Core Python: Getting Started

by Austin Bingham and Robert Smallshire

Course Overview

Course Overview

Hi, everyone. My name is Austin Bingham, and welcome to the course, Core Python: Getting Started. My coauthor, Robert Smallshire, and I are founders and principle consultants at Sixty NORTH. Python is a big language, and it's important to have a thorough grounding in its key concepts in order to be productive and create high-quality Python programs. In our experience, starting off in the right direction and avoiding early misconceptions is crucial to success with Python, and that's the kind of start we aim to provide with this course. In this course, we're going to introduce you to the essentials of the Python language, as well as important parts of the Python standard library. Some of the major topics that we will cover include Python's fundamental data types, using functions and modules to organize your code, Python's underlying object model, defining your own types using classes, and working with iteration and iterable objects. By the end of this course, you'll know what you need to work on many Python projects, and you'll be in a great position to continue learning more advanced aspects of Python. Before beginning the course, you should be familiar with basic computer concepts, like files and running programs. This course doesn't assume any specific programming experience, but a basic understanding of concepts like functions and program execution can be helpful. From here, you should feel comfortable diving into other Core Python language courses on Organizing Larger

Programs, Classes and Object-Orientation, Functions and Functional Programming, and Robust Resource and Error Handling. I hope you'll join me on this journey to learn about the Python programming language with the Core Python: Getting Started course at Pluralsight.

Installing and Starting Python

Overview

In this module, we'll cover obtaining and installing Python on your system for Windows, Ubuntu Linux, and macOS. We'll write some basic Python code and become acquainted with the essentials of Python programming culture, such as the Zen of Python. Though we'll never forget the origins of the name of the language. There are two major versions of the Python language, Python 2, which is the widely-deployed legacy language, and Python 3, which is the present and future of the language. It's now over a decade since the transition from Python 2 to Python 3 was begun, and we strongly recommend that all new projects are begun with Python 3 as Python 2 will not be maintained from the year 2020. That said, most of the Python code we will demonstrate will work without modification between the last version of Python 2, which is Python 2.7, and recent versions of Python 3, such as Python 3.8. However, there are some key differences, and in a strict sense, the languages are incompatible. We'll be using Python 3 for this course, and everything we show will work on Python 3.6 or later, and most things will work on Python 3.3 or later. We're also confident that everything we present will apply to future versions of Python 3, so don't be afraid to try those as they become available.

Installing Python

Before we can start programming in Python, we need to get hold of a Python environment. Python is a highly portable language available on all major operating systems. You will be able to complete this course on Windows, Mac, or Linux, and the only major section where we diverge into platform specifics is coming right up, as we install Python 3. As we cover the three platforms, feel free to skip over the sections which aren't relevant for you. Let's see how to install Python 3 on Windows 10. For Windows, you need to visit the official Python website at python.org and then navigate via the Downloads tab to the downloads for Windows and click the button to begin

downloading the latest Python 3 version. When given the option, choose to run the installer. After the installer starts, be sure to enable the option to add Python to the PATH environment variable before moving on by clicking Install Now. You may be asked to approve the Python Installer making changes to your device, which you should accept. After a few seconds, the installer will complete and you can close the installer and your web browser. We'll be working with Python from the command line, so via the Start button, choose Windows PowerShell. On older versions of Windows, you may need to use the Command shell instead, and start Python just by typing python, followed by Enter. Welcome to Python. The triple arrow prompt shows you that Python is waiting for your input. At this point, you might want to skip forward while we show you how to install Python on Mac and Linux. Now let's see how to install Python 3 on macOS. For macOS, you need to visit the official Python website at python.org. Navigate via the Downloads tab to the downloads for macOS, and click the button to begin downloading the latest Python 3 version. A PKG file downloads, which, when opened, launches the Python Installer. Continue through the install process, accepting the license agreement and using the default installation location. You may need to provide your password as you go. Although macOS does include a Python interpreter, it's the legacy Python 2.7 version, and for this course we use Python 3. The Python 3 version we're installing here will sit alongside your system Python 2 and won't interfere with the correct operation of your Mac. When Python is installed, you can clean up by moving the installer to the trash. To use Python, open a terminal, here we're using Spotlight to do so, and run Python 3 from the command line. Welcome to Python. The triple arrow prompt shows that Python is waiting for your input. The last operating system we'll look at is Linux, which is the easiest of all. Recent versions of Ubuntu Linux include Python 3 out of the box, so no installation is required. To begin using Python, open a terminal. On Ubuntu we can do this by using the search function accessible through the Ubuntu icon on the top left, entering terminal, and launching the terminal application. In the terminal, you should be able to start Python 3. Welcome to Python. The triple arrow prompt shows you that Python is waiting for your input. If you're using a version of Linux other than Ubuntu, you'll need to find out how to invoke and, if necessary, install Python 3 on your system.

Interactive Python

Now that Python is installed and running, we can immediately start using it interactively. This is a good way to get to know the language, as well as a useful tool for experimentation and quick testing during normal development. This Python command-line environment is a read, eval, print loop. Python will read whatever input we type in, evaluate it, print the result, and then loop back

to the beginning. You'll often hear it referred to as simply the REPL. When started, the REPL will print some information about the version of Python you're running, and then it will give you a triple-arrow prompt. This prompt tells you that Python is waiting for you to type something. Within an interactive Python session, you can enter fragments of Python programs and see instant results. Let's start with some simple arithmetic. As you can see, Python reads our input, evaluates it, prints the result, and loops around to do the same again. We can assign the variables in the REPL, print their contents simply by typing their name, and refer to them in expressions. Within the REPL, you could use the special underscore variable to refer to the most recently printed value, this being one of the very few obscure shortcuts in Python, or you can use the special underscore variable in an expression. Remember though that this useful trick only works at the REPL. The underscore doesn't have any special behavior in Python scripts or programs. Notice that not all statements have a return value. When we assigned 5 to x, there was no return value, only the side effect of bringing the variable x into being. Other statements have more visible side effects. Try typing print Hello, Python, at the prompt, you'll need parentheses after the print and quotes around the text, then press enter. You'll see that Python immediately evaluates and executes this command, printing the string Hello, Python and returning you to another prompt. It's important to understand that the response here is not the result of the expression being evaluated and displayed by the REPL. Rather, it is a side effect of the print function. As an aside, print is one of the biggest differences between Python 2 and Python 3. In Python 3, the parentheses are required, whereas in Python 2, they are not. This is because in Python 3, print is a function call. More on functions later. At this point, we should show you how to exit the REPL and get back to your system shell prompt. We do this by sending the end of file control character to Python. Although, unfortunately, the means of sending this character varies across platforms. If you're on Windows, press Ctrl+Z followed by Enter to exit. If you're on Mac or Linux, press Ctrl+D to exit. If you regularly switch between platforms and you accidentally press Ctrl+Z on a UNIX-like system, you will inadvertently suspended the Python interpreter and return to your operating system's shell. To reactivate Python by making it a foreground process again, simply run the fg command and press Enter a couple of times to get the triple-arrow Python prompt back.

Significant Whitespace

Now that you have a working Python REPL, let's look at some basic code structure. Start your Python 3 interpreter using the Python or Python 3 command for Windows or Unix-like systems, respectively. The control flow structures of Python, such as for loops, while loops, and if statements, are all introduced by statements which are terminated by a colon, indicating that the

body of the construct is to follow. For example, for loops require a body, so if you enter for i in range 5, Python will change the prompt to 3 dots to request you provide the body. One distinctive and sometimes controversial aspect of Python is that leading whitespace is syntactically significant. What this means is that Python uses indentation levels rather than the braces used by other languages to demarcate code blocks. By convention, contemporary Python code is indented by four spaces for each level, so we provide those four spaces and a statement to form the body of the loop. Our loop body will contain a second statement. So after pressing Return and getting another three dot prompt, we'll enter another four spaces, followed by a call to the built-in print function. To terminate our block, we must enter a blank line into the REPL. With the block complete, Python executes the pending code, printing out the multiples of 10, less than 50. Looking at a screen full of Python code, we can see how the indentation clearly matches and, in fact, must match the structure of the program. Even if we replace the code by gray lines, the structure of the program is clear. Each statement terminated by a colon starts a new block and introduces an additional level of indentation, which continues until the de-dent restores the indentation to a previous level. Each level of indent is typically four spaces, although we'll cover the rules in more detail in a moment. Python's approach to significant white space has three great advantages. First, it forces developers to use a single level of indentation in a code block. This is generally considered good practice in any language because it makes code much more readable. Second, code with significant whitespace doesn't need to be cluttered with unnecessary braces, and you never need to have code standard debates about where the braces should go. All code blocks in Python code are easily identifiable and everyone writes them the same way. Third, significant whitespace requires that a consistent interpretation must be given to the structure of the code by the author, the Python runtime system, and future maintainers who need to read the code. As a result, you could never have code that contains a block from Python's point of view, but which doesn't look like it from a cursory human perspective. The rules for Python indentation can seem complex, but are straightforward in practice. The whitespace you use can be either tabs or spaces. The general consensus is that spaces are preferable to tabs, and four spaces has become a standard in the Python community. One essential rule is never to mix spaces and tabs. The Python interpreter will complain, and your colleagues will hunt you down. You are allowed to use different amounts of indentation at different times, if you wish. The essential rule is that consecutive lines of code at the same indentation level are considered to be part of the same code block. There are some exceptions to these rules, but they almost always have to do with improving code readability in other ways, for example, by breaking up necessarily long statements over multiple lines. This rigorous approach to code formatting is programming as Guido intended it, or indented it. This philosophy of placing a high value on code qualities, such as

readability, gets to the very heart of Python culture, something we'll take a short break to explore now.

Python Culture

Many programming languages are at the center of a cultural movement. They have their own communities, values, practices, and philosophy, and Python is no exception. The development of the Python language itself is managed through a series of documents called Python Enhancement Proposals, or PEPs. One of the PEPs, called PEP 8, explains how you should format your code, and we follow its guidelines throughout this course. It is PEP 8, which recommends we use 4 spaces for indentation in new Python code. Another of these PEPs, called PEP 20, is called The Zen of Python. It refers to 20 aphorisms describing the guiding principles of Python, only 19 of which have been written down. Conveniently, The Zen of Python is never further away than the nearest Python interpreter as it can always be accessed from the REPL by typing import this. Throughout this course, we'll be highlighting particular nuggets of wisdom from the Zen of Python in Moments of Zen, to understand how they apply to what we have learned. As we've just introduced Python significant indentation, this is a good time for our first Moment of Zen. Readability counts, clarity matters, so readability makes for valuable code. In time, you'll come to appreciate Python's significant whitespace for the elegance it brings to your code and the ease with which you can read other's.

The Python Standard Library

As mentioned earlier, Python comes with an extensive standard library, an aspect of Python often referred to as batteries included. The standard library is structured as modules, a topic we'll discuss in depth later in this course. What's important at this stage is to know that you gain access to standard library modules by using the import keyword. The basic form of importing a module is simply the import keyword followed by a space and the name of the module. For example, let's see how we can use the standard libraries math module to compute square roots. At the Triple Arrow prompt, we type import math. Since import is a statement which doesn't return a value, Python doesn't print anything if the import succeeds and were immediately returned to the prompt. We can access the contents of the imported module by using the name of the module, followed by a dot, followed by the name of the attribute in the module that you need. Like many object-oriented languages, the dot operator is used to drill down into object structures. Being expert Pythonistas, we have inside knowledge that the math module contains a

function called SQRT. Let's try to use it. But how can we find out what other functions are available in the math module? The REPL has a special function, help, which can retrieve any embedded documentation from objects for which it has been provided, such a standard library modules. To get help, simply type help. We'll leave you to explore the first form for interactive help on your own time. We'll go for the second option and pass the math module as the object for which we want help. You can use the Space bar to page through the help. If you're on Mac or Linux, use the arrow keys to scroll up and down. Browsing through the functions, we can see that there is a math function for computing factorials. Press Q to exit the help browser and return us to the Python REPL. Practice using help to request specific help on the factorial function, press Q to return to the REPL. Let's use the factorial function, which accepts an integer and returns an integer. Notice how we need to qualify the function name with the name of the module containing it. This is generally good practice as it makes it abundantly clear where the function is coming from. That said, it can result in code that is excessively verbose. To see that, let's use factorial to compute how many ways there are to draw three fruit from a set of five fruit using some math we learned in school. This simple expression is quite verbose with all those references to the math module. The Python import statement has an alternative form that allows us to bring a specific function from a module into the current namespace. This is a good improvement, but it's still a little long winded for such a simple expression. A third form of the import statement allows us to rename the imported function. This can be useful for reasons of readability or to avoid a namespace clash. Useful as it is, we recommend that this feature be used infrequently and judiciously. Remember that when we used factorial alone, it returned an integer, but our more complex expression for combinations is returning a floating point number. This is because we've used pythons floating point division operator, the single forward slash. We can improve our expression since we know it will only ever return integral results by using Python's integer division operator, which is a double forward slash. What's notable is that many other programming languages would fail on the above expression for even moderate values of n. In most programming languages, regular garden variety signed integers can only store values less than 2 to the 31st power. However, factorials grow so fast that the largest factorial you can fit into a 32-bit signed integer is 12 factorial since 13 factorial is too large. In most widely-used programming languages, you would need more complex code or more sophisticated mathematics merely to compute how many ways there are to draw 3 fruit from a set of 13 fruits. Python encounters no such problems and can compute with arbitrarily large integers limited only by the memory in your computer. Let's try the larger problem of computing how many different pairs of fruit we can pick from 100 different fruits assuming we can lay our hands on so much fruit. Just to emphasize how large the size of the first term in that expression is, calculate 100

factorial on its own. This is a number vastly larger than even the number of atoms in the known universe with an awful lot of digits. If, like me, you're curious to know exactly how many digits, we can convert our integer to a text string and count the number of characters in it like this.

Summary

Congratulations! You've taken your first steps in Python, and you're well on your way to reading and writing much more sophisticated Python programs. In this module, we saw how to download and install Python on Windows, Linux and macOS. We covered starting your Python REPL. We evaluated some simple expressions in the REPL. We learned that in the REPL, the underscore symbol is bound to the result of the last evaluated expression. We saw how to make basic use of the print function, and we learned that the printed output is a side effect of the function, not a return value. We saw how to exit the REPL using Ctrl+Z on Windows and Ctrl+D on Linux and macOS. We were introduced to Python's use of significant whitespace. We learned that code blocks in python are initiated with a colon and comprise consecutive lines at the same indentation level. We looked at some of the advantages of significant whitespace, including clarity and consistency. We covered the basic rules for indentation in Python. On a less technical level, we covered some parts of Python's culture. We looked at the Zen of Python and saw that it could be printed by executing import this in the REPL, and we looked specifically at the idea that readability counts when writing Python code. We covered the basics of importing modules from the standard library, and we saw three forms of the import statement, importing an entire module, importing selected elements of a module, and renaming imported elements. We saw how to use Python's help system. Along the way, we saw how to use the factorial function from Python's standard math library. In the next module of Core Python: Getting Started, we'll look at Python's fundamental scalar types, integers, floats, nones, and bools, as well as some basic flow control constructs.

Scalar Types, Operators, and Control Flow

Overview

Now that you've got a functional Python REPL on your system, we can start to work with the fundamentals of the language. In this module of Core Python: Getting Started, you'll start to learn about Python's fundamental scalar types. We'll look at basic use of relational operators, and we'll introduce basic flow-control mechanisms. Python comes with a number of built-in data types. These include primitive scalar types like integers, as well as collection types like dictionaries. These built-in types are powerful enough to be used alone for many programming needs, and they can be used as building blocks for creating more complex data types. In this section, we cover the basic scalars, int for whole numbers; float for numbers with fractional parts; None, an important placeholder value; and bool, used for True and False values. We'll provide basic information about these now, showing their literal forms and how to create them. We've already seen quite a lot of Python integers in action. Python integers are signed, and have, for all practical purposes, unlimited precision, meaning they can contain as many digits as you need. Integer literals in Python are specified in decimal, and may also be specified in binary with the 0b prefix, in octal with the 0o prefix, or in hexadecimal with the 0x prefix. We can also construct integers by a call to the int constructor. This can convert from other numeric types, such as floats, to integers. Note that the rounding of integers is always towards zero. The int constructor can also convert strings to integers. You can even supply an optional number base when converting from a string. For example, to convert from base 3, provide the value 3 as the second argument to the int constructor. Floating point numbers are supported in Python by the float type. Python floats are implemented as IEEE-754 double-precision floating point numbers with 53 bits of binary precision. This is equivalent to between 15 and 16 significant digits in decimal. Any literal number containing a decimal point is interpreted by Python as a float. Scientific notation can be used, so for large numbers, such as the approximate speed of light in meters per second, 3 times 10 to the eighth, we can write 3e8, and for small numbers like Planck's constant, 1.616 times 10 to the negative thirty-fifth, we can enter 1.616e-35. Notice how Python automatically switches the display representation, that is, the format it prints to the REPL, to the most readable form. As with integers, we can convert to floats from other numeric or string types using the float constructor. We can pass int values to the float constructor, and we can pass strings. This is also how we create the special floating point values nan, or not a number, as well as positive infinity and negative infinity. One important rule to remember is that the result of any calculation involving int and float is promoted to a float. You can read more about Python's number types in Python's documentation. Python has a special null value called None, spelled with a capital N. None is frequently used to represent the absence of a value. The Python REPL never prints None results, so typing None into the REPL has no effect. None could be bounded to variable names just like any other object, and we can test whether an object is None by using Python's is operator. We

can see here that the result of the is operator in this case is True, which brings us conveniently onto the bool type. The bool type represents logical states, and plays an important role in several of Python's control flow structures, as we'll see shortly. As you probably expect, there are two bool values, True and False, both spelled with initial capitals. There is also a bool constructor which can be used to convert from other types to bool. Let's look at how it works. For integers, zero is considered Falsy and all other values Truthy. We see the same behavior with floats, where only zero is considered Falsy. When converting from collections, such as strings or lists, only empty collections are treated as Falsy. For lists, which we'll look at shortly, the empty list is Falsy, while any non-empty list is Truthy. Similarly, empty strings are Falsy, while any other strings are Truthy. It's worth noting that the bool constructor may not behave as you expect when passing in the strings True and False. Since both are non-empty strings, both result in True. These conversions to bool are important, because they are widely used in Python if statements and while loops, which accept bool values into the condition. We'll look at these constructs soon.

Relational Operators

Bool values are commonly produced by Python's relational operators, which can be used for comparing objects. These include value equality or equivalents, value inequality or inequivalence, less than, greater than, less than or equal to, and greater than or equal to. Two of the most widely used relational operators are Python's equality and inequality tests. These test whether two objects are equivalent or inequivalent, that is, whether one can be used in place of the other or not. We'll learn more about the notion of object equivalents later in the course, but for now, we'll just compare simple integers. Let's start by assigning or binding a value to the variable g. We test for equality with the == operator, and we test for inequality using the not equals operator, an exclamation point followed by an equal sign. We can also compared the order of quantities using the rich comparison operators. We check if one object is less than another with the less than operator. We can append an equal sign to this operator to test for less than or equal. Likewise, we check if an object is greater than another with the greater than operator, and as with less than, there is the related greater than or equal operator.

Control Flow

Now that we've examined some basic built-in types, we'll look at two important control flow structures, which depend on the conversions to the bool type, if statements and while loops. We'll start with if statements, also known as conditional statements. Conditional statements allow us to

branch execution based on the value of an expression. The form of the statement is the if keyword, followed by an expression, terminated by a colon to introduce a new block. Let's try this at the REPL. Remembering to indent four spaces within the block, we add some code to be executed if the condition is true. We terminate the blocked by entering a blank line. Because the condition is self-evidently true, the block executes, and we see the string, It's true!, printed to the REPL. Conversely, if the condition is false, the code in the block does not execute. The expression used with the if statement will be converted to a bool, just as if the bool constructor had been used, so explicitly constructing a bool in the if statement is exactly equivalent to using a bare string. Thanks to this useful shorthand, explicit conversion to bool using the bool constructor is rarely used in Python. The if statement supports the optional else-claus that goes in a block introduced by the else keyword followed by colon, which is indented to the same level as the if keyword. To start the else block, in this case, we just omit the indentation after the three dots. For multiple conditions, you might be tempted to nest if statements inside else blocks like this. Whenever you find yourself doing this though, you should consider using Python's lf keyword, which is a combined else-if. As the Zen of Python reminds us, flat is better than nested. This version is altogether easier to read.

While-loops

Python has two types of loop, for loops and while loops. We've already briefly introduced for loops back when we introduced significant white space, and we'll return to them soon, but right now we'll cover for loops. While loops in Python are introduced by the while keyword, which is followed by a Boolean expression. As with the condition for if statements, the expression is implicitly converted to a Boolean value, as if it had been passed to the bool constructor. The while statement is terminated by a colon because it introduces a new block. Let's write a loop at the REPL, which counts down from 5 to 1. We'll initialize a counter variable called C to 5, and keep looping until we reach zero. Another new language feature here is the use of the augmented assignment operator, a minus sign followed by an equal sign to subtract one from the value of C on each iteration. Similar augmented assignment operators exist for other basic math operators, such as plus and multiply. Because the condition, also called the predicate, will be implicitly converted to bool, just as if a call to the bool constructor present, we could replace the above code with the following version. This works because the conversion of the integer value of C to bool results in true until we get to zero, which converts to false. That said, to use this short form in this case might be described as unPythonic because, referring back to the Zen of Python, explicit is better than implicit, and we place a higher value on the readability of the first form over the

concision of the second form. While loops are often used in Python, where an infinite loop is required. We achieve this by simply passing true as the predicate expression to the while construct. Now you're probably wondering how to get out of this loop and regain control of your REPL. To do this, we press Ctrl+C. Python intercepts this to raise a special exception which terminates the loop. We'll be talking much more about what exceptions are and how to use them later in the course. Many programming languages support a loop construct which places the predicate test at the end of the loop rather than at the beginning. For example, C, C++, C #, and Java support the do while construct. Other languages have repeat until loops instead or as well. This is not the case in Python, where the idiom is to use, while true, together with an early exit facilitated by the break statement, the break statement jumps out of the loop and only the innermost loop, if several loops have nested, and then continues execution immediately after the loop body. Let's look at an example of break, introducing a few other Python features along the way. We start with a while true for an infinite loop. On the first statement of the while block, we used the built in input function to request a string from the user. We assigned that string to a variable called response. We now use an if statement to test whether the value provided is divisible by seven. We convert response to an integer, using the int constructor and then use the modulus operator, the percent symbol to divide by 7 and give the remainder. If the remainder is equal to 0, the response was divisible by 7, and we enter the body of the if block. Within the if block, now two levels of indentation deep, we start with eight spaces and use the break keyword. Break terminates the innermost loop, in this case the while loop, and causes execution to jump to the first statement after the loops. In our case, this is the end of the program. Enter a blank line that the three dots prompt to close both the if block and the while block. Our loop will start executing and will pause at the call to the input function waiting for us to enter a number. Let's try a few. As soon as we enter a number divisible by seven, the predicate becomes true. We enter the if block and then literally break out of the loop, ending the program and returning us to the REPL prompt.

Summary

We've covered quite a bit of ground in this module, so let's summarize what we've seen. We looked at the four built-in scalar types, int, float, none, and bool, and conversions between these types, and use of their literal forms. We looked at six relational operators used for equivalents and ordering. We demonstrated, structuring conditional code with the if-elif-else structures. We showed iterating with while loops, and we saw how the condition expression of the while loop is implicitly converted to a bool. We looked at how to intercept infinite loops with Ctrl+C, and we

learned that doing this generates a keyboard interrupt exception. We gave an example of how to break out of loops using the break statement. We observed that break only breaks us out of the innermost nested loop and that it takes us to the first statement immediately following the loop. Along the way, we looked at the augmented assignment operators for modifying objects, such as counter variables in place. We also looked at requesting text from the user with the built-in input function. In the next module of Core Python: Getting Started, we'll continue our exploration of Python's built-in types and control flow structures by looking at strings, lists, dictionaries, and for loops. We'll even be using Python to fetch some data from the web for processing.

Introducing Strings, Collections, and Iteration

Overview

Now that you have an initial understanding of Python's fundamental scaler types, you're ready to start exploring some of the collection types. In this module of Core Python: Getting Started, we'll look at a few of the most important collection types in Python, str, bytes, list, and dict. We'll introduce you to the for loop, a looping construct commonly used for iterating over collections. We'll apply all of this to build a small, but useful program that demonstrates the expressiveness of Python. Python includes a rich selection of collection types, which are often completely sufficient for even quite intricate programs without resorting to defining your own data structures. We'll give enough of an overview of some fundamental collection types now to allow us to write some interesting code. We'll also be revisiting each of these collection types together with a few additional ones later in the course. Let's start with these types, str, bytes, list, and dict. Along the way, we'll also cover Python's for loops.

String

Strings in Python have the data type str, spelled s-t-r, and we've been using them extensively already. Strings are sequences of Unicode code points, and for the most part, you can think of code points as being like characters, although they are not strictly equivalent. The sequence of characters in a Python string is immutable, meaning that once you've constructed a string, you can't modify its contents. Literal strings in Python are delimited by quotes, you could use single

quotes or double quotes. You must, however, be consistent. For example, you can't use single quotes on one side and double on the other, like this. Supporting both quoting styles allows you to easily incorporate the other quote character into the literal string without resorting to ugly escape character gymnastics. Notice that the REPL exploits the same, quoting flexibility when echoing the strings back to us. Beautiful text strings rendered in literal form, simple elegance. At first sight, support for both quoting styles seems to violate an important principle of Pythonic style from the Zen of Python. There should be one, and preferably only one, obvious way to do it. In this case, however, another aphorism from the same source, practicality beats purity, takes precedence. The utility of supporting two quoting styles is valued more highly than the alternative, a single quoting style combined with more frequent use of ugly escape sequences, which we'll encounter shortly.

String Literals

Adjacent literal strings are concatenated by the Python compiler into a single string, which, although at first it seems rather pointless, can be useful for a nicely formatted code, as we'll see later. If you want a literal string containing newlines, you have two options, use multiline strings or use escape sequences. First, let's look at multiline strings. Multiline strings are delimited by three quote characters rather than one. Here's an example using three double quotes. Notice how, when the string is echoed back to us, the newlines are represented by the \n escape sequence. We can also use three single quotes. As an alternative to using multiline quoting, we can just embed the control characters ourselves. To get a better sense of what we're representing, we can use print to see the string. If you're working on Windows, you might be thinking that newlines should be represented by the carriage return and newline couplet \r\n. There's no need to do that with Python. Since Python 3 has a feature called Universal Newline Support, which translates from the simple \n to the native newline sequence for your platform on input and output. You can read more about Universal Newline Support in PEP 278. We can use the escape sequences for other purposes, too, such as incorporating tabs with \t or allowing us to quote characters within strings by using \ double quote or \ single quote. See how Python is smarter than we are at using the most convenient quote delimiters, although Python will also resort to escape sequences when we use both types of quotes in a string. Because backslash has special meaning, to place a backslash in a string, we escape the backslash with itself. To reassure ourselves that there really is only one backslash in that string, we can print it. You can read more about escape sequences in the Python documentation at python.org. Sometimes, particularly when dealing with strings such as Windows file system paths or regular expression patterns, which use backslashes extensively, the

requirement to double up on backslashes can be ugly and error prone. Python comes to the rescue with its raw strings. Raw strings don't support any escape sequences and are very much what you see is what you get. To create a raw string, prefix the opening quote with a lowercase r. We can use the string constructor to create string representations of other types such as integers or floats. Strings in Python are what are called sequence types, which means they support certain common operations for querying sequences. For example, we can access individual characters using square brackets with an integer 0-based index. Note that in contrast to many programming languages, there is no separate character type distinct from the string type. The indexing operation we just used returns a full-blown string that contains a single character element, something we can test using Python's built-in type function. There will be more on types and classes later in the course. String objects also support a wide variety of operations implemented as methods. We can list those methods using help on the string type. Ignore all the hieroglyphics with underscores for now and page down until you see the documentation for the capitalized method. Press Q to quit the help browser, and we'll try to use that method. First, let's make a string that deserves capitalization, the proper noun of a capital city, no less. To call methods on objects in Python, we use the dot after the object name and before the method name. Methods are functions, so we must use the parentheses to indicate that the method should be called. Remember that strings are immutable, so the capitalized method didn't modify c in place, rather, it returned a new string. We can verify this by displaying c, which remains unchanged. You might like to spend a little time familiarizing yourself with the various useful methods provided by the string type. Finally, because strings are fully Unicode capable, we can use them with international characters easily, even in literals because the default source code encoding for Python 3 is UTF-8. For example, if you have access to Norwegian characters, you can simply enter this. Alternatively, you can use the hexidecimal representations of Unicode code points as an escape sequence prefixed by \u, which I'm sure you'll agree, is somewhat more unwieldy. Similarly, you can use the \x escape sequence, followed by a two-character hexidecimal string or an escaped octal string to include Unicode characters in a string literal. There are no such Unicode capabilities in the otherwise similar bytes type, which we'll look at next.

Bytes

Bytes are very similar to strings, except that rather than being sequences of Unicode code points, they are sequences of, well, bytes. As such, they are used for raw binary data and fixed-width single-byte character encodings such as ASCII. As with strings, they have a simple, literal form using quotes, the first of which is prefixed by a lower case b. There is also a bytes constructor, but

it's an advanced feature and we won't cover it in this fundamentals course. At this point, it's sufficient for us to recognize bytes literals, and understand that they support most of the same operations as string, such as indexing, which returns the integer value of the specified byte, and splitting, which you'll see returns a list of bytes objects. To convert between bytes and strings, we must know the encoding of the byte sequence used to represent the string's Unicode code points as bytes. Python supports a wide variety of encodings, a full list of which can be found at python.org. Let's start with an interesting Unicode string which contains all the characters of the 29-letter Norwegian alphabet, a pangram. We'll now encode that using UTF-8 into a bytes object. See how the Norwegian characters have each been rendered as pairs of bytes. We can reverse that process using the decode method of the bytes object. Again, we must supply the correct encoding. We can check that the result is equal to what we started with, and display it for good measure. This may seem like an unnecessary detail so early in the course, especially if you operate in an anglophone environment, but it's a crucial point to understand since files and network resources such as HTTP responses are transmitted as byte streams, whereas we often prefer to work with the convenience of Unicode strings.

List

Python lists, such as those returned by the string split method are sequences of objects. Unlike strings, lists are mutable, insofar as the elements within them can be replaced or removed, and new elements can be inserted or appended. Lists are a workhorse of Python data structures. Literal lists are delimited by square brackets, and the items within the list separated by commas. Here is a list of three numbers and a list of three strings. We can retrieve elements by using square brackets with a zero-based index, and we can replace elements by assigning to a specific element. See how lists can be heterogeneous with respect to the types of the objects. We now have a list containing a string, an integer, and another string. It's often useful to create an empty list, which we can do using empty square brackets. We can modify the list in other ways. Let's add some floats to the end of the list using the append method. There are many other useful methods for manipulating lists, which we'll cover in a later module. There's also a list constructor, which can be used to create lists from other collections such as strings. Finally, although the significant whitespace rules in Python can at first seem very rigid, there is a lot of flexibility. For example, if at the end of the line brackets, braces or parentheses are unclosed, you can continue on the next line. This can be very useful for long, literal collections, or simply to improve readability. See also how we're allowed to use an additional comma after the last element. This is an important maintainability feature.

Dict

Dictionaries are completely fundamental to the way the Python language works and are very widely used. A dictionary maps keys to values, and in other languages, is known as a map or an associative array. Let's look at how to create and use them in Python. Literal dictionaries are created using curly braces containing key-value pairs. Each pair is separated by a comma, and each key is separated from the corresponding value by a colon. Here, we use a dictionary to create a simple telephone directory. We can retrieve items by key using the square brackets operator and update the values associated with the key by assigning through the square brackets. If we assign to a key that has not yet been added, a new entry is created. Be aware that in Python versions prior to 3.7, the entries in the dictionary can't be relied upon to be stored in any particular order. As of Python 3.7, however, entries are required to be kept in insertion order. Similarly, to lists, empty dictionaries can be created using empty curly braces. We'll revisit dictionaries in much more detail in a later module

For-loop

Now that we have the tools to make some interesting data structures, we'll look at Python's second type of loop construct, the for loop. For loops in Python correspond to what are called for each loops in many other programming languages. They request items one by one from a collection, or more strictly, from an iterable series, but more on that later, and assign them in turn to a variable that we specify. Let's create a collection and use a for loop to iterate over it. If you iterate over dictionaries, you get the keys, which you can then use within the for loop body to retrieve values. Here we define a dictionary mapping string color names to hexadecimal integer color codes. Note that we used the ability of the built-in print function to accept multiple arguments. We passed the key and the value for each color separately. See also how the color codes returned to us are in decimal.

Putting it all Together

In this last section, before we summarize, we're going to write a longer snippet at the REPL. We're going to fetch some text data for some classic literature from the web using a Python standard library function called urlopen. To get access to urlopen, we need to import the function from the request module within the standard library urllib package. Next, we're going to call urlopen with a URL to our story, then create an empty list, which ultimately will hold all of the words from the text. Next, we open a for loop, which will work through the story. Recall that for loops request

items one by one from the term on the right of the in keyword, in this case, story, and assign them in turn to the name on the left, in this case, line. It so happens that the type of HTTP response represented by story yields successive lines of text when iterated over in this way. So the for loop retrieves one line of text a time from Dickens' classic. Note also that the for statement is terminated by a colon because it introduces the body of the for loop, which is a new block and hence a further level of indentation. For each line of text, we used the split method to divide it into words on whitespace boundaries, resulting in a list of words we call line words. Now, we use a second for loop nested inside the first to literate over this list of words, appending each in turn to the accumulating story words list. Now, we enter a blank line at the three dots prompt to close all open blocks. In this case, the inner for loop and the outer for loop will both be terminated. The block will be executed, and after a short delay, Python now returns us to the regular triple-arrow prompt. At this point, if Python gives you an error, such as a syntax error or indentation error, you should go back, review what you entered, and carefully re-enter the code until Python accepts the whole block without complaint. If you get an HTTP error, then you were unable to fetch the resource over the internet, and you should try again later, although it's worth checking that you typed the URL correctly. Finally, now that we're done reading from the URL, we need to close our handle to it, story. We can look at the words we read simply by asking Python to evaluate the value of story words. Here, we can see the list of words. Notice that each of the single-quoted words is prefixed by a lowercase letter B, meaning that we have a list of bytes objects where we would have preferred a list of strings. This is because the HTTP request transferred raw bytes to us over the network. To get a list of strings, we should decode the bytes string each line into Unicode strings. We can do this by inserting a call to the decode method of the bytes object and then operating on the resulting Unicode string. The Python REPL supports a simple command-line history, and by careful use of the up and down-arrow keys, we can re-enter our snippet. When we get to the line which needs to be changed, we can edit it using the left and right-arrow keys to insert the requisite call to decode. Then when we rerun the block and take a fresh look at story words, we should see we have a list of strings. We've just about reached the limits of what's possible to comfortably edit at the Python REPL. So in the next course module, we'll look at how to move this code into a Python module where it can be more easily worked with in a text editor.

Summary

There are a lot of details in this module, which may be difficult to remember all at once, but which you'll find you use very frequently when writing Python code. First, we looked at strings, in particular the various forms of quoting for single and multi-line strings. We saw how adjacent

string literals are implicitly concatenated. Python has support for universal newlines. So no matter what platform you're using, it's sufficient to use a single backslash n character, safe in the knowledge that it will be appropriately translated from and to the native newline during IO. Escape sequences provide an alternative means of incorporating new lines and other control characters into literal strings. The backslashes used for escaping can be a hindrance for Windows file system paths or regular expressions, so raw strings with the R prefix can be used to suppress the escaping mechanism. Other types, such as integers, can be converted to strings using the str constructor. Individual characters returned as one character strings can be retrieved using square brackets with integer zero-based indices. Strings support a rich variety of operations, such a splitting, through their methods. In Python 3, literal strings can contain Unicode characters directly in the source. The bytes type has many of the capabilities of strings, but it is a sequence of bytes rather than a sequence of Unicode code points. Bytes literals are prefixed with a lowercase b. To convert between string and bytes instances, we use the encode method of str and the decode method of bytes. In both cases, passing the encoding, which we must know in advance. Lists are mutable, heterogeneous sequences of objects. List literals are delimited by square brackets, and the items are separated by commas. As with strings, individual elements can be retrieved by indexing into a list with square brackets. In contrast to strings, Individual list elements can be replaced by assigning to the indexed item. Lists can be grown by appending to them, and they can be constructed from other sequences using the list constructor. Dictionaries associate keys with values. Literal dictionaries are delimited by curly braces. The key value pairs are separated from each other by commas, and each key is associated with its corresponding value with a colon. For loops take items one by one from an iterable object, such as a list, and binds a name to the current item. They correspond to what are called for-each loops in other languages. In the next module of Core Python: Getting Started, we'll look at functions and modules, Pythons fundamental tools for organizing code. These tools will facilitate writing larger programs and structure in your code into cohesive, reusable components.

Modularity

Overview

Modularity is an important property for anything, but trivial software systems as it gives us the power to make self-contained, reusable pieces, which can be combined in new ways to solve

different problems. In this module of Core Python: Getting Started, we'll see that in Python, as with most programming languages, the most fine-grained modularization facility is the definition of reasonable functions, we'll see how collections of related functions are typically grouped into source code files called modules, and we'll learn how modules can be used from other modules so long as we take care not to introduce circular dependencies. We'll learn more about a topic we've already seen, importing modules into the REPL. We'll show you how modules can be executed directly as programs or scripts. Along the way, we'll investigate the Python execution model to assure you have a good understanding of exactly when code is evaluated and executed. We'll round off by showing you how to use command-line arguments to get basic configuration data into your program and make your program executable. To illustrate this module, we'll be taking the code snippet for retrieving words from a web hosted text document that we developed at the end of the previous module and organizing it into a fully-fledged Python module.

Modules

Let's start with the snippet we worked with last time. Open a text editor, preferably one with syntax highlighting support for Python and configure it to insert four spaces per indent level when you press the Tab key. You should also check that your editor saves the file using the UTF-8 encoding, as that's what the Python 3 runtime expects by default. Let's get the snippet we wrote at the REPL at the end of the previous module into a text file called words.py. All Python source files used the .py extension. Now that we're using a text file for our code, we can pay a little more attention to readability. Let's put a blank line after the import statement. Save the file in the directory called corepy in your home directory. Switch to a console with your operating system's shell prompt and changed the new corepy directory. We can execute our module simply by calling python and passing the module's file name. Use python3 words.py on Mac or Linux, or python words.py on Windows. When you press Return, after a short delay, you'll be returned to the system prompt, not very impressive, but if you got no response, then the program is running as expected. If, on the other hand, you got an error, an HTTP error indicates there is a network problem, whilst other types of errors probably mean you have mistyped the code. Let's add another for loop to the end of the program to print out one word per line. If you run your module again, it will print out the downloaded words. This is much better. Now we have the beginnings of a useful program. Our module can also be imported into the REPL. Let's try that and see what happens. Start the REPL and import your module by typing import words. Note how when importing, we omit the file extension. The code in your module is executed immediately when

imported, maybe not what you expected and not very useful. To give us more control over when our code is executed and to allow it to be reused, we'll need to put our code in a function.

Functions

Let's quickly define a new function at the REPL to get the idea. Functions are defined using the def keyword followed by the function name, an argument list in parenthesis, and a colon to start a new block. The code inside the function block must be indented. We use the return keyword to return a value from the function, here returning an expression which evaluates to X squared. As we've seen previously, we call functions by providing the actual arguments in parentheses after the function name. Functions aren't required to explicitly return a value though, perhaps they produce side effects, such as this launch_missiles function. Here, the message we see printed is a side effect of calling print, not a return value from the function. Generally speaking, it's good practice to prefer functions which return values rather than cause effects, especially when the effects are as drastic as this. You can return early from a function by using the return keyword with no parameter. Both the return statement without a parameter, as well as the implicit return at the end of a function actually causes the function to return none. Remember, however, that the REPL doesn't display none results, so we don't see them. By capturing the returned object into a named variable, we can test for none. To reinforce these important ideas, let's look at another example. Here's the function nth_root, which can be used to compute the square root, cube roots, fourth roots, and so on of a number we supply, called the radicand. Internally, it uses the nth fix raised to the power, or exponentiation operator, which in Python is the double star. Let's use nth_root to compute the square root of 16, giving 4, and again, to compute the cube root of 27, which is 3. See how the calls to the function evaluate to the value of the returned expression. Our nth_root function has only one point of return. Now consider our ordinal_suffix function. Ordinals are numbers like 1st, 2nd, and 3rd, as opposed to cardinal numbers like 1, 2, and 3. The ordinal_suffix function is used to determine the suffixes ST, ND, RD, and so on, which are used for ordinal numbers in English. The function uses an if/elif structure to check the trailing digits of the string representation of the value it has passed and contains seven points of return. Now comes the ordinal function. This takes a value and concatenates its string representation with its ordinal_suffix. It's common and good practice to compose more complex functions out of more fundamental functions in this way. Function composition is the crucial technique for controlling complexity in programs. Lastly, we have the display_nth_root function. Given some radicand and n, it delegates to our nth_root function to do the mathematics, then performs the effect of printing a message, like the 2nd root of 16 is 4, delegating to ordinal and then onto ordinal_suffix

as necessary. This function has no explicit return statement at the end, so we can consider it to finish with an implicit return none. Let's try it at the REPL using it to display and compute the 4th \root of 64. Before we go on, we need to take a quick aside regarding terminology. In Python, many language features are implemented or controlled using specially named objects and functions. These special names generally have two leading and two trailing 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 pronouncing these names. Dunder is a portmanteau of the term double underscore, and we'll use it to refer to any name with leading and trailing double underscores. So, for example, when we talk about underscore underscore name underscore underscore, something you'll soon encounter, we'll say dunder name. These kinds of names play a big role in Python, so we'll be using this convention frequently.

__name__

With some background on functions in place and the terminal logical exactitude established, let's get back to organizing the code in our words module into function. We'll move all the code except the import statement, into a function called fetch words. You do that simply by adding the def statement and indenting the code below it by one extra level. Save the module, start a fresh Python REPL, and import your module again with import words. The module imports, but now the words are not fetched until we call the fetch_words function with words.fetch_words. This use of the dot is called qualifying the function name with the module name. Alternatively, we can import are specific function using a different form of the import statement, from words import fetch_words. Having imported the fetch_words function directly into our REPL session, which is itself a module, we can invoke fetch_words using its unqualified name. So far, so good. But what happens when we try to run our module directly from the operating system shell? Exit from the REPL with Ctrl+D from Mac or Linux, or Ctrl+Z for Windows, and run Python, passing the module file name. No words are printed, that's because all the module does now is define a function and then exit. The function is never called. What we'd prefer is that the module actually print something when we execute it. To make a module from which we can usefully import functions into the REPL and which can be run as a script, we need to learn a new Python idiom. As we mentioned earlier, one of the specially named variables is called dunder name, and it gives us the means to detect whether our module has been run as a script or imported into another module or the REPL. To see how, add print dunder name at the end of your module, outside of the

fetch_words function. Let's import the modified words module back into the REPL with import words. We can see that when imported, dunder name does indeed evaluate to the module's name. As a brief aside, if you import the module again in the same REPL, the print statement will not be executed. Module code is only executed once on first import. Now let's try running the module as a script from the operating system shell with Python 3 words.py. Now the special dunder name variable is equal to the string dunder main, which is also delimited by double underscores. That is, Python sets the value of dunder name differently, depending on how our module is being used. The key idea we're introducing here is that our module can use this behavior to decide how it should behave. We replaced the print statement with an if statement, which tests the value of dunder name. If dunder name is equal to the string dunder main, we execute our function. On the other hand, if dunder name is not equal to dunder main, the module knows it's being imported into another module, not executed, and so only defines the fetch_words function without executing it. We can now safely import our module without unduly executing our function, and we can usefully run our module as a script.

The Python Execution Model

It's important to understand the Python execution model and precisely when function definitions and other important events occur when a module is imported or executed. Here, we show execution of our Python module as it's imported in a graphical debugging environment. We step through the top-level statements in the module. What's important to realize here is that the def used for the fetch_words function isn't merely a declaration. It's actually a statement, which when executed in sequence with the other top-level model scope code, causes the code within the function to be bound to the name of the function. When modules are imported or run, all of the top-level statements are run, and this is the means by which the function within the module namespace are defined. We are sometimes asked about the difference between Python modules, Python scripts, and Python programs. Any .py file constitutes a Python module. But as we've seen, modules can be written for convenient import, convenient execution, or using the if dunder name = dunder main idiom, both. We strongly recommend making even simple scripts importable since it eases development and testing so much if you can access your code from within the REPL. Likewise, even modules, which are only ever meant to be imported in production settings, benefit from having executable test code. For this reason, nearly all modules we create have this form of defining one or more importable functions with a postscript to facilitate execution. Whether you consider a module to be a Python script or Python program is a matter of context and usage. It's certainly wrong to consider Python to be merely a scripting tool, in the vein of

Windows batch files or UNIX Shell scripts, as many large and complex applications are built exclusively with python.

Command Line Arguments

Let's refine our word fetching module a little further. First, we'll perform a small refactoring and separate the word retrieval in collection on the one hand from the printing of words on the other. This is because when importing, we'd rather get the words as a list. But when running directly, we prefer the words to be printed. Here we move the printing code into a new function called print_words. We'll also modify our main block to first call fetch_words and then call print_words with the list of words returned by fetch_words. Next, we'll extract the contents of our main block into a function called main. By moving this code into a function, we contest it from the REPL, something which isn't possible while it's in the module scope if block. We can now try these functions from the REPL. We'll use this opportunity to introduce a couple of new forms of the import statement. The first new form imports multiple objects from a module using a comma separated list. The parentheses are optional, but they do allow you to break this list over multiple lines if it gets too long. This form is perhaps the most widely used form of the import statement. The second new form imports everything from a module using an asterisk wild card. This latter form is recommended only for casual use at the REPL. It can wreak havoc in programs, since what is imported is now potentially beyond your control, opening yourself up to potential names space clashes at some future time. Having done this, we can get the words from the URL by calling fetch_words. We can also print any list of words by calling print_words. And indeed, we can run the main program. Notice that the print_words function isn't fussy about the types of the items in the list. It's perfectly happy to print a list of numbers. So perhaps print_words isn't the best name. In fact, the function doesn't mention lists either. It will happily print any collection that the for loop is capable of iterating over, such as a string. So let's perform another minor refactoring and rename this function to print_items, changing the variable names within the function to suit. We'll talk more about dynamic typing in Python, which allows this degree of flexibility, in the next module. Another obvious improvement to our module would be to replace the hard coded URL with the value we can pass in. Let's extract that value into an argument of the fetch_words function. Now, when running our module is a standalone program, we'll need to accept the URL as a command line argument. Access to command line arguments in Python is through the attribute of the sys module called argv, which is a list of strings. To use it, we must first import the sys module at the top of our program. Then we need to get the second argument with an index of one from the list. Now, when we run the program, it works as expected. This

looks fine until we realize that we can't usefully test main any longer from the REPL because it refers to sys.argv sub 1, which is unlikely to have any useful value in that environment. The solution is to allow the argument list to be passed as a formal argument to the main function, using sys.argv as the actual parameter in the if dunder name equals dunder main block. Testing from the REPL again, we can now pass any URL we want into main. For more sophisticated command line processing, we recommend you look at the Python standard library arg parse module or the inspired third-party doc opt module.

Moment of Zen

You'll notice that our top level functions have two blank lines between them. This is conventional for modern Python code. Two between functions, that is the number of lines PEP8 recommends. According to the PEP8 style guide, it's customary to use two blank lines between module level functions. We find this convention has served us well, making code easier to navigate. We use single blank lines for logical breaks within functions.

Docstrings

We saw previously how it was possible to ask at the REPL for help on Python functions. Let's look at how to add this self-documenting capability to our own module. API documentation in Python uses a facility called docstrings. Docstrings are literal strings, which occur as the first statement within a named block, such as a function or module. Let's document the fetch_words function. We use triple-quoted strings, even for single-line docstrings because they can be easily expanded to add more detail. One Python convention for docstrings is documented in PEP 257, although it is not widely adopted. Various tools, such as Sphinx, are available to build HTML document from Python docstrings, and each tool mandates its preferred docstring format. Our preference is to use the form presented in Google's Python Style Guide since it is amenable to being machine parsed while still remaining readable at the console. Now we'll access this help from the REPL. We'll add similar docstrings to our other functions, and we'll add one for the module itself. Module docstrings should be placed at the beginning of the module before any statements. Now, when we request help on the module as a whole, we get quite a lot of useful information, including the module docstring and each function docstring.

Comments

We believe docstrings are the right place for most documentation in Python code. They explain how to consume the facilities your model provides rather than how it works. Ideally, your code should be clean enough that ancillary explanation is not required. Nevertheless, it's sometimes necessary to explain why a particular approach has been chosen or a particular technique used, and we could do that using Python comments. Comments in Python begin with a hash, and they continue to the end of the line. Let's document the fact that it might not be immediately obvious why we're using sys.argv sub 1 rather than sys.argv sub 0.

Shebang

It's common on UNIX-like systems that have the first line of a script include a special comment rather wonderfully called a shebang. This begins with the usual hash, as for any other comment, followed by an exclamation mark, shhh bang! This allows the program loader to identify which interpreter should be used to run the program. Shebangs have an additional purpose of conveniently documenting at the top of a file whether the Python code therein is Python 2 or Python 3. The exact details of your shebang command depend on the location of Python on your system. Typical Python 3 shebangs used the UNIX env program to locate Python 3 on your path environment variable, which importantly, is compatible with Python virtual environments. On Mac or Linux, we must mark our script is executable using the command chmod +x words.py before the shebang will have any effect. Having done that, we can now run our script directly. Since Python 3.3, Python on Windows also supports the use of the shebang to make Python scripts directly excusable with the correct version of the Python interpreter, even to the extent that shebangs that look like they should only work on a UNIX-like system will work as expected on Windows. This works because Windows Python distributions now use a program called ylauncher. Pylauncher, the executable for which is called simply py.exe, will parse the shebang and locate the appropriate version of Python. For example, on Windows in CMD, words.py followed by a URL will be sufficient to run your script with Python 3, even if you also have Python 2 installed. In PowerShell, the equivalent is almost the same, .\words.py followed by the URL. You can read more about pylauncher in PEP 397.

Summary

Let's review what we've covered in this module. We learned that Python code is generally placed in .py files called modules. We saw how modules can be executed directly by passing them as the first argument to the Python interpreter, and that modules can also be imported into the REPL, at

which point all top-level statements in the module are executed in order. We looked at how named functions are defined using the def keyword followed by the function name and the argument list in parentheses, and how we can return objects from functions using the return statement. Related to this, we learned that return statements without a parameter return None, as does the implicit return at the end of every function body. We saw how we can detect whether a module has been imported or executed by examining the value of the special __name__ variable. If it is equal to the string __main__, then are module has been executed directly as a program. We learned how to use this to write the if __name__ == '__main__' check. By calling a function when this check succeeds, we can make our module both usefully importable and executable, an important testing technique even for short scripts. We saw that module code is only executed once, on first import. We learned that the def keyword is a statement which binds executable code to a function name. We investigated how command line arguments can be accessed as a list of strings accessible through the argv attribute of the sys module. The zeroth command line argument is the script file name, so the item at index 1 is the first true argument. We saw that Python's dynamic typing means our functions can be very generic with respect to the type of arguments. We looked into using literal strings as the first line of a function definition to form the function's docstring. They're typically triple quoted multi-line strings containing usage information. We learned that function documentation provided in docstrings can be retrieved using help in the REPL. Similar to function docstrings, we saw that module docstrings should be placed near the beginning of the module prior to any Python statements such as import statements. We covered how comments in Python commence with the hash character and continue to the end of the line. Finally, we looked at how the first line of the module can contain a special comment called a shebang, allowing the program loader to launch the correct Python interpreter on all major platforms. In the next module of Core Python: Getting Started, we'll dig into Python's object model, looking at how values are passed to and returned from functions. We'll investigate the nature of dynamic typing in Python, and focus on rules for variable scope. Thanks for watching, and we'll see you in the next module.

Objects and Types

Overview

While you can get a long way in Python with fairly shallow understanding of its underlying model, we find that even just a small understanding of its deeper structure can yield deep insight, making you more productive and helping you design better programs. In this module of Core Python: Getting Started, we'll seek to understand the Python object model. We'll see that the notion of named references to objects is key to how Python works. We'll discuss the important difference between value equality and identity equality. We'll see how argument passing and returning from functions in Python fits into the object model. We'll investigate Python's type system. We'll look at how Python uses scopes to limit access to names in a program. And we'll introduce you to a core insight for understanding Python programs, the idea that everything is an object. The topics in this module may seem abstract or even simplistic, but if you internalize these concepts, you'll find that you're able to reason about Python much more fluidly and with greater precision. We've already talked about and used variables in Python, but what exactly is a variable? What's going on when we do something as straightforward as assigning an integer to a variable? In this case, Python creates an int object with a value of 1000, an object reference with the name x, and arranges for x to refer to the into 1000 object. If we now modify the value of x with another assignment, what does not happen is a change in the value of the integer object. Integer objects in Python are immutable and cannot be changed. In fact, what happens is that Python creates a new immutable integer object with the value 500 and redirects the x reference to point to the new object. We now have no way of reaching the int 1000 object, and the Python garbage collector will reclaim it at some point. When we assign from one variable to another, we're really assigning from one object reference to another object reference, so both references now refer to the same object. If we now reassign x, we have x referring to an int 3000 object and y referring to a separate int 500. There is no work for the garbage collector to do, because all objects are reachable from live references. Let's dig a little deeper using the built-in id function. Id returns an integer identifier that is unique and constant for the lifetime of an object. Let's rerun the previous experiment using id. First we'll assign a to the integer 496 and check its id. Then we'll assign b to 1729, and see that it has a different id. If we now make b refer to the same object as a, we'll see that they have the same id. Note that the id function is seldom used in production Python code. Its main use is in object model tutorials such as this one, and as a debugging tool. Much more commonly used than the id function is the is operator, which tests for equality of identity. That is, it tests whether two references refer to the same object. We've already met the is operator earlier in the course when we tested for None. Even operations which seem naturally mutating in nature are not necessarily so. Consider the augmented assignment operator. If we create an integer and then increment it by 2, we see that the id of the incremented integer is different from the original. Now let's look at that pictorially. We start with t referring to an int 5 object. Augmented

assignment creates an int 2 without assigning a reference to it. It then adds the int 2 with the int 5 to create a new int 7. Finally, it assigns t to the int 7, and the remaining ints are garbage collected. Python objects show this behavior for all types. A core rule to remember is this, the assignment operator only ever binds objects to names. It never copies an object to a value. Let's look at another example using mutable objects, lists. We create a list object with three elements, binding the list object to a reference named r. We then assign r to a new reference s. When we modify the list referred to by s, by changing the middle element, we see that the r list has changed as well. This happens since the names s and r, in fact refer to the same object, which we can verify with the is operator. Let's see that again with a diagram. First we assign r to a new list. Then we assign s to r, creating a new name for the existing list. If we modify s, we also modify r, because we're modifying the same underlying object. S is r is true, because both names refer to the same object. If you want to create an actual copy of an object such as a list, other techniques must be used, which we'll look at later. It turns out that Python doesn't really have variables in the metaphorical sense of a box holding a value. It only has named references to objects, and the references behave more like labels, which allow us to retrieve objects. That said, it's still common to talk about variables in Python. We will continue to do so, secure in the knowledge that you now understand what's really going on behind the scenes. Let's contrast the behavior of the is operator with the test for value equality, or equivalence. We'll create two identical lists. First, we'll test them for equivalence with the double equals operator. Then we'll test them for identity equality with is. Here we see that p and q refer to different objects, but that the objects they refer to have the same value. Of course, an object should almost always be equivalent to itself. Here's how that looks pictorially. We have two separate list objects, each with a single reference to it. The values contained in the lists are the same, that is, they are equivalent or value equal, even though they have different identities. Value equality and identity are fundamentally different notions of equality, and it's important to keep them separate in your mind. It's also worth noting that value comparison is something that is defined programmatically. When you define types, you can control how that class determines value equality. In contrast, identity comparison is defined by the language, and you can't change that behavior.

Passing Arguments and Returning Values

Now let's look at how all this relates to function arguments and return values. Let's define a function at the REPL, which appends a value to a list and prints the modified list. First, we'll create a list. Then we'll make a function, modify, which appends to and prints the list. The function accepts a single formal argument named k. We then call modify, passing our list m as the actual

argument. This indeed prints the modified list with four elements. But what does our list reference outside the function now refer to? The list referred to by m has been modified because it is the self same list referred to by k inside the function. When we pass an object reference to a function, we're essentially assigning from an actual argument reference, in this case m, to the formal argument reference, in this case k. As we've seen, assignment causes the reference being assigned to to refer to the same object as the reference being assigned from. This is exactly what's going on here. If you want the function to modify a copy of an object, it's the responsibility of the function to do the copying. Let's look at another instructive example. First will define a new list, then will define a function which replaces the list. Now we'll call replace with f as the argument. It prints the new list, which is much as we'd expect. However, what's the value of f after the call? F still refers to the original unmodified list. This time, the function did not modify the object that was passed in. What's going on? Well, the object reference named f was assigned to the formal argument named g, so g and f did indeed refer to the same object, just as in the previous example. However, on the first line of the function, we reassigned the reference g to point to a newly constructed list 17, 28, 45. So within the function, the reference to the original list 14, 23, 37, was overwritten, although the original list was still pointed to by the f reference outside the function. So we've seen that it's quite possible to modify the objects through function argument references, but also possible to rebind the argument reference to new values. If you wanted to change the contents of the list and have the changes seen outside the function, you could modify the contents of the list by writing a function that replaces each element of this list in place. Now we define f and pass it into replace_contents. And indeed, the contents of f have been modified by the function. Function arguments are transferred by what is called pass-by-object-reference. This means that the value of the reference is copied into the function argument, not the value of the referred to object. No objects are copied. The return statement uses the same pass-by-object-reference semantics as function arguments. We can demonstrate this by writing a simple function that just returns it's only argument. Create an object such as a list and pass it through this simple function. We see that it returns the very same object we passed in, showing that no copies of the list were made.

Function Arguments

Now that we understand the distinction between object references and objects, we'll look at some more capabilities of function arguments. The formal function arguments specified when a function is defined with the def keyword are a comma-separated list of the argument names. These arguments can be made optional by providing default values. Consider a function which

prints a simple banner to the console. This function takes two arguments, the second of which is provided with a default value, in this case a hyphen, in a literal string. Since we've given it this default value, callers can choose whether they want to pass their own value for border or use the default. Note that when we define functions using default arguments, the parameters with default arguments must come after those without defaults. Otherwise, we will get a syntax error. Within the body of the function. We multiply our border string by the length of the message string. This shows how we can determine the number of items in a Python collection using the built-in lend function. Secondly, it shows how multiplying a string, in this case the single-character string border, by an integer results in a new string containing the original string repeated a number of times. We use that feature here to make a string equal in length to our message. We then print the full width border, the message, and the border again. When we call our banner function, we don't need to supply the border string because we provided a default value. We can see that the default border of hyphens has been created. However, if we do provide an optional argument, it's used. In production code, this function call is not particularly self documenting. We can improve that situation by naming the border argument at the call site. In this case, the message string is called a positional argument and the border string a keyword argument. The actual positional arguments are matched up in sequence with the formal arguments, that is by position, whereas the keyword arguments are matched by name. If we use keyword arguments for both of our parameters, we have the freedom to supply them in any order. Remember though that all keyword arguments must be specified after the positional arguments. It's crucial to have an appreciation of exactly when the expression provided as a default value is evaluated. This will help you avoid a common pitfall, which frequently ensnares newcomers to Python. Let's examine this question closely using the Python standard library time module. We can easily get the time as a readable string by using the ctime function of the time module. Let's write a function, which uses a value retrieved from ctime as a default argument value. So far, so good. But notice what happens when you call show_default again a few seconds later, and again. The displayed time never progresses. Recall how we said that def is a statement that, when executed, binds a function definition to a function name? Well, the default argument expressions are evaluated only once when the def statement is executed. Normally, when the default is a simple, immutable constant, such as an integer or a string, this causes no problems. But it can be a confusing trap for the unwary that usually shows up in the form of using mutable collections as argument defaults. Let's take a closer look. Consider this function, which uses an empty list as a default argument. It accepts a menu, which will be a list of strings, appends the item spam to the list, and returns the modified menu. Let's create a simple breakfast of bacon and eggs. Naturally, we'll want to add Spam to it. We'll, do something similar for our lunch of baked beans. Nothing

unexpected so far. But look what happens when you rely on the default argument by not passing an existing menu. When we append spam to the default value of the empty menu, we get just spam. Let's do that again. When we exercise the default argument value a second time, we get two spams, and three and four. What's happening here is that the empty list used for the default argument is created exactly once, when the def statement is executed. The first time we fall back on the default, this list has spam added to it. When we use the default a second time, the list still contains that item, and a second instance of spam is added to it, making two, ad infinitum, or perhaps ad nauseum would be more appropriate. The solution to this is straightforward, but perhaps not obvious. Always use immutable objects such as integers or strings for default values. Following this advice, we can solve this particular case by using the immutable None object as a sentinel. Now our function needs to check if menu is none and provide a newly constructed empty list, if so. The rest of the function behaves as before. Now, if we call the function repeatedly with no arguments, we get the same menu, one order of spam each time.

Python's Type System

Programming languages can be distinguished by several characteristics, but one of the most important is the nature of their type system. Python could be characterized as having a dynamic and strong type system. Let's investigate what that means. Dynamic typing means the type of an object reference isn't resolved until the program is running, and needn't be specified up front when the program is written. Take a look at this simple function for adding two objects. Nowhere in this definition do we mention any types. We can use add with integers, floats, strings, or indeed any type for which the addition operator has been defined. These examples illustrate the dynamism of the type system. The two arguments, a and b, of the add function can reference any type of object. The strength of the type system can be demonstrated by attempting to add types for which addition has not been defined, such as strings and floats. This produces a type error, because python will not, in general, perform implicit conversions between object types, or otherwise attempt to coerce one type to another. The exception to this rule is the conversion of if statement and while loop predicates to bool.

Scopes

As we've seen, no type declarations are necessary in Python, and variables are essentially just untyped name bindings to objects. As such, they can be rebound or reassigned as often as necessary, even to objects of different types. But when we bind the name to an object, where is

that binding stored? To answer that question, we must look at scopes and scoping rules in Python. There are four types of scope in Python arranged in the hierarchy. Each scope is a context in which names are stored and in which they could be looked up. The four scopes, from narrowest to broadest, are local, names defined inside the current function; enclosing, names defined inside any and all enclosing functions, this scope isn't important for the contents of this Python fundamentals course; global, names defined at the top level of a module, each module brings with it a new global scope; built-in, names built into the Python language through the special built-ins module. Together, these scopes comprise the LEGB rule. Names are looked up in the narrowest relevant context. It's important to note that scopes in Python do not correspond to the source code blocks as demarcated by indentation. For loops and the like do not introduce new nested scopes. Consider our words.py module. It contains the following global names, main bound by deaf main, sys bound by import sys, dunder name provided by the Python runtime, urlopen bound by from urllib.request import urlopen, fetch_words bound by def fetch_words, print_items bound by def print_items. Module scope name bindings are typically introduced by import statements and function or class definitions. It's possible to use other objects at module scope, and this is typically used for constants, although it can be used for variables. Within the fetch_words function, we have six local names, word bound by the inner for loop, line_words bound by assignment, line bound by the outer for loop, story_words bound by assignment, url bound by the formal function argument, and story bound by assignment. Each of these is brought into existence at first use and continues to live within the function scope until the function complete, at which point, the references will be destroyed. Very occasionally, we need to rebind a global name, that is, one defined at module scope, from within a function. Consider the following simple module, it initialize is count to 0 at module scope. The show_count function simply prints the value of count, and set_count binds the name count to a new value. When show_count is called, Python looks up the count name in the local namespace, doesn't find it, so it looks it up in the next most outer namespace, in this case, the global module namespace, where it finds the name_count and prints the referred to object. Now we call set_count with a new value and show_count again. You might be surprised that show_count displays 0 after the call to set_count 5, so let's work through what's happening. When we call set_count, the assignment, count = c, binds the object referred to by the formal argument, c, to a new name, count, in the innermost namespace context, which is the scope of the current function. No look-up is performed for the global_count at module scope. We've created a new variable which shadows and thereby prevents access to the global of the same name. To avoid this situation, we need to instruct Python to consider use of the count name in the set_count function to resolve to the count in the module namespace. We can do this by using the global keyword. Let's modify set_count to do so.

The additional call to show_count still behaves as expected. Calling set_count, however, does now modify the count preference at module scope. This is the behavior we want.

Moment of Zen

Special cases aren't special enough to break the rules. We follow patterns, not to kill complexity, but to master it. As we have shown, all variables in Python are references to objects, even basic types, such as integers. This thorough approach to object orientation is a strong theme in Python, and practically everything in Python is an object, including functions and modules.

Everything is an Object

Let's go back to our words module and experiment with it further at the REPL. On this occasion, we'll import just the module. The import statement binds a module object to the name words in the current name space. We can determine the type of any object by using the type built in function. If we want to see the attributes of an object, we can use the dir built in function in a Python interactive session to introspect it. The dir function returns assorted list of the module attributes, including the ones we defined, such as the function fetch words, any imported names, such as sys and urlopen, and various special attributes delimited by double underscores, such as dunder name and dunder doc, which reveal the inner workings of Python. We can use the type function on any of these attributes to learn more about them. For instance, we could see that fetch words is a function object. We can in turn call dir on the function to reveal its attributes. We see that function objects have many special attributes to do with how Python functions are implemented behind the scenes. For now, we'll just look at a couple of simple attributes. As you might expect, words dot fetch_words dot dunder name is the name of the function object as a string. Likewise, words dot fetch_words dot dunder doc is the doc string we provided for the function. This gives us some clues as to how the built in help function might be implemented.

Summary

We've covered a lot of important concepts about how the Python language works in this module. Let's summarize what we've been over. It's better to think of Python working in terms of named references to objects rather than variables and values. Assignment doesn't put a value in a box. It attach is a name tag to an object. Assigning from one reference to another puts two name tags on the same object. The Python garbage collector will reclaim unreachable objects, those objects with no name tag. The id function returns a unique and constant identifier, but should rarely, if

ever, be used in production. The is operator determines equality of identity, that is, whether two names refer to the same object. We can test for equivalents using the == operator. Function arguments are passed by object reference, so functions can modify their arguments if they are mutable objects. If a formal argument is rebound through assignment, the reference to the passed in object is lost. To change a mutable argument, you should replace its contents rather than replacing the whole object. The return statement also passes by object reference, no copies are made. Function arguments can be specified with defaults. Default argument expressions are evaluated only once when the def statement is executed. Python uses dynamic typing, so we don't need to specify reference types in advance. Python uses strong typing. Types are not coerced to match. Python reference names are looked up in one of four nested scopes according to the LEGB rule, local to functions, in enclosing functions, and the global or module namespace, and built-ins. Global references could be read from a local scope. Assigning to a global reference from a local scope requires that the reference be declared global using the global keyword. Everything in Python is an object, including modules and functions. They could be treated just like other objects. The import and def keywords result in binding to named references. The built-in type function could be used to determine the type of an object. The built-in dir function can be used to introspect an object and return a list of its attribute names. The name of a function or module object can be accessed through its dunder name attribute. The doc string for a function or module object can be accessed through its dunder doc attribute. In passing, we also saw that we can use len to measure the length of a string. If we multiply a string by an integer, we get a new string with multiple copies of the operand string. This is called the repetition operation. In the next module of Core Python: Getting Started, we'll revisit a few collections that we've already met for a deeper look. We'll also introduce you to a few new ones, tuple, range, and set, as well as the concept of protocols which unite the various collection types. Thanks for watching, and we'll see you in the next module.

Built-in Collections

Overview

Python comes with a powerful suite of built-in collection types, several of which you've seen already in earlier modules. To be truly fluent in Python, you need to be familiar with all of these types and how to use them. So in this module, we'll take another deeper look at the collection

types you already know, str, list, and dict. We'll introduce you to some new collection types. First, we'll look at tuple, an immutable sequence of objects. Then, we'll cover range, which represents arithmetic progressions of integers. Finally, we'll see set, immutable collection of unique, immutable objects. We'll round off with an overview of the protocols that unite these collections, which allow them to be used in consistent and predictable ways. First up is tuple.

Tuples

Tuples in Python are immutable sequences of arbitrary objects. Once created, the objects within them cannot be replaced or removed, and new elements cannot be added. Tuples have a similar syntax to lists, except that they are delimited by parentheses rather than square brackets. Here's a literal tuple containing a string, a float, and an integer. We can access the elements of a tuple by 0-based index using square brackets. We can determine the number of elements in the tuple using the built-in len function, and we can iterate over it using a for loop. We can concatenate tuples using the plus operator and repeat using the multiplication operator. Since tuples can contain any object, it's perfectly possible to have nested tuples. We use repeated application of the indexing operator to get to the inner elements of such nested collections. Sometimes a single element tuple is required. To write this, we can't just use a simple number in parentheses. This is because Python parses that as an integer enclosed in the precedence controlling parentheses of a math expression. To create a single element tuple, we make use of the trailing comma separator, which you'll recall, we're allowed to use when specifying literal tuples, lists, and dictionaries. A single element with a trailing comma is parsed as a single element tuple. This leaves us with the problem of how to specify an empty tuple. In actuality, the answer is simple. We just used empty parentheses. In many cases, the parentheses of literal tuples may be omitted. This feature is often used when returning multiple values from a function. Here we make a function to return the minimum and maximum values of a sequence, the hard work being done by two built-in functions, min and max. Returning multiple values as a tuple is often used in conjunction with a wonderful feature of Python called tuple unpacking. Tuple unpacking is a destructuring operation, which allows us to unpack data structures into named references. For example, we can assign the result of our minmax function to two new references like this. When we print the two objects, we see that the references have indeed been unpacked from the tuples returned from the function. Unpacking also works with nested tuples. Here we assigned from a triply nested tuple of integers to a triply nested tuple of references. As before, we can see that each of the references has been assigned to the corresponding value from the original tuple. This support for unpacking leads to the beautiful Python idiom for swapping two or more variables. First, we'll create two references,

a and b, referring to the strings jelly and bean, respectively. Then we use the form a, b = b, a. This first packs a and b into a tuple on the right side of the assignment. It then unpacks the tuple on the left, reusing the names a and b. If we examine a and b, we can see that they have been swapped. Should you need to create a tuple from an existing collection object, such as a list, you can use the tuple constructor. You can also do this with a string, or indeed, any type over which you can iterate. Finally, as with most collection types in Python, we can test for containment using the in operator. Similarly, we can test for non-membership with the not in operator.

Strings

We covered strings at some length already, but we'll take time now to explore their capabilities in a little more depth. As with any Python sequence, we can determine the length of a string with the built-in len function. Here we see that the name of the longest train station in the UK contains a whopping 58 characters. Concatenation of strings is supported using the plus (+) operator. We can create the string, Newfoundland, by contaminating the strings New, found, and land. We can also use the related augmented assignment operator. Here starting with the string New, we incrementally add found and land. Remember that strings are immutable. So here the augmented assignment operator is binding a new string object to s on each use. The illusion of modifying s in place is achievable because s is a reference to an object, not an object itself. While the plus (+) operator is intuitive, you should prefer the join method for joining large numbers of strings because it is substantially more efficient. This is because concatenation using the addition (+) operator or it's augmented assignment version can lead to the generation of large numbers of temporaries with consequent costs for memory, allocations, and copies. Join is a method on the string class, which takes the collection of strings as an argument and produces a new string by inserting a separator between each of them. An interesting aspect of join is how the separator is specified. It is the string on which join is called. As with many parts of Python, an example is the best explanation. To join a list of HTML color code strings into a semicolon-separated string, construct a string containing semicolon and call join on it, passing in the list of color strings to be joined as an argument. We can then split the colors up again using the split method. We've already encountered this method, but this time we're going to provide it's optional argument. A widespread and fast idiom for contaminating together a collection of strings is to join using an empty string as the separator. The way may not be obvious at first. To concatenate, invoke join on empty text. Something from nothing. This use of join is often confusing to the uninitiated, but with use, the approach taken by Python will be appreciated as natural and elegant. Another very useful string method is partition, which divides a string into three sections, the part before the

separator, the separator itself, and the part after the separator. Partition returns a tuple, so this is commonly used in conjunction with tuple unpacking. Here we partition the elements of a travel plan into its parts. Often we're not interested in capturing the separator value, so you might see the underscore variable name used. This is not treated in a special way by the Python language, but there's an unwritten convention that the underscore variable is for unused or dummy values. This convention is supported by many Python-aware development tools, which will suppress unused variable warnings for underscore. One of the most interesting and frequently used string methods is format. This supersedes, although does not replace, the string interpolation technique used in older versions of Python, which we do not teach here. The format method can be usefully called on any string containing so-called replacement fields, which are surrounded by curly braces. The objects provided as arguments to format are converted to strings and used to populate these fields. Here's an example where the arguments to format the string Jim and the integer 32 are inserted into the format string. The field names, in this case 0 and 1, are matched up with the positional arguments to format, and each argument is converted to a string. The field name may be used more than once. Here we use the first argument to format twice. However, if the field names are used exactly once and in the same order as the arguments, the field number can be omitted. If keyword arguments are supplied to format, then named fields can be used instead of ordinals. Here the keywords latitude and longitude are inserted into the corresponding named replacement fields. It's possible to index into sequences using square brackets inside the replacement field. Here we index into a tuple in the replacement fields. You can even access object attributes. Here we pass the whole math module to format, using a keyword argument, remember, modules are objects, then access two of its attributes from within the replacement fields. Format strings also give us a lot of control over field alignment and floating point formatting. Here are the same values with the constants displayed using only three decimal places. While the format method we've just covered is quite powerful and is generally preferable to its predecessors, it could be quite verbose, even for relatively simple cases. Consider this example. We assigned 4 times 20 to the name value. We then interpolate value into a string with format, using the keyword argument matching feature. Here we have to mention the name value three times. Of course, this example could be made shorter by removing value from the brackets in the string and not using keyword arguments to format. But in larger, more complex interpolations, we would want to keep those elements in place for readability and maintainability. To address this, PEP 498 from which this example is directly drawn, introduces a new string formatting approach called literal string interpolation or, more commonly, f-strings. F-strings are available in Python 3.6 and later, and in the words of PEP 498, they provide a way to embed expressions inside literal strings using a minimal syntax. An f-string is like a normal string literal,

except that it is prefixed with the letter f. Inside the string literal, Python expressions can be embedded inside curly braces, and the results of these expressions will be inserted into the string at runtime. Let's rework our previous example using f-strings. We again assign 4 times 20 to the name value. We then use an f-string to insert value into a string. Here instead of needing to pass value into a method, the f-string simply evaluates it as a normal Python expression, inserting the result into the resulting string. Because f-strings allow you to use any Python expression, you're not limited to using simple named references. You can, for example, call functions. First, let's import the datetime module. Now we use an f-string to report the current time calling datetime.datetime.now to get the time, then formatting it with isoformat. We can rewrite the math constants example from the previous section by simply accessing math.pi and math.e from within the f-string. This then lets us demonstrate that, like format, f-strings also support floating point formatting. To print these constants with three places of precision, we can put a colon after the expression in the f-string followed by the format specifier. These are the essentials of f-strings, and this may be all that you ever need to use. There's quite a bit more to know about them, though, and we'll cover f-strings in greater depth in later courses in the core Python series. We recommend you spend some time familiarizing yourself with the other string methods. Remember, you can find out what they are by simply passing str to help.

Ranges

Let's move on and look at range, which really is a collection rather than a container. A range is a type of sequence used for representing an arithmetic progression of integers. Ranges are created by calls to the range constructor, and there is no literal form. Most typically, we supply only the stop value, and Python defaults to a starting value of 0. Ranges are sometimes used to create consecutive integers for use as loop counters. Note that the stop value supplied to range is 1 past the end of the sequence, which is why the previous loop didn't print 5. We can also supply a starting value if we wish by passing two arguments to range. Wrapping this in a call to the list constructor is a handy way to force production of each item. This so-called half open range convention, with the stop value not being included in the sequence, may seem strange at first, but it actually makes a lot of sense if you're dealing with consecutive ranges because the end specified by one range is the start of the next one. Range also supports a step argument. Here we count from 0 to 9 by 2's. Note that in order to use it, you must supply all three arguments. Range is curious in that it determines what its arguments mean by counting them. Providing only one argument means the argument is a stop value, two arguments are start and stop, and three arguments are start, stop, and step. Python range works this way so the first argument, start, can

be made optional, something which isn't normally possible. Furthermore, range doesn't support keyword arguments. You might almost describe it as un-Pythonic. At this point, we're going to show you another example of poorly styled code, except this time it's one you can, and should, avoid. Here's a poor way to print the elements in a list, by constructing a range over the length of the list and then indexing into the list on each iteration. Although this works, it's most definitely un-Pythonic. Instead, always prefer to use iteration over objects themselves. If, for some reason, you need a counter, you should use the built-in enumerate function, which returns an iterable series of pairs, each pair of being a tuple. The first element of the pair is the index of the current item and the second element of the pair is the item itself. Here we construct a list, passing it to enumerate and iterate over the result, giving us the elements of the list with the corresponding positions in the list. Even better, we can use tuple unpacking to avoid having to directly deal with the tuple.

Lists

We've already covered lists a little, and we've been making good use of them. We know how to create lists using the literal syntax, add to them using the append method, and get at and modify their contents using the square brackets indexing. Now we'll take a deeper look. One very convenient feature of lists and other Python sequences, for this applies to tuples and strings as well, is the ability to index from the end rather than from the beginning. This is achieved by supplying negative indices. For example, we can access the last and second-to-last elements of the list using -1 and -2. This is much more elegant than the clunky approach of subtracting 1 from the length of a container, the approach you would otherwise need to use for retrieving the last element. Note that indexing with -0 is the same as indexing with 0, returning the first element in the list. Because there's no distinction between 0 and -0, negative indexing is essentially 1-based rather than 0-based. This is good to keep in mind if you're calculating indices with even moderately complex logic. One-off errors can creep into negative indexing fairly easily. Slicing is a form of extended indexing, which allows us to refer to portions of a list. To use it, we pass the start and stop indices of a half open range, separated by a colon, as the square brackets index argument. Here's how you can slice the first and second elements of a list. See how the second index, 3 in this case, is 1 beyond the end of the returned range. This facility can be combined with negative indices. For example, to take all elements except the first and last, use the slice 1:-1. Both the start and stop indices are optional. To slice all elements from the third to the end of the list, don't put anything after the colon. Similarly, to slice all elements from the beginning up to, but not including the third, don't put a number before the colon. Notice that these two lists together form

the whole list, demonstrating the convenience of the half open range convention. Since both start and stop slice indices are optional, it's entirely possible to omit both and retrieve all of the elements. And indeed, this last example is an important idiom for copying a list. Recall that assigning references never copies an object, but merely copies a reference to an object. We deploy the full slice to perform a copy into a new list. This gives us a new object with a distinct identity, but since it's a copy, the new object has an equivalent value. It's important to understand that although we have a new list object, which can be independently modified, the elements within it are references to the same objects referred to by the original list. In the event that these objects are both mutable and modified, as opposed to replaced, the change will be seen in both lists. We teach this full slice copying idiom because you're likely to see it in the wild, and it's not immediately obvious what it does. You should be aware that there are other, more readable ways of copying a list, such as the copy method, or you could simply call the list constructor passing the list to be copied. Largely it's a matter of taste. Our preference is for the third form, since it has the advantage of working with any iterable series as the source, not just lists. You must be aware, however, that all these techniques perform a shallow copy. That is, they create a new list containing the same object references as the source list, but they don't copy the referred-to objects. To demonstrate this, we'll use nested lists with the inner lists serving as mutable objects. Here's a list containing two elements, each of which is itself a list. We copy this list using a full slice and convince ourselves that we do, in fact, have distinct lists with equivalent values. Notice, however, that the references within these distinct lists refer not only to equivalent objects, but, in fact, to the same object, which we can see using the is operator. The first elements of a and b are the same object until we rebind the first element of a to a newly constructed list. Now the first elements of a and b refer to different lists with different values. The second elements of both a and b still refer to the same object. We'll demonstrate this by mutating that object through the a list. We can see this change reflected through the b list. For completeness, here is the final state of both the a and b lists. If you need to perform true deep copies of hierarchical data structures like this, which in our experience is a rarity, we recommend taking a look at the copy module in the Python standard library. As with strings and tuples, lists support repetition using the multiplication operator. It's simple enough to use, although it's rarely spotted in the wild in this form. It's most often useful for initializing a list of a size known in advance, with a constant value, such a 0. Be aware, though, that when using mutable objects as elements, the same trap for the unwary lurks here, since repetition will repeat the reference without copying the value. Let's demonstrate using nested lists as our mutable elements again. If we now modify the third element of our outer list, we can see the change through all five references, which comprise the outer list elements. This is because each element of the outer list is a reference to the same

nested list. To find the element in the list, use the index method, passing the object you're searching for. The elements are compared for equivalents until the one you're looking for is found. Here we create a string and split it on spaces, giving us a list of strings. We then use the index method on that list to find the index of the word fox. If you search for a value that isn't present, you receive a value error. Another means of searching is to count matching elements. Here we count how many times the word 'the' appears in our list. If you just want to test for membership, you can use the in operator, or you could test for non-membership with not in. You can remove elements from a list using a keyword with which we have not yet become acquainted, del. The del keyword takes a single parameter, which is a reference to the list element, and removes it from the list, shortening the list in the process. Here we again construct a list by splitting a string. We then remove the fourth element by calling del U [3]. We can see that the element has been removed. It's also possible to remove elements by value rather than by position by using the remove method. This is, of course, equivalent to passing the result of the index method to del. This also raises a value error if a matching element is not present. Items can be inserted into lists using the insert method, which accepts the index of the new item and the new item itself. We'll again start with a list of strings. We'll then insert the string 'destroyed' at index 2. If we join the list into a single string, we see that someone has made a terrible mistake. Concatenating lists using the addition operator results in a new list without modification of the operands, whereas the augmented assignment operator, +=, modifies the assignee in place. This can also be achieved using the extend method. All of these techniques work with any iterable series on the right-hand side. Before we move on from lists, let's look at two operations which rearranged the elements in place, reversing and sorting. A list can be reversed in place simply by calling its reverse method. A list can be sorted in place using the sort method. The sort method accepts two optional arguments, key and reverse. The latter is self-explanatory, and when set to true gives it ascending sort. The key parameter is more interesting. It accepts any callable object, which is then used to extract a key from each item. The items will then be sorted according to the relative ordering of these keys. There are several types of callable objects in Python, although the only one we have encountered so far is the humble function. For example, the len function is a callable object, which is used to determine the length of a collection, such as a string. Consider the following list of words. We can sort this list by the length of the strings by passing len as the key argument to sort. Sometimes an in situ sort or reversal is not what's required. For example, it may cause a function argument to be modified, giving the function confusing side effects which we'd not otherwise have. For out-of-place equivalents of the reverse and sort list methods, we can use the reversed and sorted built-in functions. These return a reversed iterator and a newly sorted list, respectively. For example, calling sorted on the list 4, 9, 2, 1 results in an entirely new list with the

correctly sorted elements. Calling reversed on a list doesn't give us a new list. Rather, it gives us an object of the type list_reverseiterator. We can pass this iterator to the list constructor to create an actual list. These functions have the advantage that they'll work on any finite, iterable source object.

Dictionaries

We'll now return to dictionaries, which lie at the heart of many Python programs, including the Python interpreter itself. We briefly looked at literal dictionaries previously, seeing how they were delimited with curly braces and contain comma-separated key-value pairs with each pair tied together by a colon. These values are accessible via the keys. Since each key is associated with exactly one value and lookup is through keys, the keys must be unique within any single dictionary. It's fine, however, to have duplicate values. Internally, the dictionary maintains pairs of references to the key objects and the value objects. The key objects must be immutable, so strings, numbers, and tuples are fine, but lists are not. The value objects can be mutable, and in practice often are. Our example URL map uses strings for both keys and values, which is fine. You should never rely on the order of the items in the dictionary. It's essentially random and may even vary between different runs of the same program. As for the other collections, there's also a dict constructor, which can convert other types to dictionaries. We can use the constructor to convert from an iterable series of key-value pairs stored in tuples like this. Recall that the items in the dictionary are not stored in any particular order. So long as the keys are legitimate Python identifiers, it's even possible to create a dictionary directly from keyword arguments passed to the dict. As with lists, dictionary copying is shallow by default, copying only the references to the key and value objects, not the objects themselves. There are two means of copying a dictionary, of which we most commonly see the second. The first technique is to use the copy method. The second is simply to pass an existing dictionary to the dict constructor. If you need to extend the dictionary with definitions from another dictionary, you can use the update method. This is called on the dictionary to be updated and is passed the contents of the dictionary which is to be merged in. If we create a new dictionary and then update our original dictionary with the new one, we can see the elements of the new dictionary in the original. If the argument update includes keys which are already present in the target dictionary, the values associated with these keys are replaced in the target by the corresponding values from the source. If we start with the dictionary of stock prices, we can update one entry and add another with update. As we've seen in an earlier module, dictionaries are iterable so can be used with for loops. The dictionary yields the next key on each iteration, and we retrieve the corresponding value by lookup using the

square brackets operator. If we start with the dictionary of color names to codes, we could iterate over the keys, looking up the codes with them. Notice that the keys are returned in an arbitrary order, which is neither the order in which they were specified nor any other meaningful order. If we want to iterate over only the values, we can use the values dictionary method. This returns an object which provides an iterable view onto the dictionary values without causing the values to be copied. There is no efficient or convenient way to retrieve the corresponding key from a value, so we only print the values. In the interest of symmetry, there is also a keys method. Often, though, we want to iterate over the keys and values in tandem. Each key-value pair in a dictionary is called an item, and we can get hold of an iterable view of items using the items dictionary method. When iterated, the items view yields each key-value pair as a tuple. By using tuple unpacking in the for statement, we can get both key and value in one operation without the extra lookup. The membership tests for dictionaries using the in and not in operators work on the keys. To see this, let's construct a dictionary mapping currency names to symbols. We can then use in to see if New Zealand dollars are in the dictionary. Likewise, we can use not in to see that Macedonian denars are not present. As with lists, we use the del keyword to remove an entry from a dictionary. If we create a dictionary of elements and their atomic numbers, we can remove Feynmanium by deleting the key Fy. The keys in a dictionary should be immutable, although the values can be modified. Here's a dictionary which maps the element symbol to a list of mass numbers for different isotopes of that element. See how we split the dictionary literal over multiple lines. That's allowed because the curly braces for the dictionary literal are open. Our string keys are immutable, which is a good thing for correct functioning of the dictionary. But there's no problem with modifying the dictionary values in the event, for example, that we discover some new isotopes. Here the augmented assignment operator is applied to the list object accessed through H, the key for hydrogen. The dictionary is not being modified. Of course, the dictionary itself is mutable, and we can add new items. With compound data structures such as our table of isotopes, it can be helpful to have them printed out in a much more readable form. We can do this with the Python standard library pretty-printing module called pprint, which contains a function called pprint. Note that if we didn't bind the pprint function to the different name pp, the function reference would overwrite the module reference, preventing further access to contents of the module. Arguably, it's poor design to have a module containing functions of the same name because of this issue. Be that as it may, the pprint function gives us a much more comprehensible display. Let's move on from dictionaries and look at a new built-in data structure, the set.

Sets

The set data type is an unordered collection of unique elements. The collection is mutable, insofar as elements could be added and removed from the set. But each element must itself be immutable, very much like the keys of a dictionary. Sets have a literal form very similar to dictionaries. Again, delimited by curly braces, but each item is a single object rather than a pair joined by a colon. Note that the set is unordered and, of course, our set has the type set. Recall that curly braces create an empty dictionary. So in order to create an empty set, we must resort to the set constructor. This is also the form that Python echoes back to us. The set constructor can create a set from any iterable series, such as a list, and duplicates are discarded. In fact, a common use of sets is to efficiently remove duplicate items from a series of objects. Naturally, sets are iterable, although the order is arbitrary. Membership is a fundamental operation for sets and, as with other collection types, is performed using the in and not in operators. To add a single element to a set, use the add method. Adding an element that already exists has no effect and neither doesn't produce an error. Multiple elements could be added in one go from any iterable series, including another set using the update method. Two methods are provided for removing elements from sets. The first, remove, requires that the element to be removed is present in the set. Otherwise, a key error is produced. The second method, discard, is less fussy and simply has no effect if the element is not a member of the set. As with the other built in collections, set sports a copy method, which performs a shallow copy of the set, copying references but not objects. And as we've already shown, the set constructor may be used. Perhaps the most useful aspect of the set type is the group of powerful set algebra operations, which are provided. These allow us to easily compute set unions, set differences and set intersections, and to evaluate whether two sets have subset, superset, or disjoint relations. To demonstrate these methods, we'll construct some sets of people according to various phenotypes. We'll identify a few people with blue eyes and a few with blond hair. The smell_hcn set contains those who can smell hydrogen cyanide. And taste_ptc is those who can taste phenylthiocarbamide. Finally, we'll define a set for those with O blood type, one for B blood type, one for A blood type, and finally one for AB blood. To find all the people with blond hair, blue eyes, or both, we can use the union method, which collects together all the elements which are in either or both sets. We can demonstrate that union is a commutative operation, that is, we can swap the order of the operands using the value of quality operator to check for equivalents of the resulting sets. To find all the people with blond hair and blue eyes we can use the intersection method, which collects together only the elements which are present in both sets. This is also commutative. To identify the people with blond hair who don't have blue eyes, we can use the difference method. This finds all the elements which are

in the first set, which are not in the second set. This is non-commutative because the people with blond hair who don't have blue eyes are not the same as the people who have blue eyes but don't have blond hair. However, if we want to determine which people have exclusively blonde hair or blue eyes, but not both, we can use the symmetric difference method. This collects all the elements which were in the first set or the second set, but not both, As you can tell from the name, symmetric difference is indeed commutative. In addition, three predicate methods are provided, which tell us about the relationships between sets. We can check whether one set is a subset of another using the issubset method. For example, to check whether all of the people who can smell hydrogen cyanide also have blond hair, we can call issubset on smell_hcn, passing blond_hair as a parameter. This checks that all the elements of the first set are also present in the second set. To test whether all the people who could taste phenylthiocarbamide can also taste hydrogen cyanide, use the issuperset method. This checks that all the elements of the second set are present in the first set. To test that two sets have no members in common, use the isdisjoint method. For example, your blood type is either A or O, never both.

Protocols

In Python, a protocol is a set of operations or methods that a type must support if it is to implement that protocol. Protocols needn't be defined in the source code as separate interfaces or base classes as they would in nominally typed languages, such as C# or Java. It's sufficient to simply have an object provide functioning implementations of those operations. We can organize the different collections we have encountered in Python according to which protocols they support. Support for a protocol demands specific behavior from a type. The container protocol requires that membership testing using the in and not in operators be supported. The sized protocol requires that the number of elements in a collection can be determined by calling len on the collection. Iteration is such an important concept that we're devoting a whole module to it later in this course. In short though, iterables provide a means for yielding elements one by one as they are requested. One important property of iterables is that they can be used in for loops. The sequence protocol requires that items can be retrieved using square brackets with an integer index, that items can be searched for with index, that ideas can be counted with count, and that a reversed copy of the sequence can be produced with reversed. In addition, objects that support sequence must also support iterable, sized, and container. We won't cover the mutable sequence, mutable mappings, and immutable set here. We have only covered one representative type of each protocol so the generality afforded by the protocol concept doesn't gain as much at this juncture.

Summary

Python's built-in collection types are generally intuitive for most purposes, but they're also quite sophisticated, so we've had to go through a lot of material in this module. Here are the topics we've covered. Tuples, or immutable sequence types. Literal syntax for tuples are optional parentheses around a comma-separated list. For single-element tuples, you need to use a trailing comma. Tuple unpacking is useful for multiple return values and swapping. Join concatenation is most efficiently performed with the join method rather than the addition or augmented assignment operators. The str.partition method is a useful and elegant string parsing tool. The str.format method provides a powerful means of replacing placeholders with stringified values. Python 3.6 introduced f-strings, a new kind of string literal that can interpolate Python expressions into the string. Range objects represent arithmetic progressions. Range can be called with one, two, or three arguments describing start, stop, and step values. The enumerate built-in function is often a superior alternative to range for generating loop counters. Lists support indexing from the end of the list with negative indices. Slice syntax allows us to copy all or part of a list. The full slice is a common Python idiom for copying lists, although the copy method and list constructor are less obscure. You can look for elements in a list with the index and count methods. You can remove elements from a list with the del keyword. Lists can be sorted or reversed in place with the sort and reverse methods. The sorted and reversed functions can sort or reverse any iterable. List copies and those of other collections in Python are shallow copies. References are copied, but objects are not. Dictionaries map from keys to values. Iteration and membership testing with dictionaries is done with respect to their keys. You should not assume any order when iterating over keys in a dictionary. The keys, values and items methods provide views into the different aspects of a dictionary, allowing convenient iteration. You can copy dictionaries with the copy method or the dict constructor. The update method on dictionary extends one dictionary with another. Sets store an unordered collection of unique elements. Sets support powerful set algebra operations and predicates. The built-in collections can be organized according to which protocols they support, such as iterable, sequence, and mapping. In passing, we've also found that underscore is in common usage used for dummy or superfluous variables. The pprint module supports pretty printing of compound data structures. In the next module of Core Python: Getting Started, we'll look at exceptions and how to work with them in Python. Thanks for watching, and we'll see you in the next module.

Exceptions

Overview

In most programs, there is a clear notion of the normal path through the code. But, of course, conditions can arise where this normal flow can't be followed. For example, if a program involves reading a file specified by the user, it may actually happen that the file doesn't exist. Conditions like this often need to be handled, and the standard mechanism for doing so in Python, as with many other languages, is with what are known as exceptions. In this module of Core Python: Getting Started, we will learn what exceptions are. We'll see how to introduce or raise an exception, and we'll look at how doing so interrupts the normal flow of a program. We'll learn about how you can catch exceptions to handle them, and we'll see what happens to your program if you choose not to handle exceptions. We'll talk a bit about Python's somewhat liberal approach to the use of exceptions. We'll explore some of Python's built-in exception types, and we'll see that some of these indicate programmer errors, while others represent various other kinds of conditions. And we'll look at an important mechanism for ensuring resource cleanup when exceptions are involved. Exception handling is a mechanism for stopping normal program flow and continuing at some surrounding context or code block. The event of interrupting normal flow is called the act of raising an exception. In some enclosing context, the raised exception must be handled upon which control flow is transferred to the exception handler. If an exception propagates up the callstack to the start of the program, then an unhandled exception will cause the program to terminate. An exception object containing information about where and why an exceptional event occurred is transported from the point at which the exception was raised to the exception handler so that the handler can interrogate the exception object and take appropriate action. If you've used exceptions in other popular imperative languages like C++ or Java, then you've already got a good idea of how exceptions work in Python. There have been long and tiresome debates over exactly what constitutes an exceptional event, the core issue being that exceptionality is in reality a matter of degree. Some things are more exceptional than others, whereas programming languages tend to impose a false dichotomy by insisting that an event is either entirely exceptional or not at all exceptional. The Python philosophy is at the liberal end of the spectrum when it comes to the use of exceptions. Exceptions are ubiquitous in Python, and it's crucial to understand how to handle them.

Exceptions and Control Flow

Since exceptions are a means of control flow, they can be clumsy to demonstrate at the REPL. So for this part of the course, we'll be using a Python module to contain our code. Let's start with a very simple module we can use for exploring these important concepts and behaviors. Place this

code in a module called exceptional.py. We'll define a function called convert that attempts to construct an integer from a sequence of strings describing its decimal digits. It then returns that intege. Import the convert function from this module into the Python REPL and call our function to see that it has the desired effect. This seems to work, but if we call our function with an object that can't be converted to an integer, we get a trace back from the dictionary lookup. What's happened here is that digit map raised a key error when we tried to look up the string around in it. Of course, it did this because it doesn't have an entry for around. We didn't have a handler in place, so the exception was caught by the REPL and the stack trace was displayed. The key error referred to in the stack trace is the type of the exception object, and the error message, the string around, is part of the payload of the exception object that has been retrieved and printed at the REPL. Notice that the exception propagates across several levels in the call stack.

Handling Exceptions

Let's make our convert function more robust by handling the KeyError using a try except construct. Both the try and except keywords introduce new blocks. The try block contains code that could raise an exception, and the except block contains the code, which performs error handling in the event that an exception is raised. Modify your convert function to look like this. We have decided that if an unconvertible string is supplied, will return -1. To reinforce your understanding of the control flow here, we'll add a couple of print statements. Let's test this interactively after restarting the REPL. First, we import the convert function from a module named exceptional. Then, we convert the string three four into the number 34. Finally, we try to convert the word eleventeen, which, of course, fails. Note how the print in the try block, after the point at which the exception was raised, was not executed when we passed in eleventeen. Instead, execution was transferred directly to the first statement of the except block. Our function expects its argument s to be iterable, so let's see what happens if we pass an object that isn't, for example, an integer. This time our handler didn't intercept the exception. If we look closely at the trace, we can see that this time we received a TypeError, a different type of exception. Each try block can have multiple corresponding except blocks, which intercept exceptions of different types. Let's add a handler for TypeError, too. Now, if we rerun the same test in a fresh REPL, we find that the TypeError is handled as well. We've got some code duplication between our two exception handlers with that duplicated print statement and assignment. We'll move the assignment in front of the try block, which doesn't change the behavior of the program. Then we'll exploit the fact that both handlers do the same thing by collapsing them into one, using the ability of the except statement to accept a tuple of exception types. Now we see that everything

still works as designed. We can convert the string two nine, but converting the word elephant or the integer 451 will fail and return -1.

Exceptions and Programmer Errors

Now that we are confident with the control flow for exception behavior, we can remove the print statements. But now, when we try to import our program, we get yet another type of exception, an IndentationError, because our except block is now empty, and empty blocks are not permitted in Python programs. This is not an exception that it is ever useful to catch with an except block. Almost anything that goes wrong with a Python program results in an exception. But some, such as IndentationError, SyntaxError, and NameError are the result of programmer errors, which should be identified and corrected during development rather than handled it runtime. The fact that these things are exceptions is mostly useful if you're creating a Python development tool, such as a Python IDE, embedding Python itself in a larger system to support application scripting, or designing a plugin system which dynamically loads code. With that said, we still have the problem of what to do with our empty except block. The solution arrives in the form of the pass keyword, which is a special statement that does precisely nothing. It's a no-op, and its only purpose is to allow us to construct syntactically permissible blocks that are semantically empty. Perhaps in this case, though, it would be better to simplify further and just use multiple return statements and do away with the x variable completely. Sometimes we'd like to get hold of the exception object, in this case, an object of type of KeyError or AttributeError, and interrogate it for more details of what went wrong. We can get a named reference to the exception object by tacking an as clause onto the end of the except a statement. We'll modify our function to print a message with exception details to the standard error stream before returning. To print a standard error, we need to get a reference to the stream from the sys module, so at the top of our module we'll need to import sys. We can then pass sys.stderr as a keyword argument called file to print. Here we use a feature of f strings that we haven't seen before. If you put an !r after the expression, the repr representation of the value will be inserted into your string. In the case of exceptions, this gives us more detailed information about the type of the exception. Let's see that at the REPL.

Re-raising Exceptions

Let's add a second function, string_log, to our module, which calls our convert function and computes the natural log of the result. We've written this fairly innocuous looking bit of code to

demonstrate the greatest folly of returning error codes, that they can be ignored by the caller, wreaking havoc amongst unsuspecting code later in the program. A slightly better program might test the value of v before proceeding to the log call. Without such a check, log will, of course, fail when passed the negative error code value. Naturally, the log failure causes the raising of another exception. Much better and altogether more Pythonic is to forget about error return codes completely and go back to raising an exception from convert. Instead of returning an un-Pythonic error code, we can simply omit our error message and re-raise the exception object we're currently handling. This can be done by replacing the return a -1 with raise at the end of our exception handling block. Without a parameter, raise simply re-raises the exception that is being currently handled. Testing in the REPL, we can see that the original exception type is re-raised whether it's a key error or a type error, but our Conversion error message is printed to standard error along the way.

Exceptions Are Part of the API

Exceptions form an important aspect of the API of a function. Callers of a function need to know which exceptions to expect under various conditions so that they can ensure appropriate exception handlers are in place. We'll use square root finding as an example, using a homegrown square root function, courtesy of Heron of Alexandria, although he probably didn't use Python. Place the following code in a file named roots.py. There's only one language feature in this program we haven't met yet, the logical and operator, which we use in this case to test that two conditions are true on each iteration of the loop. Python also includes a logical or operator, which can be used to test whether either or both operands are true. Running our program, we can see that Heron was really onto something. Let's add a new line to the main function, which takes the square root of -1. If we run that, we get a new exception. What has happened is that Python has intercepted a division by 0, which occurs on the second iteration of the loop and raised an exception, a ZeroDivisionError. Let's modify our code to catch the exception before it propagates up to the top of the call stack, thereby causing our program to stop, using the try-except construct. Now, when we run the script, we see that we're handling the exception cleanly. We should be careful to avoid a beginner's mistake of having too-tight scopes for exception handling blocks. We can easily use one try-except block for all of our calls to square root. We also add a third print statement to show how execution of the enclosed block is terminated. This is an improvement on what we started with but most likely, users of a square root function don't expect it to throw a ZeroDivisionError. Python provides us with several standard exception types to signal common errors. If a function parameter is supplied with an illegal value, it is customary

to raise a ValueError. We can do this by using the raise keyword with a newly created exception object, which we can create by calling the ValueError constructor. There are two places we could deal with the division by 0. The first approach would be to wrap the root finding while loop in a try-except ZeroDivisionError construct and then raise a new ValueError exception from inside the exception handler. This would be wasteful, though. We know this routine will fail with negative numbers so we can detect this precondition early on and raise an exception at that point, The test is a simple if statement and a call to raise, passing the new exception object. The ValueError constructor accepts an error message. See how we can modify the doc string to make it plain which exception type will be raised by square root and under what circumstances. But look what happens if we run the program. We're still getting a traceback and an ungraceful program exit. This happens because we forgot to modify our exception handler to catch ValueError rather than ZeroDivisionError. Let's modify our calling code to catch the right exception class and also assign the caught exception object to a named variable so that we can interrogate it after it has been caught. In this case, our interrogation is simply to print the exception object, which knows how to display itself as the message to standard error. Running the program again, we can see that our exception is being gracefully handled.

Exceptions and Protocols

Exceptions are part of a function's API, and more broadly, are part of certain protocols. For example, objects which implement the sequence protocol should raise an IndexError exception for indices which are out of range. The exceptions which are raised are as much a part of a function's specification as the arguments it accepts, and as such, must be implemented and documented appropriately. There are a handful of common exception types in Python, and usually, when you need to raise an exception in your own code, one of the built-in types is a good choice. Much more rarely, you'll need to define a new exception type, but we don't cover that in this course. Often, if you're deciding what exceptions your code should raise, you should look for similar cases in existing code. The more your code follows existing patterns, the easier it will be for people to integrate and understand. For example, suppose you were writing a key value database. It would be natural to use KeyError to indicate a request for a non-existent key because this is how dict works, that is mapping in Python follows certain patterns, and exceptions are part of that pattern. Let's look at a few common exception types. IndexError is raised when an integer index is out of range. You can see this when you index past the end of the list. ValueError is raised when an object is of the right type, but contains an inappropriate value. We've seen that already

when trying to construct an int from a non-numeric string. KeyError is raised when a lookup in a mapping fails. You can see that here when we look up a non-existent key in a dict.

Avoid Explicit Type Checks

We tend not to protect against type errors in Python. To do so runs against the grain of dynamic typing in Python and limits the reuse potential of code we write. For example, our convert function could test whether the argument was a list using the built-in isinstance function and raise a type error exception if it was not. But then we'd also want to allow arguments that are instances of tuple as well. It soon gets complicated if we want to check whether our function will work with types such as set, dict, or any other iterable type. And in any case, who is to say that it does? Alternatively, as we currently do, we could intercept TypeError inside our convert function and reraise it, but to what end? Usually in Python, it's not worth adding type checking to your functions. If a function works with a particular type, even one that you couldn't have known about when you designed the function, then that's all to the good. If not, execution will probably result in a TypeError anyway. Likewise, we tend not to catch TypeErrors very frequently.

It's Easier to Ask Forgiveness Than Permission

Now let's look at another tenant of Python philosophy and culture, the idea that it's easier to ask forgiveness than permission. There are only two approaches to dealing with a program operation that might fail. The first approach is to check that all the preconditions for a failure-prone operation are met in advance of attempting the operation. The second approach is to blindly hope for the best, but be prepared to deal with the consequences if it doesn't work out. In Python culture, these two philosophies are known as look before you leap, or LBYL, and it's easier to ask forgiveness than permission, EAFP, a term, which, incidentally, was coined by rear admiral Grace Hopper, inventor of the compiler. Python is strongly in favor of EAFP because it puts primary logic for the happy path in its most readable form with deviations from the normal flow handled separately rather than interspersed with the main flow. Let's consider an example, processing a file. The details of the processing aren't relevant. All we need to know is that the process_file function will open a file and read some data from it. First, the LBYL version. Before attempting to call process_file, we check that the file exists, and if it doesn't, we avoid making the call and print a helpful message instead. There are several problems with this approach, some obvious and some insidious. One obvious problem is that we only perform an existence check. What if the file exists, but contains garbage? What if the path refers to a directory instead of a file? According to

LBYL, we should add preemptive tests for these too. A more subtle problem is there is a race condition here. It's possible for the file to be deleted, for example, by another process between the existence check and the process_file call, a classic issue of atomicity. There's really no good way to deal with this. Handling of errors from process_file will be needed in any case. Now consider the alternative using the more Pythonic EAFP approach. Here, we simply attempt the operation without checks in advance, but we have an exception handler in place to deal with any problems. We don't even need to know in a lot of detail exactly what might go wrong. Here we catch OSError, which covers all manner of conditions such as file not found and using directories where files are expected. EAFP is standard in Python, and that philosophy is enabled by exceptions. Without exceptions, that is, using error codes instead, you're forced to include error handling directly in the main flow of your logic. Since exceptions interrupt the main flow, they allow you to handle exceptional cases non-locally. Exceptions coupled with EAFP are also superior because, unlike error codes, exceptions cannot be easily ignored. By default, exceptions have a big effect, whereas error codes are silent by default. So the exception EAFP-based style makes it very difficult for problems to be silently ignored.

Cleanup Actions

Sometimes you need to perform a cleanup action, irrespective of whether an operation succeeds. In a later module, we'll introduce context managers, which are the modern solution to this common situation, but here we'll introduce the try finally construct, since creating a context manager can be overkill in simple cases. And in any case, an understanding of try finally is useful for making your own context managers. Consider this function, which uses various facilities of the standard OS module to change the current working directory, create a new directory at that location, and then restore it to the original working directory. At first sight, this seems reasonable, but should the call to os.mkdir fail for some reason, the current working directory of the Python process won't be restored to its original value, and the make_at function will have an unintended side effect. To fix this, we'd like the function to restore the original current working directory under all circumstances. We can achieve this with a try finally block. Code in the finally block is executed whether exception leaves the try block normally by reaching the end of the block, or exceptionally by an exception being raised. This construct can be combined with except blocks, here used to add a simple failure logging facility. Now, if os.mkdir raises an OS error, the OS error handler will be run, and the exception will be re-raised, but since the finally block is always run, no matter how the try block ends, we can be sure that the final directory change will take place in all

circumstances. Errors should never pass silently, unless explicitly silenced. Errors are like bells, and if we make them silent, they are of no use.

Platform-Specific Code

Detecting a single key press from Python, such as the 'press any key to continue' functionality at the console, requires use of operating system specific modules. We can't use the built-in input function, because that waits for the user to press Return before giving us a string. To implement this, on Windows we need to use functionality from the Windows only msvcrt module, and on Linux and Mac OS X, we need to use functionality from the Unix only tty and termios modules, in addition to the sys module. This example is quite instructive, as it demonstrates many Python language features including import and def as statements, as opposed to declarations. Recall, The top-level module code is executed on first import. Within the first try block, we attempt to import msvcrt, the Microsoft Visual C Runtime. If this succeeds, we then proceed to define a function, getkey, which delegates to the msvcert getch function. Even though we're inside a try block at this point, the function will be declared at the current scope, which is the module scope. If, however, the import of msvcrt fails because we're not running on Windows, an import error will be raised, and execution will transfer to the except block. This is a case of an error being silenced explicitly because we're going to attempt an alternative course of action in the exception handler. Within the except block, we import three modules needed for a getkey implementation on Unix-like systems, and then proceed to the alternative definition of getkey, which, again, binds the function implementation to a name in the module scope. This Unix implementation of getkey uses a try finally construct to restore various terminal attributes after the terminal has been put into raw mode for the purposes of reading a single character. In the event that our program is running on neither Windows nor a Unix-like system, the import tty statement will raise a second import error. This time we make no attempt to intercept the exception. We allow it to propagate to our caller, which is whatever attempted to import this key press module. We know how to signal this error, but not how to handle it, so we defer that decision to our caller. The error will not pass silently. If the caller has more knowledge or alternative tactics available, it can in turn intercept this exception and take appropriate action, perhaps degrading to using Python's input built-in function and giving a different message to the user.

Summary

Exceptions are an essential topic in Python, and it's critical that you understand how to work with them as you develop your mastery of the language. In this module, we learned that the raising of an exception interrupts normal program flow and transfers control to an exception handler. Exception handlers are defined using the try except construct. Try blocks define a context in which exceptions can be detected. Corresponding except blocks define handlers for specific types of exceptions. Python uses exceptions pervasively, and many built-in language features depend on them. Except blocks can capture an exception object, which is often of a standard type such as value error, key error, or index error. Programmer errors, such as indentation error and syntax error, should not normally be handled. Exceptional conditions can be signaled using the raise keyword, which accepts a single parameter of an exception object. Raise without an argument within an except block re-raises the exception which is currently being processed. We tend not to routinely check for type errors. To do so would negate the flexibility afforded to us by Python's dynamic type system. Exception objects can be converted to strings using the str constructor for the purposes of printing message payloads. The exceptions thrown by a function form part of its API and should be appropriately documented. When raising exceptions, prefer to use the most appropriate built-in exception type. Cleanup and restorative actions can be performed using the try finally construct, which may optionally be used in conjunction with except blocks. Along the way, we saw that the output of the print function can be redirected to standard error using the optional file argument. The expressions in f strings can be suffixed with !r to use the repr representation of the inserted value. Python supports the logical operators and an or for combining Boolean expressions. Return codes are too easily ignored. Platform-specific actions can be implemented using an easier to ask forgiveness than permission approach facilitated by intercepting import errors and providing alternative implementations. In the next module of Core Python: Getting Started, we'll take a deeper look at the concept of iteration in Python, covering topics including comprehensions, Python's iteration protocols, and lazy evaluation. Thanks for watching, and we'll see you in the next module.

Iteration and Iterables

Overview

A central abstraction in Python is the notion of an iterable, an object from which you can fetch a sequence of other objects. The act of fetching a sequence from an iterable is known as iteration,

and you've already encountered it many times in the form of the for loop. Iteration in Python is designed to be simple to use and not require a deep understanding on the part of the user. But under this fairly unassuming surface lies a sophisticated and powerful system. In this module of Core Python: Getting Started, we'll look at comprehensions, Python's shorthand syntax for creating certain kinds of iterable objects. We'll see that some comprehensions create objects that you're already familiar with, like lists and sets, while others create wholly new kinds of objects with some surprising properties. We'll learn about the syntax for filtering in comprehensions. We'll discover Python's low-level API for working with iterable objects, and in doing so, we'll learned about another crucial element of iteration, the concept of the iterator. We'll also look into Python's somewhat surprising use of exceptions in its core iteration protocol. We'll cover generator functions, a powerful and sometimes subtle technique for defining iterable sequences as computations. In doing so, we'll talk about Python's yield keyword, and we'll look at some fascinating qualities of generators, such as statefulness, laziness, and the ability to model infinite sequences. We'll also cover what are known as generator expressions, a cross between comprehensions and generator functions. Finally, we'll look at some of the tools that Python provides for working with iteration, including several elements from the standard library's itertools module. Let's jump right into comprehensions.

List and Set Comprehensions

Comprehensions in Python are concise syntax for describing lists, sets, or dictionaries in a declarative or functional style. This shorthand is readable and expressive, meaning that comprehensions are very effective at communicating intent to human readers. Some comprehensions almost read like natural language, making them nicely self-documenting. Comprehensions are much easier to demonstrate than they are to explain, so let's bring up a Python REPL. First, we'll create a list of words by splitting a string. Now comes a list comprehension. The comprehension is enclosed in square brackets, just like a literal list, but instead of literal elements, it contains a fragment of declarative code, len(word), which describes how to construct the elements of a list, which we loop over using for word in words. Here the new list is formed by binding word to each value in words in turn and evaluating len(word) to create a new value. Each of these new values becomes an element in the newly constructed list. The general form of the list comprehension is an opening square bracket, an expression, a statement binding a name to successive elements of an iterable, and a closing square bracket. That is, for each item in the iterable on the right, we evaluate the expression on the left, which is almost always, but not necessarily in terms of the item, and use that as the next element of the list being

constructed. The comprehension is the declarative equivalent of the following imperative code. We first create the empty list lengths. We then use a for loop to iterate over words, binding each element in turn to the name word. For each iteration, we calculate the length of the word and append the length to lengths. Note that the source object over which we iterate doesn't need to be a list. It can be any object which implements the iterable protocol, such as a tuple. The expression producing the new list's elements can be any Python expression. Here we find the number of decimal digits in each of the first 20 factorials using range to generate the source of sequence. First, we import factorial from the math module. We then calculate the length of the decimal string representation of the factorial of each integer from 0 to 19. Note also that the type of the object produced by list comprehensions is nothing more or less than a regular list. Set supports a similar comprehension syntax using, as you might expect, curly braces. Our number of digits in factorials result contained duplicates. By building a set instead, we can eliminate them, although note that the resulting set is not necessarily stored in a meaningful order since sets are unordered containers.

DIctionary Comprehensions

The third type of comprehension is the dictionary comprehension. This also uses curly braces and is distinguished from the set comprehension by the fact that we now provide two colon-separated expressions for the key and the value, which will be evaluated in tandem for each new item. Let's start with the dictionary we can play with which maps countries to capital cities, United Kingdom to London, Brazil to Brasilia, Morocco to Rabat, and Sweden to Stockholm. One nice use for a dictionary comprehension is to invert a dictionary so we can perform efficient look-ups in the opposite direction. Here we use capital cities as the keys by putting capital on the left of the colon and countries as the values by putting country on the right. We get the country capital tuples by looping over country to capitol.items. We can now import the pprint function and use it to pretty print our new inverted dictionary. Note that dictionary comprehensions do not operate directly on dictionary sources. Well, they can, but recall that iterating over a dictionary yields only the keys. If we want both keys and values, we should use the items method and then tuple unpacking to access the key and value separately, as we did in this example, Should your comprehension produce some identical keys, later keys will overwrite earlier keys. In this example, we start with a list of words and map the first letter of words to the words themselves, but only the last H word is kept. Remember that there's no limit to the complexity of the expression you can use in any of the comprehensions, but for the sake of your fellow programmers, you should avoid going overboard. Extract complex expressions into separate

functions to preserve readability. The following is close to the limit of being reasonable for a dictionary comprehension. We import OS and glob. We then use a dictionary comprehension to map the real paths of files to their sizes.

Filtering Comprehensions

All three types of collection comprehensions support an optional filtering clause. This clause allows us to choose which items of the source are evaluated by the expression on the left. To make this interesting, we'll first find a primality testing predicate function. It uses the square root function, so we have to import that. We then define the function is_prime. It checks to see if the input is less than 2, returning False if so. It then checks if the input can be evenly divided by any integer up to the square root of the input, returning False, if so. Finally, it returns True if no divisors were found, We can now use is_prime as the filtering clause of a list comprehension to produce all primes less than 100. We have a slightly odd-looking x for x construct here because we're not applying any transformations to the filtered values. The expression in terms of x is simply x itself. There's nothing to stop us combining a filtering predicate with a transforming expression. Here's a dictionary comprehension, which maps numbers with exactly three divisors to a tuple of those divisors.

Moment of Zen

Code is written once but read over and over. Fewer is clearer. Comprehensions are often more readable than the alternative. However, it's possible to overuse comprehensions. Sometimes a long or complex comprehension may be less readable than the equivalent for loop. There's no hard and fast rule about when one form should be preferred. But be conscientious when writing your code and try to choose the best form for your situation. Above all, your comprehensions should ideally be truly functional. That is, they should have no side effects. If you need to create side effects such as printing to the console during iteration, use another construct, such as a for loop instead.

Iteration Protocols

Comprehensions and for loops are the most frequently used language features for performing iteration. That is, taking items one by one from a source and doing something with each in turn. However, both comprehensions and for loops iterate over the whole sequence by default, whereas sometimes more fine-grained control is needed. There are two important concepts here

on top of which a great deal of Python language behavior is constructed, iterable objects and iterator objects, both of which are reflected in standard Python protocols. The iterable protocol allows us to pass an iterable object, usually a collection or stream of objects, such as a list, to the built-in iter function to get an iterator for the iterable object. Iterator objects, in turn, support the iterator protocol, which requires that we can pass the iterator object to the built-in next function to fetch the next value from the underlying collection. As usual, a demonstration at the Python REPL will help all these concepts crystallize into something you can work with. We use a list of names of the seasons, in British English no less, as our iterable source object. We ask our iterable object to give us an iterator using the iter built in and then request a value from the iterabor using the next built in. Each call to next moves the iterator through the sequence. But what happens when we reach the end? In a spectacular display of liberalism, Python raises an exception specifically of the type StopIteration. Those of you coming from other programming languages with a more straightlaced approach to exceptions may find this mildly outrageous, but I ask you, What could be more exceptional than reaching the end of a collection? It only has one end after all. This attempt at rationalizing the language design decision makes even more sense when one considers that the iterable series may be a potentially infinite stream of data. Reaching the end in that case really would be something to write home about or, indeed, raise an exception for. With for loops and comprehensions at our fingertips, the utility of these lower-level iteration protocols may not be obvious. To demonstrate a more concrete use, here's a little utility function, which, when passed an iterable object, returns the first item in that series or, if the series is empty, raises a value error. First, it calls iter on the input interable object to produce an iterator. It then calls next on the iterator inside a try block returning the results. If the input iterable is empty, the next has no value to return, and it instead raises StopIteration. We catch that and raise a value error instead. This function works as expected on any iterable object. Here we see it getting the first element from a list, and we can see it doing the same for set. If we pass an empty set to first, it throws a value error. It's worth noting that the higher-level iteration constructs, such as for loops and comprehensions, are built directly upon this lower-level iteration protocol.

Generator Functions

Now we come to generator functions, one of the most powerful and elegant features of the Python programming language. Python generators provide the means for describing iterable series with code in functions. These sequences are evaluated lazily, meaning they only compute the next value on demand. This important property allows them to model infinite sequences of values with no definite end, such a streams of data from a sensor or active log files. By carefully

designing our generator function, we can make generic stream processing elements, which can be composed into sophisticated pipelines. Generators are defined by any Python function which uses the yield keyword at least once in its definition, they may also contain the return keyword with no arguments. And just like any other function, there is an implicit return at the end of the definition. To understand what generators do, let's start with a simple example at the python ripple. Let's define the generator, and then we'll examine how the generator works. Generator functions are introduced by def just as for a regular Python function. We'll then yield the values 1, 2, and 3 in order. Now let's call gen123 and assign its return value to g. As you can see, gen123 is called just like any other Python function, but what has it returned? G is a generator object. Generators are, in fact, Python iterators so we can use the iterated protocol to retrieve or yield successive values from the series. Take note of what happens now that we've yielded the last value from our generator. Subsequent calls to next raise a stop iteration exception just like any other Python iterator. Because generators are iterators and because iterators must also be iterable, they can also be used in all the usual Python constructs which expect iterable objects, such as for loops. Be aware that each call to the generator function returns a new generator object. Here recall a generator function two times binding the results to h and i, respectively. If we look at h and i, we see that they're different objects and that each generator object can be advanced independently. Let's take a closer look at how and crucially when the code in the body of our generator function is executed. To do this, we'll create a slightly more complex generator that traces its execution with good old fashioned print statements. We'll call this generator gen246. It will first print that it's about to yield 2 and then it will yield 2, it will do the same for 4 and finally for 6. At the end, it will print that it's about to return. Now we'll call the generator and assign it to the name g. At this point, the generator object has been created and returned, but none of the code within the body of the generator function has yet been executed, so let's fetch the first value from the generator. See how when we request the first value, the generator body runs up to and including the first yield statement, the code executes just far enough to literally yield the next value. When we request the next value from the generator, execution of the generator function resumes at the point it left off and continues running until the next yield. After the final value has returned, the next request causes the generator function to execute until it returns at the end of the function body, which in turn raises the expected StopIteration exception. Now that we've seen how generator execution is initiated by calls to next and interrupted by yield statements, we could progress to placing more complex code in a generator function body.

Maintaining State in Generators

Now, we'll look at how our generator functions, which resumes execution each time the next value is requested can maintain state in local variables. In the process of doing so, our generators will be both more interesting and more useful. The resumable nature of generator functions can result in complex control flow, so we'll be watching the execution of these generators in a graphical Python debugger. I'll be using PyCharm, but you can use any Python debugger to trace generator execution. We'll be showing two generators which demonstrate lazy evaluation, which we'll then combine into a generator pipeline. The first generator we'll look at is take, which retrieves a specified number of elements from the front of an iterable. Note that the function defines a generator because it contains at least one yield statement. This particular generator also contains a return statement to terminate the stream of yielded values. The generator simply uses a counter to keep track of how many elements have been yielded so far, ending the sequence when we reach a specified point. Now, let's bring our second generator into the picture. This generator function, called distinct, eliminates duplicate items by keeping track of which elements it's already seen in a set. In this generator we also make use of a control flow construct we have not previously seen, the continue keyword. The continue statement finishes the current iteration of the loop and begins the next iteration immediately. When executed in this case, execution will be transferred back to the for statement, but as with break, it can also be used with while loops. In this case, that continue is used to skip any values which have already been yielded. Now we'll arrange both of our generators into a lazy pipeline using take and distinct together to fetch the first three unique items from a collection. We start by creating a small list called items. We pass the result of distinct into take and loop over the results. Notice that the distinct generator only does just enough work to satisfy the demands of the take generator, which it is iterating over. It never gets as far as the last two items in the source list because they are not needed to produce the first three unique items. This lazy approach to computation is very powerful, but the complex control flows it results in can be difficult to debug. It's often useful during development to force evaluation of all the generated values, and this is most easily achieved by inserting a call to the list constructor. This interspersed call to list causes the distinct generator to exhaustively process its source items before take does its work.

Laziness and the Infinite

Generators are lazy, meaning that computation only happens just in time when the next result is requested. This interesting and useful property of generators means that they can be used to model infinite sequences, since values are only produced as requested by the caller, and since no data structure needs to be built to contain the elements of the sequence, generators can safely

be used to produce never-ending or just very large sequences like sensor readings, mathematical sequences, and the contents of multi-terabyte files. The authors of this course are sworn by sacred oath never to use either Fibonacci or quicksort implementations in demonstrations or exercises. Allow us instead to present a generator function for the Lucas series, which has nothing whatsoever to do with the order in which you should watch the episodes of Star Wars. The Lucas series starts with 2, 1 and each value after that is the sum of the two proceeding values. So the first few values of the sequence are 2, 1, 3, 4, 7, and 11. The first yield produces the value 2. The function then initializes A and B, which hold the previous two values needed as the function proceeds. Then the function enters an infinite while loop, where it yields the value of B. A and B are updated to hold the new previous two values using a neat application of tuple unpacking. Now that we have a generator, it can be used like any other iterable object. For instance, to print the Lucas numbers, you could use a loop like this. Of course, since the Lucas sequence is infinite, this will run forever, printing out values until your computer runs out of memory. Use Ctrl+C to terminate the loop.

Generator Expressions

Generator expressions are a cross between comprehensions and generator functions. They use a similar syntax as comprehensions, but they result in the creation of a generator object, which produces the specified sequence lazily. The syntax for generator expressions is very similar to list comprehensions delimited by parentheses instead of the brackets used for list comprehensions. Generator expressions are useful for situations where you want the lazy evaluation of generators with the declarative concision of comprehensions. For example, this generator expression yields a list of the first 1 million square numbers. At this point, none of the squares have been created, we just captured the specification of the sequence into a generator object. We can force evaluation of the generator by using it to create a long list. This list obviously consumes a significant chunk of memory, in this case, about 40 MB for the list object and the integer objects contained therein. Also, notice that a generator object is just an iterator, and once run exhaustively in this way, will yield no more items. Repeating the previous statement returns an empty list. Generators are single use objects. Each time we call a generator function, we create a new generator object. To recreate a generator from a generator expression, we must execute the expression itself once more. Let's raise the stakes by computing the sum of the first 10 million squares using the built-in some function, which accepts an iterable series of numbers. If this were a list comprehension, we could expect this to consume around 400 MB of memory. Using a generator expression, memory usage will be insignificant. This produces a result in a second or so and uses almost no memory.

Looking carefully, you could see that, in this case, we didn't supply separate enclosing parentheses for the generator expression, in addition to those needed for this sum function call. This elegant ability to have the parenthesis used for the function call also serve for the generator expression aides readability. You could include the second set of parentheses if you wish. As with comprehensions, you could include an IF clause in the end of the generator expression. Reusing our admittedly inefficient is_prime predicate, we can determine the sum of those integers for the first 1,000 which are prime like this. Note that this is not the same thing as computing the sum of the first 1,000 primes, which is a more awkward question because we don't know in advance how many integers we need to test before we clock up 1,000 primes.

Iteration Tools

So far, we've covered the many ways Python offers for creating iterable objects. Comprehensions, generators, and any object which follows the iterable or iterator protocols can be used for iteration, so it should be clear that iteration is a central feature of Python. Python provides a number of built-in functions for performing common iterator operations. These functions form the core of a sort of vocabulary for working with iterators, and they can be combined to produce powerful statements in very concise, readable code. We've met some of these functions already, including enumerate for producing integer indices and sum for computing summation of numbers. In addition to the built-in functions, the itertools module contains a wealth of useful functions and generators for processing iterable streams of data. We'll start demonstrating these functions by solving the first 1,000 primes problem using the built-in sum with two generator functions from intertools, islice and count. Earlier, we made our own take generator function for lazily retrieving the start of the sequence. We needn't have bothered, however, because islice allows us to perform lazy slicing similar to the built-in list slicing functionality. To get the first 1,000 primes, we need to do something like islice(all_primes, 1000). But how to generate all primes? Previously, we've been using range to create the raw sequences of integers to feed into our primality test, but ranges must also be finite, that is bounded on both ends. What we'd like is an open-ended version of range, and that is exactly what itertools count provides. Using count and islice, our first 1,000 primes expression can be written out as the islice of a generator of all prime numbers. This returns a special islice object which is iterable. We can convert it to a list using the list constructor. Answering our question about the sum of the first 1,000 primes is now easy, remembering to recreate the generators. We simply pass that islice we just defined to the sum function. Two other very useful built-ins which facilitate elegant programs, are any and all. They are equivalent to the logical operators 'and' and 'or', but for iterable series of bool values.

Any takes an iterable and tells you if any elements in it are true. All takes an iterable and tells you if all the elements in it are true. Here we'll use any together with a generator expression to answer the question of whether there are any prime numbers in the range 1328 to 1360, inclusive. Any is prime of X for X in range 1323 to 1361. For a completely different type of problem, we can check that all of these city names are proper nouns with initial uppercase letters. All of name equals name.title for each name in a list of cities. The last built-in we'll look at is zip, which, as its name suggests, gives us a way to synchronize iteration over two iterable series. For example, let's define a column of temperatures for sunday and a column for monday. If we zip these together, we can see that zip yields tuples when iterated. This, in turn, means we can use it with tuple unpacking in the for loop. In fact, zip can accept any number of iterable arguments. Let's add a third time series. We then zip together sunday, monday, and tuesday, and use min, max, sum, and len to calculate some statistics for our data points. Note how we've used string formatting features to control the numeric column width to four characters. Perhaps though, we'd like one long temperature series for sunday, monday, and tuesday. We can lazily concatenate iterables using itertools.chain, so this is different from simply concatenating three lists into a new list. We can now check that all of those temperatures are above freezing point without the memory impact of data duplication. Before we summarize, we'd like to pull a few pieces of what we've made together and leave your computer computing the Lucas primes. When you've seen enough, we recommend you spend some time exploring the itertools module.

Summary

Iteration is a pervasive and powerful concept in Python. Having a good understanding of how to work with iterables and iteration is often the key to designing elegant, sophisticated solutions to your problems. In this module, we learned that comprehensions are a concise syntax for describing lists, sets, and dictionaries. Comprehensions operate on a iterable source objects and apply an optional predicate filter and a mandatory expression, both of which are usually in terms of the current item. Iterable objects are objects for which we can iterate item by item. We retrieve an iterator from an iterable object using the built-in iter function. Iterators produce items one by one from the underlying iterable series each time they are passed to the built-in next function. Iterators raise a StopIteration exception when the collection is exhausted. Generator functions allow us to describe sequences using imperative code. Generator functions contain at least one use of the yield keyword. Generators are iterators. When the iterator is advanced with next, the generator starts or resumes execution up to and including the next yield. Each call to a generator function creates a new generator object. Generators can maintain explicit state and local variables

between iterations. Generators are lazy and so can model infinite series of data. Generator expressions have a similar syntactic form to list comprehensions and allow for a more declarative and concise way of creating generator objects. Python includes a rich set of tools for dealing with iterable series, both in the form of built-in functions such as sum, any, and zip, but also in the iterables module. In the next module of Core Python: Getting Started, we'll look a topic that you might have expected to see earlier in the course, that of classes. Classes allow you to define the structure and behavior of objects in a unified manner, and Python has strong support for defining them. Thanks for watching, and we'll see you in the next module.

Classes

Overview

You can get a long way in Python using the built-in scalar and collections types. For many problems, the built-in types and those available in the Python standard library are completely sufficient. Sometimes, though, they aren't quite what's required and the ability to create custom types is where classes come in. In this module of Core Python: Getting Started, we'll define what we mean by class. We'll investigate the relationship between classes and an object's type. We'll see how to define new classes. We'll learn about instance methods, and we'll see how to add them to classes, and we'll look into the ubiquitous self argument of instance methods. We'll discover how to add initializers to classes. We'll discover how initializers are similar to, but different from constructors that you may be familiar with in other languages, and we'll explore their role in helping enforcing variance in your programs. We'll look at how classes could collaborate, and we'll see how this could help you decompose the problem space into manageable elements. We'll discuss techniques for separating your class' public APIs from their implementation details. We'll see how you can painlessly mix object-oriented design with designs based around functions, and in doing so, hopefully gain a deeper insight into Python's everything is an object approach. We'll explore how classes intersect with Python's notion of duck typing, and we'll look at the basics of class inheritance in Python.

Classes

As we've seen, all objects in Python have a type, and when we report that type using the type built-in function, the result is couched in terms of a class. Classes are a means of defining the

structure and behavior of objects at the time we create the object. Generally speaking, the type of an object is fixed throughout its lifetime. As such, classes act as a sort of template or pattern according to which new objects are constructed. The class of an object controls its initialization and which attributes and methods are available through that object, for example, on a string object the methods we can use on that object such a split are defined in the str class. Classes are a key piece of machinery for Object Oriented Programming. And although it's often true that OOP is useful for making complex problems more tractable, it also commonly has the effect of making the solution to simple problems unnecessarily complex. A great thing about Python is that it's highly object oriented without forcing you to deal with classes until you really need them. This sets the language starkly apart from Java and C#.

Defining Classes

Python gives us the tools to define new classes, which can be completely novel or based on existing classes. Class definitions are introduced by the class keyword, followed by the class name. By convention, new class names in Python use upper CamelCase, sometimes known as Pascal Case, with a capital letter for each and every component word. Since classes are awkward to define at the REPL, we'll be using a Python module file to hold our class definitions. Let's start with the very simplest class to which we'll progressively add features. For this module, I'll be using the pycharm Python IDE so that it's easy to follow code examples and their use in the REPL. In this example, we'll model a passenger aircraft flight between two airports by putting this code into airtravel.py. The class statement introduces a new block, so we indent on the next line. Empty blocks aren't allowed, so the simplest possible class needs at least a do-nothing pass statement to be syntactically admissible. Just as with def for defining functions, class is a statement that can occur anywhere in a program and which binds a class definition to a class name. When the top-level code in the airtravel module is executed, the class will be defined. We can now import our new class into the REPL and try it out. From airtravel, import Flight. The thing we've just imported is the class object. Everything is an object in Python, and classes are no exception. To use this class to mint a new object, we call it's constructor, which is done by calling the class object just as we would a function. The constructor returns a new object, which we here assign to the name f. If we use the type function to request the type (f), we get class 'airtravel.Flight' again. The type of f literally is the class.

Instance Methods

Let's make our class a little more interesting by adding a so-called instance method, which returns the flight number. Methods are just functions to find within the class, and instance methods are functions which can be called on objects, or instances, of our class, such as f. Instance methods must accept a reference to the actual instance on which the method was called as the first argument. And by convention, this argument is always called self. We have no way of configuring the flight number value yet, so we'll just return a constant string to find the function number within the body of Flight and return this string SN060. From a fresh REPL, import Flight from airtravel. Construct a Flight instance bound to the name f and call number on f. Notice that when we call the method, we do not provide the instance f for the actual argument self in the argument list. That's because the standard method invocation form with a dot, f.number, is simply syntactic sugar for Flight.number(f). If you try the latter, you'll find that it works as expected, although you almost never see this form used for real.

Instance Initializers

This class isn't very useful, because it can only represent one particular flight. We need to make the flight number configurable at the point that the flight is created. To do that, we need to write an initialization method. If provided, the initialization method is called as part of the process of creating a new object when we call the constructor. The initializer method must be called dunder init, delimited by the double underscores used for Python runtime machinery. Like all other instance methods, the first argument to dunder init must be self. First, define dunder init in the body of the flight class. In this case, we also pass a second argument to dunder init, which is the flight number. Dunder init creates a new reference on self called underscore number and points it at the argument number. If you're coming from a Java C# or C++ background, it's tempting to think of dunder init as being the constructor. This isn't quite accurate. In Python, the purpose of dunder init is to configure an object that already exists by the time dunder init is called. The self argument is, however, analogous to this in Java, C#, or C++. In Python, the actual constructor is provided by the Python runtime system, and one of the things it does is check for the existence of an instance initializer and call it when present. Within the initializer, we assign to an attribute of the newly created instance called underscore number. Assigning to an object attribute that doesn't yet exist is sufficient to bring it into being. The initializer should not return anything; it simply modifies the object referred to by self. Just as we don't need to declare variables until we create them, neither do we need to declare object attributes before we create them. We choose underscore number with a leading underscore for two reasons. First, because it avoids a name clash with the method of the same name. Methods are functions, functions are objects, and these

functions are bound to attributes of the object. So we already have an attribute called number and we don't want to replace it. Second, there's a widely followed convention that the implementation details of an object, which are not intended for consumption or manipulation by clients of the object should be prefixed with an underscore. We also modify our number method to access the underscore number attribute and return it. Any arguments past the flight constructor will be forwarded to the initializer. So to create and configure our flight object, we can now call the flight class and pass in the flight number as a string. We can then access the flight number through the number method. We can also directly access the implementation details. Although this is not recommended for production code, it's very handy for debugging in early testing. If you're coming from a bondage and discipline language like Java or C# with public, private, and protected access modifiers, Python's everything-is-public approach can seem excessively open-minded. The prevailing culture among Pythonistas is that we're all consenting adults here. In practice, the leading underscore convention has proven sufficient protection even in large and complex Python systems we've worked with. People know not to use these attributes directly, and in fact they tend not to. Like so many doctrines, lack of access modifiers is a much bigger problem in theory than in practice. It's good practice for the initializer of an object to establish so-called class invariants. The invariants or truths about objects of that class should endure for the lifetime of the object. One such invariant for flights is that the flight number always begins with an uppercase two-letter airline code, followed by a three- or four-digit route number. In Python, we can establish class invariants in the dunder init method and raise exceptions if they can't be attained. We first used string slicing in str.isalpha to verify that the first two characters of the flight number are alphabetic, raising a ValueError if not. We then check if the same slice is uppercase by using str isupper, again, raising a ValueError if not. Finally, we use slicing, str isdigit and the int constructor to verify that the rest of number comprises only digits and that it represents a value between 0 and 9999; once again, raising a ValueError if not. For the first time in this course, we also see the logical negation operator, not. Ad hoc testing in the REPL is a very effective technique during development. We can construct flights using valid flight numbers. If we try to use a flight number without letters at the front, we get a ValueError. Similarly, if the letters at the front aren't uppercase, we get a ValueError with a different message. Here we try to use a flight number that doesn't have digits in the tail with predictable results. And in this case we get a ValueError because the flight number has too many digits. Now that we're sure the new flight objects have a valid flight number, we'll add a second method to return just the airline code. Once the class invariants have been established, most query methods could be very simple; in this case, simply returning a slice of the stored flight number.

A Second Class

One of the things we'd like to do with our flight is accept seat bookings. To do that, we need to know the seating layout, and for that we need to know the type of aircraft. Let's make another class to model different kinds of aircraft. The initializer creates four attributes for the aircraft, registration number, a model name, the number of rows of seats, and the number of seats per row. In a production code scenario, we could validate these arguments to ensure, for example, that the number of rows was not negative. The registration method returns the registration, and the model method returns the aircraft's model. This is straightforward enough, but for the seating plan, we'd like something a little more in line with our booking system. Rows in aircraft are numbered from 1, and the seats within each row are designated with letters from an alphabet, which omits I to omit confusion with 1. We'll add a seating_plan method, which returns the allowed rows and seats as a tuple containing a range object and a string of seat letters. This might be a bit much to take in all at once, so it's worth pausing for a second to make sure you understand how this function works. The range call produces an interval sequence of row numbers, up to the number of rows in the plane. The string and its slice method return a string with one character per seat. These two objects, the range and the string, are bundled up into a tuple. With that in mind, let's construct a plane with registration number G-EUPT, model Airbus A319, 22 rows of seating, and 6 seats per row. We can access its registration, model, and seating_plan through the associated instance methods. See how we used keyword arguments for the rows and seats for documentary purposes. Also recall the ranges are half open, so 23 is intentionally 1 beyond the end of the range.

Collaborating Classes

The Law of Demeter is an object-oriented design principle that says you should never call methods on objects you received from other calls, or put another way, only talk to your friends. We'll modify our Flight class to accept an aircraft object when it is constructed, and we'll follow the Law of Demeter by adding a method to report the aircraft model. This method will delegate the aircraft on behalf of the client, rather than allowing the client to reach through the flight and interrogate the aircraft object directly. We also added docstring to the class. These work just like function and module docstrings. We can now construct a flight, in this case flight number BA758, with a specific aircraft, another Airbus A319. Notice that we construct the Aircraft object and directly pass it to the Flight constructor without needing an intermediate named reference to it. Once this is constructed, we can access the aircraft's model directly from our Flight object.

Moment of Zen

Many moving parts combined in a clever box are now one good tool. The aircraft model method is an example of complex is better than complicated. The Flight class is more complex. It contains additional code to drill down through the aircraft reference to find the model. However, all clients of flight can now be less complicated. None of them need to know about the Aircraft class, dramatically simplifying the system.

Booking Seats

Now we can proceed with implementing a simple booking system. For each flight we simply need to keep track of who is sitting in each seat. We'll represent the seat allocations using a list of dictionaries. The list will contain one entry for each seat row, and each entry will be a dictionary from seat letter to occupant name. If the seat is unoccupied, it will contain none. We initialize the seating plan in the flight initializer using this fragment. In the first line, we retrieve the seating plan for the aircraft and use tuple unpacking to put the row and seat identifiers into local variables. In the second line, we create a list for the seat allocations. Rather than continually deal with the fact that row indices are one-based, whereas Python lists are zero-based, we choose to waste one entry at the beginning of the list. This first wasted entry is the single element list containing none. To this single element list, we concatenate another list containing one entry for each real row in the aircraft. This list is constructed by a list comprehension, which iterates over the rows objects, which is the range of row numbers retrieved from the _aircraft on the previous line. We're not actually interested in the row number, since we know it will match with the list index in the final list, so we discard it by using the dummy underscore variable. The item expression part of the list expression is itself a comprehension, specifically a dictionary comprehension. This iterates over each letter for the row and creates a mapping from the single letter string to none to indicate an empty seat. We use a list comprehension rather than list replication with the multiplication operator because we want a distinct dictionary object to be created for each row. Before we go further, let's test our code in the REPL. We'll create Flight BA758 from an AirbusA319, just like before. Thanks to the fact that everything is public, we can access implementation details during development, and it's clear enough that we're doing so since the leading underscores remind us what's public and what's not. That's accurate, but not particularly beautiful. Let's try again with pretty print. Import the pprint function from the pprint module and rename it to pp. Then use pp to print the seating structure. Perfect. Now we'll add behavior to flight to allocate seats to passengers. To keep this simple, a passenger will simply be a string name. Most of this code is validation of the seat designator, and it contains some interesting snippets. Methods or functions

so deserve doc strings too. We get the seat letter by using negative indexing into the seat string. We test that the seat letter is valid by checking for membership of seat letters using the in membership testing operator. We extract the row number using string slicing to take all but the last character. We try to convert the row number substring to an integer using the int constructor. If this fails, we catch the value error and, in the handler, raise a new value error with a more appropriate message payload. We conveniently validate the row number by using the in operator against the rows object, which is a range. We can do this because range supports the container protocol. We checked that the requested seat is unoccupied, using an identity test with none. If it's occupied, we raise the value error. If we get this far, everything's in good shape and we can assign the seat. It also contains a bug that we'll discover soon enough. Trying our seat allocator at the REPL, we first construct a flight. We then put Python's BDFL, Guido van Rossum, in seat 12A. Early on in your object-oriented Python career, you're likely to see type error messages like this quite often. The problem has occurred because we forgot to include the self argument in the definition of the allocate_seat method. Once we fix that, we can try again. Construct the flight object, allocate seat 12A to Guido, attempt to allocate 12A to Rasmus Lerdorf, and see that it raises a value error to prevent us from assigning two people to the same seat. Let's seat Bjarne Stroustrup and Anders Hejlsberg next to one another on Row 15. When we try to put Yukihiro Matsumoto in E27, we get a ValueError telling us that we've used an invalid seat letter. We'll put John McCarthy and Rich Hickey in Row 1, but we'll get yet another ValueError when we try to put Larry Wall in seat DD. Now we can see our seating chart using the pprint function. The Dutchman is quite lonely there on row 12, so we'd like to move him back to Row 15 with the Danes. To do so, we'll need a relocate_passenger method.

Methods for Implementation Details

First, we'll perform a small refactoring and extract the seat designator parsing and validation logic into its own method, _parse_seat. We use a leading underscore here because this method is an implementation detail. The new _parse_seat method returns a tuple with an integer row number and a seat letter string. This leaves allocate seat much simpler since it can delegate most of its work to _parse_seat. Notice that method calls within the same object, such as this call to _parse_seat, also require explicit qualification with the self. prefix. Now we've laid the groundwork for our relocate passenger method. This parses and validates the from seat and to seat arguments and then moves the passenger to the new location. It's also getting tiresome recreating the flight object each time, so we'll add a module level convenience function for that, too. We'll call the function make_flight. First, it will construct a flight object with its associated

aircraft, then we'll allocate a seat to Guido van Rossum, Bjarne Stroustrup,Anders Hejlsberg, John McCarthy, and Rich Hickey. Finally, we return the flight. It's quite normal to mix related functions and classes in the same module like this. Now from the repel, we can import the make_flight function and use it to replace the boiler code we've been using to construct flight objects. You may find it remarkable we have access to the Flight class when we have only imported a function, make_flight. This is quite normal and it's a powerful aspect of Python's dynamic type system that allows us this very loose coupling between code. Let's get on and move Guido back to Row 15 with his fellow Europeans. We'll use flight relocate_passenger method to move him from 12A to 15D. If we print the seating chart, we can see that he is now in the right place. It's important during booking to know how many seats are available. To this end, we'll write a num_available_seats method. This is achieved using two nested generator expressions. The outer expression filters for all rose, which are not none. This excludes are dummy first row. The value of each item in the outer expression is the sum of the number of none values in each row. We split this outer expression over three rows to improve readability. This inner expression iterates over values of the dictionary and adds one for each none found. We can make a flight and ask how many seats it has available. We can verify this result by multiplying the number of seats per row, 5, by the number of rows on the plane, 22, and subtracting the number of occupied seats, 5, resulting in 127, the same number we got above.

Object-Oriented Design with Function Objects

Now we'll show how it's quite possible to write nice object-oriented code without needing classes. We have a requirement to produce boarding cards for our passengers in alphabetical order, however, we realize that, by following the engineering principle of separation of concerns, the Flight class is probably not a good home for details of printing boarding passes. We could go ahead and create a boarding card printer class, but that's probably overkill. Remember that functions are objects too, and are perfectly sufficient for many cases. Don't feel compelled to make classes and objects without a good reason. Rather than have a card printer query all the passenger details from the flight, we'll follow another object-oriented design principle of Tell, Don't Ask and have the flight tell a simple card printing function what to do. First, the card_printer, which is just a module-level function. We first create the primary output string from the boarding card. The new Python feature here is the use of line continuation backslash characters, which allow us to split a long statement over several lines. This is used together with implicit string concatenation of adjacent strings to produce one long string with no line breaks. We then measure the length of the output line and build some banners and borders around it.

Finally, we concatenate the lines together using the join method called on a new line separator, before printing the whole card followed by a blank line. Note that the card_printer doesn't know anything about flights or aircraft. It's very loosely coupled. You can probably easily envisage an HTML card_printer that has the same interface. To the Flight class, we add a new method make_boarding_cards, which accepts a card_printer. This calls the implementation detail _passenger_seats method, sorts the results, and loops over the resulting passenger_seat tuples. For each of these, it then calls the card_printer that was passed in. The _passenger_seats method is, in fact, a generator function, which searches all seats for occupants, yielding the passenger and the seat number as they're found. It first determines the number of rows and seat letters in each row. Then it loops over the row_numbers, and for each row loops over the seat_letters. It finds the passenger in that seat, and if the passenger is not none, yields the passenger and seat information. Now, if we run this in the REPL, we can see that the new boarding card printing system works. We import the console_card_printer and make flight functions, make a new flight object, and then print the boarding cards with the console_card_printer function.

Polymorphism and Duck Typing

Polymorphism is a programming language feature which allows us to use objects of different types through a uniform interface. The concept of polymorphism applies to functions and more complex objects. We've just seen an example of polymorphism with the card printing example. The make_boarding_card method didn't need to know about an actual or, as we say, concrete card printing type, only the abstract details of its interface, essentially just the order of its arguments. Replacing our console_card_printer with the putative HTML card printer would exercise polymorphism. Polymorphism in Python is achieved through duck typing. Duck typing is in turn named after the duck test attributed to James Whitecomb Riley, the American poet. When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. Duck typing, where an object's fitness for a particular use is only determined at runtime, is the cornerstone of Python's object system. This is in contrast to statically typed languages where a compiler determines if an object can be used, and in particular it means that an object's suitability is not based on inheritance hierarchies, base classes, or anything except the attributes an object has at the time of use. Again, this is in contrast of languages such as Java, which depend on nominal subtyping through inheritance from base classes and interfaces. We'll talk more about inheritance in the context of Python shortly. Let's return to our Aircraft class. The design of this class is somewhat flawed in that objects instantiated using it depend on being supplied with the seating configuration that matches the aircraft model, which for the purposes

of this exercise, we assume is fixed per model. Better and simpler, perhaps, is to get rid of the Aircraft class entirely and make separate classes for each specific model of aircraft with a fixed seating configuration. Here's an Airbus A319 and a Boeing 777. These two Aircraft classes have no explicit relationship to each other or to our original Aircraft class beyond having identical interfaces to each other with the exception of the initializer, which now takes fewer arguments. As such, we can use these new types in place of each other. Let's change our make_flight method into make_flights to use them. It will first construct Flight BA758 with an AirbusA319 and allocate some seats to passengers. It will then construct Flight AF72 with the Boeing777 and seat some passengers there. Finally, it will return both flights as a tuple. The different types of aircraft both work fine when used with Flight because they both quack like ducks or fly like planes or something. We import everything from airtravel and then make the flights, binding them to the names f and g. We can see that f has model Airbus A319 and g is a Boeing 777. F has 127 seats available, while the much larger Boeing has 491 seats available. We can, of course, relocate passengers on the flights and make boarding cards as before. Duck typing and polymorphism is very important in Python. In fact, it's the basis for the collection protocols we discussed such as iterator, iterable, and sequence.

Inheritance and Implementation Sharing

Inheritance is a mechanism whereby one class, the sub class, can be derived from another class, the base class, allowing us to make behavior more specific in the sub class. In nominally-typed languages, such as Java, class-based inheritance is the means by which runtime polymorphism is achieved. Not so in Python, as we've just demonstrated. The fact that no Python method calls or attribute lookups are bound to actual objects until the point at which they're called, a quality known as late binding, means we can attempt polymorphism with any object, and it will succeed if the object fits. Although inheritance in Python can be used to facilitate polymorphism, after all derived classes will have the same interface as base classes, inheritance in Python is mostly useful for sharing implementation between classes. As usual, this will make much more sense with an example. We would like our aircraft classes, AirbusA319 and Boeing777 to provide a way of returning the total number of seats. We'll add a method called num_seats to both classes to do this. The implementation can be identical in both cases, since it could be calculated from the seating plan. Unfortunately, we now have duplicate code across two classes, and as we add more aircraft types, the code duplication will only worsen. The solution we'll look at here is to extract the common elements of AirbusA319 and Boeing777 into a base class from which both aircraft types will derive. Let's recreate the class Aircraft, this time with the goal of using it as a base

class. It contains just the method we want to inherit into the derived class, num_seats. This class isn't usable on its own because it depends on a method called seating_plan, which isn't available in Aircraft. Any attempt to use Aircraft standalone will fail. Here we construct an aircraft and ask for its number of seats. Since this aircraft instance doesn't have a seating plan method, we get an attribute error. The class is abstract insofar as it's never useful to instantiate it alone. So how can we use aircraft meaningfully? That's where derived classes come in. We specify inheritance in Python using parentheses containing the base class name immediately after the class name in the class statement. Here's the Airbus class, now inheriting from Aircraft, and this is the Boeing class. Let's exercise them at the REPL. Now, our AirbusA319 objects have a num_seats method, as do instances of Boeing777. We can see that both aircraft subclasses inherited the num_seats method, which now works as expected because the call to a seating plan, defined on both sub classes, is successfully resolved on the self object at runtime. Now that we have the base Aircraft class, we can hoist other common functionality into it. In this case, initializer and registration methods are identical between the two subtypes. Now the derived classes only contain the specifics for that aircraft type. All shared functionality is inherited from the base class. Thanks to duck-typing, inheritance is less used in python than in other languages. This is generally seen as a good thing because inheritance is a very tight coupling between classes.

Summary

Classes are an important tool for any Python programmer, and this module introduced the essentials of how classes work in Python. While there is certainly more to know about classes in Python, much more in fact, for many Python users this may be all that you need to know to get your work done. Indeed, if you follow our advice to reserve classes for when other simpler techniques fall short, you may be surprised at how infrequently you really need them at all. In this module, we learned that all types in Python have a class. Classes define the structure and behavior of an object. The classes determine when an object is created and is fixed for the lifetime of the object in the general case. Classes are the key support for object-oriented programming in Python. Classes are defined using the class keyword followed by the class name, which is in camelCase. Instances of a class are created by calling the class as if it were a function. Instance methods are functions defined inside the class, which should accept an object instance called self as the first parameter. Methods are called using the instance.method syntax, which is syntactic sugar for passing the instance as the formal self argument to the method. An optional special initializer method called dunder init can be provided, which is used to configure the self object at creation time. If present, the constructor calls the dunder init method. Dunder init is not

the constructor. The object has been constructed by the time the initializer is called. Arguments passed to the constructor are forwarded to the initializer. Instance attributes are brought into existence simply by assigning to them. Attributes and methods which are implementation details are, by convention, prefixed with an underscore. There are no public, protected or private access modifiers in Python. Access to implementation details from outside the class can be very useful during development, testing, and debugging. Class invariants should be established in the initializer. If the invariants can't be established, raise exceptions to signal failure. Methods have docstrings just like regular functions. Classes can have docstrings. Even within an object, method calls must be preceded with self. You can have as many classes and functions in a module as you wish. Related classes and global functions are usually grouped together this way. Polymorphism in Python is achieved through duck-typing, where attributes and methods are only resolved at point of use, late binding. Polymorphism in Python does not require shared base classes or named interfaces. Class inheritance in Python is primarily useful for sharing implementation, rather than being necessary for polymorphism. All methods are inherited, including special methods like the initializer. Along the way, we found that strings support slicing because they implement the sequence protocol. Following the Law of Demeter can reduce coupling. We can nest comprehensions. It can sometimes be useful to discard the current item in a comprehension using a dummy reference, conventionally, the underscore. When dealing with one-based collections, it's often easier just to waste one list entry. Don't feel compelled to use classes when a simple function will suffice. Functions are also objects. Complex comprehensions or generator expressions can be split over multiple lines to aid readability. Statements can be split over multiple lines using the backslash line continuation character. Use this feature sparingly, and only when it improves readability. Object-oriented design, where one object tells another information, can be more loosely coupled than those where one object queries another. Tell, don't ask. In the next module of Core Python: Getting Started, we'll take a look at how to work with file I/O in Python, both for working with text, as well as binary data. As part of this, we'll introduce you to the with block, Python's approach to managing resources like files. Thanks for watching, and we'll see you in the next module.

File IO and Resource Managements

Overview

File I/O, that is, reading from and writing to files, is at the heart of a great many programs. It's unsurprising then that Python provides a sophisticated array of support for working with both binary and text files, with mechanisms for doing everything from reading a specific byte in a file iterating over lines of text. Files are just one example of objects representing resources, elements of a program that should be released or closed after use. Since managing resources is critical for proper functioning of programs, Python provide special syntax and protocols that help you work with them. In this module of Core Python: Getting Started, we'll look at the core functions for opening files in Python. We'll discuss the difference between text mode and binary mode when interacting with files. We'll see how to read and write from files. And we'll see how to close them explicitly. We'll introduce you to context managers, Python's support for automatically managing resources. We'll show you Python's with keyword for using context managers. And we'll show you how to use with-blocks for running code that uses a resource. We'll take a deep look at using Python to work with binary file formats. We'll discuss the more abstract notion of file-like objects. And we'll look at some tools for creating context managers.

Opening Files

To open a file in Python, we call the built-in open function. This takes a number of arguments, but the most commonly used are: file, the path to the file, and this is required; mode: read, write, append and binary or text. This is optional, but we recommend always specifying it for clarity; explicit is better than implicit. Encoding: If the file contains encoded text data, which encoding to use. It's often a good idea to specify this. If you don't specify it, Python will choose a default encoding for you. At the file system level, files contain only a series of bytes. Python distinguishes between files opened in binary and text mode, even when the underlying operating system doesn't. Files opened in binary mode return and manipulate their contents as bytes objects without any decoding. Binary mode reflects the raw data in the file. A file opened in text mode treats its contents as if it contains text strings of the _____ type, the raw bytes having been first decoded, using a platform-dependent encoding or using the specified encoding if given. By default, text mode files also engage support for Python's universal new lines. This causes translation between a portable single new line character in our program strings, /n, in a platform-dependent new line representation of the raw bytes stored in the file system; for example carriage return, new line, /r/n on Windows. Getting the encoding right is crucial for correctly interpreting the contents of a text file, which is why we labor the point. If you don't specify an encoding, Python will use the default from sys.getdefaultencoding. Ours is utf-8, but

there's no guarantee that the default encoding on your system, is the same as the default encoding on another system with which you wish to exchange files. It's better for all concerned to make a conscious decision about the text-to-bytes encoding. You can get a list of supported text encodings in the Python documentation.

Writing Text

Let's start by writing some texts to a file. We'll be explicit about using the UTF-8 encoding because we have no way of knowing what your default encoding is. We'll use keyword arguments to make things clearer still. We'll assign to f the result of calling open. The first argument is the file name. The next argument is a string containing letters with different meanings. In this case, w means write and t means text. All mode strings should consist of a read, write, or append mode. One of the preceding should be combined with a selector for text or binary mode. So typical mode strings might be wb, write binary, or at, append text. Although both parts of the mode code support defaults, we recommend being explicit for the sake of readability. The exact type of the object returned by open depends on how the file was opened, dynamic typing in action, but for our purposes, it's sufficient to know that the object returned is a file-like object, and as such, we can expect it to support certain attributes and methods. We've shown previously how we can request help for modules and methods and types, but in fact, we can request help on instances, too. This makes sense when you remember that everything is an object. Let's call help on our file f. Browsing through the help, we can see that f supports a method write. Quit the help with Q and continue with the REPL. We'll write the string, What are the roots that clutch, to our file. The call returns the number of code points, or characters, written to the file. A few more lines, what branches grow, at 19 code points, and Out of this stony rubbish, with 27 code points. Note that it's the caller's responsibility to provide newline characters when they're needed. There's no write line method on Python file objects. When we finished writing, we should remember to close the file by calling the close method. Note that when we close the file, all the contents become visible to other programs. Closing files is important. If you now exit the REPL and look at your file system on UNIX with ls -l, you should see the wasteland.txt file with 78 bytes. On Windows, you can do the same with dir. Here you should see wasteland.txt with 79 bytes, 1 more than on UNIX, because Python's universal newline behavior for files has translated the line ending to your platform's native endings. The number returned by the write method is the number of code points, or characters in the string passed to write, not the number of bytes written to the file after encoding a universal newline translation. In general, when working with text files, you cannot sum the quantities returned by write to determine the length of the file in bytes.

Reading Text

To read the file back, we use open again, but pass a different mode string, open wasteland.txt, mode, rt, encoding= utf-8. In this case, we use the rt mode for read text. If we know how many bytes to read, or if we want to read the whole file, we can use read. Looking back through our REPL, we can see that the first write was 32 characters long. We can read that back with the call to the read method. In text mode, the read method accepts the number of characters to read from the file, not the number of bites. The call returns the text and advances the file pointer to the end of what was read. The return type is struck because we opened the file in text mode. To read all of the remaining data in the file, we can call read without any argument, giving us parts of two lines in one string. Note the new line character in the middle. At the end of the file, further calls to read return an empty string. Normally, when we have finished reading a file, we would close it, but for the purposes of this exercise, we'll keep the file open and move the file pointer back to the beginning of the file using the seek method with a 0 offset from the start of the file. The return value is the new file pointer position. As a quick aside, it's important to note that for text mode files, seek cannot be used to move to an arbitrary offset, in particular, when seeking from the beginning of the file, as we do here, the only legal values for offset are 0 and any value returned by the file's tell method. Any other values will result in undefined behavior. Using read for text is quite awkward, and thankfully Python provides better tools for reading text files line by line. The first of these is the readline function. The first call to readline reads to the first new line. The second call reads to the end of the file. The returned lines are terminated by a single new line character if there is one present in the file. The last line here does not terminate with a new line because there's no new line sequence at the end of the file. You shouldn't rely on the string returned by readline being terminated by a new line. Again, the universal newline support will have translated to /in from whatever the platform native newline sequence is. Once we reach the end of the file, further calls to readline return an empty string. Let's rewind again with g.seek0. Sometimes when we know we want to read every line in the file, and we're sure we have enough memory, we can read all lines into a list with the readlines method. This is particularly useful if parsing the file involves hopping backwards and forwards between lines. It's much easier to do this with a list of lines, then with file streams of characters. This time, we'll close the file before moving on.

Appending Text

Sometimes we'd like to append to an existing file. We can do that by opening the file with mode a, which opens the file for writing, appending to the end of the file if it already exists. In this

example, we can bind that with t to be explicit about using text mode, open('wasteland.txt', mode='at' encoding=' utf-8'. Although there's no WriteLine method in Python, there is a writelines method, which writes an iterable series of strings to a stream. If you want line endings on your strings, you must provide them yourself. This seems odd at first, but it preserves symmetry with read lines while also giving us the flexibility for using writelines to write any iterable series of strings to a file. Son of man, you cannot say or guess, for you know only a heap of broken images where the sun beats. Notice that only three lines are completed here. I say completed because the file we're appending to doesn't end with a new line.

Iterating over Files

The culmination of these increasingly sophisticated text file reading tools is the fact that file objects support the iterator protocol, with each iteration yielding the next line in the file. This means that they can be used in for loops and any other place where an iterator can be used. At this point, we'll take the opportunity to create a Python module called files.py. We can call this directly from the system command line, passing the name of our text file python3 files.py wasteland.txt. The double line spacing occurs because each line of the poem is terminated by a new line and then print adds its own. To fix that, we could use the strip method to remove the whitespace from the end of each line prior to printing. Instead, we'll use the write method of the standard out stream. This is exactly the same write method we used to write to the file, and can be used because the stream is a file-like object. We get hold of a reference to the standard out stream from the sys module. Replace print line with sys.stdout.write(line). Rerunning, we get the poem printed without extra blank lines. Now, alas, it's time to move on from one of the most important poems of the 20th century to get to grips with context managers.

Closing Files with Finally

For the next demonstration, we're going to need a data file containing some numbers. We'll write a sequence of numbers called Recaman sequence to a text file with one number per line. Recaman sequence itself isn't important to this exercise. We just needed a way of generating numeric data so we won't be explaining the sequence generator. Feel free to experiment, though. The module contains a generator for yielding the Recaman numbers and a function which writes the start of the sequence to file using the writelines method. A generator expression is used to convert each number to a string and add a new line. Enter tools.islice is used to truncate the otherwise infinite sequence. We'll write the first 1000 Recaman numbers to a file by executing the

module, passing the file name in series length as command line arguments. Python 3, recaman.py, recaman.dat 1000. We'll now take a complimentary module, series.py, which reads this data file back in. We simply read one line at a time from the open file, strip the new line with the call to the strip string method, and convert it to an integer. Running it from the command line, everything works as expected. Now let's deliberately create an exceptional situation. Open recaman.dat in a text editor and replace one of the numbers with something that isn't a stringified integer. Save the file and rerun. The int constructor raises a value error, which is unhandled, and so the program terminates with a stack trace. One problem here is that our f.close call was never executed. To fix that, we can insert a try finally block, put a try before the open call, indenting everything up to the close call, then put a finally before the close call, indenting that as well. Now the file will always be closed. Doing so opens up the opportunity for another refactoring. We can replace the for loop with the list comprehension and return this list directly. Return open bracket, int of line.strip for line in f, closed bracket. Even in this situation, close will be called. The finally block is called however the try block is exited.

With-blocks

Up to now, our examples have all followed a pattern, open a file, work with the file, close the file. The close is important because it informs the underlying operating system that you're done working with the file. If you don't close a file when you're done with it, it's possible to lose data. There may be pending writes buffered up which might not get written completely. Furthermore, if you're opening lots of files, your system may run out of resources. Since we always want to pair every open with a close, we want a mechanism that enforces that and makes sure that it happens even if we forget. This need for resource cleanup is common enough that Python implements a specific control flow structure called with-blocks to support it. With-blocks can be used with any object which supports the context-manager protocol. And this includes the file objects returned by open. Exploiting the context-manager nature of the file object and using the with-block, our read_series function can become open the file with a with-block and return the list comprehension. We no longer need to call close explicitly because the with construct will call it for us when and by whatever means execution exits the block. Now we can go back and modify our Recaman's series writing program to use a with-block too. This, again, removes the need for the explicit close.

Moment of Zen

Sugary syntax, thoughtlessness attained through sweet fidelity. The with blocks in tax is so called syntactic sugar for a much more complex arrangement of try except and try finally blocks. Few of us would want our code to look this convoluted, but for it to be reliable, this is how it would need to look without the with statement. Sugar may not be good for your health, but can be very healthy for your code, Which do you prefer?

Binary Files

To demonstrate handling of binary files, we need an interesting binary data format. The BMP file format contains device-independent bit maps, and it's simple enough that we can make a BMP file writer from scratch in this session. The code will be placed in a module bmp.py and is straightforward from a file-handling point of view. For simplicity's sake, we have decided to deal with 8-bit grayscale images, which have the nice property that they're 1 byte per pixel. The write_grayscale function accepts two arguments, the filename and a collection of pixels. As the docs string points out, this collection should be a sequence of iterable series of integers. A lists of lists of int objects would do just fine. Each int is a pixel value from 0 to 255. Each inner list is a row of pixels from left to right, and the outer list is a list of pixel rows from top to bottom. The first thing we do is figure out the size of the image by counting the number of rows to get the height and the number of items in the 0th row to get the width. We assume, but don't check, that all rows have the same length. In production code, that's a check we'd want to make. Next, we opened the file for write in binary mode using the wb mode string. We don't specify an encoding. That makes no sense for raw binary files. Inside the with block, we start writing what is called the BMP Header, which begins the BMP format. The header must start with a so-called magic byte sequence, BM, to identify it as a BMP file. We use the write method, and because the file was opened in binary mode, we must pass a bytes object. The next 4 bytes should hold a 32-bit integer containing the file size. We don't know that yet. We could have computed it in advance, but we'll take a different approach of writing a placeholder value for now, then returning to this point later to fill in the details. To be able to come back to this point, we use the tell method of the file object to give us the offset from the beginning of the file for the file pointer. We'll store this in a variable, which will act as a sort of bookmark. We write 4 0 bytes as the placeholder, using escape syntax to specify the 0s. The next 2 pairs of bytes are unused, so we just write 0 bytes of them too. The next 4 bytes are for another 32-bit integer, which should contain the offset in bytes from the beginning of the file to the start of the pixel data. We don't know that value yet either, so we'll store another bookmark using tell and write another 4-byte placeholder. We'll return here shortly when we know more. The next section is called the Image Header. The first thing we have

to do is write the length of the image as a 32-bit integer. In our case, the header will always be 40 bytes long. We just hardwire that as a hexadecimal. Notice that the BMP format is little-endian. The least significant byte is written first. The next 4 bytes are the image width as a little-endian 32-bit integer. We call a module scope implementation detail function here called _int32_to_bytes, which converts an int object into a bytes object containing exactly 4 bytes. We then use the same function again to deal with the image height. The remainder of the header is essentially fixed for 8-bit grayscale images, and the details aren't important here, except to note that the whole header does, in fact, total 40 bytes. Each pixel in an 8-bit BMP image is an index into a color table with 256 entries. Each entry is a 4-byte BGR color. For grayscale images, we need to write 256 4-byte gray values on a linear scale. This snippet is fertile ground for experimentation, and a natural enhancement to this function would be to be able to supply this pallet separately as a function argument. At last, we're ready to write the pixel data, but before we do, we make a note of the current file pointer offset using tell as this was one of the locations we need to go back and fill in later. Writing the pixel data itself is straightforward enough. We use the reversed built-in function to flip the order of the rows. BMP images are written bottom to top. For each row, we simply pass the iterable series of integers to the bytes constructor. If any of the integers are out of the range 0 to 255, the constructor will raise a value error. Each row of pixel data in a BMP file must be a multiple of 4 bytes long irrespective of image width. To achieve this, we compute the number of padding bytes required by subtracting the row length modulus 4 from 4. This value is used with a repetition operator applied to a single 0 byte to produce a bytes object containing 0, 1, 2, or 3 bytes. We write this to the file to terminate each row. After the pixel data, we are at the end of the file. We undertook to record this offset value earlier, so we record the current position using tell into an end-of-file bookmark (end_bookmark) variable. Now we can return in to fill our promises by replacing the placeholder offsets we recorded with the real values. First, the file length. To do this, we seek back to the size_bookmark we remembered back near the beginning of the file and write the size stored in eof_bookmark as a little-endian, 32-bit integer using our _int32_to_bytes function. Finally, we seek to the pixel data offset placeholder bookmarked by pixel offset bookmark and write the 32-bit integer stored in pixel_data_bookmark. As we exit the with block, we can rest assured that the context manager will close the file and commit any unbuffered writes to the file system.

Bitwise Operators

Dealing with binary files often requires pulling apart or assembling data at the byte level. This is exactly what our _n32_to_bytes function is doing. We'll take a quick look at it because it shows

some features of Python we haven't seen before. The function uses the Right-shift and Bitwise-and operators to extract individual bites from the integer value. Note that Bitwise-and uses the ampersand symbol to distinguish it from the logical and, which is the spelled out word and. The double arrow is the Right-shift operator, which shifts the binary representation of the integer right by the specified number of bits. The routine shifts the integer 1, 2, and 3 bytes to the right before extracting the least significant byte with the Bitwise-and at each position. The 4 resulting integers are used to construct a tuple, which is then passed to the bytes constructor to produce a 4-bytes sequence.

Pixel Data

In order to generate a BMP image file, we're going to need some pixel data. We've provided a simple module, fractal.py, which produces pixel values from the iconic Mandelbrot set fractal. We're not going to explain the fractal generation code in detail, still less the math behind it, but the code is simple enough and doesn't rely on any Python features we haven't encountered previously. The key takeaway is that the Mandelbrot function uses nested list comprehensions to produce a list of lists of integers in the range 0 to 255, representing an image of the fractal. The integer value at each point is produced by the mandel function. Let's fire up a REPL and use the fractal and BMP modules together. First, we use the Mandelbrot function to produce an image of 448 by 256 pixels. You'll get best results using images with an aspect ratio of 7 to 4. This last call may take a second or so. Our fractal generator is simple rather than efficient. We can take a look at the return to data structure, a list of lists of integers just as we were promised. Let's write those pixel values to a BMP file, import bmp bmp.write_grayscale ("mandel.bmp", pixels). Find the file and open it in an image viewer, for example, by opening it in your web browser.

Reading Binary Data

We're not going to write a full-blown BMP reader, although that would be an interesting exercise. We'll just make a simple function to determine the image dimension in pixels from a BMP file. We'll add the code into bmp.py. Of course, we can use a with statement to manage the file so we don't have to worry about it being properly closed. Inside the with block, we perform a simple validation check by looking for the first two magic bytes that we expect in a BMP file. If they are not present, we raise a value error, which will, of course, caused the context manager to close the file. Looking back at our BMP writer, we can determine that the image dimensions are stored exactly 18 bytes from the beginning of the file. We seek to that location and use the read method

to read two chunks of 4 bites each for the 2 32-bit integers, which represent the dimensions. Because we opened the file in binary mode, read returns a bytes object. We pass each of these two bytes objects to another implementation detailed called _bytes_to_int32, which assembles them back into an integer. The two integers representing image width and height are returned as a tuple. The _bytes_to_int32 function uses bitwise left shift and bitwise or, which is the vertical bar or pipe symbol, together with indexing of the bytes object to reassemble the integer. Note that indexing into a bytes object returns an integer. Let's use it, bmp.dimensions("mandel.bmp").

File-like Objects

There is a notion in Python of file-like objects. This isn't as formal as a specific protocol, like sequence protocol is for the tuple-like objects, but thanks to the polymorphism afforded by duck typing, it works well in practice. The reason it's not closely specified is that different types of data streams and devices have many different capabilities, expectations, and behaviors. So in fact, defining a set of protocols to model them would be quite complex without actually gaining us much in practice, other than a smug sense of theoretical achievement. This is where the EAFP, easier to ask forgiveness than permission philosophy, comes into its own. If you want to perform seek on a file-like object without knowing in advance that it supports random access, go ahead and try, literally. But be prepared to fail if the seek method doesn't exist or does exist but doesn't behave as you expect. You might say, if it looks like a file and reads like a file, then it's a file. We've actually seen this already. The objects returned to us when we open files and text in binary mode are actually of different types, although both with definite file-like behavior. There are other types in the Python standard library which implement file-like behavior. We saw one of them in action back at the beginning of the course, used to retrieve data from a URL on the internet. Let's exploit this polymorphism across file-like objects by writing a function to count the number of words per line in a file and return that information as a list. We'll call it words_per_line, accepting a single argument, flo. It'll return a list comprehension that takes the length of line.split for each line in the argument. Now we'll open a regular text file containing the fragment of T. S. Eliot's masterpiece we created earlier, with open, wasteland.txt, as real_file. And then we'll pass real_file to our new function and display the results. The actual type of real_file is _io.TextIOWrapper, which is an internal Python implementation detail. We'll now do the same using a file-like object representing a web resource referred to by a URL. First, we import urlopen from urllib.request. We then open the resource using urlopen("http://sixty-north.com/c/t.txt") as web_file. Count the words per line with words_per_line of web_file and display the results. In this case, the type of web_file is http.client.HTTPResponse, quite a different thing from what we saw before. However, since both

are file-like objects, our function can work with both. There's nothing magical about file-like objects. It's just a convenient and fairly informal description for a set of expectations we can place on an object which are exploited through duck typing.

Context Managers

The with statement construct can be used with any type of object which implements the context Manager protocol. We're not going to show you how to implement that in this course, but we will show you a simple way to make your own classes usable with the context manager, using the code in fridge.py. We'll import raid into the REPL and go on the rampage, from fridge import raid. Now we raid the fridge for bacon. Importantly, we remembered to close the door so the food will be preserved until our next raid. However, what if we raid the fridge for deep-fried pizza? This time we were interrupted by a health warning and didn't get around to closing the door. We can fix that by using a function called closing in the Python Standard Library contextlib module. After importing the function, we wrap our RefrigeratorRaider constructor call in a call to closing, which wraps our object in a context manager that always calls the close method on the wrapped object before exiting. We use this object to initialize a with block. Now we execute a raid for Spam. We see that the close is called twice. Our explicit call to close is unnecessary. So let's fix that up by removing the explicit call to close. A more sophisticated implementation, we check that the door was already closed and ignore other requests. Now, if we ill-advisedly raid for deep-fried pizza, we see that even though the health warning was triggered, the door was still closed for us by the context manager.

Summary

As you've seen, Python makes it easy to work with files or indeed, any file like object at many different levels, from byte level manipulation to high-level lines of encoded text. And Python provides excellent tools for doing the bookkeeping associated with files and other kinds of resources. Used diligently, the tools and techniques we've covered here will help ensure that you use files efficiently, safely, and with predictable results. In this module, we've learned that files are open using the built-in open function, which accepts a file mode to control, read, write, append behavior, and whether the file is to be treated as raw binary or encoded text data. For text data, you should specify a text encoding. Text files deal with string objects and perform universal newline translation and string encoding. Binary files deal with bytes objects with no newline translation or encoding. When writing files, it's our responsibility to provide newline characters for

line breaks. Files should always be closed after use. Files provide various line-oriented methods for reading and are also iterators, which yield line by line. Files are context managers, and the with statement can be used with context managers to ensure that cleanup operations, such as closing files, are performed. The notion of file-like objects is loosely defined, but very useful in practice. Exercise EAFP to make the most of them. Context managers aren't restricted to file-like objects. We can use tools in the context of standard library module, such as the closing wrapper, to create our own context managers. Along the way, we found that help can be used on instances of objects, not just types. Pythons supports bitwise operators and, or, and left and right shift. Well done on completing Core Python: Getting Started. It took you no small amount of effort to get here, but we're confident that your hard work will pay dividends when you start to use what you've learned. Python is a big language, and this course can really only hope to lay a foundation upon which you can start to build. We hope we've opened enough doors so that you can confidently and rapidly expand your Python knowledge. Look out for other core Python courses here on Pluralsight, which build on the knowledge you've gained here and which explain the many other tools and abstractions provided by Python for managing complexity. Remember to check out our Python Craftsman Book series, which covers these topics in written form. Specifically, you'll find these topics covered in The Python Apprentice, the first book in the trilogy. We'll be back with more content for the ever growing Python language and library. Please remember, though, that the most important characteristic of Python is that it's great fun to write Python software. Happy programming.

Course authors


Austin Bingham


Robert Smallshire

Course info

Level	Beginner
Rating	★★★½★ (183)

My rating ★★★★★

Duration 4h 6m

Released 12 Dec 2019

Share course