign we effectively get 53-bits of precision. This means that thinking in decimal equivalents Python floats have at least 15 digits of decimal precision, but no more than 17 digits of decimal precision. In other words, you can convert any decimals with 15 significant figures into Python floats and back again without loss of information. Python floats support a very large range of values, larger than would be required in most applications. To determine the limits of the float type, we can query the sys. float\_info object from the built-in sys module. You can see that the largest float is 1. 797 times 10 to the 308th power, and the smallest float greater than 0 is 2. 22 times 10 to the -308th power. If we want the most negative float or the greatest float smaller than 0, we can negate these two quantities respectively. So, although floats can represent a huge range of numbers, you should be aware of their limitations. First of all, you shouldn't assume in general that any Python int can be converted without loss of information to a Python float. Although the conversion is obviously possible for small magnitude integers such as 10, because the mantissa has only 53-bits of binary precision, we can't represent every integer above 2 to the 53. Let's demonstrate that with a simple experiment at the REPL. Let's compute 2 to the power 53 as an integer. This value is 9 quadrillion, 7 trillion, 199 billion, 254 million, 740 thousand, 992. Let's make a float version of that large number. As you can see, the digits in the float are identical to the digits in the integer, and so this conversion seems to have been successful. Now let's add one to that value and perform the conversion again. Notice that the value is the same as before. We seem to have lost precision. Now let's add two to 2 to the 53. The value is different this time, but we've skipped an integer as the value has gone up. Adding three this time, again we get a different integer, but two larger than the last one. And adding four, finally we get the same integer again. Furthermore, because the float type has finite precision, some fractional values can't be represented accurately in much the same way that one-third can't be represented as a finite precision decimal. For example, neither 0. 8 nor 0. 7 can be represented in binary floating point exactly, so computations involving them return incorrect answers rounded to a nearby value which can be represented. And if you're not familiar with the details of floating point mathematics, this can seem shocking at first, but it's really no less reasonable than the fraction two-thirds not displaying as an infinitely recurring series of sixes. A full treatment of careful use of floating point arithmetic is well beyond the scope of this course, but we do want to alert you to some of the issues to motivate the introduction of alternative number types supported by Python, which avoid some of these problems by making different trade-offs. If you do need to understand more about floating point mathematics, we recommend Goldberg's classic What Every Computer Scientist Should Know About Floating-Point Arithmetic.

The Decimal Module and the Decimal Type

As we have seen, the Python float type can result in problems with even the simplest of decimal values, which would be unacceptable in any application where exact arithmetic is needed such as in a financial setting. The Decimal type with an upper case D in the decimal module with a lowercase D is a fast correctly rounded number type for performing arithmetic in base 10. Crucially, the decimal type is still a floating point type, albeit with a base of 10 rather than 2 and has finite precision, although user configurable rather than fixed. Using decimal in place of float for say an accounting application can lead to significantly fewer hard to debug edge cases. Let's take a look. We'll import the module and start by calling the decimal. getcontext() function to retrieve information about how the decimal system has been configured. The most important figure here is the precision, which tells us that by default the decimal system is configured with 28 places of decimal precision. Some of the other values in here control rounding and error signaling modes, which can be important in certain applications as we'll soon see. We create decimal instances by calling the decimal constructor. This is obvious enough when creating a decimal from an integer such as 5; however, it's a little awkward to use the module name every time, so let's pull the decimal type into the current scope with a new import statement. Notice that when the REPL echos the representation of the decimal object back to us it places quotes around the 7 indicating that the constructor also accepts strings. Let's try that with 0. 8. Let's exercise this ability to accept strings and replicate the computation that gave an inexact answer with float previously, 0. 8 minus 0. 7. With a Decimal type we get an exact answer. For fractional values, passing the literal value to the constructor as a string can be very important. Consider this example without the quotes. We are back to the same problem we had with floats. To understand why, let's deconstruct what's going on here. We have typed two number, 0. 8 and 0. 7, in base 10 into the REPL. Each of these numbers represents a Python float, so Python converts our literal base 10 representations into internal base 2 representations within the float objects. Neither of the values we have chosen can be represented exactly in base 2, so some rounding occurs. These rounded float values are then passed to the decimal constructor and used to construct the internal base 10 representations, which will be used for the computation. Finally, the subtraction is performed on the decimal objects. So although the decimal constructor supports conversion from float, you should always specify fractional decimal literals as strings to avoid the creation of an inexact intermediate base 2 float object. To avoid inadvertently constructing decimal objects from floats, we can modify the signal handling in the decimal module. We do this by setting to true the value associated with the float operation key in the traps attribute of the decimal module context. With this change in place, our attempt to construct decimal objects directly from floats raises a FloatOperation exception. This also has the desirable effect of making comparisons between decimal and float objects raise an exception too. Here we carefully construct a Decimal from a string on the left hand side of the expression, but use a float on the right hand side. Decimal, unlike float, preserves the precision of numbers supplied with trailing zeros. If we set A to Decimal(3), B to Decimal('3. 0'), and C to Decimal('3. 00'), we can see that this information has been stored when the REPL echos the values back to us. Furthermore, this stored precision is propagated through computations. Here we multiply A, B, and C by 2 resulting in ('6'), ('6. 0'), and ('6. 00'). The precision of constructed values is preserved whatever the precision setting in the module context, which only comes into play when we perform computations. Let's reduce the precision down to just six significant figures and then create a value which exceeds that precision, 1. 234567. Now performing a computation by adding one to that value we see the limited context precision kick in with the resulting value 2. 23457 having only six digits of precision. We should also point out that like the float type Decimal supports the special values for Infinity and not a number, Infinity, -Infinity, and NaN as strings. These propagate as you would expect through operations. As we have seen, decimals can be combined safely with Python integers, but the same cannot be said of floats or other number types we have met in this module. Operations with floats will raise a TypeError. This is by and large a good thing since it prevents inadvertent precision and representation problems creeping into programs. Decimal objects play very well with the rest of Python, and usually once any input data has been converted into decimal objects program code and be very straightforward and proceed as for floats and ints. That said, there are a few differences to be aware of. Consider this computation in integers, (-7) % 3 with a result of 2. Let's rationalize how this result is achieved. Here's the real number line from -10 to +3. Here are all the multiples of the divisor 3, and here is our dividend -7. Minus 9 is the largest multiple of 3, which is less that -7, and -7 is two greater than -9. This is our result. Now consider the same computation with decimals, Decimal(-7) % Decimal(3). This has a result of Decimal('-1'). How is this achieved? Here's our number line again and our multiples of the divisor 3. Here's our dividend -7. Minus 6 is the next multiple of 3 towards 0, and -7 is one less than that giving us a result of -1. It may seem capricious that Python has chosen different modulus conventions for different number types, and indeed it's somewhat arbitrary which convention different programming language is used, but it works this way so that floats retain compatibility with the legacy Python versions whereas Decimal is designed to implement the IEEE854 decimal floating point standard. One result of this is that widespread implementations of common functions may not work as expected with different number types. Consider a function to test whether a number is odd, which is typically written like this by taking % 2 and comparing the result for equality with 1. This works well for integers. Is\_odd(2) is False, is\_odd(3) is True, is\_odd(-2) is False, and is\_odd(-3) is True. It also works well for these floats. Is\_odd(2. 0) is False, is\_odd(3. 0) is True, is\_odd(-2. 0) is False, and is\_odd(-3. 0) is True. But when used with Decimal it fails for negative odd numbers. Is\_odd(Decimal(2)) is False, is\_odd(Decimal(3)) is True, is\_odd(Decimal(-2)) is False, but is\_odd(Decimal(-3)) is also False. This is because -1 is not equal to +1. To fix this, we can rewrite is\_odd as a not even test, which also works for negative decimals. To maintain consistency and preserve the important identity, X is equal to (x // y) \* y + x % y, the integer division operator also behaves differently. Consider (-7) // 3 using integer division. This results in -3. How is this result achieved? Here's our number line with the multiples of the divisor 3 and our dividend -7. Minus 9 is the largest multiple of 3 less than -7. Three divides into this one, two, three times. To account for the sign, the result is -3. The same again, but with a Decimal type, Decimal(-7) // Decimal(3). This results in Decimal('-2'). Here's our number line again with the multiples of three and our dividend -7. This time -6 is the next multiple of three towards 0 from our dividend, and our divisor 3 divides into this once, twice. Again, taking account of the sign, the result is -2. It's confusing that the double slash operator is known in Python as the floor division operator, but has not been implemented in this way in the case of Decimal where it truncates towards 0, so it's perhaps better to think of the double slash operator as simply the integer division operator whose semantics are type dependent. The functions of the math module cannot be used with the Decimal type, although some alternatives are provided as methods on the decimal class. For example, to compute square roots, use the square root method. A list of other methods supported by Decimal can be found in the Python documentation. Before moving on, let's recap which kinds of values the number types we have in our toolkit so far can represent. Although float cannot exactly represent 0. 7, this number can be exactly represented by decimal. Nevertheless, many numbers such as two-thirds cannot be represented exactly in either binary or decimal floating point representations. To plug some of these gaps in the real number line, we must turn to a fourth number type for representing rational fractions.

Rational Numbers with the Fraction Type

The fractions module contains the fraction type for representing so-called rational numbers, which consist of the quotient of two integers such as the number two-thirds with enumerator of 2 and a denominator of 3 or in the number four-fifths with a numerator of 4 and a denominator of 5. An important constraint on rational numbers is that the denominator must be non-zero. Let's see how to construct fractions. First we'll import the fraction type from the Fraction module. The first form of the constructor we'll look at accepts two integers for the numerator and denominator respectively. Here we construct two\_thirds with 2 and 3 and four\_fifths with 4 and 5. Notice that this is also the form in which the fraction instances are echoed back to us by the REPL. Attempting to construct with a zero denominator raises a ZeroDivisionError. Of course given that the denominator can be one, any integer, however large, can be represented as a fraction. Fractions can also be constructed directly from float objects, although be aware that if the value you expect can't be exactly represented by the binary float such as 0. 1 you may not get the result you bargained for. Fractions support interoperability with Decimal though, so if you can represent the value as a decimal you'll get an exact result. Finally, as with decimals, fractions can be constructed from a string. Arithmetic with fractions is without surprises. Two-thirds plus four-fifths is twenty-two fifteenths, two-thirds minus four-fifths is minus two-fifteenths, two-thirds multiplied by four-fifths is eight- fifteenths, two-thirds divided by four-fifths is five-sixths, and two-thirds integer divided by four- fifths is zero, and two-fifths modulus four-fifths is two-thirds. Unlike Decimal, the Fraction type does not support methods for square roots and such like for the simple reason that the square root of a rational number such as two may be an irrational number and not representable as a fraction object. However, fraction objects can be used with the math. ceil and math. floor functions, which do return integers. Between them Python ints, floats, decimals, and fractions allow us to represent a wide variety of numbers on the real number line with various tradeoffs in precision, exactness, convenience, and performance. Later in this course module we'll provide a compelling demonstration of the power of rational numbers for robust computation.

The Complex Type and the cmath Module

Python supports one more numeric type for complex numbers. This course isn't the place to explain complex numbers in depth. If you need to use complex numbers, you probably already have a good understanding of them, so we'll quickly cover the syntactic specifics for Python and provide a motivating example. Complex numbers are built into the Python language and don't need to be imported from a module. Each complex number has a real part and an imaginary part, and Python provides special literal syntax to produce the imaginary part by placing a J suffix onto a number where a J represents the imaginary square root of -1. Here we specify the number, which is twice the square root of -1, 2j. Depending on which background you have, you may have been expecting an I here rather than a J. Python uses the convention adopted by the electrical engineering community for denoting complex numbers where complex numbers have important uses as we'll see shortly rather than the convention used by the mathematics community. An imaginary number can be combined with a regular float representing a real number using the regular arithmetic operators, so we can produce a complex number 3 + 4j. Notice that this operation results in a complex number with non-zero real and imaginary parts, so Python displays both components of the number in parentheses to indicate that this is a single object. Such values have a type of complex. The complex constructor can also be used to produce complex number objects. It can be passed one or two numeric values representing the real and optional imaginary components of the number or the constructor can be passed a single string argument containing the string delimited by optional parentheses, but which must not contain whitespace. Note that although the complex constructor will accept any numeric type so it can be used for conversion from other numeric types in much the same way as the int and float constructors can, the real and imaginary components are represented internally as floats with all the same advantages and limitations. To extract the real and imaginary components as floats, use the real and imag attributes. Complex numbers also support a method to produce the complex conjugate. The functions of the math module cannot be used with complex numbers, so a module cmath is provided containing versions of the functions, which both accept and return complex numbers. For example, although the regular math. sqrt function cannot be used to compute the square roots of negative numbers as we see here, it gives a ValueError, the same operation works fine with cmath. sqrt returning an imaginary result. In addition to complex equivalents of all the standard math functions, cmath contains functions for converting between the standard Cartesian form and polar coordinates. To obtain the phase of a complex number also known in mathematical circles as its argument, use cmath. phase, but to get its modulus or magnitude, use the built-in abs function. We'll return to the abs function shortly in another context. These two values can be returned as a tuple pair using the cmath. polar function, which of course we can use in conjunction with tuple unpacking to get the modulus and phase in a single operation. This operation can be reversed using the cmath. rect function passing in the modulus and phase and the two arguments, although note that repeated conversions may be subject to floating point rounding error as we've experienced here. To keep this grounded firmly in reality, here's an example of the practical application of complex numbers to electrical engineering, specifically analysis of the phase relationship of voltage and current in AC circuits. First we create three functions to create complex values for the impedance of inductive, capacitive, and resistive electrical components respectively. The first function for inductive components uses the reactance in ohms to specify the imaginary component of the complex impedance. Function for capacitive components negates the reactance in ohms and uses it to specify the imaginary component of the complex impedance. And finally, the function for resistive components uses the resistance in ohms to specify the real component of the complex impedance. The impedance of a circuit is the sum of the quantities for each component. We can now model a simple series circuit with an inductor of 10 ohms reactants, a resistor of 10 ohms resistance, and a capacitor with 5 ohms reactance. We now use cmath. phase to extract the phase angle from the previous result, which is 0. 463 radians and convert this from radians to degrees using a handy function in the math module, math. degrees. This means that the voltage cycle lacks the current cycle by a little over 26 degrees in this circuit.

Built-In Numeric Functions abs() and round()

As we've seen, Python includes a large number of built-in functions, and we'd like you to have seen them all excluding a few we think you should avoid by the end of this course. Several of the built-in functions are operations on numeric types, so it's appropriate to cover them here. We already briefly encountered the abs() function when looking at complex numbers where it returned the magnitude of the number, which is always positive. When used with integers, floats, decimals, or fractions, it simply returns the absolute value of the number, which is the non-negative magnitude without regards to its sign. In effect, for all number types including complex, abs returns the distance of the number from 0. Another built-in is the round() function, which rounds to a given number of decimal digits. So, for example, round 0. 2812 to three places is 0. 281, and round 0. 625 to one place is 0. 6. To avoid bias, when there are two equally close alternatives, rounding is towards even numbers. So, round 1. 5 rounds up, and round 2. 5 rounds down, in this case both to a value of 2. As for abs(), the round() function is implemented for int where it has no effect, float we've already seen, Decimal, here we round Decimal 3. 25 to one place giving Decimal 3. 2, and even for Fraction rounding 57 one- hundredths to two places gives 57 one-hundredths whereas to one place gives three-fifths, and to zero places gives one, although notice that round is not supported for complex. Be aware that when used with float, which uses a binary representation, round(), which is fundamentally a Decimal operation, can give surprising results. For example, rounding 2. 675 to two places should yield 2. 68 since 2. 675 is midway between 2. 67 and 2. 68 and the algorithm rounds towards the even digit. However, in practice we get an unexpectedly rounded down result. Rounding 2. 675 to two places gives 2. 67. As we have seen before, this is caused by the fact that our literal float represented in base 10 can't be exactly represented in base 2, so what is getting rounded is the binary value which is close to, but not quite the value we specified. If avoiding these quirks is important for your application, you know what to do. Use the Decimal type.

Number Base Conversions

All this talk of number bases brings us onto another set of built-in functions, the base conversions. Back in the Python Fundamentals course we saw that Python supports integer literals in base 2 or binary using a 0b prefix, base 8 or octal using the 0o prefix, and base 16 or hexadecimal using a 0x prefix. Using the bin, oct, and hex functions we can convert in the other direction with each function returning a string containing a valid Python expression. If you don't want the prefix, you can strip it off using string slicing. The int constructor and conversion function also accepts an optional base argument. Here we use it to pass a string containing a hexadecimal number without the prefix into an integer object. The valid values of the base argument are 0 and then 2-36 inclusive. For numbers in bases 2-36, as many digits as required from the sequence of 0-9 followed by a-z are used, although letters may be used in lowercase or uppercase. When specifying binary, octal, or hexadecimal strings, the standard Python prefix may be included. Finally, using base 0 tells Python to interpret the string according to whatever the prefix is or if no prefix is present to assume that it's decimal. Note that base 1 or unary systems of counting are not supported.

The datetime Module and date Type

The last important scaler types we consider in this part of the course come from the datetime module. The types in this module should be the first resort when needing to represent time-related quantities. The types are date, a Gregorian calendar date. Note that the type assumes a proleptic Gregorian calendar that extends backwards for all eternity and into the infinite future. For historical dates, this must be used with some care. The last country to adopt the Gregorian calendar was Turkey in 1927. The time type is the time within an ideal day, which ignores lead seconds. And then there is datetime, a composite of date and time. Each of the two value types for the time component can be used in so-called naïve or aware modes. In naïve mode the values lack time zone and daylight saving time information, and their meaning with respect to other time values is purely by convention within a particular program. In other words, part of the meaning of the time is implicit. On the other hand, in aware mode these objects have knowledge of both time zone and daylight saving time and so can be located with respect to other time objects. The abstract tzinfo and concrete timezone classes are used for representing the timezone information required for aware time objects. Finally, we have timedelta, a duration expressing the difference between two date or datetime instances. As with the other number and scaler types we have looked at, all objects of these types are immutable. Once created, their values cannot be modified. Let's start by importing the datetime module and representing some calendar dates. Here is the 6th of January 2014. The year, month, and day are specified in order of descending size of unit duration, although if you can't remember the order you can always be more explicit with keyword arguments. Each value is an integer, and the month and day values are one-based. So, as in the example here, January is month one and the 6th day of January is day six just like regular dates. For convenience, the date class provides a number of named constructors or factory methods implemented as class methods. The first of these is today, which returns the current date. There's also a constructor, which can be used to create a date from a POSIX timestamp, which is the number of seconds since the 1st of January 1970. For example, the billionth second fell on the 9th of September 2001. The third named constructor is fromordinal, which accepts an integer number of days starting with one on the 1st of January in year 1 assuming the Gregorian calendar extends back that far. The year, month, and day values can be extracted with the attributes of the same name. There are many useful instance methods on date. We cover some of the more frequently used ones here. To determine the weekday, use either the weekday or isoweekday methods. The former returns a zero-based day number in the range 0-6 inclusive where Monday is 0 and Sunday is 6. The isoweekday method uses a one-based system where Monday is 1 and Sunday is 7. To return a string in ISO 8601 format, by far the most sensible way to represent dates is text. Use the isoformat method. For more control over date formatting with strings, you can use the strftime() method, read as string- format-time using a wide variety of placeholders as specified in the Python documentation. Here %A gives the day of the week as a word, %d gives the day of the month, %B gives the month as a word, and %Y gives the year. Or you can use the format method of the string type with a suitable format placeholder format string. Unfortunately, both of these techniques delegate to the underlying platform dependent libraries on depending your Python interpreter, so the format strings can be fragile with respect to portable code. Furthermore, many platforms do not provide tools to modify the result in subtle ways such as omitting the leading zero on month days less than 10. On this computer we can inert a hyphen into the format string to suppress leading zeros, but this is not portable even between different versions of the same operating system. A better and altogether more Pythonic solution is to extract the date components individually and pick and choose between date-specific formatting operators and date attribute access for each component. This is much more powerful and portable. Finally, the limits of date instances can be determined with the min and max class attributes, and the interval between successive dates retrieved from the resolution class attribute. The response from this is in terms of a timedelta type, which we'll return to shortly.

The time Type

The time class is used to represent a time within an unspecified day with optional time zone information. Each time value is specified in terms of four attributes for hours, minutes, seconds, and microseconds. Each of these is optional, although of course the preceding values must be provided if positional arguments are used. Here is a time of 3 hours; a time of 3 hours and 1 minute, a time of 3 hours, 1 minute, and 2 seconds; and a time of 3 hours, 1 minute, 2 seconds, and 232 microseconds. As is so often the case, keyword arguments can lend a great deal of clarity to the code. Here is 23 hours, 59 minutes, 59 seconds, and 999, 999 microseconds. All values are zero-based integers. Recall that for dates they were one-based, and the value we have just created represents the last representable instant of any day. Curiously, there are no name constructors for time objects. The components of the time can be retrieved through the expected attributes hour, minute, second, and microsecond. As for dates, an ISO 8601 string representation can be obtained with the isoformat method. More sophisticated formatting is available through the strftime() method and the regular str. format method, although the same caveats about delegating to the underlying C library apply with the portability traps for the unwary. We prefer the more Pythonic, although admittedly more verbose version using string. format. The minimum and maximum times on the resolution can be obtained using the same class attributes as for dates, min, max, and resolution.

The datetime Type

You may have noticed that throughout this section we have fully qualified the types in the datetime module with the module name, and the reason will now become apparent. The composite type, which combines date and time into a single object is also called datetime with a lowercase D. For this reason, you should avoid doing from datetime import datetime because from that point on the datetime name now refers to the class rather than to the enclosing module. As such, trying to get all of the time type then results in retrieval of the time method of the datetime class. To avoid this nonsense, you could import the datetime class and bind it to an alternative name, for example, from datetime import datetime as Datetime but now with an uppercase D or use a short module name by doing import datetime as dt. We'll continue what we have been doing so far and fully qualify the name. As you might expect, the compound datetime constructor accepts year, month, day, hour, minute, second, and microsecond values of which at least year, month, and day must be supplied. The argument ranges are the same as for the separate date and time constructors. In addition, the datetime class supports a rich selection of name constructors implemented as class methods. The today and now methods are almost synonymous, although now may be more precise on some systems. In addition, the now method allows specification of a time zone, but we'll return to that topic later. Remember that these functions and all the other constructors we've seen so far return the local time according to your machine without any record of where that might be. You can get a standardized time using the utcnow() function, which returns the current coordinated universal time utc taking into account the time zone of your current locale. Note that even utcnow returns a naïve datetime, which doesn't know it is represented in utc. We'll cover a time zone aware alternative shortly. As with the date class, datetime supports the fromordinal constructor and a fromtimestamp constructor supplemented by a utcfromtimestamp constructor, which also returns a naïve datetime object. If you wish to combine separate date and time objects into a single datetime instance, you can use the combine class method. For example, to represent 8:15 this morning you can do this. We can get the current date from the today() class method of the date class and directly construct the time 8:15 combining them together with the combine class method. The final name constructor we'll look at is strptime, read as string-parse-time. It can be used to pass a date in string format according to the supplied format string using the same syntax as is used for rendering dates and times to strings in the other direction with strftime. Here we pass the date Monday the 6th of January 2014 and the time 12:13:31. To obtain separate date and time objects from a datetime object, use the date and time methods. Beyond that, the datetime type essentially supports the combination of the attributes and methods supported by date and time individually such as the day attribute and the isoformat method for ISO 8601 datetimes.

Durations with the timedelta Type

Finally we have timedelta, a duration expressing the difference between two date or datetime instances. The timedelta constructor is superficially similar to the constructor for the other types, but has some important differences. The constructor accepts a combination of days, seconds, microseconds, milliseconds, minutes, hours, and weeks. Although positional arguments could be used, we strongly urge you to use keyword arguments for the sake of anybody reading your code in future including yourself. The constructor normalizes and sums the arguments, so specifying 1 millisecond and 1000 microseconds results in a total of 2000 microseconds. Notice that only three numbers are stored internally, which represent days, seconds, and microseconds, the components for which can be accessed through attributes of the same name. No special string formatting operations are provided for timedeltas, although you can use the str function to get a friendly representation. Compare this to the repr we have already seen.

Arithmetic with datetime

Timedelta objects arise when performing arithmetic on datetime or date objects, for example, when subtracting two datetimes. (Typing) These two datetimes have a difference of 55 days and 7980 seconds. We can convert this into just seconds by using the total\_seconds method. We can also add a timedelta onto a date. For example, one way to find the date in three-weeks' time is to construct a timedelta of 1 week, multiply it by 3, and add that to today's date. Be aware that arithmetic on time objects is not supported and will result in a TypeError.

Time Zones

So far, all of the time-related objects we have created have been so-called naïve times, which represent times in local time. To create time zone aware objects, we must attach instances of a tzinfo object to our time values. Time zones and daylight saving time are a very complex domain mired in international politics and which could change at any time. As such, the Python Standard Library does not include an exhaustive time zone database. If you need up to date time zone date, you'll need to use the third-party pytz or dateutil modules. That said, Python 3, although not Python 2, contains rudimentary support for time zone specification. The tzinfo abstraction on which more complete time zone support can be added is supported in both Python 2 and Python 3. The tzinfo class is abstract and so cannot be instantiated directly. Fortunately, Python 3 includes a simple timezone concrete class, which can be used to represent time zones, which are a fixed offset from utc. For example, here in Norway we are currently in the Central European Time or CET time zone, which is utc +1. Let's construct a timezone object to represent this. To the timezone constructor we pass a timedelta object with a value of 1 hour and the cet name for the time zone. I can now specify this tzinfo instance when constructing a time or a datetime object. Here's the departure time of my flight to London tomorrow, 11:30 in the morning on the 7th of January 2014 in Central European Time. The timezone class has an attribute called utc, which is an instance of timezone configured with a zero offset with utc, useful for representing utc times. In the wintertime London is on utc, so I'll specify my arrival in London in utc. This is at 5 minutes past 1 in the afternoon in utc. By subtracting these two time zone aware datetime instances, we can determine that the flight duration is 9300 seconds, which by converting to a string we can see is more usefully formatted as 2 hours and 35 minutes. For more complete timezone support including correct handling of daylight savings time, which is not handled by the basic timezone class, you'll either need to subclass the tzinfo base class yourself for which instructions are provided in the Python documentation or employ one of the third-party packages such as pytz.

Duck Tail: Floating Point vs. Rational Numbers

Rational numbers have an interesting role in the field of computational geometry, a world where lines have zero thickness, circles are perfectly round, and points are dimensionless. Creating robust geometric algorithms using finite precision number types such as Python's float, is fiendishly difficult because it's not possible to exactly represent numbers such as one-third, which rather gets in the way of performing simple operations like dividing a line into exactly three equal segments. As such, rational numbers modeled by Python's fractions, can be useful for implementing robust geometric algorithms. These algorithms are often deeply elegant and surprising because they must avoid any detour into the realm of irrational numbers, which cannot be represented in finite precision, which means that using seemingly innocuous operations like square root for example to determine the length of a line using Pythagorus is not permitted. One example algorithm which benefits from rational numbers is a simple collinearity test, that is to determine whether three points lie on the same line. This can be further refined to consider whether a query point P is above, exactly on, or below the line. A robust technique for implementing collinearity is to use a so-called orientation test, which determines whether three points are arranged counterclockwise, in a straight line so they are neither clockwise nor counterclockwise, or clockwise. You don't need to understand the mathematics of the orientation test to appreciate the point of what we're about to demonstrate. Suffice it to say that the orientation of three two-dimensional points can be concisely computed by computing the sign of the determinant of a 3 x 3 matrix containing the X and Y coordinates of the points in question. This function returns +1 if the polyline P, Q, R executes a left turn and the loop is counterclockwise or 0 if the polyline is straight or -1 if the polyline executes a right turn and the loop is clockwise. These values can in turn be interpreted in terms of whether the query point P is above, on, or below the line through Q and R. To cast this formula in Python we need a sign function and a means of computing the determinant. Both are straightforward, although perhaps not obvious and give us the opportunity to learn some new Python. First the sign function. You may be surprised to learn, and you wouldn't be alone, that there is no built-in or library function in Python which returns the sign of a number as -1, 0, or +1. As such, we need to roll our own. The simplest solution is probably something like this where we test where the X is less than 0, greater than 0, or must be exactly 0. This works well enough, but a more elegant solution will be to exploit an interesting behavior of the bool type, specifically how it behaves in the subtraction. Let's do a few experiments. False - False is 0 whereas False - True is -1, True - False is 1, and True - True is 0. Intriguingly, subtraction of bool objects has an integer result. In fact, when used in arithmetic operations this way, True is equivalent to +1 and False is equivalent to 0. We can use this behavior to implement a most elegant sign function by subtracting x < 0 from x > 0. (Typing) Now we need to compute the determinant. In our case this turns out to reduce down to (qx - px) multiplied by (ry - py) - (qy - py) multiplied by (rx - px). So, the definition of our orientation function using tuple coordinate pairs for each point looks like this. (Typing) Let's test this on some examples. First we'll set up three points a, b, and c. Now we'll test the orientation of a, b, c. This represents a left turn, so the function returns +1. On the other hand, the orientation of a, c, b is -1. Let's introduce a fourth point d, which is colinear with a and c. As expected, our orientation function returns 0 for the group a, c, d. So far, so good. Everything we've done so far is using integer numbers, which have arbitrary position, and our function only uses multiple and subtraction with no division to result in float values, so all of that precision is preserved. But what happens if we use floating point values as our input data? Let's try some different values using floats. Here are three points which lie on a diagonal line, e, f, and g. As would expect, our orientation test determines that these three points are colinear. Furthermore, moving the point E up a little by increasing its Y coordinate by even a tiny amount gives us the answer we would expect. Our orientation test now returns 1, so E must be above the line. Now let's increase the Y coordinate just a little more. In fact, we'll increase it by the smallest possible amount, the next representable floating point number. Wow! According to our orientation function, the points e, f, and g are colinear again. This cannot possibly be. In fact, we can go through the next 23 successive floating point values up to this number with our function still reporting that the three points are colinear until we get to the next value at which point things settle down and become well-behaved again. What's happening here is that we've run into problems with the finite precision of Python floats for points very close to the diagonal line, and the mathematical assumptions we make in our formula about how numbers work break down due to rounding problems. We can write a simple program to take a slice through the space around the diagonal line printing the value of our orientation function for all representable points on a vertical line, which extends just above and just below the diagonal line. The program includes definitions of our sign and orientation functions together with a main() function, which runs the test. The main() function includes a list of the 271 nearest representable Y coordinate values to 0. 5. We haven't included the code to generate these successive float values because it's far from straightforward to do in Python and somewhat besides the point. Then the program iterates over these py values and performs the orientation test each time printing the result. The complex format string is used to get nice looking output, which lines up in columns. When we look back at the output, we see an intricate pattern of results emerge, which isn't even symmetrical around the central 0. 5 value. With increasing Y coordinates, the results go from below the line to on the line to below the line to on the line including at the only point that is actually on the line where py is equal to 0. 5 to above the line to on the line and then finally settling down to above the line. By this point you should at least be wary of using floating point arithmetic for geometric computation. Lest you think this can easily be solved by introducing a tolerance value or some other clunky solution, we'll save you the bother by pointing out that doing so merely moves these fringing effects to the edge of the tolerant zone. What to do? Fortunately, as we alluded to at the beginning of this tale, Python gives us a solution in the form of the rational numbers implemented as the Fraction type. Let's make a small change to our program converting all numbers to fractions before proceeding with a computation. We'll do this by modifying the orientation function to convert each of its three arguments into a pair of fractions. As we know, the fraction constructor accepts a selection of numeric types including float. The variable D will now also be a fraction, and the sign function will work as expected with this type since it only uses comparison to 0. Let's run our modified example. Now with increasing Y coordinates starting below the line our result values are consistently below the line. The only which returns on the line is 0. 5 itself as we would expect, and from then on all values are returned as being above the line. Using fraction internally, our orientation function gets the full benefit of exact arithmetic with effectively infinite precision and consequently produces an exact result with only one position of P being reported as being colinear with Q and R. Going further, we can map out the behavior of our orientation functions by hooking up our program to the bmp image file writer we created in the Files and Resource Management module of our Python Fundamentals course. By using our sequence of consecutive floats centered on the 0. 5 value to generate two-dimensional coordinates in a tiny square region straddling our diagonal line and evaluating our orientation function at each point, we can produce a view of how our diagonal line is represented using different number types. The code change is straightforward. First we'll import our bmp module we created in Python Fundamentals. We've included a copy in the example code for this course too. Then we replace the code which iterates through our line transect with code to do something similar in two dimensions using nested list comprehensions to produce nested lists representing pixel data. We use a dictionary of three entries called color to map from the -1, 0, and +1 orientation values to black, mid-gray, and white respectively. The inner comprehension produces the pixels within each row, and the outer comprehension assembles the rows into an image. We reverse the order of the rows using a call to reversed to get our coordinate axes to correspond to the image format conventions. The result of this call is that pixels is now a list of lists of integers where each integer has a value of 0, 127, or 255 depending on whether the pixel is below, on, or above the diagonal line. Finally, the pixel data is written out as a bmp file with a call to the bmp. write\_grayscale function passing the file name and the pixel data. We get a map of above and below the diagonal for float computations whereas with the rational number code active we get a much more sensible and intuitive result. Here are side-by-side views of our diagonal line as seen by the float and fraction versions of our orientation function. We've zoomed in on a portion of the fraction result so you can see that only the pixels exactly along the diagonal are marked as being on the line. Hopefully this section will help you understand the tradeoffs in selecting different number types and make you wary of using floating point arithmetic in certain cases where exact computation is important.

Summary

In this course module on numeric and scaler types we've reviewed the capabilities of the int and float types and looked at how to query sys. float\_info to get details of the float implementation. We understood the limitations of finite precision of the float type and the impact this has on representing numbers in binary. We introduced the decimal module, which provides another floating point numeric type founded on base 10 with user configurable precision. We explained that decimal is preferred for certain financial applications such as accounting where the problem domain is inherently decimal in nature. We highlighted some key differences in behavior between decimal and the other numeric types, particularly in the behavior of integer division and modulus operations, which for Decimal truncate towards 0, but for int and float round towards negative infinity. This has implications for writing functions, which need to work correctly with different number types. We demonstrated support for the rational numbers using the Fraction type from the fractions module showing how to construct and manipulate fraction values using arithmetic. We introduced the built-in complex type and gave an overview of complex number support in Python including the cmath module, which includes complex equivalents for the functions in the math module. We gave a brief demonstration of the utility of complex numbers in Python by using them to solve a simple electrical engineering problem determining the properties of an AC current in a simple circuit. We then moved on to cover two built-in functions which relate to numeric types, abs for computing the distance of a number from 0, which also works for complex numbers, and round, which rounds to a specified decimal precision and the surprises this can lead to when used on floats, which are internally represented in binary. We reviewed the literal syntax for numbers in different bases and showed how to convert strings in these literal forms using the built-in bin, oct, and hex functions. We demonstrated how to convert from strings in bases 2-36 inclusive by using the base argument to the int constructor and how to convert from any literal numeric in string form by specifying base 0. We covered the representation of dates, times, and compound datetime objects using the facilities of the datetime module. We explained the difference between naïve and time zone aware times and the many named constructors available for construction time-related objects. We showed string formatting of time and date objects and how to pass these strings back into datetime objects. We explained how durations can be modeled with a timedelta object and how to perform arithmetic on datetimes. We demonstrated the basic time zone support available in Python 3 with the timezone class and referred you to the third-party pytz package for more comprehensive support. We showed how regular floats can be unsuitable for geometric computation owing to finite precision and how to solve this by deploying the fraction type in geometric computation. Next time on Python: Beyond the Basics we'll look in more detail at iterables and iteration and counter the map() and reduce() algorithms and see how to implement the iterable and iterator protocols for our own objects. Thanks for watching, and we'll see you in the next module.

Iterables and Iteration

Multi-Input Comprehensions

Hello. My name is Austin Bingham, and welcome to the seventh module of Python: Beyond the Basics. In this module we'll be taking a deeper look at iterables and iteration in Python including topics such as more advanced comprehensions, some functional style tools, and the protocols underlying iteration. This builds upon our previous material-covering iteration, and as with that previous material you'll find that these techniques and tools can help you write more expressive, elegant, and even beautiful code. As you'll recall, comprehensions are a sort of short-hand syntax for creating collections and iterable objects of various types. For example, a list comprehension creates a new list object from an existing sequence, and it looks like this. Here we've taken a range sequence, in this case the integers from 0-9, and created a new list where each entry is two times the value from the original sequence. This new list is a completely normal list just like any other list made using any other approach. There are comprehension syntaxes for creating dictionaries, sets, and generators, as well as lists, and all of the syntaxes work in essentially the same way. All of the comprehension examples we've seen up to now use only a single input sequence, but comprehensions actually allow you to use as many input sequences as you want. Likewise, a comprehension can use as many if-clauses as you need as well. For example, this comprehension uses two input ranges to create a set of points on a 5 x 5 grid giving us a list containing the so-called Cartesian products of the two input ranges. The way to read this is as a set of nested for loops where the later for-clauses are nested inside the earlier for-clauses, and the result expression of the comprehension is executed inside the innermost or last for loop. So, for this example the corresponding nested for loop structure would look like this. The outer for loop, which binds to the X variable, corresponds the first for-clause in the comprehension. The inner for loop, which binds to the Y variable, corresponds to the second for-clause in the comprehension. The output expression in the comprehension where we create the tuple is nested inside the innermost for loop. The obvious benefit of the comprehension syntax is that you don't need to create the list variable and then repeatedly append elements to it. Python takes care of that for you in a more efficient and readable manner with comprehensions. As we mentioned earlier, you can have multiple if-clauses in a comprehension along with multiple for-clauses. These are handled in essentially the same way as for clauses. Later clauses in the comprehension are nested inside earlier clauses. For example, consider this comprehension. This is actually fairly difficult to read, and it might be at the limit of the utility of comprehensions, so let's improve the layout a bit. There. That's better. This calculates a simple statement involving two variables and two if-clauses. By interpreting the comprehension as a set of nested statements, the non-comprehension form of this statement is like this. This can be extended to as many statements as you want in the comprehension, though, as you can see, you might need spread your comprehension across multiple lines to keep it readable. This last example also demonstrates an interesting property of comprehensions where later clauses can refer to variables bound in earlier clauses. In this case, the last if statement refers to X, which is bound in the first for-clause. The for-clauses in a comprehension can also refer to variables bound in earlier parts of the comprehension. Consider this example which constructs a sort of triangle of coordinates. Here the second for-clause, which binds to the Y variable, refers to the X variable defined in the first for-clause. If this is confusing, just remember that you can think of this as a set of nested for loops. In this formulation it's entirely natural for the inner for loop to refer to the outer.

Nested Comprehensions

There's one more form of nesting in comprehensions that's worth noting, although it doesn't really involve new syntax or anything beyond what we've already seen. We've been looking at the use of multiple for and if-clauses in a comprehension, but it's also entirely possible to put comprehensions in the output expression for a comprehension. That is, each element of the collection produced by a comprehension can itself be a comprehension. For example, here we have two for-clauses, but each belongs to a different comprehension entirely. The outer comprehension uses another comprehension to create a list for each entry and its result. Rather than a flat list then, this produces a list of lists. The expansion of this comprehension looks like this, and the resulting list of lists looks like this. This is similar to, but different from multisequence comprehensions. Both forms involve more than one iteration loop, but the structures they produce are very different. Which form you choose will of course depend on the kind of structure you need, so it's good to know how to use both. In our discussion of multi-input and nested comprehensions, we've only shown list comprehensions in the examples; however, everything we've talked about applies equally to set comprehensions, dict comprehensions, and generator comprehensions. For example, you can use a set comprehension to create a set of all products of two numbers between 0 and 9 like this or you can create a generator version of the triangle coordinates we constructed earlier.

The map() Function

Python's concept of iteration and iterable objects is fairly simple and abstract not involving much more than the idea of a sequence of elements that can be accessed one at a time in order. This high level of abstraction allows us to develop tools that work on iterables at an equally high level, and Python provides you with a number of functions that serve as simple building blocks for combining and working with iterables in sophisticated ways. A lot of these ideas were originally developed in the functional programming community, so some people refer to the use of these techniques as functional- style Python. Whether you think of these as a separate programming paradigm or just as more tools in your programming arsenal, these functions can be very useful and are often the best way to express certain computations. The map() function is probably one of the most widely recognized functional programming tools in Python. At its core, map() does a very simple thing. Given a function and a sequence of objects, it calls the function for every element in the sequence producing a new sequence containing the return values of the functions. In other words, we map a function over a sequence to produce a new sequence. Let's see a simple example. Suppose we wanted to find the Unicode codepoint for each character in a string. The map() expression for that would look like this. This essentially says for every element in the string call the function ord with that element as an argument. Generate a new sequence comprising return values of ord in the same order as the input sequence. Graphically it looks like this. So, let's try this out in the REPL. Rather than return a list as you might expect, we instead get a map object. The map() function, it turns out, performs lazy evaluation. That is, it doesn't produce any output until it's needed. Another way to say this is that map() will not call its function or access any elements from its input sequence until they're actually needed. The map object returned by map() is itself an iterator object, and only by iterating over it can you start to produce output. To make this a bit more clear, let's reuse our Trace decorator from Module 3 to print out a message whenever map() calls its function. We won't be using Trace as a decorator, but rather we'll leverage the fact that we can call a trace instance to get a callable that does tracing for us. As a reminder, here's how Trace is defined, and here's how we use Trace to invoke the ord function as we map over a string. The function returned by trace-dunder call will print a bit of text each time it's called so we can see how map() works. It's not until we start to iterate over result that we see our function executed. Again, this gets to the heart of lazy evaluation. Map() does not call the function or iterate over the input sequence until it needs to in order to produce output. Here we're driving the process by manually iterating over the map object returned by map(). More commonly, you will not manually derive the map object, but instead you'll iterate over it using some higher-level method. For example, you could use the list constructor to read all of the list elements produced by map() or you could use a for loop. The point is that map's lazy evaluation requires you to iterate over its return value in order to actually produce the output sequence. Until you access the values in the sequence, they are not evaluated.

Multiple Input Sequences

So far we've seen examples using map() over single input sequences, but in fact map() can be used with as many input sequences as your mapped function needs. If the function you pass to map() requires two arguments, then you need to provide two input sequences to map(). If the function requires three arguments, then you need to provide three input sequences. Generally, you need to provide as many input sequences as there are arguments in the mapped function. What map() does with multiple input sequences is to take an element from each sequence and pass it as the corresponding argument to the mapped function to produce each output value. In other words, for each output value that map() needs to produce, it takes the next element from each input sequence. It then passes these in order as the arguments to the mapped function, and the return value from the function is the next output value for map(). An example will help make this clear. First we define three sequences and a function that takes three arguments. Now, if we map combine over sequences, we see that an element from each sequence is passed to combine for each output value. It's possible of course that the input sequences are not all the same size. In fact, some of them might be infinite sequences. Map() will terminate as soon as any of the input sequences is terminated. To see this, let's modify our combined function to accept a quantity argument. Now let's pass an infinite sequence to map() and see that it terminates after the finite inputs are exhausted.

map() Versus Comprehensions

You may have noticed that map() provides some of the same functionality as comprehensions. For example, both of these snippets produce the same lists. Likewise, both this generator expression and this call to map() produce equivalent sequences. In cases where either approach will work, there's no clear choice which is better. Neither approach is necessarily faster than the other, and while many people find comprehensions more readable others feel that the functional style is cleaner. Your choice will have to depend on your specific situation, your audience, or perhaps just your personal taste.

The filter() Function

The next functional style tool that we'll look at is filter(). As its name implies, filter() looks at each element in a sequence and filters out or removes those which don't meet some criteria. Like map(), filter() applies a function to each element in a sequence, and also like map() filter() produces its results lazily. Unlike map(), filter() only accepts a single input sequence, and the function it takes must only accept a single argument. The general form of filter() takes a function of one argument as its first parameter and a sequence as its second. It returns an iterable object of type filter(). Filter() applies its first argument to each element in the input sequence, and the sequence filter() returns contains only those elements of the input for which the function returns true. For example, here we use a lambda as the first argument to filter. This lambda returns true for positive arguments and false for everything else meaning that this call to filter() will return a sequence containing only the positive elements of its input sequence, and indeed that's precisely what we see. Remember that the return value from filter is a lazy iterable, so we have to use the list constructor or some other technique to force evaluation of the results. You can optionally pass none as the first argument to filter in which case filter() will filter out all input elements which evaluate to false in a Boolean context. For example, since 0, false, the empty list, and the empty string are all falsey, they were removed from the output sequence. While this course if focused on Python 3, the map() and filter() functions represent one area where it's useful to discuss a difference between Python 2 and 3. As we've just shown, in Python 3 the map() and filter() functions return a lazily produced sequence of values. In Python 2, however, these functions actually return lists. If you find yourself writing code for Python 2, it's important to keep this in mind.

The functools.reduce() Function

The final functional style tool we'll look at is the reduce() function in the functools standard library module. Reduce() can be a bit confusing at first, but it's actually quite simple and useful. Reduce repeatedly applied a function of two arguments to an interim accumulator value and each element of the series in turn updating or accumulating the interim value at each step with the result of the called function. The initial value of the accumulator can either be the first element of the sequence or an optional value we supply. Ultimately, the final value of the accumulator is returned thereby reducing the series down to a single value. Reduce() is not unique to Python by any means, and you may have come across it in other languages you've worked with. For example, reduce() is the same as fold in many functional languages. Likewise, it's equivalent to aggregate in. NET's LINQ and accumulate() in C++'s standard template library. The canonical example of reduce() is the summation of a sequence, and in fact reduce() is a generalization of summation. Here's how that looks using the add function from the standard library operator module, which is a regular function version of the familiar infix plus operator. Conceptually what's happening is something like this. (Typing) To get a better idea of how reduce() is calling the function, we can use a function which prints out its progress. Here we see that the interim result is passed as the first argument to the reducing function, and the next value in the input sequence is passed as the second. The reduce() function has a few edges that are worth noting. First, if you pass an empty sequence to reduce(), it will raise a TypeError. Second, if you pass a sequence with only one element, then that element is returned from reduce() without ever calling the reducing function. Reduce() accepts an optional argument specifying the initial value. This value is conceptually just added to the beginning of the input sequence meaning that it will be returned if the input sequence is empty. This also means that the optional initial value serves as the first accumulator value for the reduction. This optional value is very useful, for example, if you can't be sure if your input will have any values. Take care when selecting initial values since the correct value of course depends on the function you're applying. For example, you use an initial value of 0 for summation, but a value of 1 for products.

Combining map() and reduce()

You may have been wondering about the naming of these functions, and in particular you may have been wondering if there's any relationship between Python's map() and reduce() functions and the popular map-reduce algorithm. The answer is yes. Python's map() and reduce() are very much the same as those terms in map-reduce. To see this, let's write a small example that counts words across a set of documents. First we need a function which counts words in a document. This produces a dictionary mapping words to the number of times that word was found in the input string for example. With this function, we can now map it across a collection of documents. This will result in a sequence of dictionaries with word frequencies for the individual documents. Here's how that looks. (Typing) Next we need a function which takes two word count dictionaries and combines them. With this we have all of the tools we need to run a proper map-reduce. Here's how that looks. (Typing) So, it's interesting to see a somewhat real-world application of the relatively abstract notions we've presented so far in this module. And who knows. Maybe someday someone will find a way to make money using techniques like this.

The Iteration Protocols

In the previous Python course we looked at the basics of how iteration is implemented in Python. For using iterables and iterators there are simple protocols, and they are driven by the use of the iter() and next() functions. First, iter() is called on an iterable object to retrieve an iterator. Then next() is called on the iterator to sequentially retrieve elements from the iterable. When no more elements are available, next() will raise StopIteration. It's a very simple protocol, but a surprisingly powerful one, and it's pervasive in how Python works. In this section we'll look at how to implement iterable objects and iterator. You'll see that it's actually fairly simple and follows very much directly from the high-level iter() and next() APIs. An iterable object in Python is simply an object that responds to the iter() function. When iter() is called on an object, what that function does is called dunder-iter on the object, and dunder- iter is required to return an iterator, a concept that we'll discuss in just a moment. There's not much more to say about iterables right now. The protocol is very simple and doesn't make much sense on its own. To really see the point of iterables, we need to look at iterators first. An iterator is simply an object that fulfills the iterator protocol. The first part of the iterator protocol is the iterable protocol that we just described. This means that all iterators must implement dunder-iter. Generally speaking, however, iterators often just return themselves from dunder-iter, though that's not always the case. Iterators are also required to respond to the next() function. When next() is called on an object, what it does is call the special dunder-next method on that object. So, to implement the iterator protocol, an object needs to implement dunder-next, as well as dunder-iter. All that dunder-next needs to do is return the next value in whatever sequence the iterator represents or raise the StopIteration exception when the sequence is exhausted. To demonstrate the basic mechanics of developing an iterator, here's a simple example. This class is created with a list of three items. Each call to dunder-next returns a new value in this list until it's exhausted after which StopIteration is raised. Here's how it looks when we derive it with the next() function. Since iterators are also iterables, we can even use our iterator in a for loop.

Putting the Protocols Together

Now that we know about both the iterable and iterator protocols, let's put them together to create a complete if somewhat contrived example. First we'll modify ExampleIterator to take a sequence of values as an argument and iterate over it. Next our example iterable class will manage a collection of integers returning an ExampleIterator object when asked for an iterator. Now we can see the iterable and iterator protocols working together in a for loop just as we can see them in a comprehension. Excellent! These classes won't win any design awards, but they do demonstrate the fundamentals of how to construct your own iterators and iterables. It's all actually very simple in the end and just requires that you keep in mind the simple protocols involved.

Alternative Iterable Protocol

While most iterable objects are implemented using the protocol described above, there's another way to implement them. Rather than implementing the dunder-iter method, an object can implement the dunder- getitem method. For this to work, dunder-getitem must return values for consecutive integer indices starting at 0. When the index argument is out of the iterables range of data, then dunder-getitem must throw IndexError. For many container types which already support dunder-getitem, this alternate protocol means that they don't have to write any special code to support iteration. Here's a simple example of this protocol. We see that dunder-getitem simply returns values from the specified index in an underlying list. Since the list will naturally work for consecutive indices starting at 0, and since it will throw IndexError when the index is out of bounds, this works for iteration. Let's see this in action. (Typing)

Extended iter() Format

Finally, let's close out our discussion of iterators and iterables by looking at the so-called extended form of iter(). Normally iter() is called on objects that support the dunder-iter method of the iterable protocol, but iter() supports a two argument calling form that lets you use some objects which don't directly support the iterable protocol. In the extended for the first argument is a callable, which takes zero arguments. The second argument is a sentinel value. The return value from iter() in this case is an iterator, which produces values by repeatedly calling the callable argument. This iterator terminates when the value produced by the callable is equal to the sentinel. This form of iter() is not seen very often, and it may not be immediately clear why it's useful, but one relatively common use for it is to create infinite sequence iterators out of normal functions. In this example we'll create an iterator over an infinite sequence of timestamps. Here's how this works. The datetime. now function takes zero arguments, and it returns a datetime object for the current time. Since it takes no arguments, it's suitable as the first argument to the extended iter() form. For the sentinel argument to iter we've chosen None. Since None will never be returned by datetime. now, the iterator will never terminate producing timestamps for as long as we choose to call it. Another example of the extended iter() form shows how it can be used to read from a file until a specific line is read. In this case, we read from a file until a line containing just the string END is read. If ending\_file. txt contains this text, then the output from our loop will just be the first three lines from the file.

Duck Tail: Iterator for Streamed Sensor Data

This module is a bit high level and perhaps a bit abstract, so let's see if we can't come up with a small example that shows how these techniques might be useful in the real world. One situation where you might need to create an iterable object in Python is when reading data from a sensor. Often sensors produce a stream of data or can simply provide a value whenever queried, and it would be nice to be able to access these values in a loop. We're going to write a simple class which mimics a sensor and produces a stream of data. Since we don't all have simple access to a physical sensor for this example, we're going to simulate the sensor data with random values within a range. Here's our Sensor class. This is incredibly simple, and you can see that it meets the criteria for iteration. It has a dunder-iter method which returns an iterator, which in this case is the same object. Since the sensor also supports the dunder-next method, it works equally well as an iterator. The dunder-next method simply returns a random number, but you could imagine more complex code that actually read real values from a sensor. Let's combine this with our timestamp generator and create a small system for logging sensor data every second. (Typing) And this gives great results. Voila, your very own streaming iterable sensor in Python. If you want to really take this to the next level, you could modify sensor dunder-next to read, for example, your CPU's temperature sensor instead of returning a random value. How you do this depends on your system, but it would be a great exercise if you want to really make a working end-to-end sensor program in Python.

Summary

Hopefully this module has given you greater insight into Python's iteration system and given you greater confidence to develop with it. Let's review what we've covered. Comprehensions can process more than one input sequence. Multiple input sequences and comprehensions work like nested for loops. Later clauses in a comprehension can reference variables bound in earlier clauses. Comprehensions can also have multiple if clauses interspersed with the for clauses. Comprehensions can also appear in the result expression of a comprehension resulting in nested sequences. Python provides a number of functional style tools for working with iterators. Map() calls a function for each element in its input sequence. Map() returns an iterable object, not a fully- evaluated collection. Map() results are lazily evaluated meaning that you must access them to force their calculation. Map() results are typically evaluated through the use of iteration constructs such as for loops. You must provide as many input sequences to map() as the callable argument has parameters. Map() takes one element from each input sequence for each output element it produces. Map() stops producing output when its shortest input sequence is exhausted. Map() can be used to implement the same behavior as comprehensions in some cases. Filter() selects values from an input sequence, which match as specified criteria. Filer() passes each element of its input sequence to the function argument. Filter() returns an iterable over the input elements for which the function argument is truthy. Like map(), filter() produces its output lazily. If you pass None as the first argument to filter(), it yields the input values which evaluate to true in a Boolean context. Reduce() cumulatively applies a function to the elements of an input sequence. Reduce() calls the input function with two arguments, the accumulated result so far and the next element in the sequence. Reduce() is a generalization of summation. Reduce() returns the accumulated result after all of the input has been processed. If you pass an empty sequence to Reduce(), it will raise a TypeError. Reduce() accepts an optional initial value argument. This initial value is conceptually added to the front of the input sequence. The initial value is returned if the input sequence is empty. The map() and reduce() functions in Python are related to the ideas in the map-reduce algorithm. Python's next() function calls dunder-next on its argument. Iterators in Python must support the dunder- next method. Dunder-next should return the next item in the sequence or raise StopIteration if it is exhausted. Python's iter() function calls dunder-iter on its argument. Iterable objects in Python must support the dunder-iter method. Dunder-iter should return an iterator for the iterable object. Objects with a dunder-getitem method that accepts consecutive integer indices starting at 0 are also iterables. Iterables implemented via dunder-getitem must raise IndexError when they are exhausted. The extended form of iter() accepts a zero argument callable and a sentinel value. Extended iter() repeatedly calls the callable argument until it returns the sentinel value. The values produced by extended iter() are those returned from the callable. One use case for extended iter() is to iterated using simple functions. Protocol conforming iterators must also be iterable. Thanks for watching, and we'll see you in the next module.

Inheritance and Subtype Polymorphism

Inheritance Overview

Hello. My name is Austin Bingham, and welcome to the eighth module of Python: Beyond the Basics. In this module we'll take a deeper look at Python's support for inheritance including multiple inheritance and the underlying mechanisms by which Python dispatches method calls. We'll also look at techniques for runtime-type checking for those times when it's necessary. Before we look at multiple inheritance, let's do a quick review of single inheritance. The syntax of single inheritance is part of the class declaration with the BaseClass put in parentheses after the class name. What this essentially means is that SubClass will have all of the methods of BaseClass, and SubClass will be able to override these methods if it wants to. In other words, SubClass can do everything that BaseClass can do, and it can optionally modify or specialize that behavior. In general, a subclass initializer will want to call its base class initializer to make sure that the full object is initialized. Remember though that if a subclass doesn't define an initializer then the base class initializer is called when an instance of the subclass is created. To see the basics of inheritance in action, let's define a very simple base class called Base. (Typing) If we create an instance of Base, we see of course that its initializer is called, (Typing) and calling the f() method on that instance uses Base. f(). There should be nothing surprising here. Let's now define a subclass of Base called Sub. This subclass doesn't add any functionality to Base at all. Because we haven't defined an initializer for Sub, we can see that it inherits Base's initializer. And likewise, Sub also inherits Base. f(). Sub can override Base. f() by simply redefining the method itself. Now, if we create an instance of Sub, we see that the base initializer is still being called, but Sub's definition of F is now used. Finally, let's give Sub its own initializer. Now when we create a new instance of Sub we only see Sub's initializer being called. If you're used to languages like C++ and Java, you might have expected Python to also call Base's initializer when creating a Sub instance, but this isn't how Python behaves. Rather, Python treats the dunder-init method just like any other method, and it doesn't automatically call base class initializers for subclasses that define their own initializers. If you define an initializer in a subclass and still want to call the initializer of a base class, and this is often important to do, then you need to call it explicitly using the super() function. Let's finish this small example by doing that. Now when we construct an instance of Sub we see both the Sub and Base initializers being called. The fact that Python doesn't provide special support for calling initializers is important to understand, and if it seems strange to you don't worry. It soon becomes second nature. Hopefully this small example was just review for you. If you need to refresh your memory of basic inheritance in Python, you can always refer to the Python Fundamentals course, which covers it in some detail.

A Realistic Example: SortedList

As a more concrete and practical example of how to use inheritance in Python, we'll first define our own simplified list class called SimpleList. SimpleList uses a standard list internally, and it provides a smaller, more limited API for interacting with the list's data. We'll use SimpleList as the basis for the rest of our exploration of inheritance in Python. Next let's create a subclass of SimpleList which keeps the list contents sorted. We'll call this class SortedList, and it looks like this. You can see that the class declaration includes the class name followed by SimpleList in parentheses, so SimpleList is the base class of SortedList. The initializer for SortedList takes an optional argument, which is a sequence for initializing the list's contents. The initializer calls SimpleList's initializer and then immediately uses SimpleList. sort to sort the contents. SortedList also overrides the add method on SimpleList to ensure that the list always remains sorted. One aspect of this example that you'll notice are the calls to super(). We'll be covering super() in more detail later, but in this case it can be understood to mean call a method on my base class. So, for example, calling super. add X in SortedList. add means to simply call SimpleList. add X with the same self argument or in other words to use the base-class implementation of add. If we go to our REPL, we can see that SortedList works as expected. (Typing)

The Built-In isinstance() Function

Single inheritance in Python is relatively simple and should be conceptually familiar to anyone who's worked with almost any other object-oriented language. Multiple inheritance in Python is not much more complex, and as we'll eventually see both single and multiple inheritance ultimately rely on a single underlying model. Before we look at multiple inheritance though, we need to lay some ground work for the examples we'll be using. Along with the SortedList class we defined earlier, we're going to define another class, IntList, which only allows integer contents. This list subclass will prevent the insertion of non-integer elements. To do this, IntList will need to check the types of items that are inserted, and the tool it will use for this is the isinstance() built-in function. Isinstance() takes an object as its first argument and a type as its second. It then determines if the object is of the specified type returning true if it is and false otherwise. For example, here we see isinstance() applied to a number of built-in types. Instead of just checking for an exact typed match, isinstance() will also return true if the object is a subclass of the second argument. For example, we can see that a sorted list is an instance of SortedList, as well as of SimpleList. A final twist to isinstance() is that it can accept a tuple of types for its second argument. This is equivalent to asking if the first argument is an instance of any of the types in the tuple. For example, this call returns true because X is an instance of list. Now that we know how to use isinstance(), we can define our IntList class like this. You'll immediately notice that IntList is structurally similar to SortedList. It provides its own initializer and like SortedList overrides the add method to perform extra checks. In this case, IntList calls its \_validate method on every item that goes into the list. \_validate uses isinstance() to check the type of the candidates, and if a candidate is not an instance of int, \_validate raises a TypeError. Let's see how this looks in the REPL. (Typing) So, we can see how isinstance() can be used to do type checks in Python. Checks like that are uncommon in Python, and while some people consider them to be a sign of poor design, sometimes they're simply the easiest way to solve a problem.

The Built-In issubclass() Function

There is another built-in function related to isinstance(), which can be used for type checking. This function, issubclass(), operates on types only rather than operating on instances of types. As its name implies, issubclass() is used to determine if one class is a subclass of another. Issubclass() takes two arguments, both of which need to be type objects, and it returns true if the first argument is a direct or indirect subclass of the second. For example, we can see that both IntList and SortedList are subclasses of SimpleList as you would expect. On the other hand, SortedList is not a subclass of IntList. We can also use a simple example to verify that issubclass() looks at the entire inheritance graph, not just direct parents. These classes are obviously pretty silly, but they do illustrate that issubclass() recognizes that MyVerySpecialInt is indeed a subclass of int even though it only directly inherits from MyInt.

Multiple Inheritance

Now that we have defined two interesting subclasses of SimpleList, we're ready to take a look at multiple inheritance in Python. Multiple inheritance simply means defining classes with more than one direct base class. This feature is not universal among object-oriented languages. C++ supports multiple inheritance, for example, while Java does not. Multiple inheritance can lead to certain complex situations, for example, deciding what to do when more than one base class defines a particular method, but as we'll see Python has a relatively simple and understandable system for handling such cases. The syntax for defining a class with multiple inheritance is very similar to single inheritance. To declare multiple base classes, simply use a comma-separated list of classes in the parentheses after the class name. A class can have as many base classes as you want. Just like single inheritance, a class with multiple base classes supports all of the methods of its bases. As long as there is no overlap in the method names of the base classes, it's always easy to see how method calls are resolved. Find the base class with the matching method name, and that's which method gets called. If there is method name duplication across base classes, Python has a well-defined Method Resolution Order for determining which is called. We'll look at Method Resolution Order in more detail shortly. Let's jump right in and define our own class using multiple inheritance. So far in this module we've defined SortedList and IntList, both of which inherit from our SimpleList. Now we're going to define a class which inherits from both of these classes and thus has the properties of both. Here's our SortedIntList. It doesn't look like much does it? We've simply defined a new class and given it two base classes. In fact, the only new implementation in the class is there to give it a proper string representation. But if we go to the REPL, we can see that it works as we expect. The initializer sorts the input sequence, but rejects non-integer values. Likewise, the add method maintains both the sorting and type constraints defined by the base classes. You should spend some time playing with SortedIntList to convince yourself that it works as expected. It may not be immediately apparent how all of this works though. After all, both IntList and SortedList define add. How does Python know which add() to call? Or more importantly, since both the sorting and type constraints are being enforced by SortedIntList, how does Python seem to know to call both of them? The answers to these questions have to do with the method resolution order we mentioned earlier along with the details of how super() really works. We'll get to all of that very soon.

Details of Multiple Inheritance

Before that, there are a few more details about multiple inheritance that we need to cover. First, if a class uses multiple inheritance, but defines no initializer, only the initializer of the first base class is automatically called when an instance of that class is created. Consider this simple example. (Typing) If we now create an instance of Sub(), we see that only the initializer for Base1 is called. Through the use of super() we could design these classes such that both the Base1 and Base2 initializers are called automatically, and we'll see how to do that soon. Another useful thing to know when thinking about inheritance is the dunder-bases member of class objects. Dunder-bases is simply a tuple of a class's base classes. As you can see here SortedIntList inherits from both IntList and SortedList, and these show up in the dunder- bases member of the SortedIntList class object. The entries in dunder-bases are in the same order as they were declared in the class definition. You don't have to use multiple inheritance to populate dunder- bases as you can see if you look at dunder-bases for our IntList class.

Method Resolution Order

We're finally ready to talk about this notion of method resolution order that we've mentioned several times now. In Python the method resolution order or simply MRO of a class is the ordering of a class's inheritance graph used to determine which implementation to use when a method is invoked. When you invoke a method on an object which has one or more base classes, the actual code that gets run may be defined on the class itself, one of its base classes, a base class of a base class, or any other member of the class's inheritance graph. The MRO of a class determines the order in which the inheritance graph is searched to find the correct implementation of the method. That's a lot to take in, but MRO is actually very simple, and we'll look at some examples that will make things more clearer. First though, we need to look at where our class's MRO is stored. The method resolution order for a class is stored on a special member called dunder-mro. Here you can see the MRO for SortedIntList. The dunder-mro attribute is a tuple of classes defining the method resolution order. You can also call the method MRO on a class to get the same information in a list rather than as a tuple. So, how is the MRO used to dispatch method calls? The answer is that when you call a method on an object in Python, Python looks at the MRO for that object's type. For each entry in the MRO starting at the front and working in order to the back, Python checks if that class has the requested method. As soon as Python finds a class with a matching method, it uses that method and the search stops. Let's see a simple example. First we'll define a few classes with a diamond inheritance graph. Here the various func methods simply report which class they come from. If we look at D's MRO, we see that Python will check D first, then B, then C, followed by A, and finally object when resolving calls to objects of type D. By the way, object is the ultimate base class of every class in Python, and we'll discuss that more later in this module. Based on that MRO, what should we expect if we call func on an instance of D? Because B was the first class in D's MRO with the method func, Python called B. func. If C had been earlier in the MRO than B, then C. func would have been called. We can see this by changing the order of B and C in the definition of D. After this change the new MRO for D puts C before B, and indeed calling func on a new instance of D results in a call to C's implementation. That's all there really is to it. MRO is an ordering of a class's inheritance graph that Python calculates for you. When Python resolves a method call, it simply walks along that ordering until a class supports the requested method. Let's see the MRO for our SortedIntList class. As you might expect, the MRO is SortedIntList followed by IntList followed by SortedList with SimpleList and object bringing up the rear, so calls to add on a SortedIntList, for example, were resolved to IntList. add since IntList is the first class in the MRO with an add method. This raises a very interesting question, however. When we wrote IntList, it never had any connection to the SortedList class, yet our SortedIntList as we've seen is properly maintaining both the sorting constraint and the type constraint of both SortedList and IntList. If add resolves to IntList. add() and if IntList is using super() to call its base class implementation, how is SortedList being invoked to maintain the sorting? The answer to that mystery has to do with how super() actually works.

How Is Method Resolution Order Calculated?

Before we move on to looking at super() in detail, you might be curious to know how Python actually calculates the MRO for a class. The short answer is that Python uses an algorithm known as C3 for determining MRO. We won't go into the details of C3 except to mention a few important qualities of the MRO that it produces. First, a C3 MRO ensures that the subclasses come before their base classes. Second, C3 ensures that the base class order as defined in class definition is also preserved. Finally, C3 preserves the first two qualities independent of where in an inheritance graph you calculate the MRO. In other words, the MROs for all classes in graph agree with respect to relative class order. One outcome of the C3 algorithm is that not all inheritance declarations are allowed in Python. That is, some base class declarations will violate C3, and Python will refuse to compile them. Consider this simple example in the REPL. Here since B and C both inherit from A, B and C must both come before A in any MRO. This follows from one of the qualities that C3 preserves. However, since D's base class declaration puts A before C and since C3 also guarantees that base class declaration order is preserved, C3 cannot produce a consistent MRO. That is, it can't put A both before and after C. Understanding C3 is not critical or really even necessary for using Python, but it is an interesting topic for those curious about language design. If you want to learn more about it, you can find plenty of information on the web.

The Built-In super() Function

Finally we have enough background to understand super(). So far we've seen super() used to access methods on base classes, for example, in SortedList. add where we used super() to call SimpleList. add before sorting the contents. From this example you might conclude that super() somehow returns the base class of a method's class and that you can then invoke methods on the base class part of an object. This is only partly true. It's hard to sum up what super() does in a single sentence, but here's an attempt. Given a method resolution order and a class C in that MRO, super() gives you an object which resolves methods using only the part of the MRO which comes after C. In other words, super() doesn't work with the base classes of a class, but instead it works with the MRO of the type of the object on which a method was originally invoked. The distinction is subtle, but very important, and as we'll see it resolves the mystery of how SortedIntList works. First, let's look at the details of calling super(). Super() can be called in several ways, but all of them return a so-called super() proxy object. You can call any method on a super() proxy, and it will route the call to the correct method implementation for you if such a method exists. There are two high-level types of super() proxies, bound and unbound. Bound proxies, as the name suggests, are bound to instances or class objects. Unbound proxies are not bound, and in fact don't do any method dispatch themselves. Unbound proxies are primarily an implementation detail for other Python features. Some prominent Python developers consider them a bit of a wart on the language, and really they are beyond the scope of this course, so we won't discuss them in any more detail. Bound proxies, on the other hand, are an important part of mastering Python, so we'll discuss them in detail. From now on when I talk about proxies or super() proxies, understand that I'm talking about bound proxies.

Class-Bound Super Proxies

So, how do we use super() to create proxies? As I mentioned a few seconds ago, bound proxies can be bound to either classes or instances of classes. I will call these class-bound and instance-bound proxies respectively. To create a class-bound proxy you use this form. Here both arguments are class objects. The second class must be a subclass of or the same class as the first argument. When you invoke a method on the proxy, here's what happens. First, Python finds the MRO for derived-class. Second, it then finds base-class in that MRO. Third, it takes everything after base-class in the MRO and finds the first class in that sequence with a method name matching the request. Let's see that in action with our SortedIntList. First, let's see the MRO for SortedIntList. It's SortedIntList, IntList, SortedList, SimpleList, and finally object. So, what do we get if we call super() with the arguments SortedList and SortedIntList? That gives us a proxy bound to the arguments we'd expect. Well, what if you use that proxy to resolve a method, say add? Applying the algorithm described above, Python first finds the MRO for the second argument, which we just printed a few seconds ago. It then finds SortedList in that MRO and takes everything after SortedList giving us an MRO containing just SimpleList and object. It then finds the first class in that MRO with an add method, which of course is SimpleList. Let's see if we're right. Well, there you have it. Super() returned a proxy, which when asked for an add method, returned add from the SimpleList class. Now that we have a method, we should be able to call it right? Aah, right. Our proxy is bound to a class, not an instance, so we can't invoke it. If we used the proxy to look up a static method or class method, however, we could invoke it directly. To illustrate this, let's use a class-bound super proxy to call the \_validate static method on IntList. This is probably not the kind of code you'd write in practice, but it does show how super() works when bound to class objects. Note that Python will raise an exception if the second argument is not a subclass of the first.

Instance-Bound Super Proxies

Instance-bound super proxies work very much like class-bound proxies, but instead of binding to a class object they bind to an instance. To create an instance-bound proxy, call super() like this. Here the first argument must be a class object, and the second argument must be an instance of that class or any class derived from it. The behavior of the super proxy in this case is like this. First, Python finds the MRO for the type of the second argument. Second, Python finds the location of the first argument to super in that MRO. Remember that the instance must be derived from the class, so the class must be in the MRO. Third, Python finally takes everything in the MRO after the class and uses that as the MRO for resolving methods. Importantly, since the proxy is bound to an instance, we can call the methods after they've been found. Let's try that with our SortedIntList example. (Typing) So, the proxy is bound to a SortedIntList and will start method resolution from SortedIntList's MRO at the point after SortedList. As you'll recall, the next entry in SortedIntList's MRO after SortedList is SimpleList, so this super proxy will be directly using SimpleList's methods bypassing our constraint checks. Let's see if that works. Oh dear. Our SortedIntList not only isn't sorted anymore, but it also contains references to classic British TV. Clearly you need to use super() with care or you can end up breaking your designs.

Calling super() Without Arguments

We now know how super() works for creating class and instance-bound proxy objects, and we've seen both in action. The examples we've shown though all pass parameters to super() while the examples we saw earlier in this module, for example in the implementation of SortedIntList, didn't use any parameters at all. It turns our that you can call super() in a method with no arguments, and Python will sort out the arguments for you. If you're in an instance method, that is a method which takes an instance as its first argument and you call super() without arguments, that's the same as calling super() with the method's class as the first argument and self as the second. In the simple case of single inheritance then, this is equivalent to looking for a method on the base class. If you call super() without arguments in a class method, Python sets the arguments for you so that it's equivalent to calling super() with the method's class as the first argument and the method's first argument, that is the class argument, as the second. Again, in the typical case this is equivalent to calling the base class's method. In both cases then, for instance methods and class methods, calling super with no arguments puts the method's class as the first argument to super() and the first argument to the method itself as the second argument to super.

SortedIntList Explained

So, let's do a quick review of what super() does. Given a class and an MRO that contains that class, super() takes everything in the MRO after the class and uses that as a new MRO for resolving methods. This is all bundled up into proxy objects, which are returned from the super() call. Given that, let's see if we can resolve the apparent mystery of how SortedIntList works. Remember that SortedList and IntList were developed without referencing one another, yet when they're combined in a subclass both of their constraints are still properly maintained. The key to how this works is that both SortedList and IntList use super() to defer to their base class. But as we now know, super() doesn't just let us access base classes, but rather it lets us access the complete method resolution order for a class. So, when SortedList and IntList are both used as bases for SortedIntList, the MRO for SortedIntList contains both of them. A call to add on a SortedIntList resolves to a call to IntList. add, which itself calls super(). The super() call in IntList. add uses the full MRO for SortedIntList meaning that rather than resolving to SimpleList. add as we initially expected, it actually resolves to SortedList. add. This is how SortedIntList maintains two constraints without having to manually combine them. This is a fairly deep result, and if you understand how SortedIntList works then you have a good grasp on super() and method resolution order in Python. If you're still unsure about these concepts, review this module and experiment on your own until you do.

The Object Class

Finally, let's finish this module by looking at the core of Python's object model, the class called, fittingly enough, object. You saw object earlier in this module when we looked at the MROs for various classes. For example, it shows up in the MRO for IntList just as it does for SortedIntList and indeed things like list and int. The fact is object is the ultimate base class for every class in Python. At the root of every inheritance graph you'll find object, and that's why it shows up in every method resolution order. If you define a class with no base class, you actually get object as the base automatically. You can see this by looking at the dunder-bases member of a simple example class like this. What does object actually do? We won't get into the details of everything it does or how it does it, but it's good to have some idea of the role that object plays. First, let's just see what attributes object has. All of these are dunder functions, so clearly object is doing significant work to tie into Python's implementation details. For example, the dunder-eq, dunder-ge, dunder-gt, dunder-le, dunder-lt, and dunder-ne methods are all hooks into Python's comparison operators. The dunder-str and dunder-repr methods of course tie into the str and repr functions, and object provides the default implementations of those methods. More importantly, object provides the mechanisms for basic attribute lookup and management. Methods like dunder-getattribute, dunder-setattr, and dunder- delattr form the foundation on which Python objects expose attributes to callers. For the most part, you don't need to worry about object or understand how it works, but it is useful to know that it exists and that its part of every class in Python. As you become an expert Python user, you may eventually find that you need to know the details of object, so keep it in the back of your mind as you progress.

Duck Tail: Inheritance for Implementation Sharing

One of the aspects of Python that differentiates it from nominally typed languages like C++ and Java is that for the most part the specific type name of an object does not determine if it can be used in a given context. Rather, Python uses duck typing where an object's fitness for a use is only determined at the time it's actually used, and exceptions are raised when an object doesn't have the necessary attributes to fulfill a request. Functions are defined without specifying type names on their arguments, and you can pass objects of any type to any function. Likewise, you can try to call any method you want on any object, and Python won't complain until runtime. One important result of this dynamic type system has to do with how with inheritance is used in Python versus how it's used in nominally typed languages. With static nominal typing, if you want to pass an object to a function, you need to make sure that the object is of the type expected by the function. As a result, you very often end up creating specific interfaces and base classes for specific uses, and the need to satisfy the type system becomes as significant and sometimes the most significant element in your development process. In Python, since there is no static-type system to satisfy, inheritance isn't generally needed for the purposes of bestowing an object with a particular type. Rather, inheritance in Python is best used as a way to share implementation. That is, inheritance in Python is a convenient way to reuse code much more than it is a way to construct type hierarchies.

Summary

Inheritance, method resolution, and the proper use of super() can be confusing at first, so hopefully this module showed you that there's really a clear, consistent, and ultimately quiet simple system underlying Python's inheritance and object model. Here are the topics we covered. Specify single inheritance by putting a base class in parentheses after defining a class's name. Subclasses have all of the methods of their base class. If a class with a single base class doesn't define an initializer, the base class's initializer will be called automatically on construction. It's often best to explicitly call a base class initializer from a subclass's initializer. Isinstance() takes an object as its first argument and a type as its second. Isinstance() determines if its first argument is an instance of the second argument or any subclass of the second argument. Isinstance() can accept a tuple of types as its second argument in which case it returns true if the first argument is any of those types. Checking for specific types is rare in Python and is sometimes regarded as bad design. Isinstance() determines if the first argument is a direct or indirect subclass of or the same type as the second argument. Multiple inheritance means having more than one direct base class. You declare multiple base classes with a comma-separated list of class names in parentheses after a class's name in a class definition. A class can have as many base classes as you want. Python uses a well-defined method resolution order to resolve methods at runtime. If a multiply-inheriting class defines no initializer, Python will automatically call the initializer of the first base class on construction. Dunder-bases is a tuple of types on a class object, which defines the base classes for the class. Dunder-bases is in the same order as in the class definition. Dunder-bases is populated for both single and multiple inheritance. Method resolution order defines the order in which Python will search an inheritance graph for methods. MRO is short for method resolution order. MRO is stored as a tuple of types in the dunder-mro attribute of a class. The MRO method on type objects returns the contents of dunder-mro as a list. To resolve a method, Python uses the first entry in a class's MRO, which has the requested method. MRO is dependent on base class declaration order. MRO is calculated by Python using the C3 algorithm. MRO honors base class ordering from class definitions. MRO puts subclasses before base classes. The relative order of classes in an MRO is consistent across all classes. It is possible to specify an inconsistent base class ordering in which case Python will raise a TypeError when the class definition is reached. Super() operates by using the elements in an MRO that come after some specified type. Super() returns a proxy object which forwards calls to the correct objects. There are two distinct types of super proxies, bound and unbound. Unbound super proxies are primarily used for implementing other Python features. Bound proxies can be bound to either class objects or instances. Calling super() with a base class and derived class argument returns a proxy bound to a class. Calling super() with a class and an instance of that class returns a proxy bound to an instance. A super proxy takes the MRO of its second argument or the type of its second argument, finds the first argument in that MRO, and uses everything after in the MRO for method resolution. Since class-bound proxies aren't bound to an instance, you can't directly call instance methods that they resolve for you; however, class methods resolved by class-bound proxies can be called directly. Python will raise a TypeError if the second argument is not a subclass or instance of the first argument. Inappropriate use of super() can violate some design constraints. Calling super() with no arguments inside an instance method produces an instance-bound proxy. Calling super() with no arguments inside a class method produces a class-bound proxy. In both cases, the no argument form of super() is the same as calling super with the method's class as the first argument and the method's first argument as the second. Since super() works on MROs and not just a class's base classes, classes can be designed to cooperate without prior knowledge of one another. Object is the ultimate base class for all other classes in Python. If you don't specify a base class for a class, Python automatically uses object as the base. Because object is in every class's inheritance graph, it shows up in every MRO. Object provides hooks for Python's comparison operators. Object provides default dunder-repr and dunder-str implementations. Object implements the core attribute lookup and management functionality in Python. Inheritance in Python is best used as a way to share implementation. Thanks for watching, and we'll see you in the next module.

Implementing Collections

Collection Protocol Overview

Hello. My name is Robert Smallshire. Welcome to the ninth module of the Python: Beyond the Basics course where we'll demonstrate how to go beyond the built-in collection types by creating your own collections. In our Python Fundamentals course we showed how the different built-in collections can be categorized according to which protocols they support. In this module we'll build a new and fully functioning collection type of our own design. This will require us to implement a series of different protocols, each of which represents a different capability of collections. The protocols we'll cover are the Container Protocol, which allows us to test for item membership in a collection; the Sized Protocol, which allows us to determine the number of items in a collection; the Iterable Protocol, which we encountered in an earlier module of this course; here we'll utilize that knowledge directly; the Sequence Protocol, which supports random read access to collections; and the Set Protocol, which supports various set operations. There are many further protocols for different collections, which we won't cover here, but we'll give you a good grounding in how to implement protocols so you can build on the foundations we provide.

Collection Construction

Each of the collection protocols is founded on the idea of implementing a specific set of methods, usually special dunder methods. We'll work through implementing five of these protocols in the course of building a SortedSet collection, which have similar behavior to a regular set, but which stores its items in sorted order and which also supports the sequence protocol for random access. We'll follow a simple test-driven development or TDD approach to developing the collection, although we'll abbreviate a few steps for the sake of making progress quickly enough to complete this work within the time limit of a single course module. We briefly covered unit testing in Python in the last module of our Python Fundamentals course, although for a thorough treatment of the topic we strongly recommend the excellent Pluralsight course Unit Testing with Python by Emily Bache. We'll be starting with a completely fresh PyCharm project in which to build our new collection type. We'll only be using two Python source files though, one for tests, and one for the collection itself, so you can use any Python development environment with which you're comfortable. Our first test suite in test\_sorted\_set. py looks like this. First we import the Standard Library unit test module. This will provide the test framework for our test suites. Then we import the SortedSet class from the sorted\_set module, neither of which we have created yet. Now we declare our first test case inheriting form unittest. TestCase. These tests will check that we're able to construct instances of SortedSet in the way we would expect of any Python collection. At this point we're largely checking that the constructor call to SortedSet doesn't raise any exceptions. We expect to be able to construct with an empty list, a list containing some items, a list containing some duplicate items, and from an arbitrary iterable, in this case we've used the generator. The goal here is to ensure we can successfully construct a sorted set in a way that is typical for Python collections. There's no particular protocol to be implemented here, but it's sensible to ensure that our collections can be constructed from existing iterable sequences of various configurations including being empty. We don't want to rely on the argument being passed to the constructor being anything more sophisticated than an iterable, and using a generator is as good a way as any to test for that. Switching over to the REPL, our first attempt to run these tests falls at the first hurdle with a failure to import SortedSet. Given that the tests in here exercise the initializer and simply check that the initializer exists and can be called with these arguments without causing an exception to be raised, let's go ahead and implement enough of SortedSet in sorted\_set. py to pass these basic tests. To get things going, we just need a SortedSet class, which we can instantiate with existing collections. We need to be able to create our SortedSet from sequences and iterables and have it ignore duplicates. We've skipped a few TDD cycles here and made a design decision about implementing SortedSet around a regular list object, which we create using the sorted built-in function, which always returns a list, a respective of which type of iterable object it has passed. Now we can successfully run our tests by directly running the test module, although from now on we'll be using the test runner facilities built into PyCharm for convenience. We'd also like to be able to construct an empty sorted set with no arguments. Although that isn't mandated by any of the collection protocols, it will make our set consistent with the existing Python collections, which will eliminate a potential surprise to users of our class. Here is the very simple test. We just check that we can construct a SortedSet no arguments without an exception being raised. Of course we haven't yet handled that case in our class, so when executed in the test a TypeError is raised. Let's modify our implementation of the init method such that our items argument defaults to None. We then use a conditional expression to set the items member to the empty list in the case that items is None. We deliberately use None as the default argument rather than an empty list to avoid inadvertently mutating the default argument object, which is only created once when the method is first defined. With this change in place, all of our tests are now passing again.

The Container Protocol

The first protocol we'll implement is the container protocol. The container protocol is the most fundamental of the collection protocols and simply allows us to determine whether a particular item is present in the collection. Client code exercises objects supporting the container protocol by using the in and not in infix operators with which you should already be familiar. Let's put some tests together for that. We'll implement a new test case called TestContainerProtocol. We'll override the setUp function from the test framework to create a SortedSet containing 6, 7, 3, and 9. We'll then write four test cases to check for positive and negative results for the in and not in operators. (Typing) Of course running the test results in four failures. The error message are worthy of inspection because they indicate that we could fix the problem by making SortedSet iterable rather than by making it support the container protocol. While this is true, because the machinery behind the in and not in operators contains a fallback to exhaustive search through the iterable series of items, we won't take that approach here, although we will implement the iterable protocol in due course. To get our test passing, we'll implement the more restrictive container protocol, which is implemented by the special method dunder-contains. Dunder-contains accepts a single argument, which is the item to test for and returns a Boolean. Our dunder-contains implementation will just use the membership test on the enclosed list object. Doing this gets our test passing again.

The Sized Protocol

The next degree of sophistication above containment is the sized protocol, which allows us to determine how many items are in a collection by passing it to the len built-in function, which always returns a non- negative integer. Let's put some test together for that. First we define our new test case TestSizedProtocol. Going on the old adage that there are only three interesting numbers in computing science, 0, 1, and N, we test those three cases first using 10 in place of N. (Typing) Our fourth tests ensures that duplicate items passed to the constructor are only counted once, important behavior for a set. On running the test you'll get four failures of the form object of type SortedSet has no len(). To get the test passing, we need to implement the special dunder-len method to which the built-in len() method delegates. Again, we in turn delegate by calling len() on the underlying list. This improves matters, but our fourth sized protocol test is still failing with an AssertionError that three does not equal one. This test covers a combination of initializer behavior and conformance to the sized protocol, but the most important thing is that we have this test somewhere in our test suite. To fix this failure, we need to eliminate duplicate items in the series passed to the constructor. The easiest way to do this is to use them to construct a regular set, the built-in set type, which we can then use in turn to construct the sorted list. This approach has the added advantage that the set constructor is a good model for what we would like our constructor to support, so by delegating to it directly we can be sure that the SortedSet constructor is compatible with the set constructor. With this change in place, all of our tests are passing again. The sequence protocol implies the iterable protocol, so we'll have to implement that before we move on to sequence.

The Iterable Protocol

We saw how to implement the iterable protocol in an earlier course module. We simply need to provide the special dunder-iter method, which must return an iterator over the series of items. We'll write a couple of tests for that, one of which calls the iter built-in and one of which uses a for loop. We start by defining a new test case, TestIterableProtocol, and creating a SortedSet fixture in the setUp method inherited from the test framework. The test\_iter method obtains an iterator using the iter built-in and asserts that each value is the expected one and that the iterator terminates the series normally by raising a StopIteration exception. (Typing) Notice that the call to assertRaises is passed a nullary lambda, that is one with no arguments, which calls next(i). This is so that the assertion can call next(i) rather than our code calling next(i) and passing the result of that to the assertion, a common mistake. A somewhat clunky and frankly quite unpythonic formulation of the test\_for\_loop method is so that we can have the for loop directly iterate over our SortedSet. More elegant solutions, which would use for example the zip built-in to combine the expected and actual values, would be performing a quite different test on our class. Of course, currently both tests fail. Taking the lead from earlier protocol implementations, our simple implementation of dunder-iter will delegate to the underlying list. Most implementations of dunder-iter will either delegate in this way to some underlying collection or be implemented as a generator function. Remember, generator functions return generators, which are iterators, so generator functions fulfill the requirements of the dunder-iter method. Were we to do that in this case, dunder-iter might look something like this where we just use a for loop to iterate over the items in the underlying list yielding each item in turn. You can use whichever form you prefer, although the list iterator is likely to be faster. With 15 tests passing and our container, sized, and iterable protocols in place, we can now proceed to implementing the sequence protocol.

The Sequence Protocol: Indexing

The sequence protocol requires that in addition to being an iterable sized container, the collection supports element access with square brackets, slicing with square brackets, and can produce a reversed iterator when passed to the reversed built-in function. Furthermore, the collection should implement the index and count methods and support concatenation repetition through the addition and multiplication infix operators. Implementing a conforming sequence is a tall order. Let's chip away at it using our test-driven approach, and along the way we'll introduce some tools to reduce the magnitude of the task facing us. First, the indexing and slicing. Both indexing and slicing use postfix square brackets and are supported by the dunder-getitem special method, which we briefly encountered in a previous course module. Our dunder-getitem will provide for regular forward indexing and reverse indexing with negative integers. Including tests for the various edge cases, our new test sequence protocol test case looks like this. We create a test fixture, S, containing 1, 4, 9, 13, and 15. We test that we get the expected result of index\_zero at the beginning, index\_four at the end, and we check that an IndexError is raised one\_beyond\_the\_end. (Typing) Going in the reverse direction, we check that we get the expected result at index\_minus\_one at the end and index\_minus\_five at the beginning. We also test at index\_minus\_six one\_before\_the\_beginning to ensure that another IndexError is raised. In the two beyond\_the\_end tests, notice that we've used the alternative form for assertRaises using it as a context manager to wrap around the code to be tested. As you would expect, all of these tests fail. To get these tests passing, we need to implement the special dunder-getitem method, which is easy enough delegating to the underlying list indexing operator.

The Sequence Protocol: Slicing

Now let's add further tests for slicing to the test sequence protocol test case. We test from slicing from the start with :3, slicing to the end with 3:, we test for an empty test result by starting the slice beyond the end of the collection at index 10, we test an arbitrary slice in the middle of the collection with a start index of 2 and a stop index of 4, and we test the full slice supplying no indexes and just the colon. Notice that writing these tests drives us to a design decision that slicing a SortedSet should return a SortedSet. I think there's a good chance that if we would be doing tests after rather than tests first we'd have ended up with our slices returning lists instead. In fact, when we run the tests you can see that that's exactly what happens. Our tests are failing because the existing implementation, which delegates to list, returns list slices. To remedy this, we'll need a slightly more sophisticated version of dunder-getitem, which detects whether it's being called with an index or a slice and acts accordingly. But what does the index argument of dunder-getitem contain when it is called with a slice syntax? Let's temporarily instrument our method to find out. We'll print the index value and the type of the index value. Now run the tests again. This will produce function output like this representing a slice object containing the start and stop indexes and the type slice. This indicates that a slice object is passed configured with arguments representing the slice. We won't cover slice objects in detail here except to point out that they can be manually constructed by calling the slice constructor, which accepts integer start, stop, and step arguments. In our examples the step attribute for extending slicing is not used, so it defaults to None. It's sufficient for our purposes to detect when a slice object has been passed to dunder-getitem and wrap up the list we get back from delegating to the list slice into a new SortedSet. We do this by calling the list indexing operator as normal and then conditionally wrapping the result in a SortedSet on the basis of a type test on the index argument performed by the built-in isinstance function. Let's run the tests again. Oh dear. The tests are still failing, but now for a different reason. It's difficult to tell exactly what's gone wrong here. The string representations of our SortedSet aren't very informative.

Comprehensible Test Results With \_\_repr\_\_()

We know how to fix that though by implementing the dunder-repr special method, which should return a string helpful to developers like us. We won't abandon TDD though, so first a couple of new tests and a new test case TestReprProtocol. Our two repr tests cover empty and non-empty sorted sets, and we expect the repr to be a string which looks like a valid constructor call. This takes us from five to seven failing tests. Implementing dunder-repr on our SortedSet class will take us back down to five. We'll follow the lead of the built-in collections and render an argument list constructor call if the collection is empty. If there are items in the collection, we'll delegate to the list repr to render the argument. Notice also that we use implicit conversion of the self. \_items list to bool in the conditional. If the collection is empty, this expression is evaluated to false in this Boolean context. Now when we run our tests not only do the two repr tests pass, but we get much more information about what's going wrong with the other tests, or do we? In every case, our actual sets appear to be equal to the expected sets, but in fact they're not. What's going on?

Implementing Equality and Inequality

The problem here is that by default Python equality comparisons, which are inherited from the ultimate base class object, are for reference equality rather than value equality or equivalence. So, when we make a comparison between two SortedSet instances, we're really asking are these two SortedSet instances the same object? We can see this with some simple experiments at the REPL. (Typing) The equality operator is returning exactly the same result as the is operator, which tests equality of identity. In fact, this is the default behavior for all Python objects unless the equality operator has been specialized for the objects in question. The reason we get a different result with list equality is that the list type overrides the default equality implementation, which all classes inherit from the object base class. To get the same behavior for our SortedSet class, we should do the same. Continuing with our test-driven methodology, we first need a few more tests. Here's our test equality protocol test case. The first two tests check positive and negative results from an equality comparison between sorted sets. The third test has mismatched types using a comparison between a SortedSet on the left hand side and a regular list on the right hand side. The fourth test is a sanity check to ensure that a SortedSet is always equal to itself. In these tests we've deliberately used assertTrue and assertFalse rather than the assertEqual and assertNotEqual methods. This leaves the equality operator in plain sight and ensures that the tests are couched in terms of the equality operator only and not the inequality operator. When you run these tests, you'll see that the last three of these already pass, although for the wrong reasons. Let's now implement a specialized equality operator for SortedSet by overriding the dunder-eq special method. Our implementation is simple. It just delegates to the same operator with the enclosed lists of the left hand side operand, in this case self, and the right hand side operand, abbreviated rhs. Now our first equality test passes, but our type\_mismatch test fails. We need to refine our implementation to perform a type check returning the special built-in singleton value not implemented if the types don't match as is conventional in Python. Notice that we return the NotImplemented object rather than raising a NotImplementedError, an important distinction. This is something of a curiosity in the Python implementation, and the runtime will use it to retry the comparison once with the arguments reversed potentially giving a different implementation of eq on another object the chance to respond. With this change in place, all of our tests are passing again. Before returning to continue our implementation of the sequence protocol, we should ensure that value inequality works as expected. Here are the inequality tests. These all pass without further changes to our implementation demonstrating that Python will implement inequality by negating the equality operator. That said, it is possible to override the dunder-ne special method if inequality needs its own implementation, and in fact here we should really deviate from our test-driven approach. Even though our implementation works, the Python documentation is clear that when specializing dunder-eq in a subclass, and remember all classes are subclasses of object, we must also specialize dunder-ne. Our dunder-ne implementation is simply symmetrical with our dunder-eq implementation. After that extended detour into repr, equality, and inequality, it's time to get back to implementing the sequence protocol.

The Sequence Protocol: Reversing

So far our collection is an iterable sized container, which implements dunder-getitem from the sequence protocol, but still lacks support for reverse iterators and the index and count methods. The reversed built-in function should return an iterator, which yields the collection items in reverse order. Here's our test for it, which we append to the test sequence protocol test case. Running the test we can see that we don't need to do any extra work. The test passes. By default, the implementation of reversed will check for the presence of the dunder-reversed special method and delegate to that. However, if dunder-reversed has not been implemented, but both dunder-getitem and dunder-len are supported, which they are in this case, reversed will produce an iterator that internally walks back through the sequence by decrementing an index. That's fine for our purposes, so we won't go to the trouble of overriding dunder-reversed. If we did want to or need to implement dunder-reversed, it would be straightforward to implement it as a generator function.

The Sequence Protocol: index()

Next on our list are index and count. First, the index method, which should return the index of the first matching item in the collection or raise a ValueError. Let's get some tests in place. We have two tests, index\_positive for when the item does exist in the collection, and index\_negative for when it doesn't. (Typing) Both of these tests fail because we haven't yet implemented index itself, and being a regular method there are no opportunities for fallback mechanisms in the Python implementation to kick in. Clearly though, it would be possible in principle to implement index in terms of methods we already have in place such as dunder-getitem. Fortunately, such default implementations are available in the Python Standard Library in the base classes of the collections. abc module where abc is an acronym for abstract base class. This module contains many classes which can be inherited from or mixed in to our own classes taking some of the leg work out of collection protocol implementation. In our case, we want the collections. abc. sequence class, which if we provide implementations of dunder-getitem and dunder-len as we've already done, we'll provide a whole raft of mixing methods including index. We can make our SortedSet a subclass of sequence simply by adding sequence to the list of base classes at the point we define SortedSet. (Typing) Now our index unit tests pass without any further work.

The Sequence Protocol: count()

We'll round off our implementation of the sequence protocol by ensuring that count is also correctly implemented. Recall that count returns the number of times a specified item occurs in a list. Here are our tests. Given that our collection represents a set, we only need to test for zero and one occurrences of the item. (Typing) Thanks to our inheritance from the sequence abstract base class, this also works immediately.

Improving Performance from O(N) to O(log n)

The default implementations of methods like index and count inherited from the base class have no knowledge of the implementation details or characteristics of the data in our concrete class, so it may not necessarily be optimally efficient. For example, the inherited index implementation can't exploit the fact that our list is sorted, and the inherited count implementation can't exploit the fact that our collection never contains more than one item with a particular value, so we'll always inspect every element even when that's unnecessary. There's nothing to stop us overriding the inherited implementations with more efficient versions. Let's look more closely at the performance of the count method we have inherited by performing some simple experiments at the REPL. First we'll create a SortedSet from 2000 randomly selected integers in the range 0-999. We'll do this using the randrange function from the standard library random module in conjunction with a simple generator expression. We use the generator expression to evaluate random. randrange 2000 times. In our case, this resulted in the following SortedSet containing 865 elements, although owing to the random nature of how we've produced it you might get slightly different results. As you can see, some of the values such as 952, 974, and 989 are missing. Now let's use the count method to count the number of occurrences of each number in the range 0-999 inclusive. We can do this with a simple list comprehension. This gives us an overview of which numbers are missing from the SortedSet. That all seemed quick enough, but now let's evaluate that list comprehension 100 times. We'll use the timeit function from the timeit standard library module to measure how long it takes. We introduced the timeit module back in the second module of this course. In the setup statement we import S from the dunder-main module, which represents the contents of the current REPL session. Our statement is the list comprehension which calls count we just introduced. The third argument number instructs timeit to execute the statement 100 times. That takes a little over 5 seconds. We know that the count operation on a SortedSet only ever returns 0 or 1. We also know that the list inside our set is always sorted, so we should be able to perform a binary search for the element in a time proportional to log n rather than N where N is the number of elements in the set and log n will always be much smaller than N. Fortunately, we don't need to write the binary search for ourselves because there's a robust implementation to Python's Standard Library in the bisect module. Here's our new override of the count implementation. This uses the bisect\_left function from the bisect module to search for an item returning the index at which the requested item should be placed in the sequence. Then we must perform two further tests. The first checks whether the returned index is within the bounds of the collection; the second tests whether there is already the required item at that index. Finally, we covert the bool to an int, which results in 0 and 1 for true and false respectively. With this change in place, we can check that our tests are still passing, which they are. Rerunning a comparable performance test, this time the set is slightly smaller, although not significantly so, we see that the runtime is about 25 times faster with this implementation on this data set, although progressively larger improvements could be expected with larger collections.

Refactoring to Avoid Don't Repeat Yourself (DRY)

It may have occurred to you that the found variable in our new implementation of count represents exactly the same results as would be obtained from the dunder-contains method, which we wrote when implementing the container protocol. Let's refactor by extracting our efficient search implementation into dunder- contains and using the membership test within the count method. The call to bisect and the other tests are moved into dunder-contains like this. This replaces the order N complexity list membership test with a much superior order log n test. The count method is now reduced to simply converting the result from the membership test to an integer. Oh, and all the tests are still passing. The inherited index implementation is also not very efficient. Knowing nothing about the nature of SortedSet, it has no choice but to perform a relatively inefficient linear search. Let's fix that too, again using the bisect module. We already have the tests in place, so we can proceed directly to overriding the method. This code is very similar to what's now contained in the dunder-contains method, and if you're offended by the repetition between index and what we already have there, you could implement dunder-contains in terms of index like this using a try/except block to call index and covert the raised value error into a false return value. Although this works and avoids the duplication, it does strike us as perhaps a little obtuse.

Checking Protocol Implementations

We've overridden all of the methods inherited from collections. abc. sequence except for dunder-reversed, and we don't need to override that because there's already a fallback in the reversed implementation to a dunder-getitem and dunder-len based algorithm anyway. So, is there still any value in inheriting from sequence? Not that it's exercised by our tests, you can experiment by removing the inheritance relationship and noting that all of our tests still pass. Nevertheless, one of the advantages of explicitly implementing the classes in the collections. abc module is that it becomes possible in a simple way to determine whether a class implements a particular collection protocol. Here are some example protocol checks using the built-in issubclass function on other built-in collection types such as list and dictionary. One obvious question, to paraphrase the words the words of PEP 3119 where the collection abstract base classes are laid out, is whether this approach represents a retreat from duck typing. The PEP states that it's not required, only encouraged to inherit from the appropriate abc's when implementing collections. With that in mind, let's add some tests to cover the container, sized, iterable, and sequence abc's checking that we inherit from them appropriately. In test container protocol we check for inheritance from Container, in test size protocol we check for inheritance from Sized, in test iterable protocol we check for inheritance from Iterable, and in test sequence protocol we check for inheritance from Sequence. Remembering that at the current time we're not inheriting from any of these classes, you might be a little surprised, perhaps even very surprised when only the last of these tests fail. We've encountered a powerful and yet little known feature of the issubclass function in relation to the abstract base class system, which allows it to take advantage of duck typing without explicit inheritance. A full explanation is an advanced Python topic well beyond the scope of this course. By reintroducing inheritance from Sequence, which inherits all of the other relevant protocols, we can get all of our tests passing again including these new ones.

The Sequence Protocol: Concatenation and Repetition

Only a few operations remain for complete conformance with the sequence protocol, although neither are enforced or supplied by the abstract base class, concatenation and repetition with the add and multiply operators respectively. It's not entirely obvious that SortedSet should support concatenation in the usual sense since doing so the normal way would not be possible while maintaining the uniqueness and ordering invariance. We'll sidestep this issue for now by implementing concatenation as a set union operator, which results in a set containing all elements from both operands. We test concatenation of disjoint sets, equal sets, and overlapping or intersecting sets. To get these tests to pass, we need to implement support for the infix plus operator, which is done via the special method dunder-add. Rather than just simply concatenating the enclosed lists of the two operands and using that to construct a new SortedSet, which could result in large, temporary, intermediate structures, we use the itertools. chain function, which requires an additional import at the top of the module. Itertools chain allows us to stream all the values from one operand and then the other into the SortedSet constructor. This simple and elegant implementation works as expected. Finally, we should implement repetition, which for lists works like repeated concatenation to the original list. This will have no effect for SortedSet, so all we need to do is return a new object, which is equivalent to the existing one unless the multiplicand is less than one in which case we should return an empty collection. Here are the unit tests covering the case for zero repetitions and the nonzero number of repetitions. To get these tests to pass, we need to implement the infix multiplication operator, which delegates to the dunder-mul special method for cases where our class is on the left hand side of the operator. In the implementation we return self or an empty set depending on the value of the right hand side. Note that the only reason we can return simply self is because our SortedSet objects are immutable. If or when they are made mutable, it would be necessary to return a copy of the self object here. This could be achieved either by passing self to the SortedSet constructor or perhaps by implementing a more efficient copy method. Note that the dunder-mul method is not invoked if we reverse the operands as we have in these two tests, test\_repetition\_zero\_left and test\_repetition\_nonzero\_left. For cases where our class is the operand on the right hand side, the rmul operator must be implemented as well for reversed multiplication. Since the order of the operands doesn't matter for repetition, dunder-rmul simply delegates the dunder-mul. At this point we're doing very well. We have a SortedSet implementation that implements the container, sized, iterable, and sequence protocols very comprehensively, robustly, and efficiently in around 50 lines of code.

The Set Protocol

Since we're implementing a set which maintains its elements in sorted order, it seems entirely reasonable that we should support the set protocol, and that's what we'll do now. Referring to the collections. abc documentation, we can see there is an abstract base class called Set with abstract methods dunder-contains, dunder-iter, and dunder-len. This brings us a bevy of special methods, which implement the various set operators including all the relational operators. These will allow the comparison of SortedSet objects in terms of subset and superset relationships. The abstract base class only provides the special methods for the infix operator support. If we want the named method equivalents just like the built-in set type, we're going to need to implement them ourselves, although of course we can define them in terms of the operators. One important difference between the operator and method versions of these operations is that while the operators require that the operands are both of the same type, method versions will accept any iterable series as an argument. The fact that we've already overridden dunder-eq is probably a good thing. Our version is likely to be more efficient than the default implementation inherited from the base class. Let's create unit tests for these infix relational operators and then named method equivalents. We won't review each in detail now because there are quite a few, which all are included with the example source code associated with this course. Predictably, all 16 of these new tests fail. We won't fix them just yet though. We'll plow onwards through the other necessary set operators. The set base class also provides mix-ins for the special methods, which normally correspond to the bitwise operators. We covered the bitwise and and bitwise or operators in the context of processing binary data files in the Files and Resource Management module of our Python Fundamentals course. In the context of the set protocol, however, they're implemented to perform set intersection and set union operations respectively. Furthermore, the infix exclusive or operator is configured to produce the symmetric difference of two sets. We covered the long hand methods for the set algebra operations intersection(), union(), and symmetric\_difference() in the collection section of the Python Fundamentals course, although we did not cover these infix operator alternatives. The set protocol also defines the subtraction operator to perform the set\_difference() operation, and the set class provides a corresponding difference() method. As with the relational operators, the set abstract base class provides the special methods which underlie the infix operators, but does not implement the equivalent pubic methods used by the set built-in class. If we want those, we're going to have to implement them ourselves. As before, the operator versions expect operands of the same type whereas the named methods accept any iterable series as an argument. Here are the tests for the operators and for their corresponding named methods. One last mix-in method contributed by the set abstract base class is the isdisjoint() method, which tests whether two sets have any elements in common. We'll round off our test for the set protocol by adding two new tests for that to the test set relational method's test case. At this point we have a total of 78 unit tests of which 26 are failing. Let's make some code changes to get them passing. The obvious first step is to have SortedSet inherit from the set abstract base class. We'll use multiple inheritance for this so it can still inherit from sequence. This requires an additional import from collections. abc. That simple change gets 16 out of 26 failing tests to pass, so now we're down to the 10 tests which cover the named methods not inherited from the set abstract base class. Let's implement those in terms of the inherited operators remembering to support any iterable series as the argument, which we'll achieve by constructing a SortedSet from each iterable series before applying the operator version. When implementing these methods, we always pass self as the left hand side operand and an iterable used to construct a SortedSet as the right hand operand. (Typing) Our SortedSet class is now functionally complete. We'll round off by asserting that we've implemented the set protocol. (Typing) You'll notice that we've constructed an immutable set. Once constructed, its values cannot be modified. For many applications this is sufficient and indeed a good thing. Immutable objects are generally easier to reason about.

Duck Tail: Making a Mutable Set

If you need a mutable SortedSet, it would be straightforward to inherit from the MutableSet abstract base class instead of the Set abstract base class and implement the two additional required abstract methods we don't already have, add() and discard(). You may also want to follow the lead of the built-in Set class and implement named method such as update() and symmetric\_difference\_update() for completeness. This would allow SortedSet to be used as a drop-in replacement for Set in nearly all cases. Finally, you might want to follow the lead of the other collections and provide a copy method. Be aware that some of the methods we've implemented rely on assumptions which don't hold for mutable sets. For example, under certain conditions dunder-mul returns self. For a mutable set, we'd want to return a copy of self instead. We'll leave making a mutable version of SortedSet as the proverbial exercise for the viewer, which will also provide valuable experience in the use of the bisect module.

Summary

In this course module we covered how to implement your own collection classes, which conform to standard Python protocols. In particular, we reviewed the collection protocols. We showed how to implement the container protocol, which supports the in and not in membership test operators by implementing the dunder-contains special method. We also looked at how to implement the sized protocol, which allows the len built-in function to be used by implementing the len special method. We revisited the iterable protocol and its implementation using the dunder-iter special method, and we introduced the sequence protocol, which requires that a wide variety of operators, methods, and behaviors be supported. As part of that, we showed how to implement the dunder-getitem special method to support both indexing and slicing, and because of which we briefly touched on slice objects. We implemented dunder-repr to provide better diagnostics from our unit test failures, and we discovered that we needed to implement value equality via the dunder-eq special method. We showed that although value inequality seems to work automatically in terms of negated equality, the Python documentation strongly recommends that we explicitly implement dunder-ne. We understood that although obtaining a reverse iterator from a collection via the reversed built-in is a requirement for the sequence protocol, it wasn't actually necessary to implement the dunder-reverse special method because the built-in function falls back to using dunder-getitem and dunder-len. We introduced the collections. abc module, which contains base classes to define and assist with implementing the various collection protocols, and from that we used the collections. abc. sequence abstract base class to provide functioning implementations of many of the required methods and operators. We showed how to test for protocol implementation using the issubclass function in combination with the abstract base classes of the collections. abc module. And finally, we implemented the set protocol using the same techniques. In passing, we've shown how that test-driven development or TDD can lead us to discovering some important requirements we may have otherwise forgotten, and we worked to improve the algorithmic complexity of our implementations from order N to order log n by using the standard library bisect module to perform binary searches. We also introduced you to the bitwise operators being used for set algebra operations when operating on sets. You should now have all the conceptual tools you need to implement your own collection types. Thanks for watching, and we'll see you in the next module.

Exceptions and Errors

Always Specify an Exception Type

Hello. My name is Robert Smallshire. Welcome to the tenth module of Python: Beyond the Basics. In our predecessor course, Python Fundamentals, we introduced exceptions and exception techniques in a module called Handling Exceptions where we covered the basics of exceptions and the specifics of how to handle them in Python. We also gave advice on when and how to deploy exception raising and exception handling in your code. This course module builds directly on the foundation we established in Python Fundamentals, and in this module we seek to deepen our understanding of some of the tools Python provides for sophisticated error handling and reporting. We'll start by reminding ourselves of the basic exception handling and raising constructs and highlighting a practice you should avoid. Consider this simple program, which uses the randrange function from the random module to choose a number between 0 and 99 inclusive. We then ask the user to guess the number breaking out of the loop if they get the answer right. Let's give it a whirl. Python3 handler. py. We'll try 10, no luck; 37, 53, 22. This is quite a boring game. Let's press Control+C to exit. Let's have another go and see if we can cause it to break. 45, 21, 99, now seven as a word. This time the program fails with invalid input. In the program we used the int constructor to convert the string returned by the input function to an integer, and when we used the word seven rather than the digit 7 that conversion raises an exception, which is unhandled and so the program exits. Let's fix our program by incorporating an exception handler around the problem statement. We do this simply by wrapping the statement in try/except. In the exception handler we use a continue statement to proceed immediately with the next iteration of the innermost loop, the while loop in this case. Let's try it. 10, 17, seven as a word again, 9. When you're bored, press Control+C to exit the program. Oh dear! By not specifying an exception class we've handled all exceptions including the KeyboardInterrupt exception, which is raised when we press Control+C. Catching all exceptions is in general a very bad idea. Practically everything that can go wrong in a Python program manifests as an exception. Let's make another change. We'll replace our call to the int constructor with a call to a function foo. When run, this program will go into an infinite loop of repeatedly raising and handling the name error that is raised by the unknown foo function. Of course the solution here is to catch the specific exception we're interested in, which in this case is a ValueError. With that change in place, the name error caused by foo propagates out and terminates the program with a stack trace. Let's revert to using the int function, and our program now works completely as intended. In summary, you should almost always avoid admitting the exception class from an except statement since handling all exceptions is seldom required and usually a mistake.

The Standard Exception Hierarchy

The built-in exception classes, of which there are many, are arranged into a class hierarchy using inheritance. This is significant because when you specify an exception class in an except statement, any class which is a subclass of the specified class will be caught in addition to the specified class itself. Let's look at two built-in exception types with which we're already familiar, IndexError and KeyError. An IndexError is raised whenever we attempt an out of range lookup into a sequence type such as a list. A KeyError is raised when we look up a missing key in a mapping type such as a dictionary. Let's investigate the inheritance hierarchy of these two exception types. We know we can retrieve the transitive base classes from a class object using the MRO method, which returns the method resolution order of a class as a list. This shows the full exception class hierarchy from object, the root of all class hierarchies, down through BaseException, the root of all exceptions, a class called Exception, which we'll return to shortly, to LookupError, and finally IndexError. Now we'll try the same exercise for KeyError. We can see that KeyError is also an immediate subclass of LookupError, and so IndexError and KeyError must be siblings in the class hierarchy. What this means in practice is that we can catch both IndexError and KeyError exceptions by catching LookupError. Here's a short program which raises and handles one IndexError and one KeyError. It behaves as expected when run printing Handled IndexError and Handled KeyError. Now we'll modify the program to handle LookupErrors instead of the more specific exception types. When run, the program behaves identically because IndexError and KeyError are subclasses of LookupError. Let's take a look at the full hierarchy for the built-in exceptions, which we can find within the Python documentation. We've met many of these types already including KeyboardInterrupt, StopIteration, ZeroDivisionError, IndexError, KeyError, and of course the various types of SyntaxError when we've got something wrong. We must point out that over the history of Python there have been several changes to this exception hierarchy, so it's worthwhile checking the details for the exact interpreter versions you'll be using if your code needs to be portable. The hierarchy we're showing here is for Python 3. 3. You'll see that BaseException is at the root of the hierarchy. The only exceptions which derive from BaseException, other than the Exception class we'll come to shortly, are the so-called system exiting exceptions, most notably SystemExit and KeyboardInterrupt. We've already witnessed the untoward effects of intercepting and swallowing KeyboardInterrupt exceptions. Likewise, inadvertent handling of SystemExit, which is raised by the sys. exit function when a process is programatically terminated, causes similar problems. The majority of exceptions derive from Exception, so if you want to catch all exceptions except the system exiting exceptions you might be tempted to catch this. Note, however, that a whole host of exception types typically associated with programming mistakes such as SyntaxError, IndentationError, TabError, NameError, UnboundLocalError, AssertionError, and ImportError are also subclasses of Exception, so handling Exception has the potential to hide serious problems. In general, we encourage you to handle as specific exceptions as possible, although OSError in particular is useful for detecting that something has gone wrong with a file system operation without needing to worry about the details of whether it's a missing file indicated by a FileNotFoundError or a permissions problem indicated by PermissionError. And even though we may catch a general type of error, the exception object we receive retains its original type and any exception payload, so we're not passing up on any opportunities for detailed error reporting based on exception payloads.

Exception Payloads

Most exception objects carry a simple payload, which contains diagnostic information about what caused the exception. The majority of the built-in exception types accept a simple string in the constructor call. The exception type you will raise most frequently is probably ValueError, which is often used for argument validation guard clauses near the beginning of functions. Consider this function for determining the median value of an iterable series. The function accepts an iterable and then uses the sorted built-in to sort the items, computes the central index of the sequence, and depending on whether the sequence contains an odd or even number of items either returns the central item or the arithmetic mean of the two middle items. Let's try this on a few sequences. (Typing) It seems to be working just fine. So far, so good. But look what happens when we supply an empty list. We get an IndexError, which contains a message payload displayed in the stack trace, list index out of range. This is all very well since we can't define the concept of median for an empty series, but we're leaking an implementation detail of our function here, namely that internally we're using the sequence lookup to perform the computation. Let's add a guard clause which checks that the supplied series is non-empty. In the guard clause we'll raise ValueError with a more informative and relevant error message. (Typing) Now we get a much more useful error message in our stack trace. Most usually exception payloads are strings and are passed as a single argument to the exception constructor. The string should contain as helpful a message as possible. We can programatically retrieve the message using the args exception attribute. Here we add a function to exercise our median function with faulty input, catch the ValueError, and print the payload stored in its args attribute. When run, notice that args is a single element tuple containing the message that was passed to the constructor. Another way to retrieve the payload and string form is to convert the exception object to a string using the str or repr built-in functions. Although you might infer from this that multiple arguments can be passed to the exception constructor and that these will be available in the args tuple and you would be right, you should only pass a single string argument to exception constructors. PEP 352 is quite clear on the matter. No restriction is placed upon what may be passed in for args for backwards- compatibility reasons. In practice, though, only a single string argument should be used. This means that you should only expect the args attribute to contain a single string value, which in any case you could retrieve by converting the exception to a string rather than retrieving args 0. That said, specific exception classes may provide additional specific named attributes which contain further information about the cause. UnicodeError is one such example, which has five additional named attributes: Encoding, reason, object, start, and end. A rich collection of exception attributes like this can be a huge boon to debugging.

Defining New Exceptions

When your needs aren't adequately met by any of the built-in exceptions, you can define your own. Consider this function which uses Heron's formula to compute the area of a triangle given the length of three sides. This works well for side lengths that represent legitimate triangles such as the 3, 4, 5 triangle. But if no triangle were the supplied side lengths exist, we get a ValueError from an attempt to find a real square root of a negative number. Rather than the obscure math domain error message here, we prefer to raise a more specific exception here, which can carry more useful information in its payload. A good start is to define our own exception class TriangleError. When doing this, you should subclass Exception rather than BaseException. If you just want a distinct exception type with basic facilities which could be raised and handled separately from other exception types, even the most basic definition can suffice. Here the body of the TriangleError class definition contains a simple pass statement. It's empty. All the functionality we want is inherited from Exception. This is a fully functioning exception. It inherits complete implementations of dunder-init, dunder-str, and dunder-repr. Let's modify our function to identify illegal triangles. We do this by sorting the three sides we're given and check that the length of the longest side isn't greater than the length of the two shortest sides added together. In this case we raise a TriangleError with the message illegal triangle. This works as expected. There is no triangle with side lengths of 3, 4, and 10. Now let's modify our exception type to accept more data about the putative triangle. Our exception now overrides dunder-init and provides a constructor, which accepts a message and a collection of side lengths. The message is forwarded to the base class constructor for storage, and the side lengths are stored in an instance attribute in the derived class. We store the side lengths in a tuple to prevent modification and provide a read-only attribute to access them. We also override the dunder-str and dunder-repr methods using the args attribute from the base class to retrieve our message string. Of course we must also remember to modify the constructor call for the exception to pass the sides argument. Now when we feed an illegal triangle into the function, not only do we get a better error report including the side length information, but with an appropriate handler in place we get programatic access to the side lengths, which caused the problem.

Chaining Exceptions

Exception chaining allows us to associate one exception with another, and there are two main use cases. The first case is when during processing of one exception another exception occurs usually in a way incidental to the first exception. The second case is when we wish to deliberately handle an exception by translating it into a different exception type. In both cases there are good reasons for wanting to keep a reference to the original exception. It can avoid unnecessary duplication of information, and it can improve diagnostic messages. Let's look at each of these two cases in turn. The first case is called implicit chaining and occurs when one exception occurs while another is being processed. The Python runtime machinery associates the original exception with the new exception by setting the special dunder-context attribute of the most recent exception. Allow us to demonstrate by modifying our triangle area program by adding a main function, which contains two bugs. The first bug is that we tried to evaluate the area of a non-triangle with sides 3, 4, and 10. The second bug is that in the process of handling the resulting triangle exception we cause an io. unsupported operation experience by trying to print to the standard in stream instead of the standard error stream as we intended. Here's the stack trace we get when we run the program. See how although the TriangleError was handled by our except block it is still reported in the trace with a message during handling of the above exception, another exception occurred. Python is able to give us such a detailed report because the TriangleError has been attached to the dunder- context attribute of the unsupported operation exception object. We'll temporarily add some code to demonstrate this by capturing the UnsupportedOperation as F and checking whether f. \_\_context\_\_ is in fact the TriangleError object. When we run the program, we see that this is indeed the case. The converse of implicit chaining is explicit chaining. This is when we deliberately associate an existing exception instance with a new exception object at the point at which the latter is raised. This is usually done in the process of translating an exception from one type to another. Consider the following simple module. The inclination function returns the slope in degrees given the horizontal and vertical components of distance. This works fine for most slopes, but it fails with ZeroDivisionError when the horizontal component dx is 0. Now let's modify our code by introducing a new exception type, InclinationError, and an exception handler for the ZeroDivisionError that swallows the active exception and raises a new one in essence translating one exception to another. We've included the syntax for explicit exception chaining here with the from e suffix when we create the exception. This associates the new exception object with the original exception E. However, unlike the implicit chaining, which associates the chained experience through the dunder-context attribute, explicit chaining associates the chained exception through the dunder-cause attribute. Let's see what happens when we trigger the exception. With an outer exception handler in place to capture the exception object, we can programatically inspect the dunder-cause attribute to retrieve the exception which represents the underlying cause. Whenever you translate exceptions in this way, typically at module boundaries where you should be hiding implementation details by raising exceptions your clients can reasonably expect, consider whether to explicitly chain root cause exceptions to improve diagnostics and aid debugging.

Traceback Objects

We've mentioned many times that everything in Python is an object, and this even extends to tracebacks, those records of the function call stack which are printed by the interpreter when an exception is unhandled and the program exits. In Python 3 each exception has a dunder-traceback special attribute, which contains a reference to the traceback object associated with that exception. Let's add a main function to our chaining. py example to play with a traceback object. In the handler for the InclinationError we print e. \_\_traceback\_\_ to display the traceback object. Shortly it will become apparent why we've also decided to explicitly print Finished before the program exits normally. To do anything useful with the traceback object, we should use the Python Standard Library traceback module, which contains functions for interrogating traceback objects. To display a traceback, we can use the print\_tb function. See how the program continues running after we've printed the traceback. The exception is being handled, and the program has exited normally. The ability to get hold of traceback objects in this way is invaluable for logging diagnostic output. If you need to render the traceback object into a string, rather than printing it directly, you can use the format\_tb function instead of print\_tb. One word of caution here about keeping references to the traceback object. You should always render the output you need from a traceback object within the dynamic scope of the except block. That is, you shouldn't store a traceback or indeed exception object for later use. This is because the traceback object contains references to all the stack frame objects which comprise the call stack, and each stack frame contains references to all of its local variables. As such, the size of the transitive closure of objects reachable from the traceback object can be very large, and if you maintain that reference these objects will not be garbage collected. Prefer to render tracebacks into another form for even short-term storage in memory.

Assertions: Internal Invariants

The Python language includes an assert statement, the purpose of which is to help you prevent bugs creeping into your code, and when they do to help you find them more quickly. The form of the assert statement is assert condition with an optional message after a comma where condition is a Boolean expression and message is an optional string for an error message. If the condition is false, an AssertionError exception is raised causing the program to terminate. If the message is supplied, it is used as the exception payload. Here's an example. The purpose of the assertion statement is to give you a convenient means for monitoring program invariants, which are conditions which should always be true for your program. If for some reason an assertion fails, it will always point to a programming error. Either some other part of the program is wrong or at the very least the assertion statement itself isn't correct. If the assertion condition is true, the statement has no effect. Assertions are best used to document any assumptions your code makes such as a name being bound to an object rather than none or a list being sorted at a particular point in the program. There are many good and some very bad places to use assertions in your programs. Let's look at using assertions for internal invariance. Often you'll see comments in code which document an assumption particularly in conjunction with else blocks like this where we assume that in the else block R must be equal to 2. Comments such as this are much better reformulated as assertions, which can check for truth at runtime. The assertion has just as much documentary value as the comment it replaces, but will be checked. It may seem like you're paying the cost of an unneeded comparison here, but in practice we find the overhead of most assertions is small compared to the huge benefits they bring in helping us build correct programs. The benefits of such assertions become particularly apparent when people use clone and modifier programming when new code is based on existing code that has been adjusted correctly or not to suit a new purpose. Here somebody has cloned and modified our modulus\_3 function into a new function, modulus\_4. Can you see the mistake? (Typing) For some inputs the assertion is violated allowing us to identify the problem and correct the program. (Typing) An alternative formulation of this construct might be like this where all legitimate conditions are handled in the if and elif blocks and the else clause contains only an assertion statement, which always fails because we pass a constant false to it. In fact, this form would not only be perfectly acceptable, it may even be preferable because the symmetry of the other cases makes it easier to spot blunders. Note that the assertion should not be used to validate arguments to the function, only to detect if the implementation of the function is incorrect.

Assertions: Class Invariants

Now let's look at class invariance. Recall the SortedSet implementation we developed in the previous course module, which used a sorted list of items as the internal representation. All methods in that class assume that the list is indeed sorted and remains that way. It would be wise to encode this class invariant as assertions at the beginning and end of every method, especially if you perform the exercise of making a mutable version of this collection type as we suggested. For example, the index method of the class assumes that the items are already sorted because it uses a binary search, and the count method depends on there being no duplicates because it performs a simple membership test. We can assert on these assumptions being true with a helper method is\_unique\_and\_sorted. (Typing) This works by checking that each item is strictly less than its successor item for all items in the collection. We can then assert that is\_unique\_and\_sorted is a precondition to every method implementation.

Assertions: Performance

The precondition assertions we performed to test our SortedSet assumptions are relatively expensive. In the case that the class is working as expected, each test walks through the whole collection. Worse from a performance standpoint is that sometimes the test is performed more than once. See that count is implemented in terms of dunder-contains, and dunder-contains is in turn implemented in terms of index. So, when calling count, the assertion which affirms our assumption is made multiple times. This is not necessarily a bad thing, but it can be detrimental to performance. Beyond trying to keep assertions both effective and cheap to evaluate, there is the possibility to run Python programs with all assertions disabled by using the -O flag on the command line. Now we're running with active assertions without the -O option. The best run takes a little over 3 seconds. Now again without active assertions and with the -O option, ready, steady, go. Now the best time is 3. 26 ms. We encourage you to use this option only if performance concerns demand it. Running with assertions enabled in production can be a fabulous way of flushing out problems in your code. The majority of Python code is run with assertions enabled for this reason.

Duck Tail: Preconditions and Postconditions

Assertions can be used to enforce function post conditions, that is to assure ourselves that a function is returning what we think it is returning. Consider this function, which we've placed in a module wrapper. py, which is used to wrap strings of text at a specified line length. This function, I'm sure you'll agree, is fairly complex. It uses lots of mutable state, and in the words of Sir Tony Hoare this is not code in which we could claim there are obviously no deficiencies, although it may be true that there are no obvious deficiencies. Let's define some text with which to test it. We'll import our wrap function and pass the wealth\_of\_nations to it wrapping at a line length of 25 characters. Well, at least it appears to have returned a single string with embedded line endings. Let's print that string to get the results rendered in a more helpful way. Everything appears to work, but without counting the length of each line it's difficult to be sure, and in any case it's hard not to get distracted by Adam Smith's spelling of conveniences. Let's make sure this is working by adding an assertion prior to the return of the final result. Our assertion takes the string we're about to return, splits it back up into lines using the splitlines string method, and checks that the length of each line is less than or equal to the specified line length. We use the built-in all function to check that this is true for all lines. Neat! Let's reload the code and try again. Ouch! Our assertion failed, so the function isn't working as expected. The problem is that each time we add a word we account for the length of the word in current\_line\_length, but we don't account for the space that will follow it when we join all the words back together using the join expression later in the program. The fix is to account for that length when we increase the value of current\_line\_length. The simplest approach is just to do this, but that introduces a so-called magic number into our code. Why is that 1 there? Only a careful reading of the code by future maintainers will reveal why; better we think to be explicit about where the 1 comes from. It's the length of the space following the word. Rerunning with the fix in place, we can at least be sure that our function now meets its most basic requirement. Our wrap function isn't fully robust yet though. What happens when we pass a negative line length such as -25? In this case we also get an AssertionError from the same assertion because of course the algorithm has no way to build lines of negative length. In a way, this is good. We've found the problem. In other ways, this is bad because the insertion has intended to check that the lines are too long, not that the line length is too short. But there's a deeper conceptual problem here which is that assertions are intended to detect our mistakes as implementers of this function, not the clients mistakes as callers of the function. The client is clearly at fault here for passing a negative number to our function, and we should tell them so. If they were to see this assertion failure, they would assume with no little justification that they'd uncovered a defect in our code rather than a problem with their code. For this reason, it's inappropriate to use assertions to validate arguments provided by code beyond our immediate control. In other words, don't be tempted to use assertions as validation guards like this where we assert that line\_length is greater than zero. Instead, prefer to raise a specific exception and document it appropriately. Here we raise a ValueError with the message that the line\_length is not positive including the specific line\_length value that caused the problem. (Typing) Now we get a reasonable exception that is predictable by any client who has read the documentation for our function. Much better. There's still an interesting case our code doesn't handle though. What should we do if the text contains a word longer than our line length? The next train to Llanfiarpwllgwyngyllgogerychwyrndrobwllllantisiliogogogoch is at 16:32. First we have to decide what is reasonable in this case. There are a few options. Either we weaken our requirement and produce overly long lines or we reject text containing words longer than the line length or we split a single word over multiple lines. In our case we'll take the easy way out of rejecting overly long text by raising an exception. We do this by mapping the len function over all the words and checking the maximum value. (Typing) It would be good practice to polish your Python skills by trying to modify this function to use one of the other strategies for overly long words that we mentioned, although we recommend avoiding getting bogged down in the hyphenation rules for Welsh.

Summary

In this course module we've demonstrated the dangers of handling all exceptions, especially the so-called system exit exceptions such as KeyboardInterrupt. We showed how these exceptions fit into the standard built-in exception hierarchy and explained how catching base exceptions can be used to catch multiple related exception types. We informed you that you should almost never catch the BaseException or Exception types since they have subclasses which are almost always programming errors since as IndentationError. We investigated how exception payloads work and how to use them effectively, and we showed how to define your own exception classes by inheriting from Exception. At its most basic, an exception subclass need only contain a single pass statement. Sometimes though you'll want richer exception types, which you can implement by accepting additional arguments and providing additional attributes on your custom exceptions. We illustrated how to use exception chaining for both implicitly chained exceptions, which set the dunder-context attribute on successor exception objects, and explicitly chained expressions, which set the dunder-cause attribute on successor exception objects. We showed how to extract traceback objects from the dunder-traceback attribute of exceptions, and we counselled you to render tracebacks to strings rather than keeping references to them to avoid space leak problems with large object graphs reachable from the traceback instance. We finished off by looking at assertions including not only how to use them, but when to use them. With these techniques, you should be able to create robust programs and modules, which are both easy to use, easy to diagnose, and easy to debug. Thanks for watching, and we'll see you in the next module.

Defining Context Managers

What Is a Context Manager?

Hello. My name is Austin Bingham, and welcome to the eleventh module of Python: Beyond the Basics. In this module we'll be focusing on context managers, those objects designed to be used in with statements. We'll also look at some advanced syntax for with statements, as well as some tools in the standard library that help simplify context manager development. First off, what exactly is a context manager? Simply put, a context manager is an object that is designed to be used in a with statement. When a with statement is executed, the expression part of the statement, that is the part following the with keyword, evaluates to a value. This value must be a context manager, and the underlying mechanics of the with statement use this value in specific ways to implement the semantics of the with statement. All of that is true and important, but perhaps it doesn't get to the heart of what a context manager is. Conceptually, a context manager implements two methods, which are used by the with statement. The first method is called before the with statement's code block begins, and the second method is called when the with statement's code block ends even if the block exits with an exception. In other words, a context manager represents code that runs before and after the with statement's code block. You can think of these operations as setup() and teardown(), construction() and destruction(), resource allocation() and deallocation(), or any number of other ways. In general, and for reasons that will become clear soon, we'll refer to these methods as enter() and exit(). The important point is that both the enter() and exit() methods are called every time the with statement is executed no matter how the code block terminates. So, this brings us to perhaps the most useful statement of what a context manager is. It is a way to ensure that resources are properly and automatically managed around the code that uses these resources. The enter() method of a context manager ensures that the object is ready for use in the with block, and the exit() method ensures that the object is properly closed, shut down, or cleaned up when the block ends. Since you've almost certainly used with statements at some point, you've almost certainly used a context manager whether you knew it or not. Almost every Python developer will have used files in a with statement like this. The benefit of using files in a with statement is that they are automatically closed at the end of the with block. This works because files are context managers, that is they have methods which are called by the with statement before the block is started and after the block exits. The exit() method for a file, that is the code executed after the with block exits, does the work of closing the file, and this is how files work with with statements to ensure proper resource management.

The Context Manager Protocol

Now that we understand what a context manager is supposed to do, we can take a look at how to implement them. For an object to be a context manager, it needs to support the context manager protocol, which consists of only two methods, dunder-enter and dunder-exit. We'll look at the details of both of these methods, but first let's get a good idea of how these methods are used by a with statement. The first thing a with statement does is execute its expression, the code immediately following the with keyword. The expression must evaluate to a context manager, that is the expression must produce an object which supports both the dunder-enter and dunder-exit methods. Once the expression is evaluated and we have a context manager object, the with statement then calls dunder-enter on that context manager with no arguments. If dunder-enter throws an exception, execution never enters the with block, and the with statement is done. Assuming that dunder-enter executes successfully, it can return a value. If the with statement includes an as clause, this return value is bound to the name in the as clause; otherwise, this return value is discarded. It's important to really recognize what's going on here. A naïve interpretation of a with statement might lead you to think that the result of the with statement's expression is what is bound to the as variable while in fact it is the return value of the context manager's dunder-enter that is bound to the as variable. In many cases in fact, dunder-enter simply returns the context manager itself, but this is not always the case, so it's important to keep in mind what's really happening. Once dunder-enter has been executed and its return value potentially bound to a name, the with block itself is executed. The with block can terminate in one of two fundamental ways, with an exception or by running off the end of the block what we call normal termination. In both cases the context manager's dunder-exit method is called after the block. If the block exits normally, dunder-exit is called with no extra information. If on the other hand the block exits exceptionally, then the exception information is passed to dunder-exit. This means that dunder-exit can do different things depending on how the with block terminates, and this can be a very powerful tool.

A First Context Manager Example

With that in mind, let's make our first simple context manager. As with many things in Python, it's much easier to demonstrate context managers by example rather than by words alone. Start by creating the simplest possible context manager like this. As you can see, this implements both dunder-enter and dunder-exit, but neither one really does anything. Let's try that out in a with statement. (Typing) Well, Python didn't complain about anything, so it appears that we've created a valid, if somewhat useless context manager. We can see that X in the with statement is bound to our LoggingContextManager, which is what we'd expect since we return self from dunder-enter. To demonstrate that the as variable is actually bound to the return value of dunder-enter and not just the return of the expression, let's change dunder-enter to return something else. Now, if we use that context manager, we'll see that indeed the value of X is the string returned by dunder-enter. (Typing) Okay. We seem to know what we're doing. Let's now finish our context manager by having it log some text when we enter and exit it. (Typing) Notice how our dunder-exit method prints information about the exception information it recieves. We'll look at these arguments in more detail later, but for now just know that these arguments are all None if the block exits normally, and they contain other information if the block exits exceptionally. Let's test out our fully armed and operational context manger. (Typing) Exactly as we'd expect, we see the output from the with block sandwiched between the strings printed by the context manager's enter and exit functions. To cap this example off, let's throw an exception from the with block and see that indeed our dunder-exit method is still called, this time with exception information. We see the output from our dunder-enter method, and we also see that the output from dunder-exit indicates that an exception occurred. You'll notice that the exception information is printed a second time as well after our dunder-exit method. This happens because we're letting the exception propagate out of the with statement, so the REPL is printing its own information about the exception. There are ways to prevent exception propagation out of with statements, and we'll look at those in a bit.

\_\_enter\_\_()

Now that you've created a complete context manager yourself, there's not much left to say about the dunder- enter method. It's called on the context manager just before entering the with block, and its return value is bound to the as variable of the with statement. Dunder-enter is allowed to return anything at once including None, and the with statement itself doesn't ever access or use this value. It is very common, however, for context managers to simply return themselves from dunder-enter. This is the case, for example, with Python's file class. When we use code like this, we know that open returns a file object, and we also expect F to be bound to that file object, so clearly file's dunder-enter method must be returning the file object itself. We can verify this with a simple experiment in the REPL. We first open a file without using a with statement binding the file to the name F. We then use F as a context manager in a with statement binding the result of dunder-enter to the name G, and we see that F and G are the same object.

\_\_exit\_\_()

Dunder-exit is substantially more complex than dunder-enter because it performs several roles in the execution of a with statement. Its first and primary role of course is that it is executed after the with block terminates, so it is responsible for cleaning up whatever resources the context manager controls. Its second role is to properly handle the case where the with block exits with an exception. To handle this case, dunder-exit accepts three arguments: The type of the exception that was thrown, the value of the exception, that is the actual exception instance, and the traceback associated with the exception. When a with block exits without an exception, all three of these arguments are set to None, but when it exits exceptionally these arguments are bound to the exception, which terminated the block. In many cases a context manager needs to perform different dunder-exit code depending on whether an exception was thrown or not, so it's typical for dunder-exit to first check the exception information before doing anything else. A common idiom is to simply check the type argument to detect an exceptional exit. Let's update LoggingContextManager. \_\_exit\_\_ to behave differently when an exception is thrown. It first checks whether type is None. If it is, then this means that no exception was thrown and a simple message is printed. If type is not None, however, dunder-exit prints a longer message, which includes the exception information. Let's test this out in the REPL. (Typing) Here we see that the normal exits are handled properly, and here we see that exceptions are properly detected.

\_\_exit\_\_() and Exception Propagation

There's a final role that dunder-exit plays in with statements which has to do with how exceptions are propagated out of them. By default, when an exception is thrown from a with block, the context manager's dunder-exit is executed, and afterward the original exception is re-raised. We've seen this in our examples when the REPL prints exception information after our context manager is done. To really drive this point home, let's try another small experiment in the REPL. Here we use a with statement inside a try block and catch the ValueError, which is propagated out of the with statement. So, how does dunder-exit control the propagation of exceptions out of with statements? That has to do with the return value of the dunder-exit method. If dunder-exit returns a value which evaluates to False in a Boolean context, then any exception that came out of the with block will be re-raised after the dunder-exit call. You can think of it as if the with statement is asking dunder-exit should I swallow the exception. If dunder-exit says False, then the with statement re-raises the exception. If dunder-exit says True, then the with statement will exit normally, that is without raising the exception. With that in mind, remember that earlier I said the default behavior of a with statement is to propagate exceptions. What I really meant is that if your context manager's dunder-exit function doesn't return anything, then exceptions would be propagated. This is because, as you'll recall, a method that doesn't explicitly return a value will implicitly return None. Of course None evaluates to False, so a dunder-exit which doesn't return anything is instructing the with statement to propagate exceptions. Another important point to understand is that dunder-exit should not explicitly re-raise the exception that it receives. If dunder-exit wants to ensure that the exception from the with block is re-raised, it should simply return False and let the with statement re-raise it. Dunder-exit should only explicitly raise exceptions when it actually fails itself, that is when something goes wrong in the dunder-exit method. The with statement machinery will interpret exceptions from dunder- exit as a failure of dunder-exit, not as a propagation of the original exception.

The with-statement Expansion

Before we leave this section on creating context managers, let's take a second and look at exactly what Python is doing in with statements. The with statement was defined in PEP343, and in that PEP the authors provide an explicit expansion of the with statement into Python code that doesn't use the with statement. It looks like this. This is taken directly from PEP343, which you can see for yourself at www. python. org/dev/peps/pep-0343. We won't spend any time in this course going over this expansion, but if you want to really understand how context managers are used by with statements then it's worth taking some time on your own to really understand this code. It may seem like a lot to ingest at first, but it's really not saying anything more than what we've already covered in this module, and seeing it written so explicitly might help clear up any lingering questions you have. And in any case, it's just interesting to see how these kinds of language features are developed.

contextlib.contextmanager

Now that we've covered the deep, dark details of how to create context managers, let's spend some time looking at a tool that helps simplify context manager development. The contextlib package is part of Python's Standard Library. In its own words it "…provides utilities for common tasks involving the with statement. " In this course we're just going to look at one part of contextlib, the context manager decorator that it provides. There are several other interesting parts to contextlib, however, and it's well worth your time to investigate them. The contextmanager decorator is, as its name suggests, a decorator that you can use to create new context managers. Rather than continuing to call it the contextmanager decorator, I'm just going to call it contextmanager from now on, and I'll be sure to make it clear when I'm talking about context manager the concept versus contextmanager the decorator. The concept behind context manager is simple. You define a generator, that is a function which uses yield instead of return, and decorator it with the contextmanager decorator to create a context manager factory. This factory is nothing more than a callable object which returns context managers making it suitable for use in a with statement. Let's see how this looks in code since that will make everything much more clear. Here we see the contextmanager decorator applied to a generator called my\_context\_manager(). You can now use my\_context\_manager() just like any other context manager, and here's how it functions. First, the generator is executed up to its yield statement. Everything before the yield is equivalent to the dunder-enter method on a normal context manager. Next, the yield statement supplies the value, which will be bound to the as variable in the with statement. In other words, the yield is like the return value from dunder-enter in a normal context manager. Once yield is called, control leaves the contextmanager function and goes to the with block. If the with block terminates normally, then execution flow returns to the contextmanager function immediately after the yield statement. In our code snippet, this is the section marked normal exit. If on the other hand the with block raises an exception, then that exception is re-raised from the yield statement in the contextmanager. In our code snippet this means that execution would go to the except block and into the section labeled exceptional exit. In other words, the contextmanager decorator allows you to define context managers using normal control flow via the yield statement rather than breaking it up across two methods. Furthermore, since generators remember their state between calls to yield you don't need to define a new class just to create a stateful context manger. Let's see if we can rewrite our old logging\_context\_manager using the contextmanager decorator. That certainly looks simpler than the original version, so let's see if it works. That looks good for the case of a normal exit, so how about an exceptional exit. (Typing) Great! That seems to work well. One thing you may have noticed is that our new context manager didn't propagate the ValueError after it completed. Unlike standard context managers, those created with the contextmanager decorator must use normal exception handling to determine if exceptions are propagated from the with statement. If the contextmanager function propagates the exception either via re-raising it or by simply not catching it at all, then the exception will propagate out of the with statement. If the context manger catches and doesn't re-raise an exception from the with block, then the exception won't be propagated out of the with statement. In our case we caught the ValueError and printed some information about it. Since we didn't re-raise it, the ValueError was not propagated out of our contextmanager, and thus it wasn't propagated out of the with statement. Just to make this point explicit, let's update our new contextmanager to propagate exceptions. To do this we simply add a bare raise call after logging the exception. And we can see that this works as we expected since the ValueError is propagated to the REPL. (Typing) Contextlib's contextmanager decorator is a very useful tool, and it certainly eases the creation of context managers in many cases. It's good to know how to create context managers using the lower-level protocols of course, and knowing which technique to use in a given situation will require some experience and practice.

Multiple Context Managers

So far when we've used with statements we've only used a single context manager, but you can actually use as many context managers as you want in a single with statement. The syntax for this is like this. For each context manager, an optional variable binding is separated by a comma. From an execution point of view, this is exactly equivalent to using nested with statements with earlier context managers enclosing later ones. So, this multi-context manager form is the same as this nested form. Let's define a simple context manager to test this out and pass two of these to a single with statement. (Typing) We can see that this gives the same results as a nested form. (Typing) In the end, later context managers are simply part of the body as far as an earlier context manager is concerned. In fact, names bound in an as clause in earlier parts of the with statement can be used when defining later context managers. Let's modify nest\_test to yield a value and then use that yielded value when constructing the second context manager. (Typing) Again, this is precisely the behavior you would get if you simply use nested with statements. With the nested with statement notion in mind, it should be clear how exceptions are handled when using multiple context managers. In short, any exception propagated from inner context managers will be seen by outer context managers. Likewise, if an inner context manager swallows an exception, then it won't be seen by outer ones. To illustrate, let's create one more simple context manager that can be configured to either propagate or swallow exceptions. If the propagate argument to propagator is True, it will propagate any exceptions that come from the body nested beneath it; otherwise, it will swallow those exceptions. Now we can see how an inner context manager can swallow exceptions so that an outer one never sees them. (Typing) Likewise, the inner one can propagate them while the outer one swallows them. Since the REPL doesn't report an exception in this case, we can be sure that no exception escaped the with statement.

Don't Pass a List!

When passing multiple context managers to a with statement, make sure you don't accidentally try to pass a list or some other sequence of context managers. If you do this, you'll end up with a fairly mysterious looking error message. At first glance it seems to be telling you that your context manager doesn't have a dunder-exit method, which we know is wrong, so what's going on? The problem here is that we're actually trying to pass a list to the with statement as a context manger. The with statement doesn't try to unpack sequences. It simply tries to use what you pass it as a context manager. So, the with statement is looking for dunder-exit on a list you're passing it, and of course it fails with an AttributeError since lists are not context managers. To fix this, simply remove the square brackets. Because of this, if you have several context managers and you want to split them over several lines, you can't put them in parentheses or anything like that. What you need to do is use a line continuation to get the splits you want. For example, you could write this. (Typing)

Duck Tail: Context Managers for Transactions

As we've learned, developing context managers is not difficult, and with the various tools we have for creating them we should not back away from using context managers when they're appropriate. Most of the examples we've seen so far have been small and focused on presenting a particular point, so let's use this section to develop a more realistic example, something like what you might see in the wild. Our example will involve a connection class which represents some sort of database connection along with a transaction class, which manages transactions in the database. Users of our system can create connections and then create transaction objects to start transactions. To commit or rollback transactions, users can call methods on the transaction instances. Here's our connection class. Obviously this doesn't do real database work. All that it really does is increment a transaction ID whenever a new transaction is started. It also helpfully prints out what it's doing as transactions are processed. Now, here's our transaction class. This queries the connection for a new transaction ID and then calls into the connection to commit or rollback transactions. Here's how someone might use these classes without context mangers or in a function. But of course that last example is flawed because we never commit the transaction, yet it's very easy to write this kind of code. And even if we did explicitly commit the transaction, it can be tricky to get it right in the presence of exceptions. So, let's see if we can't design a context manager that starts a transaction for us, commits it if the with body exits normally, and rolls it back if there's an exception. This is very similar to the examples we've seen earlier. The beginning of the context manager starts a transaction. Then inside a try block we yield the transaction so that transactional operations can take place inside a with block. If the with block raises an exception, we catch the exception, roll the transaction back, and re-raise the exception. If the with block exits normally, we commit the transaction and continue normally. So, let's see it in action. (Typing) Great! The transaction manager detected the exception and rolled the transaction back. Let's try that again, this time without the exception. (Typing) So, now we've got a full-featured context manager that does some pretty realistic work. All it needs is a full database connected to it, but that's just a detail right?

Summary

In this module we've covered a lot of the details of context manager development, and you should be comfortable enough with them to write your own. Here's a summary of the specific points we covered. Context managers are objects designed to work in with statements. The expression of a with statement must evaluate to a context manager. Context managers have code that is run before and after with blocks. Context managers are particularly useful for resource management. Files are a common example of context managers. The context manager protocol involves two methods, dunder- enter and dunder-exit. Dunder-enter is called before the with block. The return value of dunder-enter is bound to the name in the optional as clause of a with statement. If dunder-enter raises an exception, the with block is never entered. Dunder-enter often returns its own self, but this is not required. A context manager's dunder-exit method is called when a with block terminates. Dunder-exit is called both when the with block exits normally and when it exits exceptionally. If a with block exits with an exception, dunder-exit is called with the type, value, and traceback of that exception as arguments. If a with block exits without an exception, then dunder-exit is called with None for each of those arguments. Dunder-exit can respond differently to exceptional and non-exceptional with block exits by checking its arguments. If dunder-exit returns a value that evaluates to False, then any exception that came from the with block will be propagated by the with statement. If dunder-exit returns a value that evaluates to True, then any exception that came from the with block will be swallowed by the with statement. Since functions implicitly return None in the absence of an explicit return, with statements will propagate exceptions when dunder-exit doesn't explicitly return a value. Dunder-exit should not explicitly re-raise its exception arguments. Dunder-exit should only raise exception if the problem occurs in the method itself. PEP 343 is the original PEP defining with statements. PEP 343 includes an expansion of with statements in the Python code that doesn't use with statements. Contextlib provides some utilities for working with with statements. Contextlib. contextmanager is a decorator used for creating context manager factories out of generator functions. A context manager generator yields a value, which will be bound to the name in the optional as clause. Execution moves from the context manager generator to the with block when the yield statement is executed. Execution returns from the with block to the yield call when the with block terminates. All of the code in a context manager generator before the yield is equivalent to the dunder-enter method. If an exception is raised in the with block, it is re-raised at the yield statement in the context manager generator. The code executed after the yield in a context manager generator is equivalent to the dunder- exit method. If a context manager generator wants to propagate an exception, it needs to explicitly re-raise it. With statements can take more than one context manager. Separate each context manager including the optional as clause from the next with commas. Multiple context managers in a with statement are equivalent to nested with statements. Later context managers are nested inside earlier ones. Names bound in earlier context managers can be used when creating later ones. With multiple context managers, later context managers can swallow exceptions such that earlier context managers never see them. If you mistakenly pass a list or other sequence of context managers to a with statement, you may be surprised to get an AttributeError. To put context managers on different lines in the same with statement, use line continuations. Thank you for watching, and we'll see you in the next module.

Introspection

Object Types in Depth

Hello. My name is Robert Smallshire. Welcome to the twelfth and final module of Python: Beyond the Basics where we'll look into Python's remarkable powers of introspection. Introspection is the ability of a program to examine its own structure and state, a process of looking inward to perform self examination. We've already encountered many tools which allowed programs to examine themselves in earlier modules of this course and in the preceding Python Fundamentals course. We'll quickly review some of those tools here to bring them under the umbrella of introspection, and then we'll move onto more tools in this category and some uses for them. Perhaps the simplest introspective tool we've met is the built-in function type. We've used this extensively at the REPL for displaying object types. For example, if we create an integer i = 7, we can use type(i) to return its type int. Now just enter int into the REPL and press return. This indicates that type int is just a representation of int as produced by repr, which is what the REPL does when displaying the result of an expression. We can confirm this by evaluating this expression repr(int). So, type 7 is actually returning int, which is the class type of integers. We can even call the constructor on the return type directly by doing type(i)(78). But what is the type of int? Or in other words, what type does type return? The type of type is type. Every object in Python has an associated type object, which is retrieved using the type function. In fact, the type function returns the special dunder-class attribute, so dunder-class of i is int, dunder-class of dunder-class of i is type, and dunder-class of dunder-class of dunder-class of i is also type confirming that type is its own type. Using another introspection facility we already know, we can see that type is itself an object because type is a subclass of object. And what's more, the type of object is type. What this circular dependency shows is that both type and object are fundamental to the Python object model, and neither can stand alone without the other. Notice that issubclass performs introspection. It answers a question about an object in the program as does the isinstance method we're familiar with. In general, type tests should be avoided in Python, but on the rare occasions they are necessary prefer to use the isinstance or issubclass functions rather than direct comparison of type objects.

Introspecting Objects

We've already met another important introspection function, dir, which returns a list of attribute names for the instance. Here we do dir(i), which remember is an int. Both attributes and method names are returned in this list for the simple reason that methods are attributes. They're just callable attributes. It's fascinating to see that the int object has attributes called numerator and denominator allowing it to be used like the rational number object modeled by the fraction type. It also has imag, real, and conjugate attributes allowing it to be used like a complex number under duck typing rules. Given an object and an attribute name, we can retrieve the corresponding attribute object using the built-in getattr function. Let's retrieve the denominator attribute using getattr. This returns the same value as accessing the denominator directly. Other attributes return more interesting objects. The conjugate attribute is revealed to be a method. We can check that it's a callable object using another introspection tool, the built-in callable function. By navigating around the attributes of an object and the attributes of those attributes, it's possible to discover almost anything about an object in Python. Here we retrieve the type of the int. conjugate method through it's dunder-class attribute and then the name of that class as a string through the dunder- name attribute. Trying to retrieve an attribute that does not exist will result in an AttributeError. We can determine whether a particular object has an attribute of a given name using the built-in hasattr function, which returns True if a particular attribute exists. For example, our integer I has an attribute bit\_length, but does not have an attribute index. Although in general the easier to ask forgiveness than permission style of programming using exception handlers is considered more Pythonic than look before you leap style programs which use type test and attribute existence tests, programs using hasattr can quickly get messy, particularly if you need to test for the existence of many different attributes, and perhaps surprisingly the optimistic code using try/except is faster than look before you leap code using hasattr because internally hasattr uses an exception handler anyway. Here's a function, a module numerals. py, which given an object supporting the numerator and denominator attributes for rational numbers returns a so-called mixed numeral containing the separate whole number and fractional parts. For example, it will convert 17/3 into 5 2/3. At the beginning of the function we use two calls to hasattr to check whether the supplied object supports the rational number interface. (Typing) As you would expect, we could successfully call this with a fraction object such as 11/10 which returns 1 1/10; however, we get a TypeError when we try to pass a float because float supports neither numerator nor denominator. Here's an alternative version of the same function, which does not perform the hasattr check. This version is optimistic and just assumes that the argument passed is reasonable. (Typing) The only difference in runtime behavior is that now a different exception is raised. Previously we raised a TypeError explicitly, and in this case an AttributeError is raised. This is possibly more detailed, but maybe less informative. Of course we can have the best of both worlds by using an exception handler to raise a more appropriate exception type of TypeError chaining to it the original AttributeError to provide the details. (Typing) This approach yields the maximum amount of information about what when wrong and why. Now we see that the AttributeError was the direct cause of the TypeError.

Introspecting Scopes

Python contains two built-in functions for examining the content of scopes. Let's play with them at the REPL. The first function we'll look at is globals(). This returns a dictionary which represents the global namespace. For example, we can see here the binding of the name dunder-name to the string dunder-main, which we frequently use to determine how our program is being executed with the if \_\_name\_\_ == '\_\_main\_\_' idiom. Now we'll defined a variable a = 42 and call globals() again, and we can see that the binding of the name 'a' to the value of 42 has been added to the namespace. In fact, the dictionary returned by globals() doesn't just represent the global namespace. It actually is the global namespace. Let's create a variable by assigning to new keys in the dictionary. Here we'll bind the name tau to the value 6. 283185. We can now use this variable just like any other. The second function for introspecting scopes in Python is locals(). Since we are currently operating at module or in other words global scope in the REPL, local()s returns the same dictionary as globals(). To really see locals() in action, we're going to need to create another local scope, which we can do by defining a function. Our report\_scope function accepts a single argument, imports a pretty printing function from the pprint standard module, defines a variable X to have a value of 496, and then pretty prints the locals() dictionary with a width of 10 characters. When run, we see that this function has the expected three entries in its local namespace, the function argument arg, a binding to the function imported from the pretty printing standard library module, and the variable X. Recall that extended function call syntax allows us to unpack a dictionary into function keyword arguments. Remember also that the string format method accepts named arguments, which correspond to format placeholders. By using locals() to provide the dictionary, we can easily refer to local variables in format strings. Here we set name to Joe Bloggs, age to 28, and country to New Zealand and refer to these variable names directly within the format string passing \*\*locals as the argument to format. Using this technique is less wise from a maintainability standpoint than its cleverness might suggest, so use it sparingly. That said, you might come across this in the wild, and it could be a handy time saver when debugging.

The Python Standard Library Inspect Module

The inspect module in the Python Standard Library contains advanced tools for introspecting Python objects in great detail. We won't cover all of its facilities here, but we'll get you off on the right foot. We'll use the inspect module to introspect the sorted\_set. py module we built earlier, which contains the SortedSet class. We start by importing inspect and our sorted\_set module. First we'll check that sorted\_set is indeed a module by using the ismodule predicate. No surprises here. The getmembers function retrieves members as a list of name value pairs. When we pass the module object to this, it returns a vast output including everything in the module namespace, which includes all the built-ins. Fortunately getmembers accepts a second argument, which is a predicate function to filter the list for certain categories of object. The inspect module contains 16 such predicates for identifying different object types from isabstract to istraceback. Let's pass the inspect. isclass predicate to filter for just classes. You may have been expecting only SortedSet to be returned since that's the only class we explicitly define in the SortedSet module, but in fact all the other classes we use such as the Sequence and Set abstract base classes and even chain, which you probably didn't expect to be a class, are included. Presumably since we call chain like a function, it is a class which implements the dunder-call special method. But why are these objects returned? Well, any class that ends up in the module namespace will be returned whether defined or imported. It may surprise you that we can actually import any of these names from our SortedSet module because import just binds objects in another module's namespace to names in our current namespace. So, in fact we can do things like from sorted\_set import chain, which is completely equivalent to from iter\_tools import chain. Surprised? Perhaps even more surprising is that there's no straightforward way to stop this happening. Let's dig a little deeper into our module, for example, by retrieving all of the functions of the SortedSet class. We do this by calling inspect. getmembers passing the SortedSet class and the isfunction predicate. The inspect module contains tools for interrogating individual functions. This is done by retrieving a so-called signature object for the function. Let's retrieve the signature of dunder-init on our SortedSet class and bind this to a name called init\_sig. (Typing) From this signature object we can obtain a list of the parameters, which we can see here are called self and items, and we can query individual parameters for attributes such as their default values. (Typing) Recall that items defaults to None. Handily, when we convert a signature object to a string, we get a nice output. Be aware that inspect. signature doesn't always work. Some built-in functions, which are implemented in C don't provide sufficient metadata. In this case, the function fails with a ValueError. Here we try to retrieve the signature of the built-in abs function. There are many, many tools in the inspect module, which allow programs to introspect themselves in very deep ways. We won't cover anymore of them here because they're very specialized.

Duck Tail: An Object Introspection Tool

We'll round off this course module and indeed the whole course by building a tool to introspect objects. We'll use many of the useful and interesting techniques we've picked up throughout this course bringing them together into a useful program. Our objective is to create a function, which when passed a single object prints out a nicely formatted dump of that object's attributes and methods similar to getmembers, but with rather more finesse. Let's start by creating a module, intropector. py, and defining a dump method that prints the outline of the object dump. We print four section headings, and the only object detail we print is the type retrieved using the type built-in function. Let's work through this function filling in the details as we go. Let's start by getting some detail into the documentation section. The easiest thing to do here would be to simply print the obj. \_\_doc\_\_ attribute, and indeed that would be a great start. We can do better though by using a function from the inspect module called cleandoc, which deals with tricky topics like uniformly removing leading whitespace from docstrings. Even better, there is a function getdoc in the inspect module that will combine these operations retrieving and tidying up the docstring. We'll use that instead giving us very straightforward code for our documentation section. Now for the attributes and methods. For this we need to produce a list of attributes and their values. We've just shown you how to do this using inspect. getmembers. Although that's arguably the right course of action, we'll code up our own routine for pedagogical purposes so we have an opportunity to apply some of the techniques we've picked up. We know you can get a list of attributes using the dir built-in, so we'll start with that. We'll put the attribute names into one of the SortedSet collections we defined earlier in the course. Now we'll produce another SortedSet of all method\_names by determining which of those attributes when retrieved getattr is callable. I'll use the filter built-in function to select those attributes which are callable, and our predicate which acts on the attribute name will be a lambda function. The next part of our program will assume that method names is a subset of all\_attr\_names, so we'll check that is the case at this juncture by using an assert statement. Recall that the relational operators on SortedSet can be used to implement the subset test efficiently between two objects of the same type. Now we'll use a set difference operation called for the infix subtraction operator to produce a SortedSet of all regular attributes, that is those which are not callable by subtracting the method\_names from all\_attr\_names. Now we are ready to print the attributes and their values. We'll do this by building a list of name value pairs using a list comprehension. An attribute value could potentially have a huge text representation, which would spoil our output without adding much value, so we'll use the Python Standard Library reprlib module to cut the values down to a reasonable size in an intelligent way. Then we'll print a nice table of names and values using a putative print\_table function we'll implement later. The function will need to accept a sequence of sequences representing rows and columns and the requisite number of column headers as strings. With the regular attributes dealt with, we can move onto methods. From our list of method names, we'll generate a series of method objects simply by retrieving each one with getattr. Then for each method object we'll build a full method signature and a brief documentation string using functions we'll define in a moment. The signature and documentation will be assembled into a list of tuples using this list comprehension. Finally, we'll print the table of methods using our yet to be implemented print\_table function for a second time. That completes the dump function. Now let's implement the three functions full\_sig, brief\_doc, and print\_table it depends on. First fill\_sig. In this function we retrieve the method's name via the special dunder-name attribute and then try to get the argument signatures using inspect. signature. If that fails, we fall back in the exception handler to returning a string indicating that we couldn't determine the signature. This is a nice example of easier to ask forgiveness than permission programming style. Now let's move on to implementing brief\_doc. Documentation strings can be very lengthy, and in our table we only want a brief description. Fortunately there's a widely followed convention that the first line of a doc\_string contains exactly such a brief description. This function attempts to extract that line and return it. We must account for the fact that the doc\_string attribute may be set to None in which case we wouldn't be able to call splitlines on it. Or the doc\_string may be defined, but empty in which case splitlines would return an empty list. In either of these eventualities we return an empty string. The function follows a look before you leap style of programming because we felt the result was clearer than the alternative. Now we come onto the print\_table function. Recall that the function accepts a sequence of sequences as its first argument with the outer sequence representing rows of the table and the inner sequences representing the columns within those rows. In addition, the function accepts as many string arguments as are necessary to provide the column headings. We use extended argument syntax to accept any number of header arguments. First we'll perform some basic validation by ensuring that the number of header values supplied is compatible with the first row of the table. We'll follow the lead of the built-in functions and raise a TypeError if too few arguments are provided. (Typing) See how we've split the long raise statement over several lines. First, we use the fact that we're allowed to continue over multiple lines inside open parentheses. Second, we use implicit string concatenation of adjacent string literals to glue two fragments of the message together. Now we need to determine the width of each column in the table taking into account the width of each item in the row\_of\_columns data structure, but also the header widths. To do this, we'll lazily concatenate the header row and the data rows into a value rows\_of\_columns\_with\_header using itertools chain. See how we make a list containing the single headers row for the first argument. Next we'll use the zip\* idiom to transpose the rows\_of\_columns into columns\_of\_rows. We force evaluation of the otherwise lazy zip by constructing a list binding the result of the name columns\_of\_rows. To find the maximum width of a column, we can use this expression, the maximum result of mapping the function len over the columns. Let's break this down. We pass the len built-in function to the map built-in function, which applies len to each item in the column sequence in effect yielding a series of widths. We then use the max built-in function to find the maximum width of the column. We'd like to apply this to each column, so we put this expression in a list comprehension which does exactly that. Notice that map and list comprehensions are often considered to be alternatives. Here we've combined both because we think that the result is more readable than using nested comprehensions or using two applications of map. The resulting column\_widths object is a list containing one integer for each column. We want to print fixed width columns, and we know we can do that with format specifiers and the format string method; however, we have a variable number of columns, so we need to build up our format string programatically. Each column will need to have a format specify something like this, curly braces with colon followed by a width for a column width of 13 in this case. To do this we can use the format function to insert the width, but we need to escape the outer curly braces so that they carry through to the result. This escaping is performed by doubling the curly brace character. Applying this over all columns using a generator expression we get this expression for column\_specs. We concatenate all of these column format specifications together using the str. join method inserting a space between each column. Now we're ready to begin printing our table. First, we use our format\_spec to print the column headers using extended argument unpacking to supply each header string as a separate argument to format. Now we'd like to print horizontal rules under the headers to separate them from the data below. We use string repetition with a multiply operator to create the individual rules by repeating hyphens. Then we generate a rule for each column using a generator expression. Then it's a simple matter to print the rules, again using extended argument unpacking. Finally, we'll print each row of the data using a for loop. Now we're done. Here's the complete code. Let's see if the dump function works on a simple object like the integer digit 7. Using introspection we can get very comprehensive information even for a very simple object.

Summary

Let's summarize what we've covered in this module on introspection in Python. We investigated the type of objects using the type function, and we asked deep questions such as what is the type of type learning that the answer is type. We saw that using the type function is the equivalent of retrieving the special dunder-class attribute, and we learned that both object and type are the foundation of the Python type system. We introspected objects using the dir function that we were already familiar with and also the getattr and hasattr functions, which are new to us. We showed how to determine whether an object is callable using the callable predicate function, and we showed how to retrieve the name of a class or function by fetching the special dunder-name attribute. We showed how hasattr can be used to support a look before you leap programming style, although we reminded you that easier to ask for forgiveness than permission is generally preferred in Python. We demonstrated the built-in globals() and locals() functions for gaining direct access to namespace dictionaries. We introduced the Python Standard Library inspect module and used it to extract information about certain types of module and class attributes, and we showed how to retrieve function signatures using the inspect. signature function allowing details such as default function arguments to be retrieved. We finished off by giving an extended example of an object introspection tool, which uses the techniques we have learned in this module together with many techniques from earlier in this course. Well, here we are at the very end of Python: Beyond the Basics. Thanks for watching, and be sure to enjoy putting much of what you've learned in this course into practice.