**Introduction to the Python Fundamentals Course**

Introduction

Welcome to Python Fundamentals. My name is Austin Bingham and over the coming modules myself and Robert Smallshire will guide you through the basics of the Python programming language. Python is a big language, and a single course can't possibly cover everything there is to know, but this course will serve you well as a solid foundation for becoming productive with Python. Python Fundamentals is the first in our trilogy of three Pluralsight courses which cover the core Python language. This course serves as the foundation for our Python - Beyond the Basics and Advanced Python courses. Our courses follow a thoughtfully designed spiral curriculum. We visit the same or closely related topics several times in increasing depth, sometimes multiple times in the same course. For example, later in Python Fundamentals, we will cover classes and single inheritance. In Python - Beyond the Basics, we cover multiple class inheritance. In the Advanced Python course, we cover what amounts to a type of virtual inheritance using Python's sophisticated abstract base class mechanism. In Python Fundamentals, we'll cover basic techniques you'll use constantly on your journey with Python. Starting in this module, we'll give a high-level overview of the Python language, what it is, and what's so great about it. We'll also give a bird's-eye view of the rest of this course so that you'll have an idea of what's to come.

Python Promo

First, what's so great about Python? Why do you want to learn it? There are lots of good answers to those questions. One, is that Python is powerful. The Python language is expressive and productive so you can create solutions quickly, and others can understand them easily. Often, you won't need to write much code because Python comes with a great standard library, and it's the center of a huge universe of wonderful third-party libraries. This is much of what makes Python so hugely popular across the diverse realms of web development, scientific computing, cloud configuration, data science, and education. Python has taken a prominent position in web development providing the back ends for high-traffic sites including YouTube, Instagram, Reddit, and Dropbox using Python web framework such as Django, Flask, or Pyramid. Python has become the leading environment for scientific computing, the specialist libraries ranging from Astropy to Biopython built on the foundations for fast numerical computing provided by NumPy and general scientific analysis with SciPy. Python plays an important role in deploying systems to the cloud with Python-based tools like Ansible, for configuration management, Boto, for Amazon Web Services, and Microsoft's Azure Software Development Kit for Python. Python technologies are at the center of a revolution in financial and other analytical fields thanks to wonderful tools for data analysis, visualization, and machine learning in the shape of Pandas, Matplotlib, Bokeh, TensorFlow, and scikit-learn. As you can see, with Python you can build everything from simple scripts to complex applications. You can do it quickly, you can do it safely, and you can do it with fewer lines of code than you might think possible, but that's just one part of what makes Python great. Another is that Python is wonderfully open. It's open source so you can get to know every aspect of it if you want. At the same time, Python is hugely popular and has a great community to support you when you run into trouble. This combination of openness and large user base means that almost anyone, from casual programmers to professional software developers can engage with the language at the level they need. But for many people, these reasons take backseat to something more important, Python is fun. Python's expressive, readable style, quick edit/run development cycle, and batteries-included philosophy mean that you can sit down and enjoy writing code rather than fighting compilers and thorny syntax. And Python will grow with you. As your experiments become prototypes, and your prototypes become products, Python makes the experience of writing software not just easier but truly enjoyable. In the words of Randall Munroe, "Come join us! "Programming is fun again. "

Course Structure

This course is broken up into ten separate modules. The modules build on one another so unless you've already had some exposure to Python, you'll need to follow them in order. We'll start with getting Python installed into your system and orienting you a bit. We'll then cover language elements, features, idioms, and libraries all driven by working examples that you'll be able to build along with the lecture. We are firm believers that you'll learn more by doing than just by watching so we encourage you to run the examples yourself. By the end of the course, you'll know the fundamentals of the Python language. You'll also know how to use third-party libraries, and you'll know the basics of developing them yourself. We'll even show you how to package up your code to make it easier for others to use what you've written. The course modules are: One, Getting Started, where we cover installing Python, look at some of the basic Python tools, and cover the core elements of the language and syntax. Two, Strings & Collections, where we look at some of the fundamental complex data types, strings, byte sequences, lists, and dictionaries. Three, Modularity, where we look at the tools Python has for structuring your code such as functions and modules. Four, Built-in Types and The Object Model, where we examine Python's type system and object system in detail, and where we develop a strong sense of Python's reference semantics. Five, Collection Types, where we go into more depth on some of the Python Collection Types as well as introduce a few more. Six, Handling Exceptions, where we learn about Python's exception handling system and the central role that exceptions play in the language. Seven, Comprehensions, Iterables and Generators, where we explore the elegant, pervasive, and powerful sequence-oriented parts of Python such as comprehensions and generator functions. Eight, defining new types with Classes, where we'll cover developing your own complex data types in Python using classes to support object-oriented programming. Nine, Files and Resource Management, where we'll look how to look at files in Python, and where we'll cover the tools Python has for resource management. Ten, Shipping Working and Maintainable Code, where we'll show you how to use Python's testing, debugging, and code distribution facilities to produce code that works and can be used by others. If you'd like a book to support you as you work through the material in this course, you can check out The Python Apprentice which is the companion volume to this Python Fundamentals course covering the same material in written form. By following the URL shown, you could obtain the book for a substantially discounted price. The Python Apprentice is the first book in our Python Craftsman trilogy. The next two books being The Python Journeyman and The Python Master which correspond to our Python - Beyond the Basics and Advanced Python Pluralsight courses respectively. All three are available to Pluralsight viewers at reduced prices.

Python Overview, Part 1

So what is Python? Simply put, Python is a programming language. It was initially developed by Guido van Rossum in the late 1980s in the Netherlands. Guido continues to be actively involved in guiding the development and evolution of the language, so much so that he's been given the title Benevolent Dictator For Life or more commonly BDFL. Python is developed as an open source project, and is free to download and use as you wish. The non-profit Python Software Foundation manages Python's intellectual property and plays a strong role in promoting the language and, in some cases, funding its development. On a technical level, Python is a strongly typed language in the sense that every object in the language has a definite type, and there's generally no way to circumvent that type. At the same time, Python is dynamically-typed meaning that there's no type checking of your code prior to running it. This is in contrast to statically-typed languages like C++ or Java, where a compiler does a lot of type checking for you, rejecting programs which misuse objects. Ultimately, the best description of the Python-type system is that it uses duck typing where an object's suitability for a context is only determined at run time. We'll cover this in more detail in Module Eight. Python is a general purpose programming language. It's not intended for use in any particular domain or environment, but instead can be fruitfully used for a wide variety of tasks. There are of course some areas where it's less suitable than others, for example, in extremely time sensitive or memory constrained environments, but, for the most part, Python is as flexible and adaptable as any modern programming language and more so than most. Python is an interpreted language. This is a bit of a misstatement technically because Python is normally compiled into a form of byte code before it's executed; however this compilation happens invisibly, and the experience of using Python is one of immediately executing code without a noticeable compilation phase. This lack of an interruption between editing and running is one of the great joys of working with Python. The syntax of Python is designed to be clear, readable, and expressive. Unlike many popular languages, Python uses white space to delimit code blocks and, in the process, does away with reams of unnecessary parentheses while enforcing a universal layout. This means that all Python looks alike in important ways, and you can learn to read Python very quickly. At the same time, Python's expressive syntax means that you can get a lot of meaning into a single line of code. This expressive, highly readable code means that Python maintenance is relatively easy. There are multiple implementations of the Python language. The original and still by far the most common implementation is written in C. This version is commonly referred to as CPython. When someone talks about running Python, it's normally safe to assume that they are talking about C Python, and this is the implementation that we'll be using for this course. Other implementations of Python include Jython which is written to target the Java virtual machine, IronPython which targets the. NET Runtime, and PyPy which is written in a specialized subset of Python called RPython. These implementations generally trail behind CPython which is considered to be the standard for the language. Much of what you will learn in this course will apply to all of these implementations. There are two important versions of the Python language in common use right now, Python 2, and Python 3. These two versions represent changes in some key elements of the language, and code written for one will not generally work for the other unless you take special precautions. Python 2 is older and more well established than Python 3, but Python 3 addresses some known shortcomings in the older version. Python 3 is the definite future of Python, and you should use it if at all possible. While there are some critical differences between Python 2 and 3, most of the fundamentals of the two versions are the same. If you learn one, most of what you know transfers cleanly to the other. In this course, we'll be teaching Python 3, but we'll point out important differences between the versions when necessary.

Python Overview, Part 2

Beyond being a programming language, Python comes with a powerful and broad standard library. Part of the Python philosophy is Batteries Included, meaning that you can use Python for many complex, real-world tasks out of the box, with no need to install third-party packages. This is not only extremely convenient, but it means that it's easier to get started learning Python using interesting engaging examples, something we aim for in this course. Another great effect of the batteries-included approach is that it means that many scripts, even non-trivial ones, can be run immediately on any Python installation. This removes a common annoying barrier to installing software that you face with some languages. The standard library has a generally high level of good documentation. APRS are well documented, and the modules often have good narrative descriptions with quick start guides, best practice information, and so forth. The standard library documentation is always available online at python. org, and you can install it locally if you need to. As the standard library is such an important part of Python, we'll be covering parts of it throughout this course. Even so, we won't be covering more than a small fraction of it so you're encouraged to explore it on your own. Finally, no description of Python would be complete without mentioning that to many people, Python represents a philosophy for writing code. Principles of clarity and readability are part of what it means to write correct or Pythonic code. It's not always clear what Pythonic means in all circumstances, and sometimes there may be no single correct way to write something, but the fact that the Python community is concerned about issues like simplicity, readability, and explicitness means that Python code tends to be more, well, beautiful. Many of Python's principles are embodied in the so-called Zen of Python. The Zen isn't a hardened fast set of rules, but rather a set of guidelines or touchstones to keep in mind when coding. When you find yourself trying to decide between several courses of action, these principles can often give you a nudge in the right direction. We'll be highlighting elements from The Zen of Python throughout this course. We think Python is a great language, and we're excited to help you get started with it. By the time you get through this course, you'll be able to write substantial Python programs, and you'll be able to read even more complex ones. More importantly, you'll have the foundation you need to go out and discover all the more advanced topics in the language, and hopefully, we'll get you excited enough about Python to actually do so. Python is a big language with a huge ecosystem of software built in and around it, and it can be a real adventure to discover everything it has to offer. So welcome to Python! We'll see you in the next module.

Getting Starting With Python 3

Introduction

Welcome to the second module of this Python Fundamentals course by Robert Smallshire and Austin Bingham. My name is Robert Smallshire and I'll be presenting this Getting Started module. In this module, we'll cover obtaining and installing Python on your system for Windows, Ubuntu Linux, and Mac OS. 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 be begun with Python 3, as Python 2 will not maintained from the year 2020. That said, most of the Python code we'll 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. 6. 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. 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. 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'll 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. Feel free to skip over the sections which aren't relevant to you, although we'll only spend a minute or two on each.

Installing Python 3 on Windows

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 Context Menu on the Start button choose Windows PowerShell. On older versions of Windows, you may need to use the CMD 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.

Installing Python 3 on macOS X

Now, let's see how to install Python 3 on Mac OS. For Mac OS, you need to visit the official Python website at python. org. Navigate via the Downloads tab to the Downloads for Mac OS and click the button to begin downloading the latest Python 3 version. A package 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 Mac OS 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 along side the 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.

Installing Python 3 on Linux

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, 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 possibly install Python 3 on your system.

The Read-Eval-Print-Loop or REPL

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. The Python command-line environment is a Read-Eval-Print-Loop. Python will read whatever 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 are 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. Two plus two is four and six times seven is 42. As you can see, Python reads our input, evaluates it, prints the result, and loops round to do the same again. We can assign variables in the REPL, such as x equals five, and print their contents simply by typing their name. We can refer to variables in expressions. Here, we do three times x. Within the REPL, you can use the special underscore variable to refer to the most recently printed value. This being one of 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 five 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 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 evaluated and displayed by the REPL, but is a side effect of the print function. As an aside, print is one of the most visible differences between Python 2 and Python 3. In Python 3, the parentheses are required, whereas in Python 2, they were 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 suspend the Python interpretor and return to your operating system 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 in Python

Start your Python 3 interpretor 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) terminated by a colon, Python will change the prompt to three dots to request that 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, x equals i times 10. Our loop body will contain a second statement. So, after pressing Return, at the next three dot prompt, we enter another four spaces, followed by a call to the built-in print function, print(x). To terminate our for-loop block, we must enter a blank line into the REPL at the three dots. 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 line and introduces an additional level of indentation, which continues until a dedent 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 whitespace 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. So, you can 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 they are straight-forward in practice. The whitespace you use can be either spaces or tabs. 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 interpretor 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 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 indented it. The 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 and the Zen of Python

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 four spaces for indentation in new Python code. Another of these PEPs, called PEP 20, is 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 interpretor, 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's significant indentation, this is a good time for our first Moment of Zen. (tranquil music) 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 others.

Importing From 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 the 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 library's 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 we're 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 square root. Let's try to use it. We do math. sqrt(81), which, of course, gives us nine. 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 as standard library modules. To get help, type, simply, help. We'll leave you to explore the first form for interactive help in 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 down through the help, and if you're on Mac or Linux, use the arrow keys to scroll up and down. Browsing through the functions, we can see there's 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. And press Q again to return to the REPL. Let's use the factorial function with math. factorial(5), which is 120. Or math. factorial(6), which is 720. Notice that the function accepts and returns an integer. See also how we need to qualify the function name with the module namespace. 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. Let's use factorials to compute how many ways there are to draw three fruit from a set of five fruit using some math we learned in school. We have n equals five, k equals three, the factorial of n over the factorial of k, multiplied by the factorial of n minus k, which gives 10. 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. From math import factorial. 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 only 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 Python's 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 the 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, the regular, garden-variety signed integers can only store values less than two to the power of 31. 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 three fruits 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 many fruit. We n equal to 100 and k equal to two, the result is 4, 950. Just to emphasis how large the size of the first term of that expression is, calculate 100 factorial on its own. Wow, that's a number vastly larger even than 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 the string, like this. That's our cue to move on and look at integers, strings, and other built-in types, in more detail.

Scalar Types: int, float, None and bool

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'll cover the following 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. Integer literals in Python are specified in decimal and may also be specified in binary with a zero b prefix, or octal with a zero o prefix, or hexadecimal with a zero x prefix. We can also construct integers by a call to the int constructor, which can convert from other numeric types, such as floats to integers. Here is int(3. 5), which gives three. Note that the rounding is always towards zero. So, int(-3. 5) gives minus 3. We can also convert strings to integers. Here's a string containing the digits four, nine, six, being converted to 496. And we can even supply an optional number base when converting from a string. Here we convert one zero zero zero, in base three, to 81 decimal. Floating-point numbers are supported in Python by the float type. Python floats are implemented as IEE-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 or a letter e is interpreted by Python as a float, here is 3. 125. Scientific notation can also be used. So, for large numbers, such as the approximate speed of light in meters per second, three times 10 to the eight, we can write 3e8. And for small numbers, like Planck's constant 1. 616 times 10 to the minus 35, we can enter 1. 616e minus 35. Notice how Python automatically switches the display representation to the most readable form. As for integers, we can convert to floats from other numeric or string types using the float constructor. Here's a float constructed from the integer seven and from a string containing 1. 618. This is also how we create the special floating-point values nan or not a number, and also positive and negative infinity. Be aware that the result of any calculation involving both int and float is promoted to a float. You can read more about Python's number types in the Python documentation. Python has a special nul value called None, 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 can be bound 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 response 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 would expect, there are two bool values, True and False, both with initial capitals. There's 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 falsey and all other values are truthy. So, bool(0) is False, bool(42) is True, and bool(-1) is also True. We see the same behavior with floats where only zero is considered falsey. So, bool(0. 0) is False, bool(0. 207) is True, and bool(-1. 117) is True. When converting from collections, such as strings or lists, only empty collections are treated as falsey. We'll be looking at lists in a lot more detail shortly. Converting an empty list, shown here by the square brackets to bool is False, whereas converting a list containing the integers one, five, and nine to bool is True because the list is not empty. Similar rules apply to strings. Only the empty string is False. Any other string, such as spam here, is True. In particular, you cannot use the bool constructor to convert from string representations of True and False. Converting False as a string to bool, gives True because the string False is not empty. 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.

Relational Operators

Bool values are commonly produced by Python's relational operators, which can be used for comparing objects. These include value equality or equivalence, 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 are Python's equality and inequality tests, which actually test for equivalence or inequivalence of values. That is, two objects are equivalent if one could be used in place of the other. We'll learn more about the notion of object equivalence later in the course. For now, we'll compare simple integers. Let's start by assigning or binding a value to a variable g by writing g equals 20. We test for equality with the double equals operator, g is equal to 20 is True and g is equal to 13, which is False. Or we can test for inequality using the not equals operator, g is not equal to 20 is False and g is not equal to 13 is True. We can also compare the order of quantities using the rich comparison operators, g is less-than 30 is True, g is greater-than 30, which is False, g is less-than or equal to 20, which is True, and g is greater-than equal to 20, which is also True.

Conditional Statements

Now 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. If True colon, remembering to indent four spaces within the block, we add some code to be executed if the condition is true, print("It's true! "). We'll follow this with a blank line to terminate the if block at which point, the block will execute because, self-evidently, the condition is true. Conversely, if the condition is false, the code in the block does not execute. If false, print("It's true! ") produces no output. The expression used with the if-statements will be converted to a bool, just as if the bool constructor had been used. So, if bool("eggs"): print("Yes please! ") is exactly equivalent to if "eggs": print("Yes please! "). Thanks to this useful shorthand, explicit conversion to bool using the bool constructor is rarely seen in Python. The if-statement supports an optional else clause, which goes in a block introduced by the else keyword, followed by a colon, which is indented to the same level as the if keyword. Let's assign 42 to a new variable, h. Then, do if h is greater-than 50: print("Greater than 50"). To start the else block in this case, we just omit the indentation after the three dots. Else colon and then, in the else block, we must indent by four spaces again, print("50 or smaller"). After entering a blank line to complete the structure, the code executes, printing 50 or smaller, showing that our else block executed. For multiple conditions, you might be tempted to do something like this. Nesting if-else structures within other if-else structures. Whenever you find yourself with an else block containing a nested if-statement like this, you should consider using Python's elif keyword, which is a combined else-if. As The Zen of Python reminds us, flat is better than nested. This version is all together easier to read.

While Loops

Python has two types of loop, for-loops and while loops. We've already briefly encountered for-loops back when we introduced significant whitespace and we'll return to them soon, but right now, we'll cover while loops. While loops in Python are introduced by the while keyword, which is followed by a Boolean expression. As with the condition in 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 while loop at the REPL, which counts down from five to one. We'll initialize a counter variable called c to five and keep looping until we reach zero. Another new language feature we're introducing here is the use of the augmented assignment operator to subtract one from the value of the counter on each iteration. Similar augmented assignment operators exist for the other basic math operators, such as plus and multiply. Because the condition or predicate will be implicitly converted to bool, just as if a call to the bool constructor were present, we could replace the above code with the following version where we just use while c. 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 the 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 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 simply by passing True as the predicate expression to the while construct. While True: print("Looping! "). Now you're probably wondering how we get out of this loop again and regain control of our REPL. We press Ctrl+C, which Python intercepts 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 this course. Many programming languages supports a loop construct which places the predicate test at the end of the loop, rather than at the beginning. For example, C, C++, C Sharp, 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 been nested, continuing 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 within the while block, we use the built-in input function to request a string from the user. We'll assign that string to a variable called response. We'll now use an if-statement to test whether the value provided is divisible by seven. We convert the response string to an integer using the int constructor and then use the modulus operator, with the percent symbol, to divide by seven and give the remainder. If the remainder is equal to zero, the response was divisible by seven, 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 used 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 loop or in our case, the end of the program. Enter a blank line at the three dots prompt to close both the if block and the while block. Our loop will start executing and we'll pause at the call to the input function waiting for us to enter a number. Let's try a few. 12, 67, 34, 28, 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 to end the program, returning us to the REPL prompt.

Summary

We've covered a lot of ground in this Getting Started module. Let's summarize what we've seen. We started by ensuring that Python 3 is available on Windows, Ubuntu Linux, and Mac OS. We then looked at the Read-Eval-Print-Loop or REPL and how it allows us to interactively explore Python code. We learned some simple arithmetic operators with plus, minus, multiply, divide, modulus, and the integer division operator with double slash. We discovered we could give objects names with the assignment operator, using the equal symbol. We learned how to print objects using the built-in print function. And we showed you how to exit the REPL, which is different between Windows, which uses Ctrl+Z and Linux or Mac, which uses Ctrl+D. We showed how Python uses significant indentation to demarcate code blocks. Each indent level is preferably four spaces. And we told you about Python Enhancement Proposals, the documents which govern the evolution of the Python language. In particular, we briefly looked at PEP 8, which is the Python Style Guide, which we follow in this course, and PEP 20, The Zen of Python, which gives useful advice on writing Pythonic code. We looked at importing Python standard library modules using the import statement in its three different forms. We showed how to find and browse help, particularly useful for discovering the standard library. We looked at the four built-in scalar types int, float, None, and bool and conversions between these types and how to use their literal forms. We looked at the six relational operators used for equivalence and ordering. We demonstrated structuring conditional code with if, elif, else structures. We showed iterating with while loops, with the important implicit conversion to bool, of the predicate expression. And how to interrupt infinite loops with Ctrl+C, which generates a KeyboardInterrupt exception. We gave an example of how to break out of a loop using the break statement, which breaks out of the innermost loop and jumps to the first statement immediately following the loop. And along the way, we looked at 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. Next time here on Python Fundamentals, 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. Thanks for watching and we'll see you in the next module.

# Strings and Collections

## Introduction

Hi. My name is Austin Bingham. Welcome to the second module of the Python Fundamentals course, which is about strings, collections, and iteration with for-loops.

Python includes a rich selection of collection types, which are often completely sufficient for even quite intricate programs without resorting to defining our own data structures. We'll give enough of an overview of some fundamental collection types now, enough 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.

## Strings

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 codepoints, and for the most part you can think of codepoints as being like characters, although they aren't 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 can use single quotes or double quotes. You must, however, be consistent. For example, you can't do this.

Supporting both quoting styles allows you to easily incorporate the other quote characters 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.

## Moment of Zen: Practicality Beats Purity

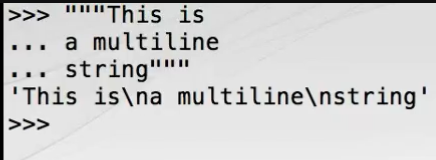
At first sight, support for both quoting styles seems to violate an important principle of Pythonic style from The Zen of Python. “There should be one and preferably only one obvious way to do it”. In this case, however, another aphorism from the same source, “practicality beats purity”, takes precedence. 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.

## Strings (Continued)

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 nicely formatting code as we'll see later.

If you want a literal string containing new lines, you have two options, use multiline strings or use escape sequences.

First, multiline strings. Multiline strings are delimited by three quote characters rather than one. Here's an example of using three double quotes. Notice how when the string is echoed back to us the new lines are represented by the slash N escapes sequence.

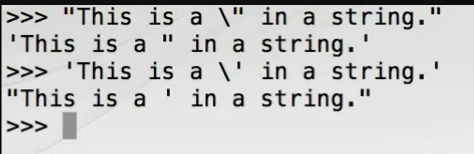


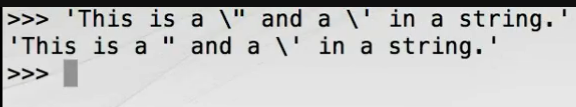
We can also use three single quotes.

As an alternative to using multiline quoting, we can just embed the control characters ourselves, or to get a better sense of what we're representing, we can use print to see the string.

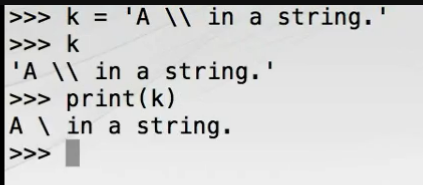
If you're working on Windows, you might be thinking that new lines should be represented by the carriage return at new line couplet \r\n. There's no need to do that with Python since Python 3 has a feature called universal newlines support, which translates from the simple \n to the native newline sequence for your platform on input and output. You can read more about universal newlines support in PEP278.

We can use the escapes sequences for other purposes too such as incorporating tabs with slash T or allowing us to use quote characters within strings by using a slash followed by the quote sign that you want. See how Python is smarter than we are using the most convenient quote delimiters, although Python will also resort to escapes sequences when we use both types of quotes in a string.

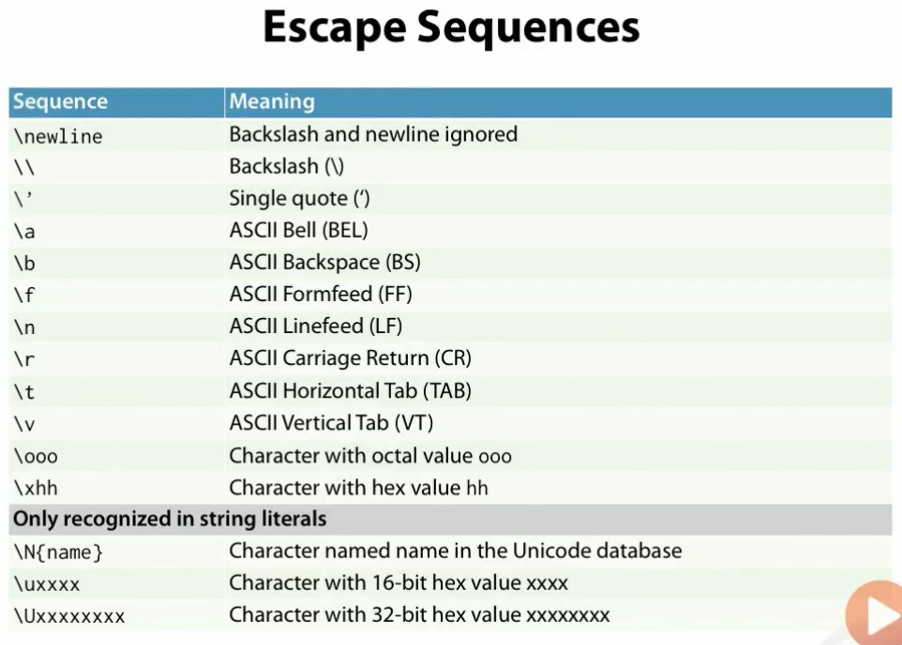




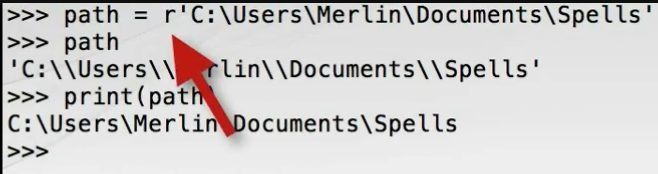
Because the backslash has special meaning, to place a backslash in a string we escape the backslash with itself. To reassure ourselves that there really is only one backslash in that string, we can print it.



You can read more about escape sequences in the Python documentation at python. org.



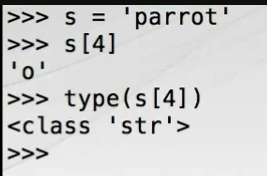
Sometimes, particularly when dealing with strings such as Windows file system paths or regular expression patterns, which use backslashes extensively, the requirement to double up on backslashes can be ugly and error prone. Python comes to the rescue with its raw strings. Raw strings don't support any escapes 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 zero-based index.

Note that in contrast to many other 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 by using help on the string type. Ignore all the hieroglyphics with underscores for now, and page down until you see the documentation for the capitalize 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 capitalize method didn't modify C in place. 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 Finally, because strings are fully Unicode capable, you 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 hexadecimal Unicode codepoints as an escape sequence prefixed by slash U, which I'm sure you'll agree is somewhat more unwieldy. There are no such Unicode capabilities in the otherwise similar bytes type, which we'll look at next.



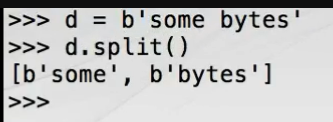
## Bytes

Bytes are very similar to strings except that rather than being sequences of Unicode codepoints, they are sequences of well, bytes. As such, they are used for raw binary data and fixed with single byte character encodings such as ASCII.

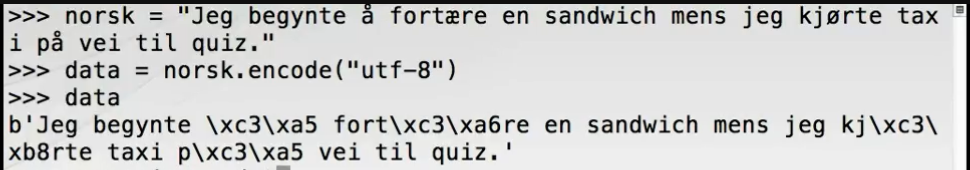
As with strings, they have a simple literal form using quotes, the first of which is prefixed by a lowercase B.

There is also a bytes constructor, but it's an advanced feature, and we won't cover it in this fundamentals course.

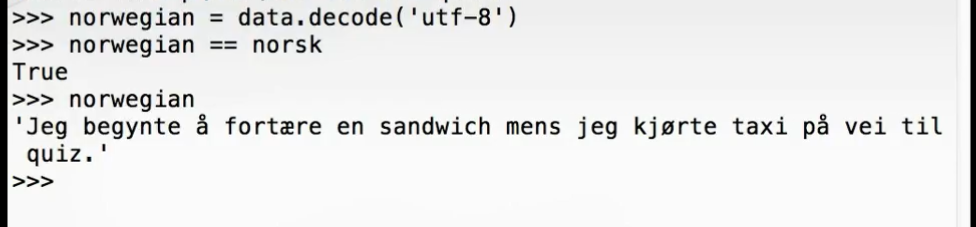
At this point, it's sufficient for us to recognize bytes literals and understand that they support most of the same operations as strings such as indexing and splitting. You'll see that the split method returns a list of bytes objects.



To convert between bytes and strings, we must know the encoding of the byte sequence used to represent the string's Unicode codepoints 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 the 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 crucial to understand since files and network resources such as HTTP responses are transmitted as byte streams whereas we prefer to work with the convenience of Unicode strings.



## Lists

Python lists such as those returned by the string's split method are sequences of objects. Unlike strings, lists are mutable in so far as the elements within them can be replaced or removed, and new elements can be inserted or appended. Lists are the workhorse of Python data structures.

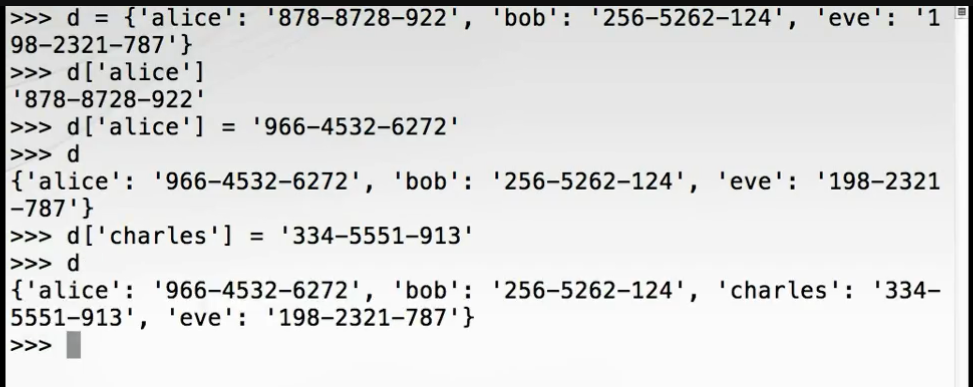
Literal lists are delimited by square brackets, and the items within the list separated by commas. Here is a list of three numbers and a list of three strings. We can retrieve elements by using square brackets with a zero-based index, and we can replace elements by assigning to a specific element. See 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 is 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, an important maintainability feature.

## Dictionaries

Dictionaries are completely fundamental to the way the Python language works and are very widely used. A dictionary maps keys to values and in some languages is known as 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 the entries in the dictionary can't be relied upon to be stored in any particular order. Similarly to lists, empty dictionaries can be created using empty curly braces. We'll revisit dictionaries in much more detail in Module 5.



## For-Loops

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 of that later, and assign them in turn to a variable 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 use the ability of the built-in print function to accept multiple arguments. We pass the key and the value for each color separately. See also how the color codes returned to us are in decimal. Now, before we put some of what we've learned together into a useful program, practice exiting the Python REPL with Control+Z on Windows or Control+D on Mac or Linux.

## Putting it All Together

In this last section before we summarize, we're going to write a longer snippet at the REPL and briefly introduce the with-statement. We're going to fetch some text data for some classic literature from the web using a Python Standard Library function called the 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. We'll be using a Python construct called a with block to manage the resource returned by the URL since under the hood fetching the resource from the web requires operating system sockets and such like. We'll be talking more about with-statements in a later module, but for now it's enough to know that using a with-statement with objects which use external resources such as this is very good practice to avoid resource leaks. The with-statement calls the urlopen function and binds the response to a variable named story. Notice that the with-statement is terminated by a colon introducing a new block, so we indent four spaces and 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 at 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 use 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 iterate over this list of words appending each in turn to the accumulating story\_words list. Finally, we enter a blank line at the three dots prompt to close all open blocks. In this case, the inner for-loop, the outer for-loop, and the with-block will all 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 have entered, and carefully reenter 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 have typed the URL correctly. We can now look at those words 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 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 byte stream in each line into UTF-8 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 arrows we can reenter 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 limit 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 within a text editor.

## Summary

Let's sum up. 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 slash N character safe in the knowledge that it will be appropriately translated to and from the native newline during I/O.

Escape sequences provide an alternative means of incorporating newlines 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 an 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 as 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 is a sequence of bytes rather than a sequence of Unicode codepoints. Bytes literals are prefixed with a lowercase B.

To covert 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.

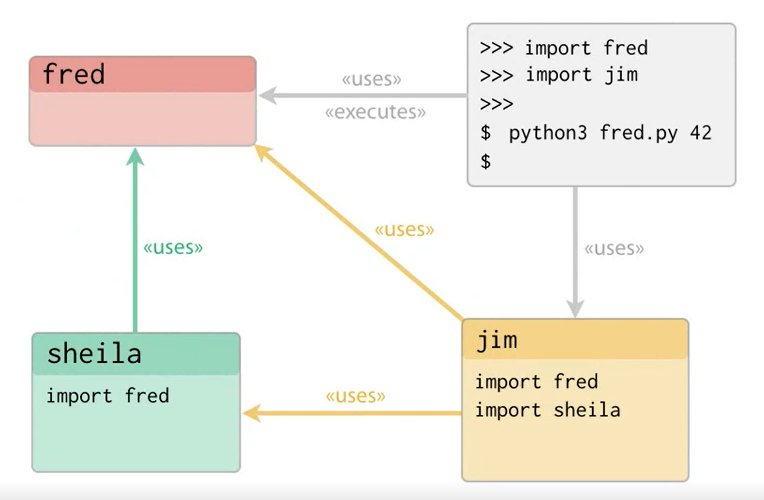
Lists are mutable, heterogeneous sequences of objects. List literals are delimited by square brackets, and the items are separated by commas. As with strings, individual elements can be retrieved by indexing into a list with square brackets. In contrast to strings, individual list elements can be replaced by assigning to the indexed item. Lists can be grown by appending to them and 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 bind the name to the current item. For-loops correspond to what are called for-each loops in other languages. Thanks for watching, and we'll see you in the next module.

# Modularity

## Introduction

Hello. My name is Robert Smallshire. Welcome to the third module of the Python Fundamentals course, which is about the structure and organization, or modularity, of Python programs. 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. As with most programming languages, the most fine-grained modularization facility is the definition of reusable functions. Collections of related functions are typically grouped into source code files called modules. Modules can be used from other modules, so long as we take care not to introduce a circular dependency. As we have seen already, we can import modules into the REPL, and 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 ensure 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 we developed at the end of the previous session and organizing it into a fully-fledged Python module.

## Creating, Running, and Importing a Module

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 use the. py extension. Now 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 a directory called pyfund in your home directory. Switch to a console with your operating system shell prompt and change to the new pyfund directory. We can execute our module simply by calling python3 and passing the module's filename. 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, the program is running as expected. If, on the other hand, you got an error, an HTTP error indicates there's a network problem while other types of error probably mean you mistyped the code.

Let's add another for loop to the end of the program to print out one word per line. 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. When importing, we omit the file extension. The code in your module is executed immediately when imported, maybe not what you'd 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.

## Defining Functions and Returning Values

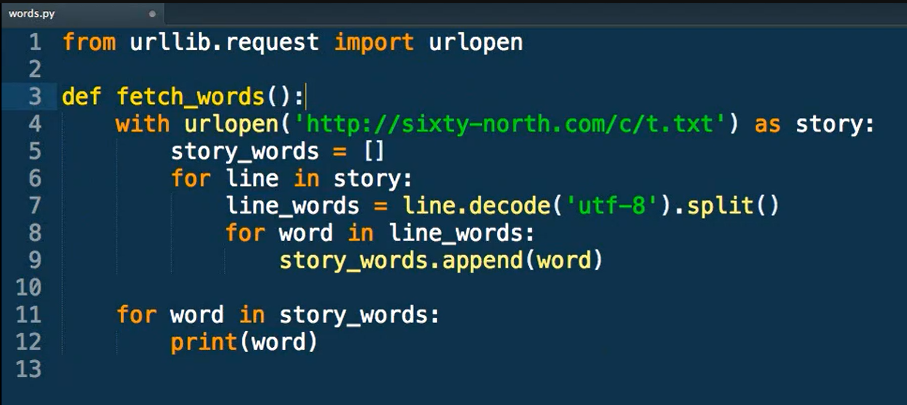
Let's quickly define a few functions at the REPL to get the idea. Functions are defined using the def keyword followed by the function name, an argument list in parentheses, and a colon to start a new block. We use the return keyword to return a value form the function. As we've seen previously, we call functions by providing the actual arguments in parentheses, if there are any, after the function name. Functions aren't required to explicitly return a value though. Perhaps they produce side effects.

You can return early from a function by using the return keyword with no parameter. A return keyword without a parameter, or the implicit return at the end of a function, actually causes the function to return None, although remember 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.



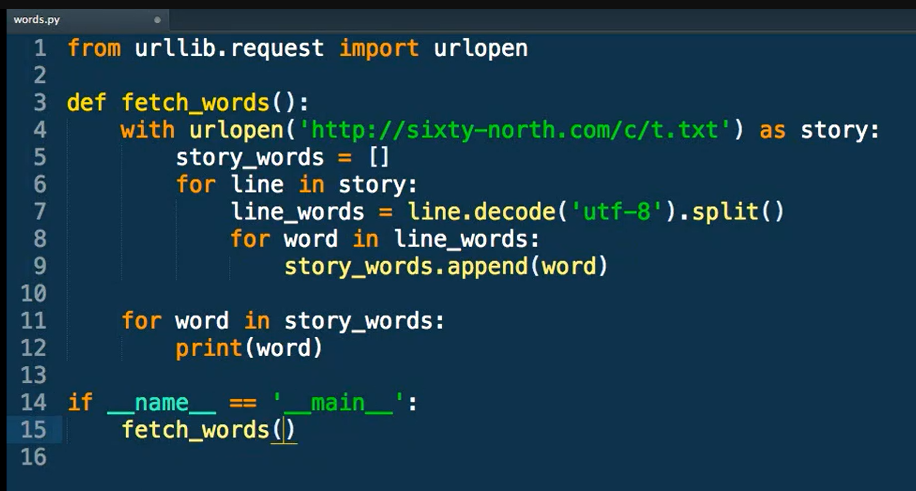
## Distinguishing Between Module Import and Module Execution

Let's organize our words module using functions. 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 and reload the module using a fresh Python REPL. The module imports, but the words are not fetched until we call the fetch\_words function. Alternatively, we can import our specific function. So far, so good. But what happens when we try to run our module directly from the operating system shell? Exit the REPL with Control +D on Mac or Linux or Control +Z on Windows, and run python3 passing the module filename words. py. No words are printed. This is because all executing the module does now is define a function and then immediately exit.

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. The Python runtime system defines some special variables and attributes, the names of which are delimited by double underscores. One such variable is called \_\_name\_\_ and gives us the means for our module to detect whether it has been run as a script or imported into another module or the REPL. To see how, add print\_\_name\_\_ at the end of your module, outside the fetch\_words function. First of all, let's import the modified words module back into the REPL with import words. We can see that when imported, double underscore name does indeed evaluate to the module's name. As a brief aside, if we import the module again, 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 with python3 words. py. Now the special name variable is equal to the string \_\_main\_\_, which is also delimited by double underscores. Our module can use this behavior to detect how it is being used. We replaced the print statement with an if statement, which tests the value of double underscore name (\_\_name\_\_), and if it is equal to double underscore main (\_\_main\_\_), executes our function.



Now we can safely import our module without unduly executing our function. And we can usefully run our function as a script.

## The Python Execution Model

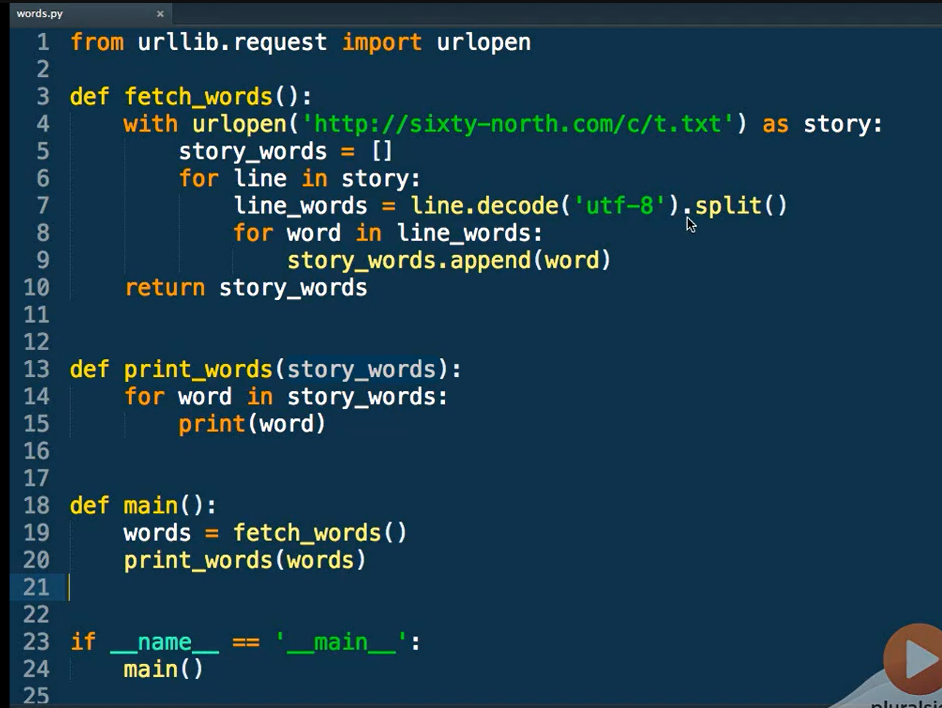
It's important to understand the Python execution model and precisely when function definitions and other important events occur when a module is imported or executed. Here we show execution of our Python module in the PyCharm graphical debugging environment. We step through the top-level statements in the module. What's important to realize here is that the def used for the fetch\_words function isn't really a declaration. It's actually a statement, which when used in sequence with the other top-level module 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 functions within the module namespace are defined.

We're sometimes asked about the differences 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 \_\_name\_\_ equals \_\_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 the Python 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 our module to be a Python script or a 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 have been built exclusively with Python

## Main Functions and Command Line Arguments

Now, we'll look at setting up a main function with a command line argument. We'll start by refining our word fetching module a little further. First, we'll perform a small refactoring and separate the word retrieval and collection on the one hand from the word printing on the other. This is because when importing, we'd rather get the words as a list, but when running directly, we'd prefer the words to be printed. Next, we'll extract a function from our if \_\_name\_\_ equals \_\_main\_\_ block called main. **By moving this code into a function, we can test it from the REPL, 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 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 wildcard. 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 namespace clashes at some future time.

Having done this, we can fetch words from the URL, print any list of words, or indeed run the main program. Notice that the print words function isn't fussy about the type of 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. Given this, we'll perform a minor refactoring and rename this function to print\_items, changing the variable names within the function to suit.

We'll talk more about the dynamic typing in Python, which allows this degree of flexibility, in the next module. One obvious improvement to our module would be to replace the hard-coded URL with a value we can pass in. Let's extract that value into an argument of the fetch\_words function called URL. When running our module as a standalone program, we need to accept the URL as a command line argument. Access to command line arguments in Python is through an attribute of the sys module called argv, which is a list of strings. To use it, we must import sys at the top of our program and then get the second argument with an index of 1 from the list. This works as expected when we pass a URL from the command line. This all looks fine until we realize that we can't usefully test the main function any longer from the REPL because it refers to sys.argv 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 \_\_name\_\_ == \_\_main\_\_ block. Testing from the REPL again, we show that main is also usable from here. For more sophisticated command line processing, we recommend you look at the Python standard library argparse module or the inspired third-party docopt module.

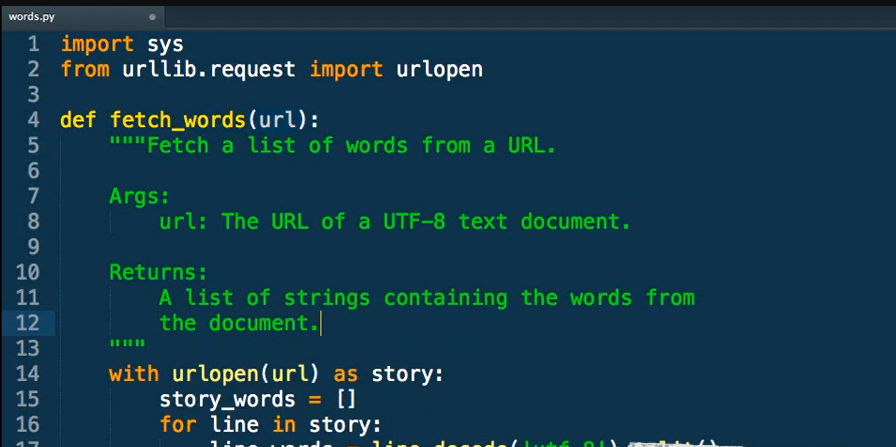
## Sparse Is Better Than Dense

You may have noticed that our top-level functions have two blank lines between them. This is conventional for modern Python code. Sparse is better than dense. Two between functions, that is the number of lines PEP 8 recommends. According to the PEP 8 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.

## Documenting Your Code Using Docstrings

Now we'll look at documenting your code using a feature called 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 easily be expanded to add more detail later. One Python convention for docstrings is documented in PEP 257, although it's not widely adopted. Various tools, such as Sphinx, are available to both HTML documentation from Python docstrings, and each tool mandates its preferred docstrings format. Our preference is to use the form presented in Googles Python Style Guide since it is amenable to being machine passed while still remaining readable at the console to humans.



Now we've added the docstring, we can access it through Help from the REPL. We'll go ahead and add similar docstrings for our other functions, print items, and main. Each docstring begins with a short description of the purpose of the function followed by, as necessary, a list of the arguments to the function and the return value. Then we'll add one for the module itself. The module docstring 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.

## Documenting Your Code With Comments

Next, we'll introduce Python code comments. We believe docstrings are the right place for most documentation Python code. They explain how to consume the facilities your module 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 can do that using Python comments. Comments in Python begin with a hash symbol and continue to the end of the line. Let's document the fact that it might not be immediately obvious why we are using sys. argv(1) rather than sys. argv(0). This is because the 0th argument of sys. argv is the module filename.

## The Whole Shebang

And now the whole shebang. It's common on Unix-like systems to have the first line of a script include a special comment called a shebang. This begins with the usual hash as for any other comment followed by an exclamation mark. The shebang 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 they're in 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 use the Unix env program to locate Python 3 on your path environment variable, which importantly is compatible with Python virtual environments. A standard Python 3 shebang passes Python 3 to the user bin/env program. Don't worry if you're on Windows. Python includes machinery to make this work on Windows too. On Mac or Linux, we must mark our script as executable using the chmod command 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 executable with a correct version of the Python interpreter, even to the extent that shebangs that look like they should only work on Unix-like systems will work as expected on Windows. This works because Windows Python distributions now use a program called PyLauncher. PyLauncher, the executable for which is simply py. exe, will pass the shebang and locate the appropriate version of Python on your system. For example, on Windows in cmd shell, simply words. py followed by the URL as the command line argument will be sufficient to run your script with Python 3, even if you also have Python 2 installed. You can read more about PyLauncher on windows in PEP 397.

## Summary

Let's review what we've covered in this module.

* Python code is placed in. py files called modules.
* Modules can be executed directly by passing them as the first argument to the Python interpreter.
* Modules can also be imported into the REPL, at which point all top-level statements in the module are executed in order.
* Named functions are defined using the def keyword followed by the function name and the argument list in parentheses.
* We can return objects from functions using the return statement, and return statements without a parameter return None, as does the implicit return at the end of every function body.
* We can detect whether a module has been imported or executed by examining the value of the special double underscore name variable. If it is equal to the string double underscore main, our module has been executed directly as a program. By executing a function if this condition is met using the top-level if \_\_name\_\_ == \_\_main\_\_ idiom at the end of our module, we can make our module both usefully importable and executable, an important technique even for short scripts.
* Module code is only executed once on first import.
* The def keyword is a statement, which binds executable code to a function name.
* Command line arguments can be accessed as a list of strings accessible through the argv attribute of the sys module. The 0th command line argument is the script filename, so the item at index 1 is the first true argument.
* Literal strings is the first line of a function definition from the function's docstring. They are typically triple quoted multiline strings containing usage information. Function documentation provided in docstrings can be retrieved using the Help function in the REPL.
* Module docstrings should be placed near the beginning of the module prior to any Python statements, such as import statements.
* Comments in Python commence with the hash character and continue to the end of the line.
* 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.

Next time here on Python Fundamentals, we'll dig into Python's object model, looking at how values are passed to and returned from functions. We investigate the nature of dynamic typing in Python and focus on the rules of variable scope. Thanks for watching, and we'll see you in the next module.

# Objects

## Introduction

Hello. My name is Austin Bingham. Welcome to the fourth module of the Python Fundamentals course where we seek to understand the Python object model, take a more in-depth look at some collection types we've met already, and introduce a few more collection types. 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?(x=1000) 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 int 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 at 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.

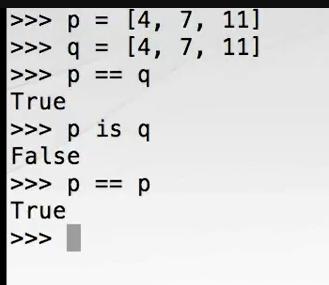
When we assign from one variable to another, we're really assigning from one object reference to another object reference, so both references now refer to the same object. If we now reassign X, we have X referring to an int 3000 object and Y referring to a separate int 500, and there is no work for the garbage collector to do because all objects are reachable from the live references.

Let's dig a little deeper using the built-in id() function, which returns an integer identifier, which is unique and constant for the lifetime of the object. Let's rerun the previous experiment using 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. 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 original int 5 with the new int 2 to produce 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. The assignment operator only ever binds to names. It never copies an object by 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, then assign R to a new reference S. When we modify the list referred to by S by changing the middle element, the R list has changed too since the names S and R in fact refer to the same object. Let's see that again with a diagram. First we assign R to a new list. We then 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.

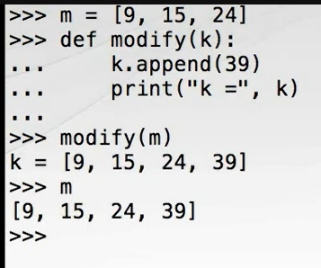
Let's contrast that behavior with a test for value equality or equivalence. We'll create two identical lists. Here we see that P and Q refer to different objects, but that the objects they refer to have the same value. Of course an object should always be equivalent to itself.



Here's how that looks pictorially. We have two separate list objects each with a single reference to it. The values contained in the objects 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 programatically. 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.

## Argument Passing

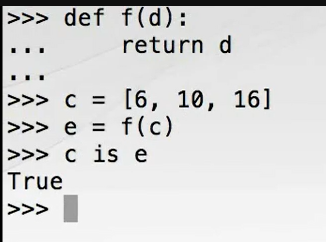
Let's look at how all of this relates to function arguments and return values. Let's define a function at the REPL, which appends a value to a list and prints the modified list. First the list and then a function modify, which appends to and prints the list. The function accepts a single formal argument named K. We call modify passing our list M as a the actual argument, which 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 the actual argument reference, in this case M, to the formal argument reference, in this case K. As we have seen, assignment causes the reference being assigned to refer to the same object as the reference being assigned from. This is exactly what's going on here. If you want a function to modify a copy of an object, it's the responsibility of the function to do the copying.

Let's look at another example, first a new list F that refers the three elements, then a function which replaces the list, which we now call with the actual argument F. This is much as we'd expect; however, what's the value of F? 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 it to a newly constructed list 17, 28, 45, so within the function the reference to the original 14, 23, 37 list was overwritten, although the unmodified object itself was still pointed to by the F reference outside the function.

So, we see 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 like this, (Typing) and indeed the contents of F have been modified.

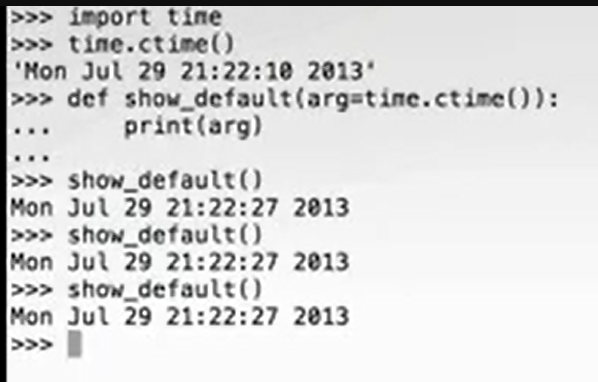
Function arguments are transferred by what is called pass by object reference. This means that the value of the reference is copied into the function argument, not the value of the referred object. No objects are copied. The return statement uses the same pass by object reference semantics as function arguments. We can demonstrate this simply by writing a simple function which returns its only argument. Creating an object such as a list and passing it through this simple function returns the very same object we passed in showing that no copies of the list were made.



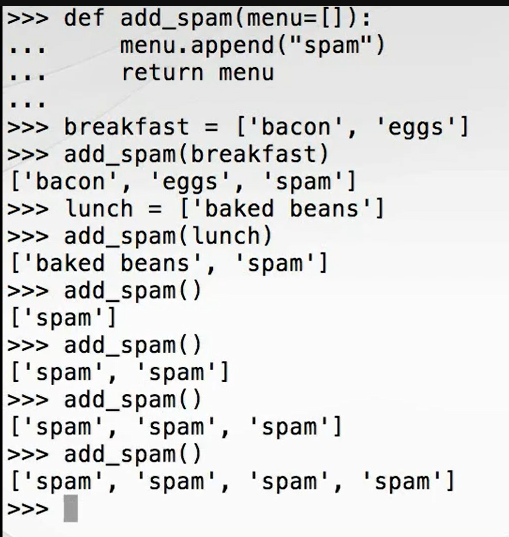
## Function Arguments in Detail

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. When we define functions using default arguments, the parameters with default arguments must come after those without default; 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 len function, and secondly 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. However, if we do provide the option argument, it is 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 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, although remember that all keyword arguments must be specified after the positional arguments.

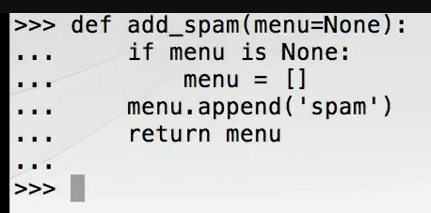
It's crucial to have an appreciation of exactly when the expression provided as a default argument value is evaluated to avoid a common pitfall which frequently ensnares newcomers to Python. Let's examine this question closely using the Python Standard Library time module. We can easily get the time 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 display 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 this causes no problems when the default is a simple immutable constant such as an integer or a string, 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 the 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 and naturally add spam to it. We'll do something similar for lunch. 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 an empty menu, we get just spam. Let's do that again. When we exercise the default the second time, we get two spams and three and four.

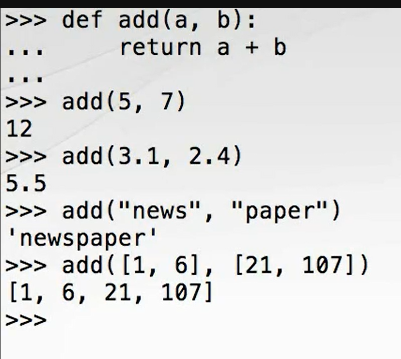


What's happening here is that the empty list used for the default argument is created exactly once when the def statement is executed. The first time we fell back on the default this list has spam added to it. When we used the default the second time, the list still contains that item, and a second instance of spam is added to it making two ad infinitum, or perhaps ad nauseam 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. And now our add\_spam function works as expected.



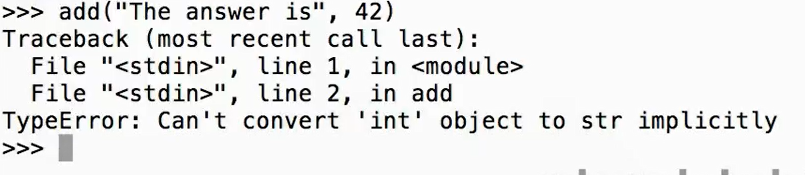
## Python's Type System

Programming languages can be distinguished by several characteristics, but one of the most important is the nature of their type systems. Python can be characterized as having a dynamic and strong type system. Let's investigate what that means. Dynamic typing means that 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 term strong type system is less rigorously defined than dynamic, but a common definition is that the language will not in general implicitly convert objects between types.

The strength of the Python 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 being the conversion to bool used for if statements and while-loop predicates.



## Variable Scoping

As we have seen, no type declarations are necessary, 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 a 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 main types of scope in Python arranged in a hierarchy. Each scope is a context in which names are stored in which they can be looked up. The four scopes from narrowest to broadest are: Local, those names defined inside the current function. Enclosing, those names defined inside any and all enclosing functions. This scope isn't important for the contents of this Python fundamentals course. Global, those names defined at the top level of a module. Each module brings with it a new global scope. And finally Built-in, those names built into the Python language through the special **builtins 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, with-blocks, and the like do not introduce new nested scopes.

Consider our words. py module. It contains the following global names: Main bound by the def main statement, sys bound when sys is imported, \_\_name\_\_ provided by the Python runtime, urlopen bound from the urllib request module, fetch\_words bound by the def fetch\_words statement, and print\_items bound by the def print\_items statement. Module scope name bindings are typically introduced by import statements and function or class definitions. It is possible to use other objects at module scope, and this is typically used for constants, although it can be used for variables.

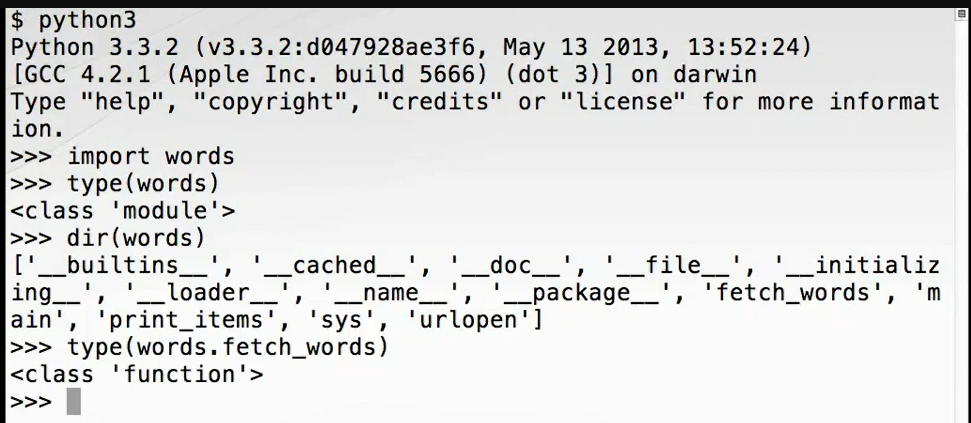
Within the fetch\_words function we have the six local names: Word bound by the inner for-loop, line\_words bound by assignment, line bound by the outer for-loop, story\_words bound by assignment, url bound by the formal function argument, and story bound by the with statement. Each of these is brought into existence at first use and continues to live within the function scope until the function completes at which point the references will be destroyed. Very occasionally we need to rebind a global name at module scope from within a function. Consider the following simple module. If we save this module in scopes. py, we can import it into the REPL for experimentation. When show\_count() is called, Python looks up the count name in the local namespace L, doesn't find it, so it looks in the next most outer namespace, in this case the global module namespace G where it finds the name count and prints the referred object. Now we call set\_count with a new value and call show\_count again. You might be surprised that show\_count displays 0 after the call to set\_count, 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 lookup is performed for the global count at module scope. We have 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. Quit and restart the Python interpreter and exercise our revised module, which now demonstrates the required behavior.

## Moment of Zen

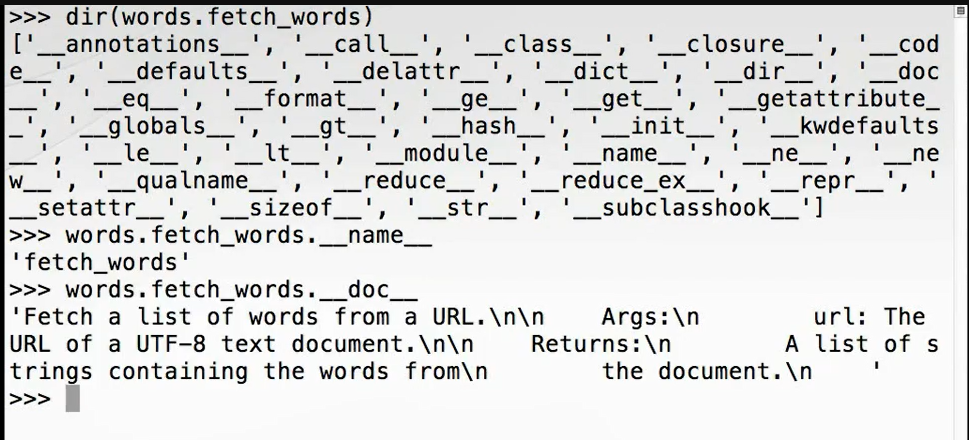
Special cases aren't special enough to break the rules. We follow patterns not to kill complexity, but to master it. As we have shown, all variables in Python are references to objects, even basic types such as integers. This thorough approach to object-orientation is a strong theme in Python, and practically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

## Everything Is an Object

In Python it's important to remember that everything is an object, primitive objects, functions, modules, and on and on. It's easy to see that when you start to explore even simple pieces of code. 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 namespace. 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 an object. The dir function returns a sorted 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 double underscore name and double underscore 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 can see that fetch\_words is a function object. We can in turn call dir on the function to reveal its attributes, and 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, this is the name of the function object as a string, and this is the docstring we provided.



These give 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. 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 attaches 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, which 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 equivalence using the double equals operator. Function arguments are passed by object reference, so functions can modify their arguments if they are mutable objects. If a formal function 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, in the Global or module namespace, and Built-ins. Global references can be read from a local scope. Assigning to global references 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 can be treated just like other objects. The import and def keywords result in binding to named references. The built-in type function can 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 double underscore name attribute. The docstring for a function or module object can be accessed through its double underscore 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. Thanks for watching, and we'll see you in the next module.

Collections

Introduction

Hi. My name is Robert Smallshire. Welcome to the fifth module of the Python Fundamentals course where we look in more depth at the built-in collection types of the Python language and explore the various protocols that unite them. In this module we'll revisit some collection types we've already explored: Str or string, the immutable sequence of Unicode codepoints; list, the mutable sequence of objects; and dict, the mutable mapping from immutable keys to mutable objects. We'll also cover some new ones: Tuple, the immutable sequence of objects; range for arithmetic progressions of integers; and set, a mutable collection of unique immutable objects. We won't cover the bytes type further here. We've already discussed its essential differences with str, and most of what we learned about str can also be applied to bytes. This is not an exhaustive list of Python collection types, but it's completely sufficient for the overwhelming majority of Python 3 programs you'll encounter in the wild or are likely to write yourself. In this module we'll be covering these collections in this order, and then 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.

Tuple

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 literal 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 zero-based index using square brackets, and we can determine the number of elements in the tuple using the built-in len function. We can iterate over tuples using the for loop, and we can concatenate tuples using the plus operator. Tuples can be repeated 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. Sometimes a single element tuple is required. To write this, we can't just use a simple number in parentheses. This is because Python pauses that as an integer enclosed in the president's controlling parentheses of a math expression. To create a single element tuple, we make use of the trailing comma separator, which we're allowed to use when specifying literal tuples, lists, and dictionaries. A single element with a trailing comma is passed 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 use 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 series, the hard work being done by the two built-in functions min and max. Returning multiple values as 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. Tuple unpacking works with arbitrarily nested tuples, although not with other data structures. This in turn leads to the beautiful Python idiom for swapping two or more variables. Should you need to create a tuple from an existing collection object such as a list, you can use the tuple constructor, here also shown for strings. Finally, as with most collection types in Python, we can test for containment using the in operator or nonmembership with the not in operator. While that just about wraps up our look at tuple, let's move onto the next collection type.

String

We've covered strings at some length already, but we'll take time now to explore their capabilities in a little more depth. Recall that str is a homogeneous immutable sequence of Unicode codepoints, which for the most part we can consider to be the same as characters. Let's put str through its paces. As with any other Python sequence, we can determine the length of a string with a built-in len function. For instance, if we wanted to know how many characters were in the Welsh place name llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, we could call len on this string and get the answer of 58. Concatenation of strings is supported using the plus operator. For instance, it's easy to turn "New" + "found" + "land" into Newfoundland, although for joining large numbers of strings the join method should be preferred because it's substantially more efficient. This is because concatenation using the addition operator or its augmented assignment version can lead to the generation of large numbers of temporaries with consequent costs for memory allocation and copies. Let's see how join is used. Join is a method on str which takes a 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, we call join on the separator we wish to use with a semicolon and pass in the list of strings to be joined. We can then split them up again using the split() method, which we've already encountered, but this time we're going to provide its optional argument. A widespread and fast path and idiom for concatenating together a collection of strings is to join using the empty string as the separator. The way may not be obvious at first. To concatenate, invoke join on empty text, something for 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 separate the string "London:Edinburgh" into three by partitioning around the colon. 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 supercedes, although does not replace, the string interpolation technique used in all the versions of Python and 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. 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. A field name may be used more than once such as in this example. However, if the field names are used exactly once and in the same order as the arguments they can be omitted. If keyword arguments are supplied to formats, then named fields can be used instead of indexes. Note in this example how we've used the ability to split long statements over multiple lines so long as the split is inside open parentheses, brackets, or braces. In this case, we split the argument list with a format() method inside its open parentheses. This technique can greatly aid the formatting of Python code whilst keeping a manageable line length. It's possible to index into sequences using square brackets inside the replacement field. Here we've assigned a tuple to a variable pos, and then we pass pos as a keyword argument to format. We access the elements of pos inside the replacement fields within the string. We can even access object attributes. Here we pass the whole math module to format using a keyword argument. Remember, modules are objects too. Then we access two of its attributes from within the replacement fields, pi and e. Format strings also give us a lot of control over field alignment and floating point formatting. Here's the same with the constants displayed to only three decimal places. We don't cover all of the intricacies of the Python formatting mini language. If you want to know more, consult the documentation. Now we've covered the fundamentals of the string type. We recommend you spend some time familiarizing yourself with the other string methods. Remember, you can find out what they are using simply help str. Let's move on now and look at the next collection type.

Range

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. In this case we create a range from 0 to 5 simply by supplying the stop value of 5. Ranges are sometimes used to create consecutive integers for use as loop counters. Note that the stop value supplied to range is one past the end of the sequence, which is why the previous loop terminated at 4 and didn't print 5. We can also supply a starting value if we wish such as in this call range 5, 10. Wrapping this in a call to the list constructor is a handy way to force production of each item. The so-called half-open range convention with a stop value not being included in the sequence at first seems strange, but 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 range. Range also supports a step argument, which controls the interval between successive numbers. Note that in order to use the step argument we must supply all three arguments. Range is curious in that it determines what its arguments mean by counting them. Providing only one argument means it is the stop value, two arguments are start and stop, and three arguments are start, stop, and step. Python's 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 unpythonic. At this point, we're going to show you another example of poorly styled Python code, one you can and should avoid. Here's a poor way to print the elements in a list using range, len, and list indexing. Although this works, it's most definitely unpythonic. 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 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. We can improve this even further by using tuple unpacking and avoiding having to deal directly with the tuple. Because of the strong iteration primitives built into Python, ranges aren't widely used in modern Python code. Let's move on and look at the next collection type.

List

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 let's take a deeper look. Let's start by making a list containing the words show how to index into sequences. We'll do this by calling the split method on a string. We're already familiar with how 0 and positive integers index the list from the front using a zero-based index. Here we extract the fifth element by using the index 4. One very convenient feature of lists and other Python sequences, for this applies to tuples too, is the ability to index from the end rather than from the beginning. This is achieved by supplying negative indexes. For example, we can access the fifth element from the end by supplying the index -5. The last element of the sequence is at index -1. Negative indexing is much more elegant than the clunky alternative of computing the forward index by subtracting the backward index from the length of the sequence. Note that indexing with -0 is the same as indexing with 0, returning the first element in the list. Because there is no distinction between 0 and -0, negative indexing is essentially one-based rather than zero-based. This is good to keep in mind if you're calculating indexes with even moderately complex logic. Off by one 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 we slice three words from the list by passing the start index 1 and the stop index 4. This facility can be combined with negative indexing. For example, to take all the elements except the first and last, slice between 1 and -1. Both the start and stop indices are optional. To slice all elements from the third to the end of the list, supply only 3: as the argument to the index operator. And to slice all elements from the beginning of the list up to, but not including the third, supply :3 as the argument to the index operator. Notice that these two lists together form the whole list demonstrating the convenience of the half-open range convention. Since both start and stop 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. This new list has a distinct identity, but 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 list 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 even a simple call to the list() constructor passing the list to be copied. Largely, it's a matter of taste. Our preference is for the third form using the list() constructor since it has the advantage of working with any iterable series as the source and not just lists. You must be aware, however, that all of these techniques perform a shallow copy. That is, they create a new list containing the same object references as the source list, but don't copy the referred to objects.

Shallow Copies

To demonstrate this, we'll used nested lists with the inner list serving as mutable objects. Here's a list containing two elements, each of which is itself a list. Let's look at what's going on under the covers as Python constructs this data structure. First, two integer objects are created containing the values 1 and 2 respectively. The elements of the first inner list are references to these two integer objects. Now two more integer objects are created containing the values 3 and 4 respectively. The elements of the second inner list are references to these integer objects. Now the outer list is created, its elements containing references to the two inner lists. Finally, the reference named A is bound to the outer list. What happens when we copy the list? Here we use the full slice technique, but any of the three techniques we've shown will have the same effect. We're requesting a copy of outer list, so it's elements, which contain references to the two inner lists, are duplicated. These references refer to the same inner lists as the original list A. Once the list copy is complete, we bind a new reference named B to the new list. We can confirm that the lists are distinct objects by testing with A is B, which returns False. They do, however, contain equivalent values, which we can test with A == B, which returns True. Not only are the elements at a(0) and b(0) equivalent, they actually refer to the same inner list object. Now let's replace the element at a(0) with a new list containing 8 and 9. This results in the construction of two new integer objects containing 8 and 9 respectively and the new list, the elements of which are references to these two new integer objects. The reference in a(0) is redirected to point to this new inner list object, and we can confirm that a(0) now indeed points to the new list while b(0) is unmodified. What happens if we now append to the inner list referred to by a(1)? Let's append a new integer object containing the value 5. The new integer object is created and is referred to by an additional element in the inner list referred to by a(1), so a(1) now refers to a list containing integers 3, 4, 5. Significantly, because a(1) and b(1) refer to the same inner list, the list accessible through b(1) has also been modified. Following these manipulations, the data structure referred to by A is a list containing two elements, each of which itself is a list. The first inner list contains 8 and 9, and the second inner list contains 3, 4, and 5. The data structure referred to by B is also a list containing two inner lists. The first element refers to a list containing 1 and 2, and the second element also refers to the 3, 4, 5 list.

List Repetition

As for strings and tuples, lists support repetition using the multiplication operator. It's simple enough to use. Here we repeat a list containing the integers 21 and 37 four times. This form is rarely spotted in the wild. It's most often used for initializing a list of size known in advance to a constant value such as 0. Here we create a list initialized with nine 0 elements. Be aware though that in the case of mutable elements the same trap for the unwary that occurred with list copying lurks here since repetition will repeat the reference without copying the value. Let's demonstrate using nested lists as our mutable elements again. We'll repeat a list five times. The mutable element it contains will be another list containing elements, two -1 and +1. Let's see what Python needs to do to construct this data structure. First, two integer objects are created containing the values -1 and +1 respectively. These are referred to by the two elements of the inner list. Python then creates the outer list and the ranges for its single element to contain a reference to the inner list. Now the repetition operation is applied. A new list is created containing five elements, each of which contains a copy of the single element in the original outer list. All of these elements contain references to the same inner list object. Now the temporary single element outer list can be disposed of. Finally, this whole data structure is bound to a new named reference, S. As expected, S contains five elements, each of which is the -1, +1 in a list. Now let's append the integer 7 to the fourth inner list at index 3 in the outer list. This creates a new integer object and a new element on the inner list containing a reference to that integer object. We can see that all of the elements of the outer list have been modified because they do in fact all refer to the same inner list.

More on List

To find an element in a list, use the index method passing the object you're searching for. The elements are compared for equivalence or value equality until the one you're looking for is found and its index returned. Here we create a list W containing the words the quick brown fox jumps over the lazy dog using the string split method. Searching for fox using the index method returns the integer 3, which of course allows us to get ahold of that element. If you search for a value that isn't present like unicorn, you will receive a ValueError, which we'll learn how to handle gracefully in the next module. Another means of searching is to count matching elements using the count method. Here we count occurrences of the word the. If you just want to test for membership, you can use the in operator or for nonmembership using the not in operator. Elements are removed from lists using a keyword with which we have not yet become acquainted, del. The del keyword takes a single parameter, which is a reference to a list element and removes it from the list shortening the list in the process. Here we make a string U containing the words jackdaws love my big sphinx of quartz using the string split method. We can delete the fourth element using del u(3) leaving jackdaws love my sphinx of quartz. It's also possible to remove elements by value rather than by position using the remove method. Here we pass jackdaws to that method giving us love my sphinx of quartz. This is of course equivalent to using the del statement and the index method in combination. Attempting to remove an item which is not present such as pyramid results in a ValueError. Items can be inserted into lists using the insert method, which accepts the index of the new item and the new item itself. Here we create a list A containing the words I accidently the whole universe. We insert the word destroyed at index 2 using a call to the insert method giving us I accidentally destroyed the whole universe. We can covert this list of words back into a string by calling the string join operator on a space separator.

Growing Lists

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() list method. All of these techniques work with any iterable series, not just lists on the right hand side.

Reversing and Sorting Lists

Before we move on from lists, let's look at two operations which rearrange the elements in place, reversing and sorting. A list can be reversed in place simply by calling its reverse() method. Similarly, 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 a descending 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 built-in len function is a callable object, which is used to determine the length of the collection such as a string. Consider the following list of words: Not perplexing do handwriting family where I illegibly know doctors. By passing a reference to the len function as the key argument to the sort() method of the H list, we can order these words in order of length. Again, we can join the list of words back together with a call to the join operator on a space separator giving the almost sentence I do not know where family doctors illegibly perplexing handwriting. Sometimes an insitu sort or reversal is not what is required. For example, it may cause a function argument to be modified giving the function confusing side effects, which it would not otherwise have. For out of place equivalence of the reverse and sort list methods, you can use the reversed and sorted built- in functions, which return a reverse iterator and a new sorted list respectively. Here we use sorted() to sort a list of numbers, X, which returns a new list, Y. Notice that the original list X is unmodified. Here we use reversed() to reverse a list of numbers, P. Notice how we need to use the list constructor to evaluate the result of reversed. This is because reversed returns an iterator, a topic we will cover in much more detail later. These two functions have the advantage that they'll work on any finite length iterable source object. We've learned a lot about lists in this section together with some more general techniques such as slicing that can be used with other sequences including tuples and strings. Let's move on and look at the next collection type.

Dictionary

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 are delimited with curly braces and contain comma separated key-value pairs with each pair tied together by a colon. The values are accessible via the keys. The keys must be unique within any single dictionary, although it's fine to have duplicate values so long as they're associated with different keys. 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 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 named constructor, dict(), which can covert other types to the dictionaries. We can use the constructor to copy from an iterable series of key-value pairs stored in tuples like this. Recall that the items in a 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 the dict. As with lists, dictionary copying is shallow by default copying only the references to the key and value objects, not the value 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. Here we copy a dictionary of color codes. Note that we supply them in hexadecimal, but they're echoed back to us in decimal. The second means of copying is simply to pass an existing dictionary to the dict constructor. If you need to extend a dictionary with definitions from another dictionary, you can use the update() method. This is called on the dictionary to be updated and passed the contents of the dictionary, which is to be merged in. Here we merge in some additional color codes into our existing dictionary. If the argument to 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. As we have seen in an earlier module, dictionaries are iterable, so they 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. Notice that the keys are returned in an arbitrary order, which is neither the order in which they were specified nor any other meaningful sort 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 here in this example. In the interest of symmetry, there's also a keys() method which provides a view onto the keys of the dictionary, although it's not often used because default iteration of dictionaries is by key. 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 ahold of an iterable view of the items using the items() dictionary method. When iterated over, the items view yields each key-value pair as a tuple. By using tuple unpacking in the for statement, we can get both the key and value in one operation without the extra lookup. The membership test for dictionaries use the in and not in operators and work only on the keys. Here we create a dictionary mapping ISO currency codes to Unicode currency symbols. We use the membership testing operators to show that the New Zealand dollar is in the list of symbols whereas the Macedonian Denar is not in the list of symbols. Here we use the del keyword to remove the fictional element unobtainium from our periodic table. 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 at the point of the line break as string keys are immutable, which is a good thing for the correct functioning of the dictionary, but there's no problem with modifying the dictionary values in the event that we discover some new isotopes. Here the augmented assignment operator is applied to the list object accessed through the H for hydrogen key. The dictionary is not being modified at all here, but the list is being extended. Of course the dictionary itself is mutable. We know we can add new items. Here we add three isotopes of nitrogen. 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. To use it, it's best to do something like from pprint import pprint as pp. If we didn't bind the pprint function to a different name pp, the function reference would overwrite the module reference preventing further access to the contents of the module. Arguably it's poor design to have a module containing functions of the same name because of this issue. Anyway, by using pp(m) we get a much more comprehensible display of our table of isotopes. Let's move on from dictionaries and look at a new built-in data structure we have not yet met, the set.

Set

The set data type is an unordered collection of unique elements. The collection is mutable in so far as elements can 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 has type set. Recall that empty curly braces create an empty dictionary, so to create an empty set we must resort to the set() constructor with no arguments. This is the form that Python echoes back to us too. The set() constructor can create a set from any iterable series such as a list. Duplicates are discarded. In fact, a common use of sets is to efficiently remove duplicate items from a series of objects. Here we remove duplicates from the list 1, 4, 2, 1, 7, 9, 9 resulting in the set 1, 2, 4, 9, 7. Naturally sets are iterable, although the order is arbitrary. Membership is a fundamental operation for sets, and as with the other collection types is performed using the in and not in operators. To add a single element to the set, simply use the add method. Adding an element that already exists has no effect, and neither does it produce an error. Multiple elements can 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 KeyError is given. The second method discard is less fussy and simply has no effect if the item 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 the objects they refer to. As we have already shown, the set constructor may also be used. Perhaps the most useful aspect of the set type is 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. Olivia, Harry, Lily, Jack, and Amelia have blue eyes. Harry, Jack, Amelia, Mia, and Joshua have blond hair. Only Harry and Amelia can smell hydrogen cyanide. Harry, Lily, Amelia, and Lola can taste phenylthiocarbamide or PTC. Mia, Joshua, Lily, and Oliva have O type blood, Amelia and Jack have B type blood, Harry has A type blood, and Joshua and Lola have AB type blood. To find all the people with blond hair, blue eyes, or both, we can use the union method, which collects together all of the elements, which are in either or both set. We can demonstrate that union is a commutative operation. That is, we can swap the order of the operands using the value equality operator to check for equivalence of the resulting sets. To find all the people with blond hair and blue eyes, we can use the set 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 set difference method. This finds all of 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 you want to determine which people have exclusively blond hair or blue eyes, but not both, we can use the symmetric\_difference method. This collects all the elements which are 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 the subset of another set using the issubset method. For example, to check whether all of the people who can smell hydrogen cyanide also have blond hair, you smell\_hcn. issubset(blond\_hair). This checks that all of the elements of the first set are also present in the second set. To test whether all of the people who can taste phenylthiocarbamide can also taste hydrogen cyanide, use the issuperset method. 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. We encourage you to experiment with these methods and become comfortable using them. They can make for some very elegant solutions.

Collection Protocols

That completes our tour of the collection types in Python, but we're not quite finished. Let's complete this module by looking at the protocols which unite these collections and allow them to be used in standard ways. 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 an anomaly typed language 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've encountered in Python according to which protocols they support. Support for protocol demands specific behavior from a type. As you can see from this table, most of the collection types we've looked at support the container, size, and iterable protocols. Many of them are also sequences. The immutable protocols are each represented by only one type we have encountered. The container protocol requires that membership testing using the in and not in operators be supported. The size protocol requires that the number of elements in a collection can be determined by passing the collection to the built-in len function. Iteration is such an important concept we're devoting a whole module to it later in this course. In short though, iterables provide a means of yielding elements one-by-one as they're requested. One important property of iterables is that they can be used with 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 the index method, that items can be counted with the count method, and that a reversed copy of the sequence can be produced with the reversed built-in function. We won't cover the mutable sequence, mutable mapping, and mutable set protocols here since we've only covered one representative of each protocol, so the generality afforded by the protocol concept doesn't gain as much as this juncture.

Summary

We've covered a lot of ground in this module. Let's try to summarize. Tuples are immutable sequence types. They have a literal syntax, which is optional parentheses, around a comma separated list. Notably, the optional trailing comma must be used for single element tuples to disambiguate them from parenthesized integers. Tuples are particularly useful for multiple return values from functions, and tuple unpacking useful for destructuring those return values. Tuple unpacking also has an interesting idiomatic use for swapping variables. Strings are also immutable sequence types, but specifically of Unicode codepoints. String concatenation is most efficiently performed with the join() method rather than with the addition or augmented assignment operators. The partition() method is a useful and elegant string parsing tool, and the format() method provides a powerful means of replacing placeholders with stringified values. Range objects are immutable sequence types which represent arithmetic progressions of evenly spaced integer values. Range is sometimes used to generate counters in loops, although the enumerate() built-in function is often a superior alternative. Lists are the only mutable sequence type we have seen and are very widely used. Lists support indexing forward using positive or 0 integers and the indexing from the end with negative indices. The 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 the list() constructor are less obscure. List and other collection copies in Python are shallow copies. References are copied, but objects are not. Dictionaries are mapping types, which map from keys to values. The keys must be immutable objects whereas the values may be mutable. Iteration and membership testing with dictionaries is done with respect to the keys. The keys(), values(), and items() methods provide views onto the different aspects of a dictionary allowing convenient iteration. Sets store an unordered collection of unique elements. Sets support powerful set algebra operations and predicates. The built-in collections are organized according to which protocols they support such as iterable, sequence, and mapping. In passing we have also found out that underscore is in common usage used for dummy or superfluous variables and that the pprint module supports pretty printing of compound data structures. Well, that just about wraps it up for this module on Python's collections. Next time we'll look at exception handling so we can deal with those occasional value and key errors we've encountered, learn the error of our ways with respect to return codes, and see that it's always easier to ask for forgiveness than for permission. Thanks for watching, and we'll see you in the next module.

Handling exceptions

Introduction

Hello. My name is Austin Bingham. Welcome to the sixth module of the Python Fundamentals course where we introduce exception handling techniques. 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 raise exception must be handled upon which control flow if transferred to the exception handler. If an exception propagates up the call stack to the start of the program, then an unhandled exception will cause the program to terminate. And 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 impose a false dichotomy by insisting that an event is 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. Import the convert function from this module into the Python REPL and call our function with a string to see that it has the desired effect. But if we call our function with an object that can't be converted to an integer, we get a traceback from the int call. What's happened here is that the int constructor raised an exception because it couldn't sensibly perform the conversion. We didn't have a handler in place, so it was caught by the REPL and the stack trace was displayed. The ValueError referred to in the stack trace is the type of the exception object, and the error message invalid literal for int() with base 10: 'hedgehog' is part of the payload of the exception object that has been retrieved and printed by the REPL. Notice the exception propagates across several levels in the call stack.

Handling Exceptions

Let's make our convert function more robust by handling the ValueError 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. We have decided that if a non-integer string is supplied we'll return -1. To reinforce your understanding of the control flow here, we'll add a couple of print statements. Modify the convert function to look like this. Let's test this interactively after restarting the REPL. (Typing) Note how the print in the try block after the point at which the exception was raised was not executed when we passed in giraffe. Instead, execution was transferred directly to the first statement of the except block. The int constructor only accepts numbers in strings, so let's see what happens if we feed an object of another type into it, say a list. This time our handler didn't intercept the exception. If we look closely at the trace, we can see that this time we receive 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 TypeError is handled too. 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 did 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.

Programmer Errors

Now that we're 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 type that is ever useful to catch with an except block. Almost anything that goes wrong with the 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 at 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 plug-in 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 which does precisely nothing. It's a NOOP, and its only purpose is to allow us to construct syntactically permissible blocks which 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 ahold of the exception object, in this case an object of type ValueError or TypeError 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 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 the print function. We take advantage of the fact that exception objects can be converted to strings using the str constructor. Now let's see that at the REPL.

Imprudent Error Codes

Let's add a second function string\_log to our module, which calls our convert function and computes the natural log of the result. At this point we must confess that we've gone out of way here to be deeply unpythonic by wrapping the perfectly good int conversion included with Python, which raises exceptions on failure, in our convert function, which returns a good old fashioned negative error code. Rest assured that this unforgivable Python heresy has been committed solely to demonstrate the greatest folly of error return 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 codes completely and go back to rasing an exception from convert.

Re-Raising Exceptions

Instead of returning an unpythonic 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 -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 ValueError or a TypeError, but our conversion error message has printed a standard error along the way.

Exceptions as APIs

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 a square root finding as an example using a home-grown square root function courtesy of Heron of Alexandria, although he probably didn't use Python. Place the following code in a file called roots. py. There's only one language feature in this program we haven't met before, 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 raises 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 two tight scopes for exception handling blocks. We can easily use one try except blocks for all of our calls to square root. We also need 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 zero. 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 free condition early on and raise an exception at that point. This 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 also modify the docstring to make it plain which exception type will be raised by a 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 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 handled gracefully.

Exceptions, APIs, and Protocols

Again, exceptions are part of a function's API, and more broadly they are part of certain protocols. For example, objects which implement the sequence protocol should raise and 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 new exception types, 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 are writing a key value database. It would be natural to use KeyError to indicate a request for a nonexistent key because this is how dictionary 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 pass the end of a list. ValueError is raised when the 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 look-up in a mapping fails. You can see that here when we look up a non-existent key in a dictionary.

Do Not Guard Against Type Errors

We tend to not protect against TypeErrors in Python. To do so runs against the grain of dynamic typing in Python and limits the reuse potential of the code that we write. For example, we could test whether an argument was an int using the built-in isinstance function and raise a TypeError exception if it was not, but then we'd also want to allow arguments that are instances of float as well. It soon gets complicated if we want to check whether our function will work with types such as rational, complex, or any other kind of number. And in any case, who's to say that it does? Alternatively, we could intercept TypeError inside our square root function and re-raise 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 you couldn't have known about when you designed your function, then that's all to the good. If not, execution will probably result in a TypeError anyway. Likewise, we tend to not catch TypeErrors very frequently either.

EAFP vs. LBYL

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, LBYL, and It's Easier to Ask Forgiveness than Permission, EAFP, 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 files 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 that 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 atomicity issue. 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 are forced to include error handling directly in the main flow of the 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- base style makes it very difficult for problems to be silently ignored.

Clean-Up 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 Library's OS module to change the current working directory, create a new directory at that location, and then restore 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 had 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 execution 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 further logging facility. Now if the os. mkdir call raises an OSError, the OS 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.

Moment of Zen

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 keypress 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 msvcrt. getch() function. Even though we are 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 ImportError 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 this exception. We allow it to propagate to our caller, which is whatever attempted to import this keypress 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

We've tackled a complicated subject in this module, so let's see if we can sum up. 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 a ValueError, KeyError, or IndexError. 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 with an except block re-raises the exception which is currently being processed. We tend to not to routinely check for TypeErrors. 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 directed to standard error using the optional file argument. Python supports the logical operators and and 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. Thanks for watching, and we'll see you in the next module.

Iterables

Introduction

Hello. My name is Robert Smallshire. Welcome to the seventh module of the Python Fundamentals course where we'll cover comprehensions, iterable objects and iterators, lazy evaluation with generators, and various tools included with Python for working with iterable objects.

List Comprehensions

Comprehensions in Python are a 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. Let's start with list comprehensions. 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, "Why sometimes I've believed as many as six impossible things before breakfast, " by splitting a string. Now comes the 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, which describes how to construct the elements of the list. 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. The general form of list comprehensions is expression of item for item in iterable. That is, for each item in the iterable object 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 this new list. This comprehension is the declarative equivalent of the following imperative code, which uses a for loop and the list to accumulate the lengths found so far. Notice that the source object over which we iterate doesn't need to be a list. It can be any iterable object such as a tuple. The expression which is in terms of the item can be any Python expression. Here we find the number of decimal digits in each of the first 20 factorials using the range function to generate the source sequence. Note also that the type of object produced by list comprehensions is nothing more or less than a regular list.

Set Comprehensions

Set support is similar comprehension syntax using, as you might expect, curly braces instead of square brackets, and number of digits in factorial's result contain 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 value, which will be evaluated in tandem for each item. Here's a dictionary we can play with, which maps countries to their capital cities, the U. K. to London, Brazil to Brazilia, Morocco to Rabat, and Sweden to Stockholm. One nice use for a dictionary comprehension is to invert a dictionary so we can perform efficient lookups in the opposite direction. This example converts from a capital to country mapping into a country to capital mapping. Note that dictionary comprehension is to not usually operate dictionary on sources while they can, but recall that iterating over a dictionary yields only the keys. If we want both the keys and the values, we should use the items() method of the dictionary, and then use tuple unpacking to access the key and values separately. Should your comprehension produce some identical keys, later keys will override earlier keys. In this example we map the first letters 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 and extract complex expressions into separate functions to preserve readability. The following is close to the limit of being reasonable for a dictionary comprehension. It uses the Standard Library glob module to find all of the Python source files in a particular directory, and then creates a mapping from the full path name for those files to the size of the file in bytes. The file path and file size are derived using facilities of the OS module.

Filtering Predicates

To make this interesting, we'll first define a primality testing predicate function. This function for identifying prime numbers won't win any prizes for efficiency, but it's reliable, simple to understand, and doesn't use any features of Python we haven't yet encountered. All three types of collection comprehension support an optional filtering clause, which allows us to choose which items of the source are evaluated by the expression on the left. We can use our new is\_prime function as the filtering clause of a list comprehension to produce all the primes less than 100 with the numbers up to and including 100 being produced by a call to the range constructor. We have a slightly odd looking X for X construct here because we're not applying any transformation 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

Simple is better than complex. 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 comprehension should ideally be purely 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-grain control is needed. There are two important concepts here on 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 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 crystalize into something you can work with. We'll use a list of the 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 built-in iter function, and then request a value from the iterator using the next function. Each call to next moves the iterator through the sequence returning Spring, Summer, Autumn, and Winter. But what happens when we reach the end? In a spectacular display of Python's liberal attitude to errors, Python raises an exception, specifically of the type StopIteration. Those of you coming from other programming languages with a more straight-laced 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 from that series, or if the series is empty raises a ValueError. This works as expected on any iterable object, in this case demonstrated on both a list and a set. 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.

Generators

Now we come onto 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 and 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 as streams of data from a sensor or active log files. By carefully designing our generator functions, 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's an implicit return at the end of the definition. To understand what generators do, let's start with a simple example at the Python REPL. We'll define the generator, and then we'll examine how the generator works. Here we write a generator called gen123(), which yields successively 1, 2, and 3. The generator function is introduced by def just as a regular Python function, and as we've seen, we must use the yield keyword at least once within the definition. Here I'm yielding a value of 1, and then 2, and then 3. We'll enter a blank line to complete the definition, and there we are. 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 standard ways of working with iterators to retrieve or yield successive values from the sequence. To retrieve the next value from an iterator, we use the built- in next function passing the iterator or generator in this case to the function. So, next(g) returns 1, and again 2, and again 3. Take note of what happens now that we've yielded the last value from our generator. Subsequent calls to next raise a StopIteration exception just like any other Python iterator. Because generators are iterators, they can be used in all the usual Python constructs which expect iterators such as for loops. Here we call the generator again, and in each iteration of the loop we print the return value 1, 2, 3 just as we'd expect. Be aware that each call to the generator function returns a new generator object. Here we call gen123() again assigning to H, and again assigning to I. We can see when we display these objects that they have distinct addresses. This means that each generator can be advanced independently. Let's request the first value from H, the second value from H, and now the first value from I. 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. As we enter the generator, we'll print a message. Then we will yield 2 and print a message about yielding 4, and then yield 4, print a message about yielding 6, and then yield 6. And then at the generator function we'll print a message indicating that the function is about to return. A blank line to complete the definition, and now we can call our new generator assigning to G. Note that at this point the generator object has been created and returned, but none of the code within the generator body has yet been executed. Now let's call the built-in next function on our 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 call next(g) again, execution of the generator function resumes at the point it left off and continues running until the next yield. And then the final value. The function resumes again until it yields 6. After the final value is 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.

Stateful Generator Functions

Now we'll look at how our generator functions, which resume 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 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 will then combine into a generator pipeline. The first generator we'll look at is called take, which retrieves a specified number of elements from the front of a sequence. 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 count. Since generators are lazy and only produce values on request, we'll drive execution with a for loop and a run\_take function. In run\_take we create a source list called items, which we pass to our generator function along with a count of 3. Internally the for loop will use the iterator protocol to retrieve values from the take generator until it terminates. Once execution begins, the for loop in run\_take requests a value from take, which runs until it encounters the yield statement. At this point control returns to run\_take, which prints the value until we get to the next iteration of what is in effect the outer for loop. Each iteration of the outer loop causes execution to be transferred back into run\_take just sufficiently to yield the next value. Eventually the counter variable is equal to the counter argument and will literally return from run\_take. Behind the scenes this raises a StopIteration exception, which is caught by the internal machinery of the for loop in run\_take. This is the signal that take has done its work, so the for loop exits, and the program completes. 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, the continue is used to skip any values which have already been yielded. As execution starts, the for loop and run\_distinct() request a value from distinct. This runs until it reaches yield, which on this iteration simply returns the first item retrieved by this inner for loop form the items list. Control flow transfers to run\_distinct, which prints the value and proceeds with its next iteration. Note how when execution of distinct resumes we actually complete the work of the previous iteration by remembering the value just yielded before getting on with the most recently requested value. We can do this because item is not reassigned until we get back to the for statement. On the third iteration, the next item retrieved from the source is already present in the seen set. Control enters the if block, and the continue statement is executed transferring control back to the beginning of the innermost loop. Eventually the items list is exhausted. The generator returns implicitly at the end, the for loop and run\_distinct completes, and the program exits. 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. A coherent verbal description of step-by-step execution of a generator pipeline may not even be possible, but that's not going to stop us trying. When execution starts, distinct must be called first in order to produce the argument for take. Of course distinct returns the generator object over which take will be iterating. In turn, the generator object returned by take will be iterated over by the for loop in our run\_pipline driver function. This part of the process is hidden from the view of the debugger. When the outermost loop in run\_pipeline requests its first value, execution is transferred to take. Remember, the iterable over which take is looping is the generator produced by distinct. When the for loop and take request the first value from this generator, control is transferred to distinct. Distinct now runs until it reaches the yield at which point it is returning the first item from the source list, 3. The value is yielded back to the for loop intake, which executes until it in turn yields the value back to the loop in run\_pipeline, which only now starts its first iteration printing the first value. Execution continues in this interleaved manner until the first three distinct values have been yielded and printed. The advantage of this approach is that distinct only does just enough work to yield the first three distinct values rather than processing its whole source list. It pays to be lazy.

Laziness and the Infinite

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 such as primes or factorials, or perhaps the contents of multi-terabyte files. The authors of this course are sworn by sacred oath never to use either Fibonacci or quick sort implementations in demonstrations or exercises. Allow us to present the 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 and 1, and each value after that is the sum of the two preceding 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 first it yields the value of B, and then second A and B are updated to hold the new previous two values using a neat application of tuple unpacking. Now that we have the lucas() generator, it can be used like any other iterable object. For instance, to print the lucas numbers you could use a loop like this: For x in lucas() print (x). Of course since the lucas sequence is infinite, this will run forever printing out values until your computer runs out of memory. Use Control+C to terminate the loop.

Generator

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, the expression of item for item in iterable 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 one million square numbers. At this point, none of the squares have been created. We've 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 megabytes for the list object and the integer contained therein. Also notice that a generator object is just an interator 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 sum function, which accepts an interable series of numbers. If we were to use a list comprehension, we could expect this to consume around 400 megabytes 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'll seen that in this case we didn't supply separate enclosing parentheses for the generator expression in addition to those needed for the sum function call. This elegant ability to have the parentheses used for the function call also serve for the generator expression aids readability. You can include the second set of parentheses if you wish, but it's not required. As with comprehensions, you can include an if clause at the end of the generator expression. Reusing an admittedly inefficient is\_prime predicate, we can determine the sum of those integers from the first 1000, which are prime, like this.

Batteries Included for Iteration

So far we've covered the many ways Python offers for creating iterable objects. Comprehensions, generators, and any object that 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 call 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 thousand primes problem using the built-in sum function with two generator functions from itertools, islice() and count(). Earlier we made our own take generator function for lazily retrieving the start of a sequence. We needn't have bothered, however, because islice allows us to perform lazy slicing similar to the built-in list slicing functionality. Once we've imported the itertools module, to get the first 1000 primes we need to do something like this. Itertools. 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 always be finite, that is banded at both ends. What we'd like is an open-ended version of range, and that is exactly what itertools count() provides. This returns a special islice object, which is iterable. We can covert it to a list using the list constructor. Answering our question about the sum of the first 1000 primes is now easy so long as we remember to recreate the generator when we pass it to sum. The answer is 3, 682, 913. Two other very useful built-ins which facilitate elegant programs are any and all. They're equivalent to the logical operators and or, but for iterable series of bool values. Any returns true of any of the series it's passed are true, and all returns true if all of the series passed to it are true. Here we 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, and remarkably the answer is false. For a completely different type of problem, we can check that all of the city names are proper nouns with initial uppercase letters. The last built-in we'll look at is zip, which as its name suggests gives us a way to synchronize iterations over two iterable series. For example, if we have two columns of temperature data, one from Sunday and one from Monday, we can combine them together into pairs of corresponding readings. 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 and use other built-ins to calculate statistics for corresponding times. Here we use the min and max built-ins for the minimum and maximum temperatures and the sum and len built-ins to compute the average temperature. Perhaps though we'd like one long temperature series for Sunday, Monday, and Tuesday. We can then lazily concatenate iterables using itertools chain, so this is different from simply concatenating the 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 have made together and leave your computer computing the lucas primes. This shows the beautiful composability of generator functions, generator expressions, pure predicate functions, and for loops. When you've seen enough or broken some records, we recommend you spend some time exploring the itertools module.

Summary

In this module we've covered the techniques that we'll run through some of the most elegant Python code you'll produce. Let's recap what we've learned. Comprehensions are a concise and readable syntax for describing lists, sets, and dictionaries in a declarative way. These comprehensions iterate on an iterable source object and apply an optional predicate filter and a mandatory expression. Both filter and expression are usually in terms of the current item. Iterable objects are objects over which we can iterate item-by-item. We retrieve an iterate all from an iterable 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. When the series is exhausted, iterators raise a StopIteration exception. Generator functions allow us to describe sequences using imperative code. Generator functions look just like regular functions and have all the same facilities, but they must contain at least one instance of the yield keyword. The generator is produced when you call a generator function are iterators. When the iterator is advanced with next(), the generator starts or resumes execution up to and including the next yield statement. Although generators can't be reused, generator functions can. Each call to a generator function creates a new generator object. Generators can maintain state between calls in local variables and because they are lazy can model infinite series of data. Generator expressions are a sort of hybrid of generator functions and list comprehensions. These 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 itertools module. We've given you some powerful tools in this module for writing functional style programs. In the next module we'll experience a paradigm shift as we get to grips with object-oriented programming with classes. See you there.

Classes

Introduction

Hello. My name is Austin Bingham. Welcome to the eighth module of the Python Fundamentals course. Here we'll cover classes, Python's tools for creating new types. You can get a long way in Python using the built-in scaler and collection 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. 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 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 in which attributes and methods are available through the object. For example, on a string object the methods we can use on that object such as 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, the new class name in Python uses CamelCase, sometimes known as PascalCase 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 the 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 syntactially 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. 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 its constructor, which is done by calling the class as we would a function. The constructor returns a new object, which here we assign to a name F. If we use the type function to request the type of F, we see that it's of class 'airtravel. Flight. ' 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 defined 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 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. Now from a fresh REPL let's import that and see how it works. Notice that when we call the method we do not provide the instance F for the actual argument itself in the argument list. That's because the standard method invocation form with the dot is simply syntactic sugar for the class name followed by a dot followed by the method name with the instance passed as the first argument. If you try the latter, you'll find that it works as expected, although you'll almost never see this form used for real.

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 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 to double underscore init delimited by the double underscores used for Python runtime machinery. Like all other instance methods, the first argument to double underscore init must be self. In this case, we also pass a second argument to double underscore init, which is the flight number. The initializer should not return anything. It simply modifies the object referred to by self. If you're coming from a Java, C#, or C++ background, it's tempting to think of double underscore init as being the constructor. This isn't quite accurate. In Python, the purpose of double underscore init is to configure an object that already exists by the time it's 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 \_number. Assigning to an object attribute that doesn't yet exist is sufficient to bring it into being. 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 \_number with the 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 is a widely followed convention that the implementation details of objects 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 \_number attribute and return it. Any arguments passed to the flight constructor will be forwarded to the initializer. So, to create and configure our flight object we can now do this. We can also directly access the implementation details. Although this is not recommend for production code, it's very handy for debugging and 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 have 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 are truths about the objects of that class that 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 establish class invariants in the double underscore init method and raise exceptions if they can't be attained. We use string slicing and various methods of the string class to perform validation. For the first time in the course we also see the logical negation operator not. Ad hoc testing in the REPL is a very effective technique during development. If we construct a flight with a valid flight number, everything works as expected. However, if that flight number doesn't have an airline code, we get a ValueError. Likewise, if the airline code is lowercase, we get a ValueError, and if we construct a flight that has letters instead of numbers you get another ValueError. And if we try to use five numbers instead of four, we also get a ValueError. Now that we're sure of having 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 can be very simple.

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 would validate these arguments to ensure for example that the number of rows was not negative. This is straightforward enough, but for the seating plan we'd like something a little more inline 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 avoid confusion with 1. We'll add a seating plan method, which returns the allowed rows and seats as tuples containing a range object and a string of seat letters. It's worth pausing for a second to make sure you understand how this function works. The range call produces an iterable 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 row. These two objects, the range and the string, are bundled up into a tuple. With that in mind, let's construct a plane. See how we use keyword arguments for the rows and seats for documentary purposes. Also recall that ranges are half open, so 23 is intentionally one 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 receive 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's constructed, and we'll follow the Law of Demeter by adding a method to report the aircraft model. This method will delegate to aircraft on behalf of the client rather than allowing the client to reach through the flight and interrogate the aircraft object directly. We also add a docstring to the class. These work just like function and module docstrings. We can now construct a flight with a specific aircraft. Notice that we construct the aircraft object and directly pass it to the flight constructor without needing an intermediate named reference for it.

Moment of Zen

Complex is better than complicated. Many moving parts combined in a cleaver 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, that is 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.

Example: 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 object, which is the range of row numbers retrieved from the aircraft in the previous line. We're not actually interested in the row number since we know it will match up with the list index in the final index, so we discard it by using the dummy underscore variable. The item expression part of the list comprehension is itself a comprehension, specifically a dictionary comprehension. This iterates over each letter for the row and creates a mapping from the single character 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. Here's the code after we put it into the initializer. Before we go further, let's test our code in the REPL. 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. 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 contains some interesting snippets. Methods are functions so deserve docstrings 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 ValueError and in the handler raise a new ValueError 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 objects support the container protocol. We check that the requested seat is unoccupied using an identity test with None. If it's occupied, we raise a ValueError. If we get this far, everything is in good shape, and we can assign the seat. It also contains a bug, but we'll discover that soon enough. Let's try our seat allocator at the REPL. Doing so, we see that we get a TypeError in allocate\_seat. Early on in your object-oriented Python career you're likely to see TypeError 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. If we create our flight object as normal, we then allocate\_seat 12A, but if we try to allocate\_seat 12A a second time we get a ValueError. Now we allocate 15F and 15E, but if we try to allocate E27 we get a ValueError because E27 is obviously not a seat. We now allocate 1C and 1D, but if we try to seat Larry Wall in seat DD we get yet another ValueError because of course DD is not a valid seat. After that we pretty print to see how things look. The Dutchman is quite lonely there in 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.

Defining 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 leave allocate\_seat much simpler. See the call to parse\_seat, and notice that method calls within the same object 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. It's quite normal to mix related functions and classes in the same module. Now let's see that at the REPL. You may find it remarkable that 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 go on and move Guido back to 15 with his fellow Europeans. 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 rows, which are not None. This excludes our dummy first row. The value of each item in the outer expression is the sum of the number of None values in each row. This inner expression iterates over values of the dictionary and adds one for each None found. See how we split the outer expression over three lines to improve readability. (Typing) And just to check, let's do a little bit of math.

OO 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 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 good reason. Rather than have a card printer query all the passenger details from the flight, we'll follow the 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. 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 this output line, build some banners and borders around it, and then concatenate the lines together using the join method called on newline operator 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 the card\_printer. This tells the card\_printer to print each passenger having sorted a list of passenger seat tuples obtained from a \_passenger\_seats() implementation detail method. Note the leading underscore. That method is in fact a generator function which searches all seats for occupants yielding the passenger and the seat number as they are found. Now if we run this on the REPL we can see that the new boarding card print system works.

Polymorphism and Duck Typing

Polymorphism is a programing 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 William 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. 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 a seating configuration that matches the aircraft model, which for the purposes of this exercise we assumed is fixed per model. Better and simpler perhaps to get rid of the Aircraft class entirely and make separate classes for each specific model of aircraft with a fixed seating configuration. Here is an AirbusA319, and here's a Boeing777. 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. The different types of aircraft work fine when used with flight because they both quack like ducks or fly like planes or something. 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 can be derived from a 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 have just demonstrated. The fact that no Python method calls or attribute lookups are bound to actual objects until the point at which they are called known as late binding means we can attempt polymorphism with any object, and we'll succeed if the object fits. Although inheritance in Python can be used to facilitate polymorphism, after all derived classes will have the same interfaces as the base classes, inheritance in Python is most useful for sharing implementation between classes. As usual, this will make much more sense with an example. We would like our aircraft class's 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. Unfortunately, we now have duplicate code across two classes, and as we add more aircraft types, the code duplication will worsen. The solution we 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 classes. This class isn't usable on its own because it depends on a method called seating\_plan, which isn't available at this level. Any attempt to use it standalone will fail. The class is abstract in so far as it is never useful to instantiate it alone. Now the derived classes. 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, and this is the Boeing class. Let's exercise them at the REPL. We can see that both subtype aircraft inherited the num\_seats() method, which now works as expected because the call to seating\_plan 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 and inheritance are an important topic in Python, so let's sum up. All types in Python have a class. Classes define the structure and behavior of an object. The class is determined 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 \_\_init\_\_() can be provided, which is used to configure the self object at creation time. If present, the constructor calls the \_\_init\_\_() method. Double underscore 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 can 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 generally grouped together this way. Polymorphism in Python is achieved through duck typing where attributes and methods are only resolved at point of use. This is called 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 character. 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. Thanks for watching, and we'll see you in the next module.

Files and Resource Management

Introduction

Hello. My name is Robert Smallshire. Welcome to the ninth module of the Python Fundamentals course where we'll cover reading and writing data from text and binary files and discover how to manage resources such as files elegantly using context managers. 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. This is required. Mode, which specifies read/write/append, and binary or text mode. This is optional, but we always recommend specifying it for clarity. Explicit is better than implicit. Encoding. If the file contains encoded text data, this is the text 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, which may not be what you want. At the file system level, files contain only a series of bytes. Python distinguishes between files opened in binary and text modes 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 files reflect the raw data in the file. A file opened in text mode treats its contents as if it contains text strings of the str type, the raw bytes having first been decoded using a platform dependent encoding or using the specified encoding if given. By default, text mode also engages support for Python's universal newlines. This causes translation between a single portable newline character in our program strings, /n, and a platform-dependent newline representation in the raw bytes stored in the file system, for example carriage return newline /r/n on Windows. Getting the encoding right is crucial for correctly interpreting the contents of a text file, which is why we label the point. If you don't specify an encoding, Python will use the default from sys. getdefaultencoding. In our case, that's a 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. Better for all concerned to make a conscious decision about the text bytes encoding. You can get a list of supported text encodings in the Python documentation.

Writing Text Files

Let's start by writing some text 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. The first argument is the filename wasteland. txt. The mode 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 R, W, or A with the optional plus modifier should be combined with a selective text or binary mode T or B. 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 from 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. Browsing through the help we can see that F supports a method write. Quit the help with Q, and continue at the REPL. The write call returns the number of codepoints or characters written to the file. Let's write a few more lines. It is the caller's responsibility to provide newline characters where they are needed. There is no writeline method. (Typing) When we finish writing, we should remember to close the file by calling the close method, f. close(). Note that when we close the file all the contents become visible. Closing files is important. If you now exit the REPL and look in your file system on Unix with ls - l, you should see the wasteland. txt file with 78 bytes. On Windows with dir, you should see wasteland. txt with 79 bytes. This is because Python's universal newline behavior for files has translated the line endings to your system's native endings. The difference is because Python's universal newline behavior for files has translated the line endings to your platform's native endings. The number returned by the write method is the number of codepoints 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 Files

To read the file back, we use open again, but pass a different mode string, in this case 'rt' for read text. If we know how many bytes to read or if we want to read the whole file, we can use the read function. Looking back through our REPL, we can see that the first write was 32 characters long. We can read that back with a 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 bytes. The call returns the text and advances the file pointer to the end of what was read. The return type is str because we opened the file in text mode. To read all the remaining data in the file, we can call read without an argument. This gives us parts of two lines in one string. Note the newline character in the middle. At the end of the file, further calls to read return an empty string. Normally when we are 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. Using read for text is quite awkward, and thankfully Python provides us with better tools for reading text files line-by-line. First of these is the readline() function. The returned lines are terminated by a single newline character if there is one present in the file. The last line here does not terminate with a newline because there is no newline sequence at the end of the file. You shouldn't rely on the string returned by readline being terminated by a newline. Again, the universal newline support will have translated to /n 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 seek. 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 pausing the file involves hopping backwards and forwards between lines. It's much easier to do this with a list than with a file stream of characters. This time we'll close the file before moving on with g. close().

Appending to Text Files

Sometimes we would 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 combine that with T to be explicit about using text mode. Although there is 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 readlines whilst also giving us the flexibility of using writelines to write any iterable series of strings to a file. Notice that only three lines are completed in this example. I say completed because the file we're appending to doesn't end with a newline.

Files as Iterators

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 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, files. py. We'll use this to contain the code for the following example. When run, this program opens the filename passes its command line argument, and iterates over the file with a for loop line-by-line printing each line as it goes. Let's run this program directly from the system command line passing the name of our text file wasteland. txt. The double line spacing occurs because each line of the poem is terminated by a newline, 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 used to write to the file. Files and streams are closely related and can be used because the stream is a file-like object. We can get hold of a reference to the standard out stream from the sys module. Rerunning, we now get single-space text. Now alas it's time to move on from one of the most important poems of the 20th century and get to grips with context managers.

Managing Files With Try..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 need 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 a file using the writelines method. A generator expression is used to convert each number to a string and add a newline. Itertool's 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 filename and series length as command line arguments. Now let's make a complimentary module, series. py, which reads this data file back in. This simply uses a for loop to iterate over the file reading one line at a time and strips the newline with a call to the strip string method and coverts 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 data file, and rerun series. py. The int constructor raises a ValueError, which is unhandled, 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. Now the file will always be closed. Doing so opens up the opportunity for another refactoring. We can replace the for loop with a list comprehension and return the list directly. Even in this situation close will be called. The finally block is called; however, the try block is exited.

Context Managers and with-blocks

Up to now, our examples have all followed a pattern. Open a file, work with the file, and close the file. The close is important, because it informs the underlying operating system that you're done working with a file. If you don't close a file when you're done with it, it's possible to lose data. There may be pending rights 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 it happens even if we forget. This need for resource cleanup is common enough that Python implements a specific control flow structure called with-block to support it. With-blocks can be used with any object which supports the context-manager protocol, and that includes the file objects returned by open(). Exploiting the context-manager nature of the file object and using a with-block, our read\_series function can become very simple. 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 series writing program to use a with-block too. This also removes the need for an explicit close.

Simple Is Better Than Complex

Beautiful is better than ugly. Sugary syntax, fewer defects attained through sweet fidelity. The with-block syntax 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 it can be very healthy for your code.

Writing Binary Files

To demonstrate handling of binary files, we need an interesting binary data format. The BMP file format contains device-independent bitmaps and is 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 sake, we have decided to deal with 8-bit grayscale images, which have the nice property that they are 1 byte per pixel. The write\_grayscale function accepts two arguments, the filename and a collection of pixels. As the docstring points out, this collection should be a sequence of iterable series of integers. A list of lists of int objects will 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 give the height and 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 would want to make. Next we open 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 four bytes should hold the 32-bit integer containing the file size, but we don't know that yet. We could've 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 four 0 bytes as a placeholder using escaping syntax to specify the zeros. The next two pairs of bytes are unused in this format, so we'll just write 0 bytes for them too. The next four bytes are for another 32-bit integer, which should contain the offsetting 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 header as a 32-bit integer. In our case, the header will always be 40 bytes long. We'll just hardwire that in 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 blue, green, red color. For grayscale images, we need to write 256 four 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 an argument to the function. At last we are ready to write the pixel data. But before we do, we make a note of the current file pointer offset using tell() as this is one of the locations we need to go back and fill in. Writing the pixel data is straightforward enough. We use the reversed built-in function to flip the order of the rows. BMP images are written from bottom to top. For each row, we simply pass the iterable series of integers to bytes constructor. If any of the integers are out of the range 0-255, the constructor will raise a ValueError. After the pixel data, we are at the end of the file. We undertook to record this offset value earlier, so we record the precision using tell() into an end of file bookmark variable. Now we can rewind and fulfill our promises by replacing the placeholder offsets we recorded with the real thing. To do this, we seek back to the size\_bookmark we remembered back near the beginning of the file and write to the size stored in the 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 buffered 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 \_int32\_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 bitwise shift and bitwise and operators to extract individual bytes from the integer value. Notice that bitwise and uses the ampersand symbol to distinguish it from the logical and, which is the spelled out word. 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 four resulting integers are used to construct a tuple, which is then passed to the bytes constructor to produce a 4-byte sequence.

Fractal Images

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 for the iconic Mandelbrot set fractal. We're not going to explain the fractal generation code in detail, and still less the math behind it, but the code is simple enough and doesn't rely on any Python features we haven't encountered already. The key takeaway is that the Mandelbrot function uses nested list comprehensions to produce a list of lists of integers in the range 0-255 representing an image of the fractal. The integer value of each point is produced by the mandel function. Let's fire up the REPL and use the fractal and BMP modules together. First we use the Mandelbrot function to produce an image of 448 x 256 pixels. You'll get best results using images with an aspect ration of about 7:4. 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 data structure, a list of lists of integers just as we were promised. Let's write those pixel values to a BMP file. Find the file, and open it in an image viewer, for example by opening it in your web browser.

Reading Binary Files

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 and pixels from a BMP file. We'll add the code into bmp. py. Of course we 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 at the first two magic bytes that we expect in a bmp file. If they're not present, we raise a ValueError, which will of course cause 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 bytes each for the two 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 detail method \_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 all, which is the vertical bar or the pipe symbol together with indexing of the bytes object to reassemble the integer. Notice that indexing into a bytes object returns an integer. Let's try our new BMP dimensions function on the mandel. bmp file.

File Like Objects

There's a notion in Python of file-like objects. This isn't as formal as a specific protocol like sequence protocol is for tuple objects, but thanks to the polymorphism afforded by duck typing it works well in practice. This is where the 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 exit 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 is a file. We've actually seen this already. The objects returned to us when we opened files in text and binary mode are actually of different types, although both with definitely 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 to the number of words per line in a file and return that information as a list. Now we'll open a regular text file containing the fragment of T. S. Eliot's masterpiece we created earlier and pass it to our new function. 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. (Typing) In this example, the type of web file is http. client. HTTPReponse, a quite different thing. However, since they are both 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.

Closing With 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 useable with a context manager using the code in fridge. py. The module includes a class RefrigeratorRaider, which well raids the refrigerator. It has three methods: Open, which opens the fridge door; take, which gets some food from the refrigerator; and close, which closes the fridge door. There's also a module scope driver function raid, which performs a full raid on the refrigerator. We'll import raid into the REPL and go on the rampage. First, let's take some bacon. As expected, the code opens the fridge doors, finds the bacon, takes the bacon, and closes the fridge door. Very responsible. It's important that we remember to close the door so the food will be preserved until our next raid. Now let's try another raid for deep fried pizza. This time we were interrupted by the 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 call to the RefrigeratorRaider constructor in a call to the closing function, 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 the with-block. Now when we execute a raid the fridge door is closed twice. We see that our explicit call to close in unnecessary, so let's fix that up too. A more sophisticated implementation might check that the door was already closed and ignore subsequent requests. Let's try again. This time, even though the health warning was triggered, the door was still closed for us afterwards by the context manager.

Summary

Let's summarize what we've covered in this module. Files are opened using the built-in open() function, which accepts a file mode. This controls read/write/append behavior and also whether the file is treated as binary or encoded text data. For text data, it's good practice to always specify an encoding. Text files differ from binary files by dealing with string objects and performing universal newline translation and string encoding. Binary files deal with bytes objects with no newline translation or encoding. When you write text files, it's up to us to provide newline characters for line breaks. Files should always be closed after use to prevent resource leaks and to ensure that all data has been committed to the file system. Files provide various convenient methods for working with lines, but are also iterators, which yield values line-by-line. Files are also context mangers and can be used with the with-statement. This ensures that cleanup operations such as closing the files are performed. The notion of file-like objects is loosely defined, but very useful in practice. Exercise easier to ask forgiveness than permission to make the most of them. Context managers aren't restricted to file-like objects. We can use the tools in the contextlib standard library module such as the closing() wrapper to create our own context managers. Along the way in this module we found that help() can be used on instance objects, not just types, and Python supports bitwise operators bitwise &, bitwise or, and left- and right-bitwise shifts. We've given you some powerful tools in this module for getting data into and out of your programs. Next time we'll look at how to make your programs correct with debuggers, reliable with unit testing, and deployable to other computers and people with packaging. Thanks for watching, and we hope to see you in the next and last module.

Shipping Working and Maintainable Code

Introduction and unittest

Hello, my name is Austin Bingham. Welcome to the 10th module of the Python Fundamentals course. Here, we'll cover some topics related to producing working maintainable code in Python. We'll look at the unittest module which you can use to automate testing of your code and we'll look at the debugger that comes with standard library. After that, we'll briefly cover the basics of packaging and distributing your code as well as how to install third party packages. When we build programs at even minor complexity, there are countless ways for defects to creep into our code. This can happen when we initially write the code but we're just as likely to introduce defects when we're making modifications to it. To help get a handle on defects and keep our code quality high, it's often very useful to have a set of tests that you can run which will tell you if you code is acting as you expect it to. To help with this, the Python standard library includes the unittest module. Despite its name, this module provides a flexible framework for automating tests of all sorts from acceptance tests to integration tests to unit tests. Its key feature, like many testing framework in many languages, is that it makes automated and repeatable tests. Meaning that you can cheaply and easily verify your code at any time. The unittest module is built around a handful of key concepts. At the center of these concepts is a notion of a TestCase which groups together a set of related individual test functions. A TestCase is the basic unit of test organization in the unittest framework. The next important concepts is that of fixtures. Fixtures are pieces of code which run before and/or after every test method. Fixtures are used to make sure that the test environment is in an expected state before a test is run. For example, to create a necessary database table or populate a cache. Fixtures are then used to clean up any resources that may have been used in a test method. The final key concept is that of assertions. Assertions are how you can tell the unittest framework to make specific checks which determine whether a test passes or fails. Among other things, assertions can make simple Boolean checks, perform object equality tests, or verify that the proper exceptions are thrown. If an assertion fails, then a test function fails. So assertions are really the lowest level of testing you can perform. With those concepts in mind, let's see how to actually use the unittest module. For this, we'll use test-driven development to write a simple text analysis function. This function will take in a filename, read the file, and calculate both the number of lines in the file and the number of characters in the file. Since this will be an iterative development process, we'll put the code in the file text\_analyzer. py rather than work at the REPL. So to start, let's create our first test with enough supporting code to actually run it. First, we import the unittest module. Next, we create our test case by creating a class, TextAnalysisTests which derives from unittest. TestCase. This is how you create test cases with the unittest framework. To define individual test methods in a test case, you simply create methods to start with test\_. These are automatically discovered by the unittest framework and don't require any sort of explicit registration. In this case, we define the simplest possible test. Does the analyze\_text function run at all? This function doesn't make any explicit checks, but rather relies on the fact that a test method will fail if it throws any exceptions. In this case, if analyze\_text isn't defined, then this test will fail. Finally, we define the idiomatic main block which calls unittest. main when this module is executed. unittest. main will search for all TestCase subclasses in a module and execute all of their test methods. Since we're using test-driven design here, we expect our test to fail at first, and indeed, this fails spectacularly for the simple reason that we haven't defined the necessary function yet. unittest. main produces a simple report telling us how many tests run and how many failed. It also shows us how the test failed. In this case, it's showing us that we got a NameError when we tried to run the non-existent function analyze\_text. Let's fix that by adding the function. Remember that in test-driven development, we only write enough code to satisfy our tests, so all we do now is create an empty function. Running the test again, we find that we now pass. The next thing you want to do is to be able to pass a filename to analyze\_text so that it knows what to process. For this to make sense, we want the filename to represent an actual file. To make sure that a file exists, we're going to define some fixtures. The first fixture we can define is a method on the TestCase called setUp. This function is run before each test method. In this case, we'll use setUp to create a file for us and remember the filename as a member of the TestCase. The second fixture available to us is another TestCase method called tearDown. tearDown is run after each test method. And in this case, we're going to use it to delete the file we created in setUp. Since we're using the os module in tearDown, we need to import it at the top of the file. Notice how tearDown swallows any exceptions thrown by os. remove. We do this because tearDown can't actually be certain that the file exists, so it simply tries to remove the file and it assumes that any failure is acceptable. With our two fixtures in place, we now have a file that is created before each test method and we just delete it after each test method. This means that each test method is starting in a stable known state. This is critical to making reproducible tests. You may have noticed that the setUp and tearDown function names aren't inline with what PEP 8 prescribes. This is because the unittest module predates those parts of PEP 8 which specify the convention of function names being in lowercase with underscores. There are several such cases in the Python standard library but most new Python code follows the PEP 8 style. Now that we have a finalname that we can pass to analyze\_text, let's pass it in by modifying our existing test. Since the self argument passed to the fixture is the same instance as that passed to the test method, our test can access the filename attribute created in setUp. Of course, this test fails because analyze\_text doesn't accept any arguments yet. That's simple enough to fix. Now we're passing again. Now that we're satisfied that the function exists and accepts the right number of arguments, let's see if we can make it do real work. The first thing we want is for the function to return the number of lines in the file, so let's define that test. Here, we see our first example of an assertion. The TestCase class has many assertion methods and in this case, we use assertEqual to check that the number of lines counted by our function is equal to four. If the value returned by analyze\_text is not equal to four, this assertion will cause the test method to fail. And that is precisely what happens. We can now see that we're running two tests, that one of them passes and the new one fails with an AssertionError. Let's break from the TDD rules and move a bit faster now. First, we'll update the function to return the number of lines in the file. This gives good results. So let's add a test for the other feature we want which is to count the number of characters in the file. Since the function is now supposed to return two values, we'll have it return a tuple with a line count in the first position and character count in the second. Our new test looks like this. And it fails as expected. This tells us that it can't index into the integer returned by analyze\_text. So let's fix analyze\_text to return the proper tuple. This fixes our new test, but we find we've broken an old one. But that's easy enough to fix because all we need to do is account for the new return type. Now, everything is passing again. Another thing we want to test for is that analyze\_text throws the correct exception when it is passed a non-existent filename. Here we see the use of assertRaises assertion. This assertion checks that the specified exception type, in this case, IOError, is thrown from the body of the with block. Since open raises IOError for nonexistent files, our test already passes. Finally, we can see one more very useful type of assertion if we write a test to verify that analyze\_text doesn't delete the file, a reasonable requirement for the function. assertTrue simply checks that the value passed into it evaluates to true. There's an equivalent assertFalse which does the same thing but for false values. As you might imagine, this test passes already as well. So now we've got a useful passing set of tests. This example is small but it demonstrates many of the important parts of the unittest module. There are many more parts of unittest but you can get quite far using just the techniques we've seen here.

Debugging With PDB

Even with the comprehensive automated test suite, we can still get into situations where we need a debugger to figure out what's going on. Fortunately, Python includes a powerful debugger with the standard library, PDB. PDB is a command line debugger and if you're familiar with tools like GDB, then you already have a good idea of how to use PDB. PDB is different from many debugging tools in that it's not really a separate program but a module just like any other Python module. You can import PDB into any program and start the debugger using the set\_trace function call. This function simply starts the debugger at whatever point you are at in the program's execution. For our first look at PDB, let's use a REPL and start the debugger with set\_trace. You'll see that even after you execute set\_trace, your prompt changes from the triple chevron to Pdb in parenthesis. This is how you know you're in the debugger. The first thing we'll do is simply see what commands are available in the debugger by typing help. This lists a few dozen commands and some of which you'll use in almost every debugging session and some of which you may never use at all. You can get specific help on a command by typing help followed by the command name. For example, to see what continue does, type help continue. The curious parenthesis in the commands tell you that continue can be activated by typing c, C-O-N-T or the full word continue. Knowing the shortcuts for common PDB commands can greatly increase your comfort and speed at debugging. Rather than simply list all the commonly useful PDB commands, we're going to instead debug a simple function. Our function, is\_palindrome, takes in an integer and determines if the digits of the integer are a palindrome or not. A palindrome of course is a sequence which is the same both forwards and backwards. The first thing we'll do is create a new file, palindrome. py, with this code. As you can see, our code has three main parts. The first is the digits function which converts an integer into a list of digits. The second is the is\_palindrome function which first calls digits and then checks if the resulting list is a palindrome. The third part is a set of unit tests. We'll use these tests to drive the program. As you might expect, since this is a section on debugging, there's a bug in this code. We're going to first run the program and notice the bug, and then we'll see how to use PDB to find the bug. So, let's simply run the program. We have three tests that we expect to run, and since this is a relatively simple program, we expect it to run very quickly. Instead of running quickly, we see that this program seems to run forever. If you look at its memory usage, you'll also see that it grows in size the longer it runs. Clearly, something is wrong. So let's use Control + C to kill the program. We'll now use PDB to try to understand what's going on here. Since we don't know where our problem might lie, we don't know where to put a set\_trace call so we're going to instead start the program under the control of the PDB using a command line invocation. Here, we're using the -m argument which tells Python to execute this specific module, in this case, pdb, as a script. The remaining arguments are passed to that script. Here, we're telling Python to execute the pdb module as a script and we're passing the name of our broken file to it. What we're seeing is that we're immediately taken to a Pdb prompt. The arrow pointing to import unittest is telling us that this is the next instruction that will be executed when we continue. But where is that instruction? Let's use the where command to find out. Where reports are current call stack with the most recent frames at the bottom. And we can see that PDB has paused execution at the very first line of palindrome. py. This reinforces an important aspect of Python execution which we've discussed before. Everything is evaluated at run time. In this case, we pause execution right before an import statement. We can execute this import by running to the next statement using the next command. We see that this takes us to the def call for the digits function. When we execute another next, we move to the definition of the is\_palindrome function. You may be wondering why the debugger didn't step into the body of digits. After all, isn't it evaluated at run time like everything else? The answer is that the body of the function can only be evaluated when there are arguments supplied to it. So we will only be run when the function is called. The body of functions are checked for proper syntax when they're imported, but PDB doesn't let us debug that part of the process. We could continue using next to move through our program's execution, but since we don't know where the bug lies, this might not be a very useful technique. Instead, remember that the problem with our program is that it seemed to be running forever. This sounds a lot like an infinite loop. So rather than stepping through our code, we'll simply let it execute and then we'll use Control + C to break back into the debugger when we think it might be in that loop. After letting the program run for a few seconds, we press Control + C which halts the program and shows us that we're in the digits function of palindrome. py. If we want to see the source code at that line, we can use the PDB command list. We see that this is indeed inside a loop which confirms our suspicion that an infinite loop might be involved. We can use the return command to try to run to the end of the current function. If this doesn't return, we'll have very strong evidence that this is an infinite loop. We let this run for a few seconds to confirm that we never exit the function and then we press Control + C. Once we get back to our PDB prompt, let's exit PDB with the quit command. Since we know the problem lies in digits, let's set an explicit breakpoint in there using the pdb. set\_trace function mentioned earlier. Remember that the set\_trace function will halt execution and enter the debugger. Now we can just execute our script without specifying the PDB module. And we see that we almost immediately go to a PDB prompt with execution halted at the beginning of our digits function. To verify that we know where we are, let's use where to see our call stack. Remember that the most recent frames are at the end of this listing. After a lot of unittest functions, we see that we are indeed in the digits function and that it was called by is\_palindrome just as we expected. What we want to do now is watch execution and see why we never exit the function's loop. Let's use next to move to the first line of the loop body. Now let's look at the value of some of our variables and try to decide what we expect to happen. We can examine values by using the print command. This looks correct. The digs list which will contain the sequence of digits in the end is empty, and x is what we passed in. We expect the divmod function to return 123 and 4, so let's try that. This looks correct, divmod has clipped off the least significant digit from our number. The next line puts that digit into our results list. If we look at digs, we'll see that it now contains mod. The next line will now update x so that we can continue clipping digits from it. We see that execution goes back up to the while loop as we expected. Let's look at x to make sure it has the right value. Wait a second. We expect x to hold the digits that aren't already in the results list. Instead, it contains only the digit in the results list. Clearly, we've made a mistake in updating x. If we look at our code, it quickly becomes apparent that we should have assigned div rather than mod to x. Let's exit PDB. Note that you may have to run quit a few times because of how PDB and unittest interact. After you're out of PDB, let's remove the set\_trace call and modify digits to fix the problem we found. If we run our program now, we see that we're passing all tests and it runs very quickly. That's a basic PDB session and it demonstrates some of the core features of PDB. PDB has many other commands and features, however, and the best way to learn them is to simply start using PDB and trying out the commands. This palindrome program can serve as a good example for learning most of the features of PDB.

Virtual Environments

Before we move on to the next sections, we'll need to quickly look at how to create virtual environments for Python. A virtual environment is a light-weight, self-contained Python installation that users can create without needing administrator rights on their system. If you're using Python 3. 3 or later, then you should already have a module called venv installed with the standard library. You can verify this by running this from the command line. If you don't have venv installed, there is another tool, virtualenv that you can get from the Python Package Index which works very similarly. You can use either venv or virtualenv for this course, but we'll be using venv in the examples. Using the venv module is very simple. You simply specify the directory to contain your new virtual environment. The tool creates the new directory and populates it with the installation. Once the environment is created, you can activate it by using the activate script in the environment bin directory. On Linux or Mac OS, you have to source the script. On Windows, you simply run it. Once you do this, your prompt will change to remind you that you are in a virtual environment. And most importantly, the Python that will execute when you run Python is from the virtual environment. To leave the virtual environment, you use the deactivate command. This will return you to the parent shelf from which the virtual environment was activated. We'll be using virtual environments in the following sections, so make sure that you can create them before moving on.

Distributing Your Programs

Packaging and distributing your Python code can be a complex and sometimes confusing task, especially if your projects have lots of dependencies or involved components more exotic than straight Python code. However for many cases, it's very straightforward to make your code accessible to others in a standard way. We'll see how to do that using the standard distutils module in this section. The distutils modules allows you to write a simple Python script which knows how to install your Python modules into any Python installation. By convention, this script is called setup. py and it exists at the top of your project structure. This script can then be executed to perform the actual installation. Let's see a simple example of distutils. We'll create a basic setup. py installation script for the palindrome module we wrote in the previous section. The first thing we want to do is to create a directory to hold our project. Let's call this palindrome. Let's put a copy of our palindrome. py in this directory. Finally, let's create our setup. py. The first line in the file imports the functionality we need from the distutils. core module namely the setup function. This function does all of the work of installing our code, so we need to tell it about the code we're installing. We do this of course with the arguments we pass to the function. The first thing we tell setup is the name of the project. We've chosen palindrome in this case, but you can choose any name you like. In general though, it's simplest to just keep the name the same as your project name. The next argument we pass to setup is the version. Again, this can be any string you want. Python doesn't rely on the version to follow any rules. The next argument, py\_modules, is probably the most interesting. We use this to specify the Python modules we want to install. Each entry in this list is the name of the module without the. py extension. setup will look for the matching. py file and install it. So in our example, we've asked setup to install palindrome. py which of course is a file in our project. The rest of the arguments we're using here are fairly self-explanatory and are there mostly to help people to use your module correctly and to know who to contact if they have problems. Before we start using our setup. py, we first need to create a virtual environment into which we'll install our module. In your palindrome directory, create a virtual environment called palindrome\_env. When this completes, activate the new environment. On Linux or Mac OS, source the activate script. On Windows, call the script directly. Now that we've got our setup. py, we can use it to do a number of interesting things. The first and perhaps most obvious thing we can do is install our module into our virtual environment. We do this by passing the install argument to setup. py. setup prints out a few lines to tell you about its progress. The most important line for us is where it actually copies palindrome. py to the installation folder. The site-packages directory of a Python installation is where third party packages such as ours are normally installed. So it looks like the installation worked properly. Let's verify this by running Python and seeing that our module can be imported. Note that we want to change directories before we do this. Otherwise, when we import palindrome, Python will simply load the source file in our current directory. Here, we use the file attribute on the module to see where it was imported from, and see that we're importing it from our virtual environment's site-packages which is exactly what we wanted. Don't forget to switch back to your source directory after exiting the Python REPL. Another useful feature of setup is that it can create various types of distribution formats. It will take all of the modules you specified and bundle them up into packages that are easy to distribute to others. You can do this with the sdist command which is shorthand for source distribution. If we look, we'll see that this command created a new directory dist which contains the newly generated distribution file. If we unzip that file, we'll see that it contains our project's source code including the setup. py. So now you can send this zip file to anyone who wants to use your code, and they can use the setup. py to install it into their system, very convenient. Note that the sdist command can produce distributions of various types. To see the available options, you can use the --help-formats option. This section really just touches on the very basic of distutils. You could find out more about how to use distutils by passing --help to setup. py. For most simple projects, however, you'll find that what we've just covered is almost all you need to know.

Installing Third-party Modules

Packaging in Python has a troubled and confusing history. Thankfully, the situation has settled down and the tool called pip has emerged as the clear winner among package installation tools for general purpose Python use. For more specialist uses such as numerical or scientific computing which rely on the NumPy or SciPy packages, you should consider Anaconda as a strong alternative to pip. In this section, we'll focus on pip as it is officially blessed by the core Python developers and comes with support out of the box. The pip tool is included and installed with Python since version 3. 4. For older versions of Python 3, you'll need to look up specific instructions on how to install pip for your platform as you may need to use your operating system's package manager depending on how you originally installed Python. The best place to start is the Python package user guide. The pip tool can search for packages in the central repository. The Python Package Index or PyPI also known by the nickname CheeseShop and then download and install them along with their dependencies. You can browse the PyPI at pypi. python. org/pypi. This is an extremely convenient way to install Python software so it's good to understand how to use it. We'll demonstrate how to use pip by installing the nose testing tool. nose is a sort of power tool for running unittest based tests such as those we developed at the beginning of this module. One really useful thing it can do is discover all of your tests and run them. This means that you don't need to add unittest. main into your code. You can just use nose to find and run your tests. First though, we need to do some groundwork. Let's create a virtual environment so we don't inadvertently install nose into our system Python installation. Create a virtual environment using venv and activate it. As pip is updated much more frequently than Python itself, it's good practice to upgrade pip in any new virtual environment, so let's do that. Fortunately, pip is capable of updating itself with pip install --upgrade pip. If you don't upgrade pip, it will give you warnings everything you use it if a new version has come available since you last upgraded. Now let's use pip to install nose. The pip tool uses subcommands to decide what to do. To install modules, you use pip install package name. If this succeeds, nose is ready to use in our virtual environment. Let's check that it's available by trying to import it at the REPL and introspecting the path at which it was installed. As well as installing a module, nose installs the nosetests program in the bin directory of the virtual environment. To really put the icing on the cake, let's use nosetests to run the test from the palindrome. py script we were working with earlier by changing into the palindrome project directory and passing the palindrome. py source file as the only argument to the nosetests program. You can also use pip to install for local packages and files rather than from the Python Package Index. To do this, pass the filename of the package distribution to pip install. Earlier, we showed how to build the source distribution of palindrome using distutils. To install this source distribution in our test environment using pip, do pip install palindrome-1. 0. zip. A key advantage to installing packages with pip rather than directly invoking the setup. py of the source distribution is that pip knows how to uninstall packages. To do so, use the uninstall subcommand, passing the name of the package you used at the time it was installed. So in this case, invoke pip uninstall palindrome-1. 0. zip.

Moment of Zen

In the face of ambiguity, refuse the temptation to guess. To guess is to know that you have left something out. What are you missing? Temptation to guess or to ignore ambiguity with wishful thinking can lead to short-term gains, but it can often lead to confusion in the future and bugs which are difficult to understand and fix. Before you make that next quick fix, ask yourself what information do you need to do it correctly?

Summary

We've really come full circle in this module. We started by learning how to develop automated tests using unittest. We then learned how to debug our code with PDB followed by sections on packaging and distributing our code with distutils and we finished off by learning how to install third party code into our Python installations. One of the programs we were able to install in the end was a tool to help us run the tests we developed earlier. That's quite a trip. Here's a more detailed list of the topics we covered. The unittest module is a framework for developing reliable automated tests. You define test cases by subclassing from unitttest. TestCase. The unittest. main function is useful for running all of the tests in a module. The setUp and tearDown fixtures are used to run code before and after each test method. Test methods are defined by creating method names that start with test\_ on test case objects. The various TestCase. assert methods can be used to make a test method fail when the right conditions aren't met. Use TestCase. assertRaises in a with statement to check that the right exceptions are thrown in the test. Python standard debugger is called PDB. PDB is a command line debugger. The PDB. set\_trace method can be used to stop program execution and enter the debugger. Your REPL's prompt will change to Pdb in parenthesis when you're in the debugger. You can access PDB's built-in help system by typing help. You can use python -m pdb followed by a script name to run a program under PDB from the start. PDB's where command shows the current call stack. PDB's next command lets execution continue to the next line of code. PDB's continue command lets program execution continue indefinitely, or until you stop it with Control + C. PDB's list command shows you the source code at your current location. PDB's return command resumes execution until the end of the current function. PDB's print command lets you see the value of objects in the debugger. Use quit to exit PDB. Virtual environments are light-weight, self-contained Python installations that any user can create. venv is the standard tool for creating virtual environments. venv accepts both a source-installation argument as well as a directory name into which it creates the new environment. To use a virtual environment, you need to run its activate script. When you activate a virtual environment, the prompt is modified to remind you. The distutils package is used to help you distribute your Python code. distutils is generally used inside a setup. py script which users run to install your software. The main function in distutil is setup. setup takes a number of arguments describing both the source files as well as metadata for the code. The most common way to use setup. py is to install code using python setup. py install. setup. py can also be used to generate distributions of your code. Distributions can be zip files, tarballs, or several other formats. Pass --help to setup. py to see all of its options. Common tools for installing third party software are distutils and pip. The central repository for Python packages is the Python Package Index also called PyPI or cheeseshop. To install modules with pip, use the subcommand notation pip install package-name. Along the way, we found that divmod calculates the quotient and remainder for division operation at one time. The reversed function can reverse a sequence. You can pass -m to your Python command to have it run a module as a script. Debugging makes it clear that Python is evaluating everything at run time. You can use the \_\_file attribute on a module to find out where its source file is located. Third party Python packages are generally installed into your installation's site-packages directory. nose is a useful tool for working with unittest based tests. Well done on completing our Python Fundamentals course. We hope we've given you a firm foothold on the start of your journey with Python. The knowledge we impart in this course is sufficient to create and maintain basic Python programs. But there's so much more to learn. Python is a large and complicated language with many moving parts. While we find it remarkable that much of this complexity is well hidden for much of the time, the moment will soon arrive when you need to deepen your Python language skills. Recall that this Python Fundamentals course is the first in our trilogy covering the core Python language and standard library. From here, you can proceed directly to our Python - Beyond the Basics course. Then, onwards again to the lofty heights of Advanced Python. If you like written materials to go along with this course, remember to check out our Python Craftsman book series. The book correspond to our Pluralsight courses which cover the core Python language. The first book in the Craftsman series, The Python Apprentice, corresponds to this Python Fundamentals Pluralsight course. The second book, The Python Journeyman, corresponds to our Python - Beyond the Basics Pluralsight course. The trilogy is completed by The Python Master which corresponds to our Advanced Python course. Pluralsight viewers can follow the URLs shown below each book to get them at a deep discount. In addition to the courses and books just mentioned, we'll doubtlessly back with more content for the ever growing Python language and library. Please remember though that the most important characteristic of Python, it's great fun to write Python software, so enjoy yourselves.