Course Overview

Course Overview

[Autogenerated] Hi, everyone. My name is Austin Bingham and welcome to the course or Python getting started. My co author, Robert Small Share and I are founders in principle consultants at 60. North Python is a big language, and it's important to have a thorough grounding in its key concepts in order to be productive and create high quality python programs. In our experience, starting off in the right direction and avoiding early misconceptions is crucial to success with Python, and that's the kind of start we aim to provide with this course. In this course, we're going to introduce you to the essentials of the Python language as well as important parts of the pipe in Standard Library. Some of the major topics that we will cover include pythons, fundamental data types using functions and modules to organize your code pythons, underlying object model, defining your own types, using classes and working with adoration and in terrible objects. By the end of this course, you'll know what you need to work on many python projects, and you'll be in a great position to continue learning more advanced aspects of Piper before beginning the course. You should be familiar with basic computer concepts like files and running programs. This course doesn't assume any specific programming experience, but a basic understanding of concepts like functions and program execution can be helpful. From here, you should feel comfortable diving into other core python language courses on organizing larger programmes, classes and object orientation functions and functional programming, and robust resource and error handling. I hope you'll join me on this journey to learn about the Python programming language with E Corp. I think getting started course at plural site.

Installing and Starting Python

Overview

[Autogenerated] in this module, we'll cover obtaining and installing python on your system for Windows, you boon to Lennox and Mac OS. We'll write some basic python code and become acquainted with the essentials of python programming culture, such as the scent 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 to which is the widely deployed legacy language, and Python three, which is the present and future of the language. It's now over a decade since the transition from python to Do Python three was begun, and we strongly recommend that all new projects are begun with Python three as python to will not be maintained from the year 2020. That said, most of the Python code we will demonstrate will work without modification between the last version of Python, too, which is Python 2.7 and recent versions of Python three, such as Python 3.8. However, there are some key differences, and in a strict sense, the languages are incompatible. We'll be using Python three for this course and everything we show we'll work on Python 3.6 or later, and most things will work on Python 3.3 or later. We're also confident that everything we present will apply to future versions of Python three, so don't be afraid to try those as they become available.

Installing Python

[Autogenerated] Before we can start programming in Python, we need to get hold of a python environment. Python is a highly portable language available on all major operating systems. You will be able to complete this course on Windows, Mac or Lennox, and the only major section where we diverge into platform specifics is coming right up as we install Python three. As we cover the three platforms, feel free to skip over the sections, which aren't relevant for you. Let's see how to install Python three on Windows 10 for Windows. You need to visit the official Python website at python dot org's and then navigate via the downloads tab to the downloads for windows and click the button to begin downloading the latest pipe in three version. When given the option, choose to run the installer after the installer starts. Be sure to enable the option to add pipe into the path environment variable before moving on by clicking install. Now you may be asked to approve the python installer, making changes to your device, which you should accept. After a few seconds, the installer will complete and you can close the installer and your Web browser. We'll be working with python from the command line. So via the start button, choose Windows Power Shell on older versions of Windows. You may need to use the command shell instead and start python just by typing python followed by enter. Welcome to Python. The Triple A row 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 Lennox. Now let's see how to install Python three on Mac OS For Mac OS, you need to visit the official Python website at python dot or GE Navigate via the downloads tab to the downloads for Mac OS and click the button to begin downloading the latest Python three version AP KG 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 three. The Python three version we're installing here. We'll sit alongside your system Python, too, 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 were using Spotlight to do so and run Python three from the command line. Welcome to Python. The Triple Aargh Prompt shows that Python is waiting for your input. The last operating system will look at is Lennox, which is the easiest of all. Recent versions of a boon to Lennox include Python three out of the box, so no installation is required to begin using python. Open a terminal on New Boon to weaken. Do this by using the search function accessible through the you boon to Aiken on the top. Left entering terminal and launching the terminal application in the terminal, you should be able to start Python three. Welcome to Python. The Triple Aargh prompt shows you that python is waiting for your input. If you're using a version of Linux other than new boon to, you'll need to find out how to invoke and if necessary, install pipes and three on your system

Interactive Python

[Autogenerated] Now that python is installed and running, we can immediately start using it interactively. This is a good way to get to know the language as well as a useful tool for experimentation and quick testing. During normal development. This python command line environment is a read evil print loop. Python will read whatever input we type in evaluated, print the result and then loop back to the beginning. You'll often hear it referred to as simply the rebel. When started, the rebel will print some information about the version of Python you're running, and then it will give you a triple arrow prompt. This prompted tells you that Python is waiting for you to type something within an interactive python session. You can enter fragments of python programs and see instant results. Let's start with some simple arithmetic. As you can see, Python reads. Our input evaluates. It prints the result and loops round to do the same Again. We can assign two variables in the rebel, print their contents simply by typing their name and refer to them in expressions within the rebel. You could use the special underscore variable to refer to the most recently printed value this being one of the very few of skier shortcuts in python, or you can use the special underscore variable in an expression. Remember, though, that this useful trick only works at the rebel. The underscored doesn't have any special behavior in python scripts or programs. Notice that not all statements have a return value. When we assigned 52 x, there was no return value on Lee. The side effect of bringing the variable X into being. Other statements have more visible side effects. Try typing print Hello, Python at the Prompt. You'll need parentheses after the print and quotes around the text. Then press enter. You'll see that Python immediately evaluates and executes this command, printing the string Hello Python and returning you to another prompt. It's important to understand that the response here is not the result of the expression being evaluated and displayed by the ripple. Rather, it is a side effect of the print function. As an aside, print is one of the biggest differences between python, too, and Python three in Python three. The parentheses are required, whereas in python to they are not. This is because in Python three print is a function call \_\_\_\_\_ functions later. At this point, we should show you how to exit the rebel 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 Control Z followed by enter to exit. If you're on Mac or Lennix, press control D to exit. If you regularly switch between platforms and you accidentally press control Z on a UNIX like system, you will inadvertently suspended the Python interpreter and return to your operating system shell to reactivate python by making it a foreground process again. Simply run the F G command and press. Enter a couple of times to get the triple. Aargh Python prompt back.

Significant Whitespace

[Autogenerated] Now that you have a working python Ruppel, let's look at some basic code structure. Start your python three interpreter using the python or Python three command for Windows or UNIX like systems, respectively, the control flow structures of python, such as four 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, four loops require a body, so if you enter four I in range five python will change the prompt to three dots to request you provide the body. One distinctive and sometimes controversial aspect of python is that leading white space is sin tactically 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 intended by four spaces for each level, so we provide those four spaces and a statement to form the body of the loop. Our loop body will contain a second statement. So after pressing return and getting another three dot prompt will enter another four spaces, followed by a call to the built in print function determinate R block, we must enter a blank line into the rebel. With the block complete. Python executes the pending code, printing out the multiples of 10 less than 50. Looking at a screen full of python code, we can see how the indentation clearly matches and in fact, must match the structure of the program. Even if we replace the code by gray lines, the structure of the program is clear. Each statement terminated by a colon starts a new block and introduces an additional level of indentation, which continues until the D Dent restores the indentation toe a previous level. Each level of intent is typically four spaces, although we'll cover the rules in more detail in a moment. Pythons approach to significant white space has three great advantages. First, it forces developers to use a single level of indentation in a code block. This is generally considered good practice in any language because it makes code much more readable. Second code with significant white space 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 and python code are easily identifiable and everyone writes them the same way. Third, significant white space requires that a consistent interpretation must be given to the structure of the code by the author, the python runtime system and future maintainers who need to read the code. As a result, you could never have code that contains a block from pythons point of view, but which doesn't look like it From a cursory human perspective, the rules for pipe an indentation can seem complex but are straightforward. In practice, The white space you use convey either tabs or spaces. The general consensus is that spaces are preferable to tabs, and four spaces has become a standard in the pipes in community. One essential rule is never to mix spaces and tabs. The python interpreter will complain, and your colleagues will hunt you down. You are allowed to use different amounts of indentation at different times, if you wish. The essential rule is that consecutive lines of code at the same indentation level are considered to be part of the same code block. There are some exceptions to these rules, but they almost always have to do with improving code readability in other ways. For example, by breaking up necessarily long statements over multiple lines. This rigorous approach to code formatting is programming as \_\_\_\_\_ intended it or indented it. This philosophy of placing a high value on code qualities such as readability gets to the very heart of python culture something we'll take a short break to explore now.

Python Culture

[Autogenerated] 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 Pepes. One of the Pepes, called Pep eight, explains how you should form at your code, and we follow its guidelines. Throughout this course. It is Pep eight, which recommends we use four spaces for indentation in new python code. Another of these peps, called Pep 20 is called the Zen of Python. It refers to 20 aphorisms describing the guiding principles of python, only 19 of which have been written down. Conveniently. The Zen of Python is never further away than the nearest python interpreter, as it can always be accessed from the rubble by typing import. This throughout this course will be highlighting particular nuggets of wisdom from the set of python in moments of Zen. To understand how the apply to what we have learned as we've just introduced Python significant indentation, this is a good time for our first moment of Zen readability counts, clarity matters, so readability makes for valuable code in time, you'll come to appreciate pythons, significant white space for the elegance it brings to your code and the ease with which you can read others.

The Python Standard Library

[Autogenerated] as mentioned earlier. Python comes with an extensive standard library, an aspect of python often refer to as batteries included. The standard library is structured as modules. A topic will discuss in depth later in this course. What's important at this stage is to know that you gain access to standard library modules by using the import keyword. The basic form of importing a module is simply the import keyword, followed by a space and the name of the module. For example, let's see how we can use the standard libraries math module to compute square roots at the Triple Arrow prompt we type import math. Since import is a statement which doesn't return a value, Python doesn't print anything. If the import succeeds and were immediately returned to the prompt. We can access the contents of the imported module by using the name of the module, followed by a dot, followed by the name of the attributes in the module that you need. Like many object oriented language is, the dot operator is used to drill down into objects structures. Being expert Python Isa's. We have inside knowledge that the math module contains a function called sq Artie. Let's try to use it, But how can we find out what other functions are available in the math module? The rebel has a special function help, which can retrieve any embedded documentation from objects for which it has been provided. Such a standard library modules to get help, simply type help. We'll leave you to explore the first form for interactive help on your own time. We'll go for the second option and passed the math module as the object for which we want to help. You can use the space bar to page through the help. If you're on Mac or Lennox, use the arrow keys to scroll up and down. Browsing through the functions, we can see that there's a math function for computing. Factorial Sze Press cue to exit the help browser and return us to the python Ripple Practice using help to request specific help on the factorial function, press Q. To return to the rebel, let's use the factorial function, which accepts an integer and returns an integer. Notice how we need to qualify the function name with the name of the module containing it. This is generally good practice, as it makes it abundantly clear where the function is coming from. That said, it can result in code that is excessively verbose. To see that let's use factorial to compute how many ways there are to draw three fruit from a set of Fife ERT. Using some math we learned in school, this simple expression is quite verbose. With all those references to the math module, the Python import statement has an alternative form that allows us to bring a specific function from a module into the current name space. This is a good improvement, but it's still a little long winded for such a simple expression. 1/3 form of the import statement allows us to rename the imported function. This could be useful for reasons of readability were to avoid a name space clash. Useful as it is, we recommend this feature be used infrequently and judiciously. Remember that when we used factorial alone, it returned an integer but are more complex expression for combinations is returning a floating point number. This is because we've used pythons floating point division operator, the single forward slash We can improve our expression since we know it will only ever return into real results by using pythons Integer division operator, which is a double forward slash. What's notable is that many other programming languages would fail on the above expression for even moderate values of end in most programming languages. Regular garden variety signed integers can only store values less than two to the 31st power. However, factorial grow so fast that the largest factorial you can fit into a 32 bit signed imager 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 fruit from a set of 13 fruits. Python encounters no such problems and can compute with arbitrarily large integers limited only by the memory in your computer. Let's try the larger problem of computing how many different pairs of fruit we can pick from 100 different fruits, assuming we can lay our hands on so much fruit just to emphasize how large the size of the first term in that expression is, calculate 100 factorial on its own. This is a number of vastly larger than even the number of atoms in the known universe with an awful lot of digits. If, like me, you're curious to know exactly how many digits we can convert our interview to a text a ring and count the number of characters in it like this.

Summary

[Autogenerated] congratulations. You've taken your first steps in python, and you're well on your way to reading and writing much more sophisticated python programs. In this module, we saw how to download and install Python on Windows, Lennox and Mac OS. We covered starting your python rebel. We evaluated some simple expressions in the rebel. We learned that in the rebel, the underscore symbol is bound to the results of the last evaluated expression. We saw how to make basic use of the print function, and we learned that the printed output is a side effect of the function, not a return value. We saw how to exit the rebel using Control Z on Windows and Control D on the Knicks and Mac OS. We were introduced to Python's use of significant white space. We learned that code blocks in python are initiated with the colon and comprised consecutive lines at the same indentation level. We looked at some of the advantages of significant white space, including clarity and consistency. We covered the basic rules were indentation in python. On a less technical level, we covered some parts of pythons culture. We looked at the Zen of python and saw that it could be printed by executing Import this in the Rebel, and we looked specifically at the idea that readability counts when writing python code. We covered the basics of importing modules from the Standard Library, and we saw three forms of the import statement importing an entire module, importing selected elements of a module and renaming imported elements. We saw how to use pythons help system. Along the way, we saw how to use the factorial function from Pythons Standard Math Library In the next module of core python getting started, we'll look at pythons, fundamental scaler types, integers, floats, nuns and bulls as well as some basic flow control constructs.

Scalar Types, Operators, and Control Flow

Overview

[Autogenerated] Now that you've got a functional python ripple on your system, we could start to work with the fundamentals of the language. In this module of core python getting started, you'll start to learn about pythons. Fundamental scaler types. Well, look at basic use of relational operators, and we'll introduce basic flow control mechanisms. Python comes with a number of built in data types. These include primitive scaler types like integers, as well as collection types like dictionaries. Thes built in types are powerful enough to be used alone for many programming needs, and they could be used as building blocks for creating more complex data types. In this section, we cover the basics scale. ER's in't for whole numbers, float for numbers with fractional parts none. An important placeholder, value and bull 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 half for all practical purposes unlimited precision, meaning they can contain as many digits as you need. Integer, liberals and python are specified in decimal and may also be specified in binary with E zero B prefix in Octel with e zero prefix or index a decimal with the zero X prefix. We can also constructing teachers by a call to the into constructor. This can convert from other numeric types such as floats to integers. Note that the rounding of integers is always towards zero. Thean constructor can also convert strings to integers. You can even supply an optional number base when converting from a string. For example, to convert from base, three provide the value. Three As the second argument to the into constructor, floating point numbers are supported and python by the float type Python floats are implemented as IEEE 754 double precision floating point numbers with 53 bits of binary precision. This is equivalent to between 15 and 16 significant digits and decimal. Any literal number containing a decimal point is interpreted by python as a float. Scientific notation can be used so for large numbers such as the approximate speed of light and meters per 2nd 3 times tended the eighth weaken right three e eight, and for small numbers like planks, constant 1.616 times 10 to the negative 35th. We can enter 1.616 e minus 35 Notice how python automatically switches the display representation. That is the format it prints to the rebel to the most readable form. As with integers, we can convert to floats from other numerical string types. Using the float constructor, we can pass into values to the float constructor, and we can pass strings. This is also how we create the special floating point values nan or not a number as well as positive infinity and negative infinity. One important rule to remember is that the result of any calculation involving into in float is promoted to afloat. You can read more about pythons number types in pythons. Documentation. Python has a special no value called none. Spelled with a capital end, none is frequently used to represent the absence of a value. The python rebel never prints. None results, so typing none into the rebel has no effect. None could be bounded to variable names just like any other object, and we contest whether an object is none by using pythons is operator. We can see here that the result of the is operator in this case is true. Which brings us conveniently onto the bull type. The bull type represents logical states and plays an important role in several of pythons control flow structures as well. See, shortly, as you probably expect, there are two bull values true in false, both spelled with initial capitals. There is also a bull constructor which could be used to convert from other types, too. Bull. Let's look at how it works. Foragers zero is considered false e and all other values. Truth e We see the same behavior with floats where only zero is considered false. E When converting from collections such as strings or lists on lee, empty collections are treated as false E for lists which will look at shortly. The empty list is false. E While any non empty list is truth E. Similarly empty strings are false e while any other strings are truth e, it's worth noting that the bull constructor may not behave as you expect when passing in the strings. True and false, since both are non empty strings. Both result in true thes conversions to bull are important because they're widely used in python. If statements and wild loops which except bull values into the condition, we look at these constructs soon

Relational Operators

[Autogenerated] bull values are commonly produced by pythons relational operators, which could be used for comparing objects. These include value equality or equivalents, value inequality or, in equivalence, less than greater than less than or equal to and greater than or equal to. Two of the most widely used relational operators are pythons. Equality and inequality tests. These tests, whether to objects, are equivalent or in equivalent. That is whether one could be used in place of the other or not. We'll learn more about the notion of object equivalents later in the course, but for now we'll just compare simple integers. Let's start by assigning or binding a value to the variable G. We test for equality with the double equals operator, and we test for inequality using the not equals operator an exclamation point followed by an equal sign. We can also compared the order of quantities using the rich comparison operators we check of. One object is less than another. With the less than operator, we can depend on equal sign to this operator to test for less than or equal. Likewise, we check if an object is greater than another with the greater than operator and as with less than there is the related greater than or equal operator

Control Flow

[Autogenerated] Now that we've examined some basic built in types, we look at two important control flow structures which depend on the conversions to the bull type. If statements and wild 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 cooling to introduce a new block. Let's try this at the rebel remembering to end in four spaces within the block, we add some code to be executed. If the condition is true, we terminate the blocked by entering a blank line because the condition is self evidently true. The block executes and we see the string. It's true, printed to the rebel. Conversely, if the condition is false, the code on the block does not execute the expression used with Theis. Statement will be converted to a bull, just as if the bull constructor had been used so explicitly. Constructing the bull in the s statement is exactly equivalent to using a bear string. Thanks to this useful shorthand explicit conversion to bull using the bull constructor is rarely used in python. The if statement supports the optional else Claus that goes in a block introduced by the else keyword, followed by colon, which is indented to the same level as the if keyword to start the else block. In this case, we just omit the indentation after the three dots for multiple conditions, you might be tempted to nest if statements inside else blocks like this. Whenever you find yourself doing this, though, you should consider using pythons LF key word, which is a combined else. If, as the Zen of Python reminds us Flat is better than nested, this version is altogether easier to read.

While-loops

[Autogenerated] Python has two types of loop four loops and wild loops. We've already briefly introduced four loops back when we introduce significant white space and will return to them soon. But right now we'll cover four lives, while loops and python are introduced by the wild keyword, which is followed by a Boolean expression. As with the condition for if statements, the expression is implicitly converted to a Boolean value, as if it had been passed to the bull constructor. The wild statement is terminated by a cold in because it introduces a new block. Let's write a loop at the rebel, which counts down from 5 to 1. We'll initialize a counter variable called C 25 and keep looping until we reach zero. Another new language feature here is the use of the augmented assignment operator, a minus sign followed by an equal sign to subtract one from the value of C on each generation. Similar augmented assignment operators exist for other basic math operators, such as plus and multiply, because the condition, also called the predicate, will be implicitly converted to bull. Just as if a call to the bull constructor present, we could replace the above code with the following version. This works because the conversion of the integer value of C two bull results in true until we get to zero, which converts to false. That said to use this short form in this case might be described as unpaid thon IQ because referring back to the Zen of Python explicit is better than implicit, and we place a higher value on the readability of the first form over the concession at the second form. While loops are often used in python, where an infinite loop is required, we achieved this by simply passing true as the predicate expression to the wild construct. Now you're probably wondering how to get out of this loop and regain control of your rebel. To do this, we press control see Python intercepts is to raise a special exception which terminates the loop. We'll be talking much more about what exceptions are and how to use them later. In the course. Many programming languages support a loop construct which places the predicate test at the end of the loop rather than at the beginning. For example, C C plus plus C Sharp and Java support the do wild 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 brake statement jumps out of the loop and only the innermost loop. If several loops have invested and then continues execution immediately after the loop body, let's look at an example of break introducing a few other python features along the way, we start with a wild true for an infinite loop. On the first statement of the wild block, we used the built in input function to request a string from the user. We assigned that string to a variable called response. We now use an if statement to test whether the value provided is divisible by seven. We convert response to an integer, using the into constructor and then use the module is operator the percent simple 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 F block within the F block. Now two levels of indentation deep. We start with eight spaces and use the brake keyword break terminates the innermost loop in this case the wild loop and causes execution to jump to the first statement after the lube. In our case, this is the end of the program. Enter a blank line that the three dots prompted to close both e. If block and the wild block, our loop will start executing and will pause it. The call to the input function waiting for us to enter a number. Let's try a few. As soon as we enter a number divisible by seven, the predicate becomes true. We enter the F block and then literally break out of the loop, ending the program and returning us to the rebel prompt.

Summary

[Autogenerated] We've covered quite a bit of ground in this module, so let's summarize what we've seen. We looked at the four built in scaler types into float none in bull and conversions between these types and use of their literal forms, we looked at six relational operators used for equivalents and ordering. We demonstrated, structuring conditional code with the if, LF LC structures. We showed iterating with wild loops and we saw how the condition expression of the wild is implicitly converted to a bull. We looked at how to intercept infinite loops with control, See, and we learned that doing this generates a keyboard interrupt exception. We gave an example of how to break out of loops. Using the break statement, we observed that break on Lee breaks us out of the innermost nested loop and that it takes us to the first statement immediately following the loop. Along the way, we looked at the augmented assignment operators for modifying objects such as counter variables in place. We also looked at requesting text from the user with the built in input function in the next module of core python getting started. We'll continue our exploration of pythons built in types and control flow structures by looking at strings lists, dictionaries and four loops, we'll even be using python to fetch some data from the Web for processing.

Introducing Strings, Collections, and Iteration

Overview

[Autogenerated] Now that you have an initial understanding of pythons fundamental scaler types, you're ready to start exploring some of the collection types in this module of core python getting started. We'll look at a few of the most important collection types in python Stra Bites List and dicked. We'll introduce you to the Four Loop, a looping construct commonly used for iterating over collections. We'll apply all of this to build a small but useful program that demonstrates the expressiveness of python. Python includes a rich selection of collection types, which are often completely sufficient for even quite intricate programs. Without resorting to defining your own data structures, we'll give enough of an overview of some fundamental collection types now to allow us to write some interesting code. We'll also be revisiting each of these collection types together with a few additional ones later in the course. Let's start with these types \_\_\_\_\_\_ bites list and dicked along the way. We'll also cover pythons four loops

String

[Autogenerated] strings and python have the data type stra spelled S t R, and we've been using them extensively already. Strings are sequences of Unicode code points, and for the most part, you can think of code points as being like characters, Although they are not strictly equivalent. The sequence of characters and a python string is immutable, meaning that once you've constructed a string, you can't modify its contents. Literal strings and python order limited by quotes, you could use single quotes or double quotes. You must, however, be consistent. For example, you can't use single quotes on one side and double on the other like this. Supporting both quoting styles allows you to easily incorporate the other quote character into the literal string without resorting to ugly escape character gymnastics. I noticed that the rebel exploits the same, quoting flexibility when echoing the strings back to us. Beautiful text strings rendered in literal form, simple elegance at first sight. Support for both quoting styles seems to violate an important principle of pipe phonics 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 to quoting Stiles is valued more highly than the alternative, a single quoting style combined with more frequent use of ugly escaped sequences, which will encounter shortly.

String Literals

[Autogenerated] adjacent literal strings are contaminated by the python compiler into a single string, which, although it first it seems rather pointless, can be useful for a nicely for writing code. As we'll see later. If you want a literal string containing new lines, you have two options. Use multi line strings or use escape sequences. First, let's look at multi line strings. Multi line strings aren't a limited by three quote characters rather than one. Here's an example using three double quotes. Notice how, when the string is echoed back to us, the new lines are represented by the backslash in escape sequence. We can also use three single quotes as an alternative to using multi line quoting weaken just in bed. The control characters ourselves To get a better sense of what we're representing. We can use print to see the string. If you're working on Windows, you might be thinking that new lines should be represented by the carriage return and New line couple it back slash our backslash end. There's no need to do that with Python, since Python three has a feature called Universal New Line Support, which translates from the simple backslash end to the native New line sequence for your platform on input and output, you can read more about universal New Line support in Pep to 78 We can use the escape sequences for other purposes, too, such as incorporating tabs with backslash tea or allowing us to quote characters within strings by using backslash. Double quote war backslash. Single quote. See how python is smarter than we are using the most convenient quota limiters, although Python will also resort to escape sequence is when we use both types of quotes in a string. Because backslash has special meaning to place a backslash industry, we escaped 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 dot or GE sometimes, particularly when dealing with strings such as windows file system paths or regular expression patterns, which use backslash, is extensively. The requirement to double up on back slashes could be ugly and error prone python comes to the rescue with its raw strings. Raw strings don't support any escape sequences and are very much What you see is what you get to create a raw string prefix. The opening quote With a lower case R we can use the string constructor to create string representations of other types, such as integers or floats. Strings and 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 programming languages, there is no separate character type distinct from the string type. The indexing operation We just use returns, a full blown string that contains a single character element, something we contest using pythons built in type function. There will be more on types and classes later in the course. String objects also support a wide variety of operations implemented as methods. We can list those methods using help on the string type. Ignore all the higher Olympics with underscores for now and page down until you see the documentation for the capitalized method. Press cute to quit the help rouser and will try to use that method first. Let's make a string that deserves capitalization. The proper noun of the capital city no less to call methods on objects. In python, we use the dot after the object name and before the method Name methods are functions, so we must use the parentheses to indicate that the method should be called. Remember that strings are immutable, so the capitalized method didn't modify. See in place. Rather, it returned a new string. We can verify this by displaying see which remains unchanged. You might like to spend a little time familiarizing yourself with the various useful methods provided by the string type. Finally, because strings are fully Unicode capable, we can use them with international characters easily, even in literal Sze. Because the default source code encoding for pipe and three is utf eight, for example, if you have access to the region characters, you can simply enter this. Alternatively, you can use the Hexi decimal representations of Unicode code points as an escape sequence prefixed by backslash you, which I'm sure you'll agree, is somewhat more unwieldy. Similarly, you can use the backslash X escape sequence, followed by a two character Hexi Decimal string or an escaped Octel string to include Unicode characters in a string literal. There are no such Unicode capabilities in the otherwise similar bites type, which will look at next

Bytes

[Autogenerated] bites are very similar to strings, except that rather than being sequences of unit code code points there sequences of well bites. As such, they're used for raw binary data and fixed with a single bite character and coatings such as asking. As with strings, they have a simple, literal form using quotes, the first of which is prefixed by a lower case, be. There was also a bites 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 bites liberals and understand that they support most of the same operations as string, such as indexing, which returns the integer value of the specified bite and splitting, which you'll see returns a list of bites objects to convert between bites and strings. We must know the encoding of the bite sequence used to represent the strings. Unicode code points as bites. Python supports a wide variety of in codings, a full list of which can be found at python dot warg. Let's start with an interesting Unicode string, which contains all the characters of the 29 letter Norwegian alphabet, a pan graham Well, now, in code that using UTF eight into a bites object, see how the Norwegian characters have each been rendered as pairs of bites. We can reverse that process using the decode method of the bites object. Again, we must supply the correct and coding. We can check that the result is equal to what we started with and display it for good measure. This may seem like an unnecessary detail so early in the course, especially if you operate in an anglophone environment. But it's a crucial point to understand. Since files and network resource is such as http, responses are transmitted as bite streams where, as we often prefer to work with the convenience of Unicode strings.

List

[Autogenerated] python lists, such as those returned by the string 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 upended lists or a workhorse of python data structures. Literalists 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 zero based index, and we can replace elements by assigning to a specific element. See how lists could be heterogeneous. With respect to the types of the objects, we now have a list containing a string manager and another string. It's often useful to create an empty list, which we can do using empty square brackets. We could modify the list in other ways. Let's add some floats to the end of the list. Using the upend method. There are many other useful methods for manipulating lists, which will cover in a later module. There's also a list constructor, which could be used to create lists from other collections. Such a strings. Finally, although the significant white space rules and 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 unclos dhe, you can continue on the next line. This could be very useful for long, literal collections or simply to improve readability. See also how we're allowed to use an additional comma after the last element. This is an important maintain ability feature.

Dict

[Autogenerated] dictionaries are completely fundamental to the way the python language works and are very widely used. A dictionary maps, keys to values and in other languages, is known as a map or an associative array. Let's look at how to create and use them. In python, literal dictionaries are created using curly braces containing key value pairs. Each parece 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 could 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 assigned to a key that has not yet been added, a new entry is created. Be aware that in python versions prior to 3.7, the entries in the dictionary can't be relied upon to be stored in any particular order. As of Python 3.7, however, entries are required to be kept in insertion order. Similarly, two lists empty dictionaries can be created using empty curly braces. We'll revisit dictionaries in much more detail in a later module

For-loop

[Autogenerated] now that we have the tools to make some interesting data structures. Well, look att pythons, second type of loop construct The four loop four loops and 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 unendurable Siri's, but more on that later and assign them in turn toe a variable that we specify. Let's create a collection and use a four loop to generate over it. If you reiterate over dictionaries, you get the keys, which you can then use within the four loop body to retrieve values. Here we define a dictionary mapping string color names to Hexi decimal integer color codes. Note that we used the ability of the built in print function to accept multiple arguments. We passed the key and the value for each color separately. See also how the color codes return to us are in decimal

Putting it all Together

[Autogenerated] in this last section. Before we summarize, we're going to write a longer snippet at the rebel. We're going to fetch some text data for some classic literature from the Web using a Python standard library function called U R L Open. To get access to your L Open, we need to import the function from the request module within the standard library. You are l Lib Package. Next we're going to call you Are l open with a Eurail to our story, then create an empty list which ultimately will hold all of the words from the text. Next, we opened a four loop which will work through the story. Recall that four loops request items one by one from the term on the right of the in key word 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 four loop retrieves one line of text a time from Dickens Classic Note also that the four statement is terminated by a colon because it introduces the body of the four 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 white space boundaries, resulting in a list of words we call line words. Now we use a 2nd 4 loop nested inside the 1st 2 literate over this list of words a pending each in turn to the accumulating story words list. Now we enter a blank line at the three dots prompt to close all open blocks. In this case, the Inter four loop in the outer four loop will both be terminated. The block will be executed, and after a short delay, python now returns us to the regular triple arrow prompt. At this point, if python gives you an error such as a syntax error or indentation error, you should go back, review what you entered and carefully re enter the code until Python accepts the whole block without complaint. If you get an http error than you were unable to fetch the resource over the Internet, and you should try again later, although it's worth checking that you type the U R L correctly. Finally, now that we're done reading from the girl, we need to close our handle to it story. We can look at the words we read simply by asking Python to evaluate the value of story words. Here we can see the list of words notice that each of the single quoted words is prefixed by a lower case letter B, meaning that we have a list of bites objects where we would have preferred a list of strings. This is because the http request transferred. Raul Bites tow us over the network to get a list of strings. We should decode the bites, dreaming each line into Unicode strings. We can do this by inserting a call to the decode method of the bites object and then operating on the resulting Unicode string. The python Ruppel supports a simple command line history and by careful use of the up and down arrow keys, we can re enter our sniff it. When we get to the line which needs to be changed, we can edit it using the left and right arrow keys to insert the requisite call to decode. Then when we rerun the block and take a fresh look at story words we should see. We have a list of strings we've just about reached the limits of what's possible to comfortably edit at the Python rebel. So in the next course module, we'll look at how to move this coat into a python module, where it can be more easily worked with in a text editor.

Summary

[Autogenerated] There are a lot of details in this module, which may be difficult to remember all at once, but which you'll find you use very frequently when writing python code. First we looked at strings in particular the various forms of quoting for single and multi line strings. We saw how adjacent string liberals are implicitly contaminated. Python has support for universal new lines. So no matter what platform you're using, it's sufficient to use a single backslash in character, safe in the knowledge that it will be appropriately translated from and to the native new line during Io. Escape sequences provide an alternative means of incorporating new lines and other control characters into literal strings. The backslash is used for escaping can be a hindrance for Windows file system paths or regular expressions, so raw strings with E. R prefix can be used to suppress the escaping mechanism. Other types, such as integers, can be converted to strings using the struck instructor. Individual characters returned as one character strings can be retrieved using square brackets with injured your zero based indices. Strings support a rich variety of operations such a splitting through their methods in python, three literal strings can contain Unicode characters directly in the source. The bites type has many of the capabilities of strings, but it is a sequence of bites rather than a sequence of Unicode. Code points bites literal Czar prefixed with a lower case be to convert between string and bites instances. We use Thean code, Methodist \_\_\_\_\_\_ and the decode method of bites. In both cases, passing the encoding, which we must know in advance lists are immutable Hedorah genius sequences of objects list literal czar 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 a pending to them, and they could 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. Four loops take items one by one from unutterable object, such as a list, and binds a name to the current item. They correspond to what are called for each loops in other languages. In the next module of core python getting started, we'll look at functions and modules. Pythons, fundamental tools for organizing code. These tools will facilitate writing larger programs and structure in your code into cohesive, reusable components.

Modularity

Overview

[Autogenerated] modularity is an important property for anything but trivial software systems as it gives us the power to make self contained reusable pieces, which can be combined in new ways to solve different problems. In this module of core python getting started, we'll see that in python, as with most programming, language is the most fine grained modular ization facility is the definition of reasonable functions. We'll see how collections of related functions are typically grouped into source code files called modules, and we'll learn how modules can be used from other modules. So long as we take care not to introduce circular dependencies, we'll learn more about a topic We've already seen importing modules into the rebel. We'll show you how modules could be executed directly as programs or scripts. Along the way, we'll investigate the Python execution model to assure you have a good understanding of exactly when code is evaluated and executed. We'll round off by showing you how to use command line arguments to get basic configuration ditty into your program and make your program execute herbal. To illustrate this module, we'll be taking the code snippet for retrieving words from a Web host to text document that we developed at the end of the previous module and organizing into a fully fledged python module

Modules

[Autogenerated] Let's start with the snippet we worked with less 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 eight encoding, as that's what the Python three runtime expects by default. Let's get this snippet we wrote at the Rebel at the end of the previous module into a text file called words dot p y all Python source files used the dot p y extension. Now that we're using a text file for our code, we could pay a little more attention to readability. Let's put a blank line after the import statement. Save the file in the directory called Core Pie in your Home Directory. Switch to a console with your operating systems. Shell prompt and changed the new Corp I directory. We can execute our modules simply by calling Python and passing the modules. File name. Use Python three words dot p y on Mac or Lennix or python Words stop e y on windows when you press return. After a short delay, you'll be returned to the system problems. Not very impressive, but if you got no response than the program is running as expected, if, on the other hand, you got an error, an http error indicates there's a network problem. Whilst other types of errors probably mean you have mistyped the code, let's add another four loop to the end of the program to print out one word per line. If you run your module again, it will print out the downloaded words. This is much better now. We have the beginnings of a useful program. Our module can also be imported into the rebel. Let's try that and see what happens. Start the rebel and import your module by typing import words. Note how, when importing we amid the file extension, the code on your module is executed immediately when imported. Maybe not what you expected and not very useful to give us more control over when our code is executed and to allow it to be reused. We'll need to put our code in a function

Functions

[Autogenerated] let's quickly to find a new function at the rebel. To get the idea. Functions are defined using the deaf keyword followed by the function. Name an argument list in parenthesis and a colon to start a new block. The code inside the function block must be indented. We use the return key word to return a value from the function here, returning an expression which evaluates to X squared as we've seen previously. We call functions by providing the actual arguments in parenthesis after the function. Name functions aren't required to explicitly return a value, though perhaps they produce side effects such as this. Launch missiles function here. The message we see printed is a side effect of calling print, not a return value from the function. Generally speaking, it's good practice to prefer functions which returned values rather than cause effects, especially when the effects are as drastic as this. You can return early for a function by using the return key word with no parameter. Both the return statement without a parameter as well as the implicit return at the end of a function, actually causes the function to return None. Remember, however, that the rebel doesn't display none results, so we don't see them by capturing the returned object into a named variable. We contest for none to reinforce these important ideas. Let's look at another example. Here's a function and through which could be used to compute the square root cube roots, fourth roots and so on of a number we supply called the Radic and internally uses Thean Fix. Raised to the power or exponentially ation operator, which in python is the double star. Let's use an through to compute the square root of 16 giving four and again to compute the cube root of 27 which is three. See how the calls to the function evaluate to the value of the returned expression. Our in through it function has only one point of return. Now consider our orginal suffix function. Organelles are numbers like 1st 2nd and third, as opposed to cardinal numbers like 12 and three. The orginal Suffolk's function is used to determine the Suffolk's S t A, N, D. R D, and so on, which are used for orginal numbers in English. The function uses an if LF structure to check the trailing digits of the string representation of the value it has passed and contains seven points of return. Now comes the orginal function. This takes a value and can captain eights. It's string representation. With its orginal Suffolk CE. It's common and good practice to compose more complex functions out of more fundamental functions. In this way, function composition is the crucial technique for controlling complexity and programs. Lastly, we have the display and through function, given some Radic and and end it delegates to RN through function to do the mathematics, then performs the effect of printing a message like the second route of 16 is four delegating toe orginal and then onto orginal suffix as necessary. This function has no explicit return statement at the end, so we can consider it to finish with an implicit return. None. Let's try it at the rebel, using it to display and compute the 43 to 64 Before we go on, we need to take a quick aside regarding terminology. In python, many language features are implemented or controlled, using specially named objects and functions. Thes special names generally have two leading and two trailing underscores. This has the benefit of making them visually distinct fairly easy to remember and unlikely to collide with other names. This game has the disadvantage, however, of making these names difficult to pronounce a problem we face when making courses like this. To resolve this issue, we have chosen to use the term dunder when pronouncing these names. Thunder is a portmanteau of the term double underscore and will use it to refer to any name with leading and trailing. Double underscores. So, for example, when we talk about underscore underscore name underscore, underscore something you'll soon encounter. Well, say Dunder name. These kinds of names play a big role in Python, so we'll be using this convention frequently.

\_\_name\_\_

[Autogenerated] with some background on functions in place and the terminal logical exactitude established. Let's get back to organizing the code in our words module into function. We'll move all the code except the import statement, into a function called fetch words. You do that simply by adding the death statement and inventing the code below it by one extra level. Save the module. Start afresh python ripple and import your module again with import words. The module imports, but now the words are not fetched until we call the fetch. Words function with words dot fetch words. This use of the DOT is called qualifying the function name with the module name. Alternatively, we could import are specific function using a different form of the import state from words import fetch words. Having imported the fetch words function directly into our rep recession, which is itself a module we can invoke fetch words using its unqualified name. So far, so good. But what happens when we try to run our module directly from the operating system? Shell exit from the rubble with control D from Mac or clinics or control Z for windows and run python? Passing the module file name, no words are printed. That's because all the module does now is defined a function and then exit. The function is never called. What we'd prefer is that the module actually print something when we executed to make a module from which we can useful the import functions into the rebel and which can be run as a script, we need to learn a new python idiom. As you mentioned earlier, One of the specially named Variables is called Dunder Name, and it gets us the means to detect whether our module has been run. It's a script more imported into another module or the rebel to see how ad print Dunder name at the end of your module. Outside of the fetch words function, let's import the modified words module back into the rebel with import words. We can see that when imported Dunder name does indeed evaluate to the modules name. As a brief aside, if you import the module again in the same ripple, the print statement will not be executed. Module code is only executed once on first import. Now let's try running the module as a script from the operating system shell with python three words dot p y. Now the special Dunder name variable is equal to the string. Dunder Maine, which is also delimited by double, underscores, that is, python sets the value of Dunder name differently, depending on how our module is being used. The key idea we're introducing here is that our module can use this behavior to decide how it should behave. We replaced the print statement with an if statement, which tests the value of Dunder name. If Dunder name is equal to the string done, Jermaine, we execute our function. On the other hand, if Dunder name is not equal to Dunder Main, the module knows it's being imported into another module not executed, and so only defines the fetch words function without executing it. We can now safely in port, are module without undue the executing our function, and we can usefully run our module as a script

The Python Execution Model

[Autogenerated] it's important to understand the Python execution model and precisely when function definitions and other important events occur when a module is imported or executed. Here we show execution of our Python module as it's imported in a graphical debugging environment, we stepped through the top level statements in the module. What's important to realize here is that the death used for the fetch words function isn't merely a declaration. It's actually a statement which, when executed in sequence with the other top level model scope code, causes the code within the function to be bound to the name of the function. When modules are imported or run, all of the top level statements are run. And this is the means by which the function within the module name space are defined where sometimes asked about the difference between python modules, python scripts and python programs, any dot P Y file constitutes a python module. But as we've seen, modules could be written for convenient import, convenient execution or using the if dunder name equals dunder main idiom. Both we strongly recommend making even simple scripts and portable, since it eases development and testing so much. If you can access your code from within the rebel. Likewise, even modules, which are only ever meant to be imported in production settings benefit from having executed all test code. For this reason, near the all modules we create have this form of defining one or Maurin portable functions with a postscript to facilitate execution. Whether you consider a model to be a Python script or Python program is a matter of context and usage. It's certainly rolling to consider Python to be merely a scripting tool in the vein of Windows Batch files or UNIX Shell scripts, as many large and complex applications are built exclusively with python.

Command Line Arguments

[Autogenerated] let's refine our word fetching module a little further. First, we'll perform a small re factoring and separate the word retrieval in collection on the one hand from the printing of words on the other. This is because when importing, we'd rather get the words as a list. But when running directly, we prefer the words to be printed. Here we move the printing coat into a new function called print words. We'll also modify our main block to first Call, fetch words and then call print words with the list of words returned to buy. Fetch words. Next, we'll extract the contents of our main block into a function called Maine. By moving this code into a function, we contest it from the rebel something which isn't possible while it's in the module scope. If block we can now try these functions from the rebel will 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 air optional, but they do allow you to break this list over multiple lines if it gets too long. This form is perhaps the most widely used form of the import statement. The second new form imports everything from a module using an asterisk wild card. This latter form is recommended on Lee for casual use at the rebel. It could wreak havoc in programs, since what is imported is now potentially beyond your control, Opening yourself up to potential names space clashes at some future time. Having done this, we can get the words from the U. R L by calling fetch words. We can also print any list of words. Buy calling print words. And indeed, we can run the main program. Notice that the print words function isn't fussy about the types of the items in the list. It's perfectly happy to print a list of numbers. So perhaps print words isn't the best name. In fact, the function doesn't mention lists either. It will happily print any collection that the four loop is capable of iterating over, such as a string. So let's perform another minor re factoring and rename this function to print items, changing the variable names within the function to suit. We'll talk more about dynamic typing in python, which allows this degree of flexibility in the next module. Another obvious improvement tri module would be to replace the hard coded URL with the value we could pass in. Let's extract that value into an argument of the fetch words function. Now, when running our module is a standalone program, we'll need to accept the euro as a command line argument. Access to command line arguments in Python is through the attribute of the cyst module called RG, which is a list of strings to use it. We must first import this this module at the top of our program. Then we need to get the second argument with an index of one from the list. Now, when we run the program, it works as expected. This looks fine until we realize that we can't usefully test main any longer from the rebel because it refers to assist our RGB someone which is unlikely to have any useful value in that environment. The solution is to allow the argument list to be passed as a formal argument to the main function, using cyst RV as the actual parameter in the if dunder name equals under main block testing from the rebel again, we could now past any \_\_\_\_ we want into Maine for more sophisticated command line processing, we recommend you look at the Python Standard Library. Argh! Parse module or the inspired third party Doc Opt module.

Moment of Zen

[Autogenerated] you'll notice that our top level functions have to blank lines between them. This is conventional for modern python code to between functions. That is the number of lines Pepe recommends. According to the Pep Aid style guide, it's customary to use to blank lines between module level functions. We find this convention has served us well, making code easier to navigate. We use single blank lines were logical breaks within functions.

Docstrings

[Autogenerated] we saw previously how it was possible to Ascot the rebel for help on python functions. Let's look at how to add this self documenting capability to our own module. AP I Documentation in Python uses a facility called Doc Strings Doc. Strings 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 doc strings. Because they can be easily expanded toe. Add more detail. One Python convention for Doc Strings is documented in pepper 257 Although it is not widely adopted. Various tools such a sphinx are available to build HTML document from Python Doc Strings, and each tool mandates its preferred DOC string format. Our preference is to use the form presented in Google's Python style guide, since it is amenable to being machine parched while still remaining readable at the console. Now we'll access this help from the rebel. We'll add similar doc strings to our other functions, and we'll add one for the module itself. Module doc strings should be placed at the beginning of the module before any statements Now, when we request help on the module as a whole, we get quite a lot of useful information, including the module doctoring and each function doctoring.

Comments

[Autogenerated] we believe Doc Strings are the right place for most documentation in python code, they explained how to consume the facilities your model provides rather than how it works. Ideally, your code should be clean enough. That ancillary explanation is not required. Nevertheless, it's sometimes necessary to explain why a particular approach has been chosen or a particular technique used. And we could do that using Python comments, comments and python. Begin with a hash and to continue to the end of the line. Let's document the fact that it might not be immediately obvious why we're using system RGB sub one rather than cyst RGB subzero.

Shebang

[Autogenerated] it's common on UNIX like systems that have the first line of a script include a special comment rather wonderfully called a shebang. This begins with the usual hash. As for any other comment, followed by an exclamation mark, she bang. This allows the program loader to identify which interpreter should be used to run the program. She bangs have an additional purpose of conveniently documenting at the top of a file, whether the python code there, Rayna's Python to or Python three. The exact details of your shebang command depend on the location of python on your system. Typical python three She bangs used the UNIX E N V program toe. Locate Python three on your path environment variable, which importantly, is compatible with python virtual environments on mackerel. Lennox, we must mark our script is execute a ble using the command ch mod plus X words dot p y before the shebang will have any effect. Having done that, we can now run our script directly since Python 3.3 Python on Windows also supports the use of the shebang to make Python scripts directly excusable with the correct version of the Python interpreter, even to the extent that she bangs that look like they should only work on a UNIX like system will work as expected on Windows. This works because Windows Python distributions now use a program called Pi Launcher Pie launcher, the Execute Herbal for which is called simply Pie. Dottie XY will parse the shebang and locate the appropriate version of Python, for example, on Windows in C M d ord stop P y, followed by a U. R L will be sufficient to run your script with by thin three, even if you also have python to installed in power shell. The equivalent is almost the same dot backslash words dot p y followed by the U. R L. You can read more about pie launcher in pep 397

Summary

[Autogenerated] Let's review what we've covered in this module. We learned that python code is generally placed in dot p Y files called modules. We saw how modules could be executed directly, bypassing them as the first argument to the Python interpreter. And that modules can also be imported into the rebel, at which point all top level statements in the module are executed in order. We looked at how named functions are defined using the deaf keyword, followed by the function name and the argument list in parentheses and how we can return objects from functions. Using the return statement related to this, we learned that return statements without a parameter returned none, as does the implicit return. At the end of every function body, we saw how we can detect whether a module has been imported or executed by examining the value of the special Dunder name variable. If it is equal to the string Dunder Main that are module has been executed directly as a program. We learned how to use this to write the If Dunder name equals Dunder Main check by calling a function. When this check succeeds, we can make our module both usefully in portable and execute herbal, an important testing technique even for short scripts. We saw that module code is only executed once on first import, we learned that the deaf keyword is a statement which binds execute herbal code toe a function name. We investigated how command line arguments can be access as a list of strings accessible through the RV attributes of the cyst module zero with command line argument is the script file name. So the item it in next one is the first true argument we saw that pythons dynamic typing means our functions can be very generic with respect to the type of arguments we looked into using literal strings as the first line of a function definition to form the function stock string. They're typically triple quoted multi line strings containing usage information. We learned that function documentation provided in doc strings could be retrieved using help in the ripple similar to function Doc Strings, we saw that module Doc strings should be placed near the beginning of the module. Prior to any pipes and statements such as import statements, we covered how comments and python commence with the hash character and continue to the end of the line. Finally, we looked at how the first line of the module can contain a special comment called a shebang, allowing the program loader tow. Launch the correct Python interpreter on all major platforms in the next module of core python getting started, we'll dig into pythons Object model, looking at how values are passed to and returned from functions. We'll investigate the nature of dynamic typing and python and focus on rules for variable scope. Thanks for watching, and we'll see you in the next module.

Objects and Types

Overview

[Autogenerated] While you can get a long way in python with fairly shallow understanding of its underlying model, we find that even just a small understanding of its deeper structure Camille Deep Insight, making you more productive and helping you design better programs in this module of core python getting started, we'll seek to understand the Python object model. We'll see that the notion of named references to objects is key to how Python works. We'll discuss the important difference between value, equality and identity equality. We'll see how argument passing and returning from functions and python fits into the object model will investigate pythons type system. Well, look at how Python uses scopes to limit access to names in a program, and we'll introduce you to a core insight for understanding python programs the idea that everything is an object. The topics in this module may seem abstract or even simplistic, but if you internalize these concepts, you'll find that you're able to reason about Python much more fluidly and with greater precision. We've already talked about and used variables in Python, but what exactly is a variable? What's going on when we do something a straightforward as a signing an integer to a variable. In this case, Python creates an object with the value of 1000 and object reference with the name X and arranges for X to refer to the into 1000 object. If we now modify the value of X with another assignment, what does not happen is a change in the value of the integer object entered. Your optics and python are immutable and cannot be changed. In fact, what happens is that python creates a new immutable, interred your object with the Value 500 redirects the ex reference to point to the new object. We now have no way of reaching the end 1000 object, and the python garbage collector will reclaim it. At some point when we assigned 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 ex referring to an end 3000 object and why, referring to a separate in 500 there's no work for the garbage collector to do because all objects are reachable from live references. Let's take a little deeper using the built in i D function I de Returns an integer identifier that is unique and constant for the lifetime of an object. Let's rerun the previous experiment using I. D. First we'll assign a to the integer 496 and check its i D. Then we'll assign be to 1729 and see that it has a different I. D. If we now make be referred to the same object is a we'll see that they have the same. I d note that the I D function is seldom used in production Python code. Its main use is an object model tutorials such as this one and is a debugging tool. Much more commonly used in the I. D function is tthe e is Operator, which tests for equality of identity. That is, it tests, whether to references, refer to the same object. We've already met Theis operator earlier in the course when we tested for none, even operations which seem naturally mutating in nature, are not necessarily so. Consider the augmented assignment operator. If we create an integer and then in commented by two, we see that the I D of the incremental integer is different from the original. Now let's look at that pictorially. We start with T, referring to an end five object augmented assignment creates and into two without assigning a reference to it, and then adds the institute with Ian five to create a new and seven. Finally, it assigns t toothy into seven and the remaining in our garbage collected python objects show this behavior for all types. A core rule to remember is this. The assignment operator only ever binds objects to names. It never copies an object. Why value? Let's look at another example. Using mutable objects lists we create a list object with three elements binding the list Object to a reference named R. We then assign are to a new reference s When we modified the list referred to by s By changing the middle element, we see that the our list has changed as well. This happens since the name's S and R, in fact, refer to the same object which we can verify with the is operator. Let's see that again with a diagram. First we assign are to a new list. Then we assign esto are creating a new name for the existing list. If we modify s, we also modify our because we're modifying the same underlying object s is our 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 will look at later. It turns out that Python doesn't really have variables in the metaphorical sense of a box holding a value. It only has named references to objects, and the references behave more like labels, which allow us to retrieve objects. That said, it's still common to talk about variables in Python. We will continue to do so secure in the knowledge that you now understand what's really going on behind the scenes. Let's contrast the behavior of the is operator. With the test for value, equality or equivalents, we'll create two identical lists. First, we'll test them for equivalents with the double equals operator. Then we'll test them for identity. Equality with is here we see that P and Q referred to different objects, but that the objects that refer to have the same value of course an object should almost always be equivalent to itself. Here's how that looks pictorially. We have two separate list objects, each with a single reference to it. The values contained in the lists are the same that is their equivalent or value equal, even though they have different identities. Value, equality and identity are fundamentally different notions of equality, and it's important to keep them separate in your mind. It's also worth noting that value comparison is something that is defined programmatically when you define types you can control how that class determines value equality. In contrast, identity comparison is defined by the language, and you can't change that behavior.

Passing Arguments and Returning Values

[Autogenerated] Now let's look at how all this relates to function arguments and return values. Let's define the function at the rebel, which depends a value to a list and prints the modified list. First, we'll create a list. Then we'll make a function modify, which depends, too, and prints the list. The function accepts a single formal argument named K. We then call modify, passing our list M as the actual argument. This indeed prints the modified list with four elements. But what does our list reference outside the function now refer to The list referred to by M, has been modified because it is the self same list referred to by K. Inside the function when we pass an object reference to a function were essentially assigning from an actual argument reference in this case, em to the formal argument reference. In this case, Kay, as we've seen assignment, causes the reference being assigned to to refer to the same object as the reference being assigned from This is exactly what's going on here. If you want to function to modify a copy of an object, it's the responsibility of the function to do the copying. Let's look at another instructive example. First will define a new list, then will define a function which replaces the list. Now we'll call replace with f s. The argument. It prints the new list, which is much as we'd expect. However, what's the value of F after the coal 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 N. F did indeed refer to the same object just as in the previous example. However, on the first line of the function, we reassigned the reference G to point to a newly constructed list 17 28 45. So within the function, the reference to the original list 14 23 37 was overwritten, although the original list was still pointed to by the F reference outside the function. So we've seen that it's quite possible to modify the objects through function argument references, but also possible to re bind the argument reference to new values. If you wanted to change the contents of the list and have the changes seen outside the function. You could modify the contents of the list by writing a function that replaces each element of this list in place. Now we define F and pass it into replace contents. And indeed, the contents of F have been modified by the function function. Arguments are transferred by what is called passed by object reference. This means that the value of the references copied into the function argument, not the value of the referred to object. No objects are copied. The return statement uses the same passed by object reference semantics as function arguments. We can demonstrate this by writing a simple function that just returns. It's only argument. Create an object such as a list and pass it through this simple function. We see that it returns the very same object we passed in showing that no copies of the list were made

Function Arguments

[Autogenerated] Now that we understand the distinction between object references and objects, we look at some more capabilities of function arguments. The formal function arguments specified when a function is to find with the deaf keyword are a comma separated list of the arguments names. These arguments can be made optional by providing default values. Consider a function, which prints a simple banner to the console. Dysfunction takes two arguments, the second of which is provided with the default value in this case Ah, hyphen in a literal string. Since we've given it this default value, callers can choose whether they want to pass their own value for border or use the default. Note that when we define functions using default arguments, the parameters with default arguments must come after those without defaults. Otherwise, we'll get a syntax error within the body of the function. We multiply our border string by the length of the message string. This shows how we can determine the number of items in a python collection using the built in lend function. Secondly, it shows how multiplying a string in this case the single character string border by an integer results in the 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 with border, the message and the border again. When we call our banner function, we don't need to supply the border string because we've provided a default value. We can see that the default border of hyphens has been created. However, if we do provide an optional argument, it's used in production code. This function call is not particularly self documenting. We can improve that situation by naming the border argument at the call site. In this case, the message string is called a Positional argument and the Border string a keyword argument. The actual positional arguments are matched up in sequence with the formal arguments that is by position, whereas the keyword arguments are matched by name. If we use keyword arguments for both of our parameters, we have the freedom to supply them in any order. Remember, though, that all keyword arguments must be specified after the positional arguments. It's crucial to have an appreciation of exactly when the expression provided as a default value is evaluated. This will help you avoid a common pitfall, which frequently ensnares newcomers to python. Let's examine this question closely. Using the Python standard Library Time module we can easily get the time is a readable string By using the sea time function of the time module, let's write a function, which uses a value retrieved from see time as a default argument value. So far, so good. But notice what happens when you call show default again A few seconds later and again, the displayed time never progresses. Recall how we said the death is a statement that, when executed, binds a function definition toe a function name. Well, the default argument expressions are evaluated on Lee once when the death statement is executed. Normally, when the default is a simple, immutable constant, such as an integer or a string, this causes no problems. But it could be a confusing trap for the unwary that usually shows up in the form of using mutable collections as argument defaults. Let's take a closer look. Consider this function, which uses an empty list as a default argument. It accepts the menu, which will be a list of strings appends the items spam to the list and returns the modified menu. Let's create a simple breakfast of bacon and eggs naturally, will want to add spam to it. Well, do something similar for lunch of baked beans. Nothing unexpected so far. But look what happens when you rely on the default argument by not passing an existing menu. When we append spam to the default value of the empty menu, we get just spam. Let's do that again. When we exercise the default argument value a second time, we get to 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 death statement is executed the first time we fall back on the default. This list has spam added to it. When we use the default the second time, the list still contains that item, and a second instance of spam is added to it. Making to ad infinitum, or perhaps ad nauseum, would be more appropriate. The solution to this is straightforward, but perhaps not obvious. Always use immutable objects such as integers or strings for default values. Following this advice, we can solve this particular case by using the immutable none object as a sentinel. Now our function needs to check if menu is none and provide a newly constructed empty list of, so the rest of the function behaves us before Now. If we call the function repeatedly with no arguments, we get the same menu, one order of spam each time.

Python's Type System

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

Scopes

[Autogenerated] as we've seen no type declarations are necessary in python, and variables are essentially, just untie ped name bindings to objects. As such, they could be rebound or reassigned as often as necessary, even two objects of different types. But when we bind the name to an object, where's that binding stored? To answer that question, we must look at scopes and scoping rules. In Python, there are four types of scope and python arranged in the hierarchy. Each scope is a context in which names are stored and in which they could be looked up. The four scopes from Narrowest broadest, our local names defined inside the current function in closing names to find inside any and all enclosing functions. The scope isn't important for the contents of this Python fundamentals course global names to find at the top level of a module. Each module brings with it a new global scope built in names built into the python language through the special built INS module. Together, these scopes comprised the L E G B rule names. Air looked up in the narrowest relevant context. It's important to note that scopes and python do not correspond to the source code blocks as demarcated by indentation, four loops and the like. Do not introduce new nested scopes. Consider our words dot P Y module. It contains the following global names main bound by deaf main cysts bound by imports cysts, Dunder name provided by the python runtime u R L Open bound by from your lip that request import you are all open. Fetch words bound by death. Fetch words. Print items bound by death print items. Model scope named bindings are typically introduced by import statements and function or class definitions. It's possible to use other objects at module scope, and this is typically used for Constance, although it can be used for variables within the fetch words function. We have six local names, word bound by the Inter four loop line. Words bound by assignment line bound by the outer four loop story words bound by Assignment Earl. Bound by the formal function, argument and story bound by assignment. Each of these has brought into existence at first use and continues to live within the function scope until the function complete, at which point the references will be destroyed. Very occasionally, we need to re bind a global name that is one to find it. Module scope from within a function. Consider the following simple module it initialize is count to zero at module scope. Thes show count function simply prints the value of count and set count binds the name count to a new value When show count is called. Python looks up the count name in the local name Space doesn't find it. So looks it up in the next most outer name space. In this case, the global module named Space where it finds the name count in Prince the referred to object. Now we call set count with a new value and show count again. You might be surprised that show count displays zero after the call to set count five. So let's work through what's happening when we call set count the assignment count equal c binds the object referred to by the formal arguments See to a new name count in the innermost name space context which is the scope of the current function. No look up has performed for the global count At module scope, we've created a new variable which shadows and thereby prevents access to the global of the same name tow. Avoid this situation. We need to instruct Python to consider use of the account name in the set count function to resolve to the count in the module name space. We can do this by using the global keyword. Let's modify said Count to do so The additional called the show count Still behaves is expected. Calling set count, however, does now modify the count Reference that module scope This is the behavior we want.

Moment of Zen

[Autogenerated] 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 approached object orientation is a strong theme in python, and practically everything in python is an object, including functions and modules.

Everything is an Object

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

Summary

[Autogenerated] We've covered a lot of important concepts about how the Python language works in this module. Let's summarize what we've been over. It's better to think of Python working in terms of named references to objects rather than variables and values. Assignment doesn't put a value in a box. It attach is a name tag to an object. Assigning from one reference to another puts to name tags on the same object. The python garbage collector will reclaim unreachable objects. Those objects with no name tag, the I D function returns a unique and constant identify her, but should wear the ever be used in production. The is operator determines equality of identity. That is, whether to names refer to the same object we contest for equivalents using the double equals operator function. Arguments are passed by object reference, so functions can modify their arguments if they're mutable objects. If a formal argument is rebound through assignment, the reference to the Paston object is lost. To change a beautiful argument, you should replace its contents rather than replacing the whole object. The return statement also passes by object reference. No copies were made function arguments could be specified with defaults default argument. Expressions are evaluated on Lee once when the death 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 referenced names are looked up in one of four nested scopes, according to the L E G B Rule local two functions in enclosing functions in the global or module named space and built ins. Global references could be read from a local scope. Assigning to a global reference from a local scope requires that the reference be declared global using the global keyword. Everything in python is an object, including modules and functions. They could be treated just like other objects. The import and deaf keywords result in binding to named references. The built in type function could be used to determine the type of an object. The built in dirt function can be used to introspect an object and return a list of its attributes. Names the name of a function or module object can be accessed through its Dunder name. Attributes the Doc string for a function or module object. It can be accessed through its Dunder Doc attribute in passing. We also saw that we can use lend 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 operating string. This is called the repetition operation. In the next module of core Python getting started, we'll revisit a few collections that we've already met for a deeper look. We'll also introduce you to a few new ones to pull, range and set, as well as the concept of protocols which unite the various collection types. Thanks for watching, and we'll see you in the next module.

Built-in Collections

Overview

[Autogenerated] Python comes with a powerful sweet of built in collection types, several of which you've seen already in earlier modules. To be truly fluent in Python, you need to be familiar with all of these types and how to use them. So when this module will take another deeper look at the collection types you already know stra list and dicked, we'll introduce you to some new collection types. First, we'll get to bowl in immutable sequence of objects. Then we'll cover range, which represents arithmetic progressions of integers. Finally, we'll see set immutable collection of unique, immutable objects. We'll round off with an overview of the protocols that unite these collections, which allowed them to be used in consistent and predictable ways. First up is to pull.

Tuples

[Autogenerated] two bulls 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. Two polls have a similar syntax toe lists, except that they're delimited by parentheses rather than square brackets. Here's a literal trouble containing a string afloat ended into JER, we can access the elements of a to pull by zero based index using square brackets. We can determine the number of elements in the to pull using the built in lend function we can generate over using a four loop. We can concoct innate to Paul's using the plus operator and repeat using the multiplication operator. Since two Poles can contain any object, it's perfectly possible toe have nested tubules. We use repeated application of the indexing operator to get to the inner elements of such nested collections. Sometimes a single element to polis required to write this. We can't just use a simple number in parentheses. This because python parses that as an integer enclosed in the president's controlling parentheses of a math expression to create a single element to pull, we make use of the trailing comma separator, which you'll recall were allowed to use when specifying literal tubules lists and dictionaries. A single element with the trailing comma is parsed as a single element to pull. This leaves us with the problem of how to specify an empty to pull. In actuality, the answer is simple. We just used empty parentheses. In many cases, the parentheses of literal two pools may be omitted. This feature is often used when returning multiple values from a function. Here we make a function to return the minimum and maximum values of a sequence. The hard work being done by two built in functions men and Max. Returning multiple values as a to pull is often used in conjunction with a wonderful future of python called to pull unpacking. To pull unpacking is a de structuring operation, which allows us to unpack data structures in the named references, for example, we can assign the result of our men Max function to two new references like this. When we print the two objects, we see that the references have indeed been unpacked from the two poles return from the function. Unpacking also works with nested tubules. Here we assigned from a triply nested to pull of integers to a triply nested Tupelov references. As before, we could see that each of the references has been assigned to the corresponding value from the original to bowl. This support for unpacking leads to the beautiful python idiom for swapping two or more variables. First, we'll create two references a and be referring to the strings jelly and being respectively. Then we use the form. A comma B equals B comma A. This first packs A and B into A to pull on the right side of the assignment. It then unpacks the to pull on the left, reusing the name's A and B. If we examine a and B, we could see that they have been swapped. Should you need to create a to pull from an existing collection object, such as a list you can use the to pull constructor. You can also do this with a string or indeed, any type over which you considerate. Finally, as with most collection types and python, we contest for containment using the end operator. Similarly, we contest from non membership with the knot in operator

Strings

[Autogenerated] we covered strings at some length already, but we'll take time now to explore their capabilities in a little more depth. As with any python sequence, we can determine the length of a string with the built in Len function. Here we see that the name of the longest train station in the UK contains a whopping 58 characters. Concatenation of strings is supported using the plus operator. We can create the string newfound land by contaminating the strings, new found and land. We can also use the related augmented assignment operator here, starting with the string knew we incrementally add found and land. Remember that strings are immutable. So here the augmented assignment operator is binding a new string object to s on each use. The illusion of modifying s in place is achievable because S is a reference to an object, not an object itself. While the plus operator is intuitive, you should prefer the joint method for joining large numbers of strings because it is substantially more efficient. This is because concatenation using the addition operator or it's augmented assignment version can lead to the generation of large numbers of temporaries with consequent costs for memory. allocations and copies. Join is a method on the string class, which takes the collection of strings as an argument and produces a new string by inserting a separator between each of them. An interesting aspect of joint is how the separator specified. It is the string on which a joint is cold. As with many parts of python, an example is the best explanation. To join a list of HTML color code strings into a semicolon separated string construct a string containing semi colon called joint on it, passing in the list of color strings to be joined as an argument. We can then split the colors up again using these split method. We've already encountered this method, but this time we're going to provide it's optional argument. A widespread and fast idiom for contaminating together a collection of strings is to join using an empty string as the separator. The way may not be obvious at first to concoct innate invoke, join on empty text something for nothing. This use of joint 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 to pull, so this is commonly used in conjunction with two bull unpacking. Here we partition the elements of a travel plan into its parts. Often we're not interested in capturing the separator value, so you might see the underscore variable name used. This is not treated in a special way by the Python language, but there's an unwritten convention that the underscore variable is for unused or \_\_\_\_\_ 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 use string methods is format. This supersedes, although does not replace the string interpolation technique used in older versions of python, in 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 races. The objects provided his arguments to format, are converted to strings and used to populate these fields. Here's an example where the arguments to format the string gym in the interview. 32 are inserted into the format string. The field names in this case zero and one are matched up with the positional arguments to format, and each argument is converted to a string. The field name may be used more than once here. We used the first argument to format twice. However, if the field names are used exactly once and in the same order as the arguments, the field number could be omitted. If keyword arguments are supplied to format than named, fields could be used instead of Ordina. Lt's here. The keywords, latitude and longitude are inserted into the corresponding named replacement fields. It's possible to index into sequences using square brackets inside the replacement field. Here we index into a to pull in the replacement fields. You can even access object attributes. Here we passed the whole math module to format, using a key word argument. Remember, modules are objects, then access to of its attributes from within the replacement fields. Format strings also give us a lot of control over field alignment and floating point formatting. Here are the same values with the Constance displayed using only three decimal places, while the format method we've just covered is quite powerful and is generally preferable to its predecessors. It could be quite for both, even for relatively simple cases. Consider this example. We assigned four times 22 the name value. We then interpret late value into a string with format, using the keyword argument matching feature here, we have to mention the name value three times, Of course, this example could be made shorter by removing value from the brackets in the string and not using keyword arguments to format. But in larger, more complex interpretations. We would want to keep those elements in place for readability and maintain ability to address this pep for 98 from which this example is directly drawn, introduces a new string formatting approach called literal string interpretation or, more commonly, F strings. F strings are available in Piketon 3.6 and later, and in the words of PET for 98 they provide a way to embed expressions inside literal strings using a minimal syntax. Ministering is like a normal string literal, except that it is prefixed with the letter F inside the string. Literal python expressions can be embedded inside curly braces, and the results of these expressions will be inserted into the string at runtime. Let's rework our previous example. Using F strings, we again assigned four times 22 the name value. We then use an F string toe insert value into a string here, Instead of needing to pass value into a method, the F string simply evaluates it as a normal python expression, inserting the result into the resulting string. Because F strings allow you to use any python expression, you're not limited to using simple named references. You can, for example, call functions. First, let's import the date time module. Now we use an F string to report the current time holding date time dot date time dot now to get the time, then formatting it with ice. Oh, format. We can rewrite the math Constance example from the previous section by simply accessing math dot pie and math dot e from within the F string. This then lets us demonstrate that like format that strings also support floating point formatting. To print these constants with three places of precision, we can put a colon after the expression in the S string, followed by the format. Specify WR thes air the essentials of F strings, and this may be all that you ever need to use. There's quite a bit more to know about them, though, and we'll cover F strings in greater depth in later courses. In the core python Siri's, we recommend you spend some time familiarizing yourself with the other string methods. Remember, you can find out what they are by simply passing stra to help.

Ranges

[Autogenerated] Let's move on and look at range, which really is a collection rather than a container, arranges 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 stock value, and python defaults to a starting value of zero ranges are sometimes used to create consecutive integers for use as a loop counters note that the stock value supplied to range is one past the end of the sequence, which is why the previous loops in print. Five. We can also supply a starting value if he wished, bypassing two arguments to range. Wrapping this in a call to the list constructor is a handy way to force production of each item. This so called half open range convention, with the stock value not being included in the sequence, may seem strange at first, but it actually makes a lot of sense if you're dealing with consecutive ranges because the end specified by one range is start of the next one. Range also supports a step argument. Here we count from 0 to 9 by twos. Note that In order to use it, you must supply all three arguments ranges, curious in that it determines what its arguments mean by counting them providing only one argument means the argument is a stop. Value to arguments are start and stop, and three arguments are start, stop and step Python range works this way, so the first argument start can be made optional something which isn't normally possible. Furthermore, Range doesn't support keyword arguments. You might almost describe it as on python IQ. At this point, we're going to show you another example of poorly styled code. Except this time it's one you can and should avoid. Here's a poor way to print the elements in the list by constructing a range over the length of the list and then indexing into the list on each iteration. Although this works, it's most definitely UNP ironic. Instead, always prefer to use it oration over objects themselves. If for some reason you need a counter, you should use the built in the numerator function, which returns an honorable Siri's of pairs. Each pair of being A to pull. The first element of the pair is the index of the current item and The second element of the pair is the item itself. Here we construct a list, passing it to enumerate and illiterate over the result giving us the elements of the list with the corresponding positions in the list. Even better, we can use to pull unpacking tow, avoid having to directly deal with the two bull.

Lists

[Autogenerated] 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 upend method and get at and modify their contents using the square brackets. Indexing. Now we'll take a deeper look. One very convenient feature of lists and other pipes in sequences for the supplies to two pools and strings as well is the ability to index from the end rather than from the beginning. This is achieved by supplying negative indices. For example, we can access the last and second to last elements of the list using negative one and negative, too. This is much more elegant than the clunky approach of subtracting one from the length of a container. The approach you would otherwise need to use for retrieving the last element note that indexing with negative zero is the same as indexing, with zero returning the first element in the list. Because there's no distinction between zero and negative zero. Negative indexing is essentially one based rather than zero based. This is good to keep in mind if you're calculating indices with even moderately complex logic. One off errors can creep into negative indexing fairly easily. Slicing is a form of extended indexing, which allows us to refer to portions of a list to use it. We pass the start and stop indices of 1/2 open range, separated by a colon as the square brackets index argument. Here's how you can slice the first and second elements of a list. See how the second index three in this case is one beyond the end of the returned range. This facility can be combined with negative indices, for example, to take all elements except the first and last used the slice one colon negative one. Both the start and stop indices are optional. To slice all elements from the third to the end of the list. Don't put anything after the colon similarly, to slice all elements from the beginning up to, but not including the third. Don't put a number before the colon. Notice that these two lists together form the whole list, demonstrating the convenience of the half open range convention. Since both start and stop slice indices are optional, it's entirely possible to admit both and retrieve all of the elements. And indeed, this last example is an important idiom for copying a list recall that assigning references never copies an object but merely copies a reference to an object. We deploy the full slice to perform a copy into a new list. This gives us a new object with a distinct identity, but since it's a copy, the new object has an equivalent value. It's important to understand that although we have a new list object, which could be independently modified, the elements within it are references to the same objects referred to by the original list. In the event that these objects are both mutable and modified as opposed to replaced, the change will be seen in both lists. We teach this full slice copying idiom because you're likely to see it in the wild, and it's not immediately obvious what it does. You should be aware that there are other, more readable ways of copying a list such as the copy method, or you could simply call it's the list constructor passing the list to be copied largely. It's a matter of taste. Our preference is for the third form, since it has the advantage of working with any honorable Siri's as the source, not just lists. You must be aware, however, that all these techniques perform a shallow copy. That is, they create a new list containing the same object references as the source list. But they don't copy the referring to objects to demonstrate. This will use nested lists with the inner lists serving as mutable objects. Here's a list containing two elements, each of which is itself a list. We copy this list using a full slice and convince ourselves that we do, in fact, have distinct lists with equivalent values. Notice, however, that the references within these distinct lists refer not only to equivalent objects but in fact, to the same object which we can see using. The is operator. The first elements of a N B are the same object until we re bind the first element of a toe a newly constructed list. Now the first elements of A and B referred to different lists with different values. The second elements of both A and B still refer to the same object. We'll demonstrate this by mutating that object through the A list. We can see this change reflected through the B list for completeness. Here's the final state of both E A and B lists. If you need to perform true deep copies of hierarchical data structures like this, which in our experience is a rarity, we recommend taking a look at the copy module in the pipes and standard library. As with strings and two poles, lists support repetition using the multiplication operator. It's simple enough to use, although it's really spotted in the wild. In this form, it's most often useful for initializing a list of a size known in advance, with a constant value such a zero. Be aware, though, that when using mutable objects as elements, the same trap for the unwary lurks here. Since repetition will repeat the reference without copying the value, let's demonstrate using necid lists as our mutable elements. Again, if we now modify the third element of our outer list, we can see the change through all five references, which comprise the outer list elements. This is because each element of the outer list is a reference to the same nested list. To find the element in the list used, the index method passing the object you're searching for the elements air compared for equivalents until the one you're looking for is found here. We create a string and split it on space is giving us a list of strings. We then use the index method on that list to find the index of the word fox. If you search for a value that isn't present, you receive a value error. Another means of searching is to count matching elements. Here we count how many times the word the appears in our list. If you just want to test for membership, you can use the in operator, or you could test for non membership with not in. You can remove elements from a list using a key word with which we have not yet become acquainted. Del. The Del keyword takes a single parameter, which is a reference to the list element and removes it from the list, shortening the list in the process. Here we again construct a list by splitting a string. We then remove the fourth element by calling Dell you sub three. We could see that the element has been removed. It's also possible to remove elements by value rather than by position by using the remove method. This is, of course, equivalent to passing the result of the index method to DEL. This also raises a value error. If a matching element is not present, items could be inserted into lists using the insert method, which accepts the index of the new item and the new item itself. Well again, start with a list of strings. Well, then insert the string destroyed at index to. If we join the list into a single string, we see that someone has made a terrible mistake. Could Captain Ating lists using the addition operator result in a new list without modification of the operations? Whereas the augmented assignment operator, plus equal modifies the assigning in place? This can also be achieved using the extent method. All of these techniques work with any honorable Siris on the right hand side. Before we move on from lists, let's look at two operations, which rearranged the elements in place, reversing and sorting. A list could be reversed in place simply by calling it's reverse method. A list could be sorted in place using these sort method, the sort method accepts to optional arguments. He and reverse the latter is self explanatory, and when set, the truth gives it ascending sort. The key parameter is more interesting. It accepts any call oval 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 land function is a callable object, which is used to determine the length of a collection such as a string. Consider the following list of words. We can sort this list by the length of the strings, bypassing Len as the key argument to sort, sometimes an in situ sort. A reversal is not what's required. For example, it may cause a function argument be modified, giving the function confusing side effects which would not otherwise have for out of place equivalents of the reverse and sort list methods. We can use the reversed and sorted built in functions thes return, a reverse it aerator and a newly sorted list, respectively. For example, calling sorted on the list for 9 to 1 result in an entirely new list, with the quickly sorted elements calling reversed on a list doesn't give us a new list. Rather, it gives us an object of the type list. Reverse it aerator. We can pass this iterated to the list constructor to create an actual list. These functions have the advantage that they'll work on any finite, durable source object.

Dictionaries

[Autogenerated] will now return to dictionaries, which, like the heart of many Python programs, including the Python interpreter itself. We briefly looked a literal dictionaries, previously seeing how they were delimited with curly braces and contain comma separated key value pairs with each pair tied together by a colon. These values are accessible via the keys, since each key is associated with exactly one value and look up his through keys. The keys must be unique within any single dictionary. It's fine, however, to have duplicate values internally. The dictionary maintains pairs of references to the key objects and the value objects. The key objects must be immutable, so strings numbers and two pills are fine. But lists are not the value. Objects can be mutable, and in practice often are our example. You are l'm app uses strings for both keys and values, which is fine. You should never rely on the order of the items in the dictionary. It's essentially random and may even vary between different runs of the same program As for the other collections, there's also addict constructor, which can convert other types to dictionaries. We can use the constructor to convert from unutterable Siri's of key value. Pairs stored into pools like this recall that the items in the dictionary are not stored in any particular order, so long as the keys are legitimate python identifiers. It's even possible to create a dictionary directly from keyword arguments. Past addict As with lists, dictionary copying is shallow by default, copying on Lee the references to the key and value objects, not the objects themselves. There are two means of copying a dictionary, of which we most commonly see the second. The first technique is to use the copy method. The second is simply to pass an existing dictionary to the addict constructor. If you need to extend the dictionary with definitions from another dictionary, you can use the update method. This is called on the dictionary to be updated, and it's past the contents of the dictionary, which is to be merged in. If we create a new dictionary and then update our original dictionary with the new one, we can see the elements of the new dictionary in the original. If the argument update includes keys which are already present in the Target Dictionary, the values associated with these keys are replaced in the target by the corresponding values from the source. If we start with the Dictionary of Stock Prices, we can update one entry and add another with update as we've seen in an earlier module. Dictionaries are honorable, so can be used with four loops. The dictionary yields the next key on the generation, and we retrieve the corresponding value by look up using the square brackets. Operator. If we start with the Dictionary of color names to codes, we could generate over the keys. Looking up the codes with them noticed that the keys are returned in an arbitrary order, which is neither the order in which they were specified nor any other meaningful order. If we want to reiterate over only the values, we can use the Values Dictionary method. This returns an object which provides an honorable view onto the dictionary values without causing the values to be copied. There was no efficient or convenient way to retrieve the corresponding key from a value, so we only print the values in the interest of symmetry. There was also a keys method. Often, though, we want to generate over the keys and values in tandem. Each key value pairing a dictionary is called an item and we can get hold of unutterable view of items using the items Dictionary method. When iterated the items of you, yields each key value pair as a to pull by using to pull unpacking. In the four statement, we can get both key and value in one operation without the extra look up. The membership tests for dictionaries using the inn and not in operators work on the keys to see this. Let's construct a dictionary mapping currency names two symbols we can then use in to see if New Zealand dollars are in the dictionary. Likewise, we can use not in to see that Macedonian dinners are not present. As with lists, we use the del keyword to remove an entry from a dictionary. If we create a dictionary of elements in their atomic numbers, we can remove Find Manny Um by deleting the key. F y. The keys and a dictionary should be immutable, although the values could be modified. Here's a dictionary, which maps the element symbol to a list of mass numbers for different isotopes of that element. See how we split the dictionary literal over multiple lines that's allowed because the curly braces for the dictionary literal are open. Our string keys are immutable, which is a good thing for correct functioning of the dictionary. But there's no problem with modifying the dictionary values in the event, for example, that we discover some new isotopes here. The augmented assignment operator is applied to the list object accessed through H, the key for hydrogen. The dictionary is not being modified, of course. The dictionary itself is mutable, and we can add new items with compound data structures such as our table of isotopes. It could be helpful to have them printed out in a much more readable form. We can do this with the Pythons Standard Library Pretty printing module called P print, which contains a function called P print. Note that if we didn't find the P print function to the different name PP, the function reference would overwrite the module reference, preventing further access to contents of the module. Arguably, it's poor designed to have a module containing functions of the same name. Because of this issue, be that as it may, the P print function gives us a much more comprehensible display. Let's move on from dictionaries and look at a new built in data structure. The set

Sets

[Autogenerated] the set Data type is an a Nordic collection of unique elements. The collection is mutable, insofar as elements could be added and removed from the set. But each element must itself be immutable, very much like the keys of a dictionary. Sets have a literal form very similar to dictionaries. Again be limited by curly braces, but each item is a single object rather than a pair joined by a colon. Note that the set is a kn ordered and, of course, our set has the type set. Recall that curly braces create an empty dictionary. So in order to create an empty set, we must resort to the set constructor. This is also the form that Python echoes back to us. The set constructor can create a set from any honorable Siri's, such as a list, and duplicates are discarded. In fact, a common use of sets is to efficiently remove duplicate items from a series of objects. Naturally, sets are in terrible, although the order is arbitrary, membership is a fundamental operation for sets and, as with other collection types, is performed using the inn and not in operators. Toe add a single element to a set used the ad method, adding an element that already exists has no effect and neither doesn't produce an error. Multiple elements could be added in one go from any honorable Siri's, 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, aqui error is produced. The second method discard is less fussy and simply has no effect if the element is not a member of the set. As with the other built in collections, set sports a copy method with performs a shallow copy of the set copying references but not objects. And as we've already shown, the set constructor may be used. Perhaps the most useful aspect of the set type is the group of powerful set algebra operations, which are provided. These allow us to easily compute set unions, set differences and set intersections, and to evaluate whether to sets have subset super set or dis joint relations. To demonstrate these methods will construct some sets of people. According to various fino types. We'll identify a few people with blue eyes and a few with blond hair. The smell H. C N set contains those who can smell hydrogen, cyanide and taste. PTC is those who could taste final. Theo Carpet meid finally will define a set for those with no blood. Type one for B blood type, one for a blood type and finally one for a B blood. To find all the people with blond hair, blue eyes or both. We can use the union method, which collects together all the elements which are an either or both sets. We can demonstrate that union is a community of operation. That is, we can swap the order of the operations using the value of quality operator to check for equivalents of the resulting sets. To find all the people with blond hair and blue eyes. We can use the intersection method which collects together only the elements with their present in both sets. This is also community ve to identify the people with blond hair who don't have blue eyes. We can use the difference method. This finds all the elements which are in the first set, which are not in the second set. This is non communicative because the people with blond hair who don't have blue eyes are not the same as the people who have blue eyes but don't have blond hair. However, if we want to determine which people have exclusively blonde hair or blue eyes but not both, we can use the symmetric difference method. This collects all the elements which were in the first set or the second set, but not both, As you can tell from the name. Symmetric difference is indeed communicative. In addition, three predicated methods are provided, which tell us about the relationships between sets. We can check whether one said is a subset of another using the is subset method, for example, to check whether all of the people who can smell hydrogen cyanide also have blond hair we can call is subset on smell H. C n passing blond hair as a perimeter. This checks that all the elements of the first set are also present in the second set to test whether all the people who could taste fennel file carver mind can also taste hydrogen cyanide used the is super set method. This checks that all the elements of the second set are present in the first set to test that two sets have no members in common used the is destroyed method. For example, your blood type is either a or O never both.

Protocols

[Autogenerated] 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 are based. Classes as they would in nominally typed language, is such a c sharp 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 a protocol demands specific behavior from a type. The container protocol requires that membership testing using the inn and not in operators be supported. Thes sized protocol requires that the number of elements in a collection can be determined by calling Len on the collection. It aeration is such an important concept that we're devoting a whole module to it later in this course. In short, though, Terrible's provide a means for yielding elements one by one, as they were requested. One important property of honorables that could be used in four loops. The sequence protocol requires that items could be retrieved using square brackets with an integer index that items could be searched for with index that ideas can be counted with count. And that reversed copy of the sequence could be produced with reversed In addition objects. That support sequence was also supported. Terrible sized and container. We won't cover the mutable sequence. Mutable mapping is immutable set here we have only covered one representative type of each protocol. So the generality afforded by the protocol concept doesn't gain this much at this juncture.

Summary

[Autogenerated] pythons built in collection types are generally intuitive for most purposes, but there are also quite sophisticated, so we've had to go through a lot of material in this module. Here are the topics we've covered. Two polls are immutable sequence types. Literal syntax for two poles is optional parenthesis around a comma separated list. For single element to pulls, you need to use a trailing comma. Tupelo unpacking is useful for multiple return values and swapping drinking. Captain Ation is most efficiently performed with the joint method rather than the addition or augmented assignment operators. The string partition method is a useful and elegant string parsing tool. The string format method provides a powerful means of replacing placeholders with string ified values. Python 3.6 introduced F strings, a new kind of string literal that can interpret late python expressions into the string range. Objects represent arithmetic progressions. Range could be called with 12 or three arguments describing start stop and step values. The numerator built in function is often a superior alternative to range for generating. Loop counters elicits support indexing from the end of the list with negative indices, sliced syntax allows us to copy all or part of a list. The full slices. A common python medium for copying lists, although the copy method and list constructor or less obscure. You can look for elements in a list with the index and count methods. You can remove elements from a list with the Dell. Keyword lists can be sorted a reversed in place with these sort and reverse methods. Thes sorted and reversed functions. Consort Reverse any adorable list copies and those of other collections in python are shallow copies references. Air copied, but objects or not dictionaries map from keys to values, generation and membership. Testing with dictionaries is done with respect to their keys. You should not assume any order when iterating over keys in a dictionary. The Key's values and items methods provide views into the different aspects of a dictionary allowing convenient adoration. You can copy dictionaries with the copy method or the \_\_\_\_ constructor. The Update Methadone Dictionary extends one dictionary with another sets store and on ordered collection of unique elements. Sets support powerful set algebra operations and critic. It's the built in collections could be organized according to which protocols they support, such as it horrible sequence and mapping in passing. We've also found that underscore is in common usage used for \_\_\_\_\_ or superfluous variables. The P print module supports pretty printing of compound data structures in the next module of core python Getting started. Well, lookit exceptions and how to work with them in Python. Thanks for watching, and we'll see you in the next module.

Exceptions

Overview

[Autogenerated] In most programs, there is a clear notion of the normal path through the code. But of course, conditions can arise where this normal flow can't be followed. For example, if a program involves reading a file specified by the user, it may actually happen that the file doesn't exist. Conditions like this often need to be handled, and the standard mechanism for doing so in python, as with many other languages, is with what are known as exceptions. In this module of core python getting started, we will learn what exceptions are. We'll see how to introduce or raise an exception, and we'll look at how doing so interrupts the normal flow of a program. We'll learn about how you can catch exceptions to handle them, and we'll see what happens to your program. If you choose not to handle exceptions, we'll talk a bit about pythons somewhat liberal approach to the use of exceptions. We'll explore some of pythons built in exception types, and we'll see that some of these indicate programmer errors, while others represent various other kinds of conditions and we'll look at it an important mechanism for ensuring resource clean up when exceptions are involved, exception. Handling is a mechanism for stopping normal program flow and continuing at some surrounding context or code block. The event of interrupting normal flow is called the active, raising an exception in some enclosing context. The race exception must be handled upon which control flow is transferred to the exception handler. If an exception propagates up the call stack to the start of the program than unhand Aled exception will cause the program to terminate. An exception object containing information about where and why an exceptional event occurred is transported from the point at which the exception was razed to the exception handler, so that the handler can interrogate the exception, object and take appropriate action. If you've used exceptions and 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 entire Cem debates over exactly what constitutes an exceptional event. The core issue being that exception ality is in reality a matter of degree. Some things are more exceptional than others, whereas programming languages tend to impose a false dichotomy by insisting that an event is either entire the exceptional or not at all exceptional. The Python philosophy is that the liberal end of the spectrum when it comes to the use of exceptions, exceptions are ubiquitous and python, and it's crucial to understand how to handle them.

Exceptions and Control Flow

[Autogenerated] Since exceptions are a means of control flow, they could be clumsy to demonstrate at the rebel. 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 dot p y well defined a function called Convert that attempts to construct an integer from a sequence of strings describing its decimal digits. It then returns that integer import to convert function from this module into the python ripple and call our function to see that it has the desired effect. This seems to work, but if we call our function with an object that can be converted to an integer, we get a trace back from the dictionary. Look up. What's happened here is that digit map raised a key error when we tried to look up the string around in it. Of course it did this because it doesn't have an entry for around. We didn't have a handler in place, so the exception was caught by the rebel and the stack trace was displayed. The Kierra referred to in the stack. Trace is the type of the exception object in the error message. The string around is part of the payload of the exception object that has been retrieved and printed at the rebel notice that the exception propagates across several levels in the call stack.

Handling Exceptions

[Autogenerated] Let's make our convert function more robust by handling the key error. Using a try except construct both the try and accept keywords introduced new blocks. The try block contains code that could raise an exception, and the except Block contains the code, which performs error handling in the event that an exception is raised. Modify your convert function To look like this. We have decided that if a nun convertible string is supplied, will return negative one. To reinforce your understanding of the control flow. Here, we'll add a couple of prints statements. Let's test this interactively. After restarting the rebel first, we import to convert function from a module named Exceptional. Then we convert the string 34 into the number 34. Finally, we try to convert the word 11 teen, which, of course, fails. Note how the print in the tri block, after the point at which the exception was raised, was not executed when we passed in 11 teen. Instead, execution was transferred directly to the first statement of the except Block. Our function expects its argument s to be a terrible so let's see what happens if we pass an object that isn't for example, an integer. This time our handler didn't intercept the exception. If we look closely at the trace, we could see that this time we received a type error, a different type of exception. Each tribe lock can have multiple corresponding except blocks, which intercept exceptions of different types. Let's add a handler for type error, too. Now, if we re run the same test in a fresh ripple, we find that the type error is handled as well. We've got some code duplication between our two exception handlers. With that duplicated print statement and assignment, we'll move the assignment in front of the tribe lock, which doesn't change the behavior of the program. Then we'll exploit the fact that both handlers do the same thing by collapsing them into one, using the ability of the except statement to accept a to pull of exception types. Now we see that everything still works as designed. We can convert the string to nine, but converting the word elephant or the entity or 451 will fail in return. Negative one

Exceptions and Programmer Errors

[Autogenerated] 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 indentation error. Because our except block is now empty and empty blocks are not permitted in python programs. This is not an exception that it is ever useful to catch within except block. Almost anything that goes wrong with the Python program results in an exception. But some, such as indentation, error, syntax, error and name error are the result of programmer errors, which should be identified and corrected during development rather than handled it run time. The fact that these things are exceptions is mostly useful. If you're creating a python development tools such as a python i d e. Embedding python itself in a larger system to support application, scripting or designing a plugin system which dynamically loads code with that said, we still have the problem of what to do with our empty except block. The solution arrives in the form of the Pass Key Word, which is a special statement that does precisely nothing. It's a no up and its only purpose is to allow us to construct us and tactically permissible blocks that are semantically empty. Perhaps in this case, though, it would be better to simplify further and just use multiple return statements and do away with the X variable completely. Sometimes we'd like to get hold of the exception object in this case, an object of type of key error or a tribute error and interrogated for more details of what went wrong. We can get a named reference to the exception object by tacking in as Klaus under the end of the except a statement will 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 system module, so at the top of our module will need to import cysts. We can then pass sis standard error as a keyword argument called file to Print. Here we use a feature of F strings that we haven't seen before. If you put an exclamation, are after the expression the rep, a representation of the value will be inserted into your string. In the case of exceptions, this gives us more detailed information about the type of the exception. Let's see that at the rebel

Re-raising Exceptions

[Autogenerated] let's add a second function string log to our module, which calls are convert function and computes the natural log of the result. We've written this fairly innocuous looking bit of code to demonstrate the greatest folly of returning error codes that they could be ignored by the caller wreaking havoc amongst unsuspecting code later in the program. A slightly better program might test the value of E before proceeding to the log. Cole without such a check log will, of course, fail when past the negative error code value. Naturally, the log failure causes the raising of another exception. Much better and altogether. More python IQ is to forget about error, return codes completely and go back to raising an exception from convert. Instead of returning an unfit tonic error code, we can simply admit our error message and reraise the exception object we're currently handling. This could be done by replacing the return and negative one with rays. 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 rebel. We can see that the original exception type is re raised, whether it's a key error or a type error. But our conversion error messages printed two standard error along the way

Exceptions Are Part of the API

[Autogenerated] exceptions form an important aspect of the A P I 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 will use square root finding as an example using ah, homegrown square root function, courtesy of Heron of Alexandria, although he probably didn't use Python Place the following code in a file named Roots dot p y. There's only one language feature in this program we haven't met yet. The logical and operator, which we use in this case to test that two conditions are true on each iteration of the loop. Python also includes a logical or operator, which could be used to test whether either or both operations are true. Running our program, we can see that \_\_\_\_\_\_ was really onto something. Let's add a new line to the main function, which takes the square root of negative one. If we run that, we get a new exception. What has happened is that Python has intercepted a division by zero, which occurs on the second iteration of the loop and raised an exception, a zero division error. Let's modify our code to catch the exception before it propagates up to the top of the call stack, thereby causing our program to stop using the try except construct. Now, when we run the script, we see that we're handling the exception cleanly. We should be careful to avoid a beginner's mistake of having too tight scopes for exception handling blocks, we can easily use one try except block for all of our calls to square root. We also add 1/3 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 zero division error. Python provides us with several standard exception types to signal common errors. If a function parameter is supplied with an \_\_\_\_\_\_\_ value, it is customary to raise a value error. We can do this by using the Rays keyword with a newly created exception object, which we can create by calling the value error constructor. There are two places we could deal with the division by zero. The first approach would be to wrap the route finding while loop in a try except zero division error construct and then raise a new value error exception from inside the exception handlers. This would be wasteful, though we know this routine will fail with negative numbers so we can detect this precondition early on and raise an exception. At that point, the test is a simple if statement and a call to raise passing. The new exception object. The value error constructor accepts an error message. See how we can modify the DOC string to make it plain which exception type will be raised by square root and under what circumstances. But look what happens if we run the program. We're still getting a trace back in an ungraceful program exit. This happens because we forgot to modify our exception handler to catch value error rather than zero division error. Let's modify our calling code to catch the right exception class and also assigned the caught exception object to unnamed variable so that we can interrogate it after it has been caught. In this case, our interrogation is simply to print the exception object, which knows how to display itself as the message to standard error running the program again, we can see that our exception is being gracefully handled

Exceptions and Protocols

[Autogenerated] exceptions are part of a functions, a P I and more broadly, are part of certain protocols. For example, objects which implement the sequence protocol should raise an index error exception for indices, which are out of range. The exceptions, which are raised are as much a part of a functions specifications as the arguments that accepts and as such, must be implemented and documented appropriately. There are a handful of common exception types in python, and usually when you need to raise an exception in your own code, one of the built in types is a good choice. Much more rarely. You'll need to define a new exception type, but we don't cover that in this course. Often, if you're deciding what exceptions your code should raise, you should look for similar cases in existing code. The more your code follows existing patterns, the easier it will be for people to integrate and understand. For example, suppose you were writing a key value database. It would be natural to use key error to indicate a request for a non existent key because this is how dicked works, that is. Mapping and python follow certain patterns, and exceptions are part of that pattern. Let's look at a few common exception types. Index error is raised when an integer index is out of range. You can see this when you index past the end of a list. Value error is raised when an object is of the right type but contains an inappropriate value. We've seen that already. When trying to construct an end from a non numeric string. Key error is raised. When I look up in a mapping fails. You can see that here when we look up a non existent key, an addict.

Avoid Explicit Type Checks

[Autogenerated] we tend not to protect against type errors in python. To do so runs against the grain of dynamic typing and python and limits the reuse potential of code rewrite. For example, our convert function could test whether the argument was a list using the built in is instance function and raise a type error exception if it was not. But then we also want to allow arguments that are instances of two. Poulos. Well, it soon gets complicated if we want to check whether our function will work with types such as set dicked or any other endurable type. And in any case, who is to say that it does? Alternatively, as we currently do, we could intercept, type, error inside or convert 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 that you couldn't have known about when you designed the function and that's all to the good. If not execution will probably result in a type here anyway. Likewise, we tend not to catch type errors very frequently.

It's Easier to Ask Forgiveness Than Permission

[Autogenerated] 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 the program operation that might fail. The first approach is to check that all the preconditions for a failure prone operation are met in advance of attempting the operation. The second approach is to blindly hope for the best, but be prepared to deal with the consequences if it doesn't work out in Python culture. These two philosophies are known as Look Before You Leap or L B Y L. And it's easier to ask forgiveness than permission. E a F p A term, which, incidentally, was coined by Rear Admiral Grace Hopper, inventor of the compiler Python, is strongly in favor of E. A. F 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 L B Y El version before attempting to call process file. We checked that the file exists, and if it doesn't, we avoid making the call in 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 in existence. Check what if the file exists but contains garbage? What if the path refers to a directory instead of a file? According to L B Y el, we should add preemptive tests for these two. A more subtle problem is there is a race condition here. It's possible for the file to be deleted, for example, by another process between the existence check and the process file call a classic issue of Adamis ity. 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 Python IC E. A. F. P 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 OS error, which covers all manner of conditions such as file not found and using directories where files are expected, E. A, F, P s Standard and python. And that philosophy is enabled by exceptions without exceptions, that is, using error codes. Instead, you're forced to include error handling directly in the main flow of your logic. Since exceptions interrupt the main flow, they allow you to handle exceptional cases. Non locally exceptions coupled with E a, F P R. 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 E. F P based style, makes it very difficult for problems to be silently ignored.

Cleanup Actions

[Autogenerated] Sometimes you need to perform a cleanup action, irrespective of whether on operation succeeds in a later module, will introduce context managers, which are the modern solution to this common situation. But here will introduce the tri finally construct. Since creating a context manager can be overkill in simple cases and in any case and understanding of try finally is useful for making your own context. Managers consider this function, which uses various facilities of the standard OS module to change the current working directory, create a new directory at that location and then restored to the original working directory at first sight. This seems reasonable, but should the call to O s make her fail? For some reason, the current working directory of the Python process won't be restored to its original value, and the make at function will have an unintended side effect. To fix this, we'd like the function to restore the original current working directory. Under all circumstances, we can achieve this with a try. Finally, block code in the finally block is executed. Whether exception leaves the tribe Locke, normally by reaching the end of the block or exceptionally by an exception being raised this construct could be combined with, except blocks here used to add a simple failure logging facility. Now, if OS maker raises an OS error, the OS error handler will be run, and the exception will be re raised. But since the finally block has always run, no matter how the tribe lock ends, we could be sure that the final directory change will take place in all circumstances. Errors should never pass silently unless explicitly silenced errors or like bells. And if we make them silent, they were of no use.

Platform-Specific Code

[Autogenerated] detecting a single key press from python such as the press. Any key to continue functionality at the console requires use of operating systems 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 M S V C R T module in on Lennox and Mac OSX. We need to use functionality from the units on Lee T T y and term IOS modules. In addition to the CIS module. This example is quite instructive as it demonstrates many python language features, including import and death 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 and Ms V C R t the Microsoft Visual see run time. If this succeeds, we then proceed to define a function. Get key which delegates to the M S V c R t get ch function even though we're inside a tribe. Look at this point. The function will be declared at the current scope, which is the module scope. If, however, the import of M S V. C. R T fails because we're not running on Windows in import, error will be raised and execution will transfer to the except block. This is a case of an error being silenced explicitly because we're going to attempt an alternative course of action in the exception handler within the except Block, we import three modules needed for a get key implementation on UNIX like systems and then proceed to the alternative definition of get Key, which again binds the function implementation to a name in the module scope. This unit's implementation of get key 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 Nora UNIX like system the import T T Y statement will raise a second import error. This time we make no attempt to intercept the exception. We allow it to propagate to our caller, which is what ever attempted to import this key press module. We know how to signal this error, but not how to handle it. so we differ. 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 pythons input built in function and giving a different message to the user.

Summary

[Autogenerated] exceptions are an essential topic and python, and it's critical that you understand how to work with them as you develop your mastery of the language in this module, we learned that the raising of an exception interrupt normal program flow and transfers control to an exception. Handler exception. Handlers are defined using the try except construct. Try blocks to find a context in which exceptions can be detected corresponding except blocks to find handlers for specific types of exceptions. Python uses exceptions pervasively, and many built in language features depend on them, except blocks can capture an exception. Object, which is often of a standard type such as value error, Kierra or index error. Programmer error such as indentation, error and syntax error should not normally be handled. Exceptional conditions can be signaled using the rays keyword, which accepts a single parameter of an exception. Object raise without an argument within an except block re raises the exception, which is currently being processed. We tend not to routinely check for type errors. To do so would negate the flexibility afforded to us by pythons dynamic type system exception. Objects can be converted to strings using the straw constructor for the purposes of printing message payloads. The exceptions thrown by a function form part of its A p I 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 tri finally construct, which may optionally be used in conjunction with except blocks. Along the way, we saw that the output of the print function can be redirected to standard error using the optional file argument. The expressions and F strings can be Suffolk's with exclamation are to use the rep. A representation of the inserted value python supports the logical operators and an or for combining Boolean expressions. Return codes are too easily ignored. Platform specific actions can be implemented using an easier to ask forgiveness than permission approach facilitated by intercepting import errors and providing alternative implementations. In the next module of core python getting started, we'll take a deeper look at the concept of federation and python covering topics including comprehension, Sze pythons iteration protocols and lazy evaluation. Thanks for watching and we'll see you in the next module

Iteration and Iterables

Overview

[Autogenerated] a central abstraction in python is the notion of an honorable, an object from which you can fetch a sequence of other objects. The act of fetching a sequence from in a terrible is known as federation, and you've already encountered it many times in the form of the four loop Adoration and Python is designed to be simple to use and not require a deep understanding on the part of the user. But under this fairly unassuming surface lies a sophisticated and powerful system in this module of core python getting started, we'll look at comprehension. Sze pythons, a shorthand syntax for creating certain kinds of terrible objects. We'll see that some comprehension create objects that you're already familiar with, like lists and sets, while others create wholly new kinds of objects with some surprising properties. We'll learn about the syntax for filtering incomprehension. Sze will discover pythons low level A P I for working with a terrible objects. And in doing so, we learned about another crucial element of adoration, the concept of the it aerator. We'll also look into python. Somewhat surprising use of exceptions in its core. It oration protocol will cover generator functions, a powerful and sometimes a subtle technique for defining a terrible sequences as computations. In doing so, we'll talk about pythons yield key word, and we'll look at some fascinating qualities of generators such as state fullness, laziness and the ability to model infinite sequences. We'll also cover what are known as generator expressions, a cross between comprehension tze and generator functions. Finally, we'll look at some of the tools that Python provides for working with adoration, including several elements from the standard libraries Inter Tools module. Let's jump right into comprehension, Sze.

List and Set Comprehensions

[Autogenerated] comprehension is in python are concise syntax for describing lists, sets or dictionaries in a declarative or functional style. This shorthand is readable and expressive, meaning that comprehension czar very effective at communicating intent to human readers. Some comprehension is almost read like natural language, making them nicely. Self documenting comprehension are much easier to demonstrate than they are to explain. So let's bring up a python ripple First will create a list of words. Buy splitting a string. Now comes a list comprehension. The comprehension is enclosed in square brackets, just like a literalist. But instead of literal elements, it contains a fragment of declared of code Len Word, which describes how to construct the elements of the list, which we loop over using forward in words. Here, the new list is formed by binding word teach value in words in turn and evaluating lend word to create a new value. Each of these new values becomes an element in the newly constructed list. The general form of the list comprehension is an opening square racket, an expression, a statement binding and name two successive elements of unutterable and a closing square racket that is for each item in the interval on the right, we evaluate the expression on the left, which is almost always, but not necessarily in terms of the item and use that as the next element of the list being constructed. The comprehension is the declarative equivalent of the following imperative code. We first create the empty list lengths. We then use a four loop to generate over words, finding each element in turn to the name word For each generation. We calculate the length of the word and depend the length two lengths. Note that the source object over which reiterate doesn't need to be a list. It could be any object which implements the adorable protocol such a za to pull the expression. Producing the new lists elements could be any python expression. Here we find the number of decimal digits in each of the 1st 20 Factorial is using range to generate the source of sequence. First we import factorial from the math module. We then calculate the length of the decimal string representation of the factorial of you Genta Jer from 0 to 19. Note also that the type of the object produced by list comprehension XYZ nothing more or less than a regular list set support a similar comprehension. Syntax using, as you might expect, curly braces our number of digits. In fact, Orioles result contained duplicates by building a set. Instead, we can eliminate them, although note that the resulting set is not necessarily stored in a meaningful order, since sets are a Nordic containers.

DIctionary Comprehensions

[Autogenerated] the third type of comprehension is the dictionary comprehension. This also uses curly braces and this distinguished from the set comprehension by the fact that we now provide to Colin separated expressions for the key and the value, which will be evaluated in tandem for each new item. Let's start with the dictionary We can play with which maps countries to capital cities United Kingdom, Toa London, Brazil to Brasilia, Morocco, to Robot and Sweden to Stockholm. One. Nice use for a dictionary comprehension is to invert a dictionary so we can perform efficient look ups in the opposite direction. Here we use Capital Cities as the keys by putting capital on the left of the colon and countries as the values. By putting country on the right, we get the country capital to polls. By looping over country to capitol dot items, we can now import the P print function and use it to pretty print our new inverted dictionary. Note. The dictionary comprehension is do not operate directly on dictionary sources. Well, they can but recall that iterating over a dictionary yields on Lee the keys. If we want both keys and values, we should use the items method and then to pull unpacking to access the key in value separately, as we did in this example, Should your comprehension produce some identical keys later, Keys will overwrite earlier keys. In this example, we start with a list of words and map, the first letter of words to the words themselves. But only the last H word is kept. Remember that there's no limit to the complexity of the expression you can use in any of the comprehension. But for the sake of your fellow programmers, you should avoid going overboard. Extract complex expressions into separate functions to preserve readability. The following is close to the limit of being reasonable for a dictionary comprehension. We import OS and glob. We then use a dictionary comprehension to map the real paths of files to their sizes.

Filtering Comprehensions

[Autogenerated] all three types of collection comprehension support an optional filtering claws. This clause allows us to choose which items of the source are evaluated by the expression on the left to make this interesting. Well, first, find a prime ality testing predicate function. It uses the square root function, so we have to import that. We then define the function is prime. It checks to see if the input is less than two returning false. If so, it then checks of the input could be evenly divided by any imager up to the square root of the input returning false. If so, finally, it returns. True, if no devices were found, we can now use is prime as the filtering claws of a list comprehension to produce all primes less than 100. We have a slightly odd looking X for X construct here because we're not applying any transformations to the filtered values. The expression in terms of X is simply X itself. There's nothing to stop us combining a filtering predicate with the transforming expression. Here's a dictionary comprehension, which maps numbers with exactly three. Devise ear's to a tube full of those devices

Moment of Zen

[Autogenerated] code is written once but read over and over. Fewer is clearer comprehension czar often more readable than the alternative. However, it's possible to overuse comprehension. Sze sometimes a long or complex comprehension, maybe less readable to 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 tried 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 adoration, use another construct, such as a four loop instead.

Iteration Protocols

[Autogenerated] comprehension. Tze and four Loops are the most frequently used language features for performing it oration that is, taking items one by one from a source and doing something with each. In turn, however, both comprehension Zen four loops iterated over the whole sequence by default, whereas sometimes more fine grained control is needed. There are two important concepts here, on top of which a great deal of python language behavior is constructed, a terrible objects and it aerator objects, both of which are reflected in standard python protocols. Thean Terrible Protocol allows us to pass an honorable object, usually a collection or stream of objects, such as a list to the built in inter function to get an iterated for the interval object. Iterated objects, in turn, support the iterated protocol, which requires that we can pass the IT aerator object to the built in next to function to fetch the next value from the underlying collection. As usual, a demonstration at the Python Ripple will help all these concepts crystallized into something you can work with. We use a list of names of the seasons in British English, no less is Ari terrible source object. We ask our edible object to give us an ID. Aerator, using the teeter built in and then request of value from the generator using the next built in each call to next, moves the iterated through the sequence. But what happens when you reach the end? In a spectacular display of liberalism? Python raises an exception specifically of the type stop iteration those of you coming from other programming languages with the more straight laced approach. Two 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 edible Siri's maybe 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 four loops and comprehension at our fingertips, the utility of these lower level it oration protocols may not be obvious to demonstrate a more concrete use. Here's a little utility function, which, when past unutterable object, returns the first item in that Siri's or if the series is empty, raises a value error. First, it calls it err on the input in terrible object to produce, innit? Aerator. It then calls next on the it aerator inside a try block returning the results. If the input in Terrible is empty, the next has no value to return, and it instead raises Stop it oration. We catch that and raise a value ever. Instead, this function works as expected, on any honorable object. Here we see it getting the first element from a list, and we can see it doing the same for set. If we pass an empty set to first, it throws a value error. It's worth noting that the higher level it oration constructs such as four loops and comprehension are built directly upon this lower level it oration protocol.

Generator Functions

[Autogenerated] Now we come to generator functions, one of the most powerful and elegant features of the Python programming language. Python generators provide the means for describing a terrible Siri's with code in functions. Thes sequences are evaluated lazily, meaning they only compute the next value on demand. This important property allows them to model infinite sequences of values with no definite end such a streams of data from a sensor or active log files. By carefully designing our generator function, we can make generic stream processing elements, which could be composed into sophisticated pipelines. Generators are defined by any python function, which uses the yield key word at least once. In its definition, they may also contain the return key word with no arguments. And just like any other function, there's an implicit return at the end of the definition toe. Understand what generators do? Let's start with a simple example. Att, the python ripple. Let's define the generator, and then we'll examine how the generator works. Generator functions are introduced by death. Just as for a regular python function, well, then yield the values 12 and three in order. Now let's call Jen 123 and a sign. It's return value to G. As you can see Jen 12 threes called just like any other python function. But what has it returned? G is a generator object generators are, in fact, python. It aerators so we can use the iterated protocol to retrieve or yield successive values from the series. Take note of what happens now that we've yielded the last value from our generator. Subsequent calls, too. Next, raise a stop iteration exception just like any other python. It aerator because generators are it aerators. And because interpreters must also be a terrible, they can also be used in all the usual python constructs, which expected terrible objects such as four loops. Be aware that each call to the generator function returns a new generator. Object here. Recall a generator function two times, binding the results to H and I, respectively. If we look at H and I, we see that they're different objects and that each generator object could be advanced independently. Let's take a closer look at how and crucially when the code in the body of our generator function is executed. To do this will create a slightly more complex generator that traces its execution with good old fashioned print statements. We call this generator jen 246 It will first print that it's about to deal, too, and then it will yield to. He will do the same for four and finally for six. At the end, it will print that it's about to return. Now we call the Generator and assign it to the name G. At this point, the generator object has been created and returned, but none of the code within the body of the generator function has yet been executed. So let's fetch the first value from the generator. See how, when we request the first value the generator body runs up to and including the first yield statement, the code executes just far enough to literally yield the next value. When we request the next value from the generator, execution of the generator function resumes at the point it left off and continues running until the next yield after the final value has returned. The next request causes the generator function to execute until it returns at the end of the function body, which in turn raises the expected stop it oration exception. Now that We've seen how generator execution is initiated by calls, too next and interrupted by yield statements, we could progress to placing more complex code and a generator function body.

Maintaining State in Generators

[Autogenerated] Now we look at how our generator functions, which resume execution each time the next value is requested can maintain state and local variables. In the process of doing so, our generators will be both more interesting and more useful. The resume Herbal nature of generator functions can result in complex control flow. So we'll be watching the execution of these generators and a graphical python de \_\_\_\_\_\_. I'll be using pi charm, but you can use any python de \_\_\_\_\_\_ 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 will look at is take, which retrieves a specified number of elements from the front of unutterable. 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 sequence when we reach a specified point. Now let's bring our second generator into the picture. This generator function, called distinct, eliminates duplicate items by keeping track of which elements it's already seen in a set in this generator. We also make use of a controlled flow construct we have not previously seen. To 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 four statement. But as with brake, it can also be used with wild loops in this case that continues, used to skip any values which have already been yielded. Now we'll arrange both of our generators into a lazy pipeline using take indistinct together to fetch the 1st 3 unique items from a collection, we start creating a small list called items. We passed the result of distinct and to take and loop over. The results noticed that the distinct generator only does just enough work to satisfy the demands of the take generator, which it is iterating over. It never gets us far as the last two items in the source list because they're not needed to produce the 1st 3 unique items. This lazy approach to computation is very powerful, but the complex control flows it results in can be difficult to debug it's often useful during development to force evaluation of all the generated values, and this is most easily achieved by inserting a call to the list constructor. This interspersed called TAU list, causes the distinctive generator to exhaustively process its source items before take does its work.

Laziness and the Infinite

[Autogenerated] generators are lazy, meaning that computation on Lee 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 air only produced as requested by the collar. And since no data structure needs to be built to contain the elements of the sequence, generators could safely be used to produce never ending or just very large sequences like sensor readings, mathematical sequences and the contents of multi terabyte files. The authors of this course are sworn by Sacred Oath never to use either Fibonacci or quick sort implementations in demonstrations or exercises. Allow us instead to present a generator function for the Lucas Siri's, which has nothing whatsoever to do with the order in which you should watch the episodes of Star Wars. The Lucas series starts with 21 and each value after that is the sum of the two proceeding values. So the first few values of the sequence are 21347 and 11. The first yield produces the value to the function, then initialize is A and B, which hold the previous two values needed as the function proceeds, then the function enters an infinite wild loop, where it yields the value of B, A and B are updated toe hold the new previous two values. Using a neat application of tupelo unpacking. Now that we have a generator, it can use like any other edible object. For instance, to print the Lucas numbers, you could use a loop like this. Of course, since the Lucas sequences infinite, this will run forever. Printing out values into your computer runs out of memory. Use control C to terminate the loop.

Generator Expressions

[Autogenerated] generator expressions are a cross between comprehension tze and generator functions. They use a similar sin taxes comprehension sze, 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 comprehension Sze delimited by parentheses instead of the brackets used for list comprehension. Sze generator expressions are useful for situations where you want the lazy evaluation of generators with the declarative incision of comprehension. Sze, for example, this generator expression yields a list of the 1st 1 million square numbers. At this point, none of the squares have been created. We just captured the specification of the sequence into a generator object. We can force evaluation of the generator by using it too great a long list. This list obviously consumed a significant chunk of memory in this case, about 40 megabytes for the list object and the interviewer objects contained therein. I also noticed that a generator object is Justin. It aerator 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 some of the 1st 10 million squares using the built in some function, which accepts an honorable series of numbers. If this were 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 could see that in this case, we didn't supply separate enclosing parenthesis for the generator expression. In addition to those needed for these, some function called this elegant ability to have the parenthesis used for the function. Call also served for the generator expression AIDS readability. You could include the second set of parentheses if you wish. As with comprehension, Sze, you concluded. If Klaus in the end of the generator expression reusing our admittedly inefficient is prime predicate, we can determine that some of those indicators for the 1st 1000 which are prime like this, note that this is not the same thing as computing. The sum of the 1st 1000 primes, which is a more awkward question because we don't know in advance how many interviews we need to test before we clock up 1000 crimes.

Iteration Tools

[Autogenerated] So far, we've covered the many ways Python offers for creating a terrible objects. Comprehension generators and any object which follows the edible or iterated protocols could be used for adoration, so it should be clear that it aeration is a central feature of Python. Python provides a number of built in functions for performing common it aerator operations. These functions form the core of a sort of vocabulary for working with it aerators, and they could be combined to produce powerful statements in very concise, readable code. We've met some of these functions already, including enumerates for producing integer indices and some for computing. Summation of numbers. In addition to the built in functions, Theater Tools module contains a wealth of useful functions and generators for processing a terrible streams of data. We'll start demonstrating these functions by solving the 1st 1000 primes problem. Using the built in some with two generator functions from inter tools I slice and count. Earlier, we made our own take generator function for a leisurely retrieving the start of the sequence. We needn't have bothered, however, because I slice allows us to perform lazy slicing similar to the built in list slicing functionality to get the 1st 1000 crimes we need to do something like I slice all primes 1000. But how to generate all primes? Previously, we've been using range to create the raw sequences of vintage ear's to feed into our formality test. But ranges must also be finite that is bounded on both ends. What we'd like is an open ended version of range, and that is exactly what it turtles count provides using count and I slice our 1st 1000 crimes expression could be written out as the ice slice of a generator of all prime numbers. This returns a special I slice object, which is in terrible. We can convert it to a list using the list constructor. Answering our question about that some of the 1st 1000 crimes is now easy remembering to recreate the generators. We simply pass that I slice. We just defined to the some function. Two other very useful built ins which facilitate elegant programs, are any and all their equivalent to the logical operators and an or but for a terrible Siri's is of bull values. Any takes an interval and tells you if any elements in it are true, all takes an interval and tells you have all the elements in it are true here will use any together with the generator expression to answer the question of whether there are any prime numbers in the range. 1328 2 1360 inclusive. Any is prime of X for X in range 1323 2 1361 For a completely different type of problem, we can check that all of these city names are proper. Announce with initial uppercase letters, all of name equals name dot title for each name in a list of cities the last building will look at is it, which, as its name suggests, gives us a way to synchronize iteration over to it horrible Siri's. For example, let's define a column of temperatures for Sunday and a column for Monday. If we zip these together, we could see that zip yields to pools when iterated. This in turn, means we can use it with two pull unpacking in the four loop. In fact, Zip can accept any number of terrible arguments. Let's add 1/3 time Siri's. We then zip together Sunday, Monday and Tuesday and use men Max some and lend to calculate some statistics for our data points. Note how we view string formatting features to control the New America column with 24 characters. Perhaps though we'd like one long temperature. Siri's for Sunday, Monday and Tuesday. Weaken lazily can captain eight intervals using it or tools dot chain. So this is different from simply contaminating 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've made together and leave your computer computing the Lucas primes. When you've seen enough, we recommend you spend some time exploring the Inter Tools module.

Summary

[Autogenerated] generation is a pervasive and powerful concept in python. Having a good understanding of how to work with terribles in federation is often the key to designing elegant, sophisticated solutions to your problems in this module. We learned that comprehension czar Concise syntax for describing lists, sets and dictionaries. Comprehension. Sze operate on a terrible source objects and apply an optional predicated filter and a mandatory expression, both of which are usually in terms of the current item. Adorable objects are objects for which we can generate. Item by item. We retrieve innit Aerator from unutterable object. Using the built in inter function, it aerators produce items one by one from the underlying a terrible Siri's each time their past to the built in. Next function, it aerators. Raisa, stop it! Oration exception. When the collection is exhausted, generator functions allow us to describe sequences using imperative code. Generator functions contain at least one use of the yield. Keyword generators are iterated er's When the generator's advanced with next, the generator starts or resumes execution up to including the next yield. Each call to a generator function creates a new generator. Object generators can maintain explicit state and local variables between it orations. Generators are lazy, and so can model infinite series of data. Generator expressions have a similar syntactic form to list comprehension and allow for more declarative and concise way of greeting generator objects. Python includes a rich set of tools for dealing with honorable Siri's, both in the form of built in functions such as some any and zip, but also in the intertitles module in the next module of core python getting started. Well, look a topic that you might have expected to see earlier in the course that of classes. Classes allow you to define the structure and behavior of objects in a unified manner, and Python has strong support for defining them. Thanks for watching, and we'll see you in the next module.

Classes

Overview

[Autogenerated] you can get a long way in python using the built in scaler and collections types. For many problems, the built in types and those available in the Python standard library are completely sufficient. Sometimes, though, they aren't quite what's required. And the ability to create custom types is where classes come in. In this module of corporate thing getting started well, define what we mean by class will investigate the relationship between classes and in objects type. We'll see how to define new classes. We learned about instance methods, and we'll see how to add them to classes. And we look into the ubiquitous self argument of instance methods. We'll discover how to add initialize. These two classes. We'll discover how initialize is are similar to, but different from constructors that you may be familiar with in other languages, and we'll explore their role in helping enforcing variance in your programs. We look at how classes could collaborate, and we'll see how this could help you decompose the problem space into manageable elements. We'll discuss techniques for separating your classes. Public AP eyes from their implementation details. We'll see how you can painlessly mix object oriented design with designs based around functions, and in doing so hopefully get a deeper insight into pythons. Everything is an object approach. We'll explore how classes intersect with pythons, notion of duck typing, and we'll look at the basics of class inheritance in python.

Classes

[Autogenerated] as we've seen all objects and python have a type, and when we report that type using the type built in function, the result is couched in terms of a class. Classes are a means of defining the structure and behavior objects at the time we create the object. Generally speaking, the type of an object is fixed throughout its lifetime. As such classes act as a sort of template or pattern according to which new objects are constructed. The class of an object controls its initialization and which attributes and methods are available through that object, for example, on a string object the methods we can use on that object such a split are defined in the struck class. Classes are key piece of machinery for object oriented programming. And although it's often true that Opie 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 sharp

Defining Classes

[Autogenerated] by THING gives us the tools to define new classes, which could be completely novel or based on existing classes. Class definitions are introduced by the class keyword, followed by the class name by convention, new class names and python used Upper Camel case, sometimes known as Pascal Case, with a capital letter for each and every component word since classes air awkward to define it, the Ripple will be using a python module file to hold our class definitions. Let's start with the very simplest class to which will progressively add features for this module. I'll be using the pie charm Python I D so that it's easy to follow code examples. And their use in the rebel in this example will model a passenger aircraft flight between two airports by putting this code into air travel dot p y. The class statement introduces a new block, so we indented on the next line. Empty blocks aren't allowed, said the simplest possible class needs at least a do nothing past statement to be seen tactically admissible. Just as with death for defining functions, Class is a statement that can occur anywhere in a program and which binds a class definition toe a class name. When the top level code in the air travel module is executed, the class will be defined. We could now import our new class into the rebel and try it out from air travel Import flight. The thing we've just imported is the class object. Everything is an object in python, and classes are no exception. To use this class to mint a new object we call it's constructor, which is done by calling the class object just as we would have function, the constructor returns a new object which we hear assigned to the name F. If we use the type function to request the type of F, we get class air travel dot flight again. The type of f literally is the class.

Instance Methods

[Autogenerated] let's make our class a little more interesting by adding a so called instance method, which returns the flight number. Methods are just functions to find. Within the class and instance, methods are functions which could be called on objects or instances of our class, such as F. Instance. Methods must accept a reference to the actual instance on which the method was called as the first argument. And by convention, this argument is always called self. We have no way of configuring the flight number value yet, so we'll just return a constant string to find the function number within the body of flight and return this string sn 060 from a fresh Ruppel import flight from air travel. Constructive flight instance bound to the name F and call number on F notice that when we call the method, we do not provide the instance f for the actual argument self and the argument list. That's because the standard method invocation form with the dot F dot number is simply syntactic sugar for flight dot number of F. If you try the ladder, you'll find that it works as expected, although you almost never see this form used for real

Instance Initializers

[Autogenerated] this class isn't very useful because it can only represent one particular flight. We need to make the flight number configurable at the point that the flight is created. To do that, we need to write an initialization method if provided. The initialization method is called this part of the process of creating a new object when we call the constructor. The initial Isar method must be called Dunder innit delimited by the double underscores used for python runtime machinery. Like all other instance methods, the first argument to dunder in it must be self first defined under in it in the body of the flight class. In this case, we also pass a second argument to Dunder in it, which is the flight number Dunder, and it creates a new reference on self called underscore number and points it at the argument number. If you're coming from a Java Sea sharper c++ background, it's tempting to think of Dunder in it as being the constructor. This isn't quite accurate in Python, the purpose of Dunder and it is to configure an object that already exists by the time Dunder and it is called. The self argument is, however analogous to this in Java Sea sharp or C plus plus and 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 initialize er. Call it when present within the initial Isar. We assigned to an attribute of the newly created instance called Underscore number. Assigning to an object attribute that doesn't yet exist is sufficient to bring it into being. The initial Isar should not return anything. It's simply modifies the object referred to by self. Just as we don't need to declare variables until we create them. Neither do we need to declare object attributes before we create them. We choose underscore number with a leading underscore for two reasons. First, because it avoids a name clash with the method of the same name methods, air functions, functions or objects. And these functions are bound attributes of the object. So we already have an attribute called number and we don't want to replace it. Second, there's a widely followed convention that the implementation details of an object which are not intended for consumption or manipulation by clients of the object should be prefixed with an underscore. We also modify our number method to access the underscore number attributes and return it. Any arguments past the flight constructor will be forwarded to the initial Isar. So to create and configure our flight object, we can now call the flight class and pass in the flight number. As a string. We can access the flight number through the number method. We could also directly access the implementation details. Although this is not recommended for production code, it's very handy for debugging and early testing. If you're coming from a \_\_\_\_\_\_\_ and discipline language like Java or C sharp with public, private and protected access modifiers, pythons, everything is public approach can seem excessively open minded. The prevailing culture among Python East is is that we're all consenting adults here. In practice, the leading Underscore convention has proven sufficient protection even in large and complex piping systems. We've worked with people know not to use thes 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 initial izer of an object to establish so called class and variance Thean variants are truths about objects of that class should endure for the lifetime of the object. One such in variant for flights is that the flight number always begins with an upper case to letter airline code, followed by a three or four digit route number in Python. We can establish class and variance in the dunder innit method and raise exceptions if they can't be attained. We first used during slicing in straw. DOT is Alfa to verify that the 1st 2 characters of the flight number are alphabetic, raising a value error. If not, we then check if the same slices upper case by using straw is upper again, raising a value error? If not, finally, we use slicing. Straw is digit and the into instructor to verify that the rest of number comprises only digits and that it represents a value between zero and 9999 once again raising a value error if not for the first time in this course, we also see the logical negation operator, not ad hoc. Testing in the ripple is a very effective technique. During development, we can construct flights using valid flight numbers. If we try to use a flight number without letters at the front, we get a value error. Similarly, if the letters at the front aren't upper case, we get a value error with a different message. Here we try to use a flight number that doesn't have digits in the tail with predictable results, and in this case we get a value error because the flight number has too many digits. Now that we're sure the new flight objects have a valid flight number will add a second method to return just the airline code. Once the class and variance have been established, most query methods could be very simple in this case, simply returning a slice of the stored flight number.

A Second Class

[Autogenerated] one of the things we'd like to do with our flight is except 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 initial Isar creates four attributes for the aircraft registration number. A model name the number of rows of seats on the number of seats per row in the production code scenario. We could validate these arguments to ensure, for example, that the number of rose was not negative. The registration method returns the registration and the model method returns the aircraft's model. This is straightforward enough, but for the seating plan, we'd like something a little more in line with our booking system. Rose and aircraft are numbered from one, and the seats with any TRO are designated with letters from an alphabet which omits I tow. Avoid confusion with one well at a seating plan method, which returns the allowed rose and seats as a to pull containing a range object and a string of seat letters. This might be a bit much to take in all at once, so it's worth pausing for a second to make sure you understand how this function works. The Range call produces an interval sequence of road numbers up to the number of rows in the plane. The string and it sliced method. Return a string with one character per seat. Thes two objects, the range and the string are bundled up into a to pull. With that in mind, let's construct a plane with registration number G, U P T Model Airbus A 319 22 rows of seating and six seats. Paro. We can access its registration model and seating plan through the Associated instance methods. See how we use keyword arguments for the rose and seats for documentary purposes. Also recall the Rangers air half open, so the 23 is intentionally one beyond the end of the range.

Collaborating Classes

[Autogenerated] The Law of the Meter is an object oriented design principle that says you should never call methods on objects you received from other calls. Or, put another way on Lee. Talk to your friends. We'll modify our flight class to accept an aircraft object when it is constructed, and we'll follow the law of the meter by adding a method to report the aircraft model. This method will delegate aircraft on behalf of the client rather than allowing the client to reach through the flight and interrogate the aircraft object directly. We also added DOC string to the class thes work. Just like function and module Doc strings. We can now construct a flight in this case flight number be a 758 with a specific aircraft. Another Airbus A 319 notice that we construct the aircraft object and directly pass it to the flight constructor without needing an intermediate named reference to it. Once this is constructed, we can access the aircraft's model directly from our flight object

Moment of Zen

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

Booking Seats

[Autogenerated] 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 will 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 seed is unoccupied, it will contain none. We initialize the seating plan in the flight initialize er. Using this fragment in the first line, we retrieve the seating plan for the aircraft and used to pull unpacking to put the road 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 rowing to seize Air 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 can captain ate another list containing one entry for each real row in the aircraft. This list is constructed by a list comprehension which generates over the rose objects, which is the range of road numbers retrieved from the Underscore aircraft on the previous line. We're not actually interested in the road number, since we know it will match with the list index in the final list. So we discard it by using the \_\_\_\_\_ underscore variable the item expression. Part of the list Ex Russian is itself a comprehension, specifically, a dictionary comprehension. This iterated over each letter for the row and creates a mapping from the single letter string to none to indicate an empty seat. We use a list comprehension rather than list replication with the multiplication operator because we want a distinct dictionary object to be created for each row before we go further. Let's test our code in the rebel would create flight be a 758 from an Airbus A 319 Just like before, thanks to the fact that everything is public, we can access implementation details during development, and it's clear enough that we're doing so since the leading underscores remind us what's public and what's not. That's accurate but not particularly beautiful. Let's try again with pretty print, import the P print function from the P print module and rename it to \_\_\_ \_\_\_, Then use peopIe to print the seeding structure. Perfect. Now we'll add behavior to flight to allocate seats. Two passengers To keep this simple, a passenger will simply be a string name. Most of this code is validation of the seat designator, and it contains some interesting snippets methods. Air functions. So deserve doc strings, too. We get the seat letter by using negative indexing into the seat string. We test that the seat letter is valid by checking for membership of seat letters. Using the in membership testing operator, we extract the road number using string slicing to take all but the last character. We try to convert the road number sub string to an integer using the constructor. If this fails, we catch the value error and, in the handler, raise a new value error with a more appropriate message. Payload. We conveniently validate the road number by using the in operator against the Rose object, which is a range. We could do this because range supports the container protocol. We checked that the requested seat is unoccupied, using an identity test with none. If it's occupied, we raise the value error. If we get this far. Everything's in good shape and we can assign the seat. It also contains a bug that will discover soon enough trying our seat Allocator at the rubble. We first constructive flight. We then put pythons be DFL \_\_\_\_\_ Van Ross, Um, and seat 12 A. Early on in your object oriented python career, you're likely to see type error messages like this. Quite often, the problem has occurred because we forgot to include these self argument in the definition of the allocate seat method. Once we fix that, we can try again, construct the flight object allocate, see 12 8 to \_\_\_\_\_, attempt to allocate 12 8 to Rasmus Larry Dorf and see that it raises a value error to prevent us from assigning two people to the same seat. Let's see Bjarne Astrue Strip and undersea Ellsberg next to one another on Row 15. When we try to put Yukihiro Matsumoto in E 27 we get evaluate our telling us that we've used an invalid seat letter. We'll put John McCarthy and Rich icky in Row one, but we'll get yet another value error when we try to put Larry Wall in seat D d Now we can see our seating chart using the P print function. The Dutchman is quite lonely there on road 12. So we'd like to move him back to Row 15 with the Danes to do so, we'll need a relocate passenger method.

Methods for Implementation Details

[Autogenerated] first will perform a small refectory and extract the seat designator parsing invalidation logic into its own method. Underscore Parks Seat. We use a leading underscore here because this method is an implementation detail. The new Underscore parse eat method returns a tuba with an integer road number and a seat letter string. This leaves allocate seat much simpler, since it Condell gate most of its work to underscore parse, eat notice that method calls within the same object, such as this call to underscore Par seat also require explicit qualification withy self dot prefix. Now we've laid the groundwork for our relocate passenger method. This parses invalidates the front seat and two seed arguments and then moves the passenger to the new location. It's also getting tiresome, recreating the flight object each time so we'll add a module level convenience function for that, too. We'll call the function make flight First, it will construct a flight object with disassociated aircraft. Then we'll allocate a seat to \_\_\_\_\_ Van Ross. Um, be honest. Truth strip undersea Ellsberg, John McCarthy and Rich Hickey. Finally, we return the flight. It's quite normal to mix related functions and classes in the same module like this now from the rebel, we can import the make slight function and use it to replace the boiler coat we've been using to construct flight objects. 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 pythons dynamic type system that allows us this very loose coupling between code. Let's get on and move \_\_\_\_\_ back to Row 15 with this fellow Europeans, we'll use flights relocate passenger method to move him from 12 8 to 15 d. If we print the seating chart, we can see that he's now in the right place. It's important during booking to know how many seats are available to this end, will write a numb available seats method. This is achieved using two nested generator expressions. The outer expression filters for all rose, which are not none. This excludes are \_\_\_\_\_ first row. The value of each item in the outer expression is the sum of the number of none values a TRO. We split this outer expression over three rows to improve readability. This inner expression iterated over values of the dictionary and adds one for each. None found. We can make a flight and ask how many seats it has available. We can verify this result by multiplying the never seats Paro six but the number of rows on the plane 22 subtracting the number of occupied seats. Five, resulting in 127 the same number we got above.

Object-Oriented Design with Function Objects

[Autogenerated] Now we'll show how it's quite possible to write nice object oriented code without needing classes. We have a requirement to produce boarding cards for our passengers in alphabetical order, however, we realize that by following the engineering principle of separation of concerns, the flight class is probably not a good home for details of printing boarding passes. We could go ahead and create a boarding card printer class, but that's probably over killed. Remember that functions are objects to and are perfectly sufficient from many cases. Don't feel compelled to make classes and objects without a good reason. Rather than have a card printer query, all the passenger details from the flight will follow. Another object. Ori to design principle of tell, don't ask and have to flight tell a simple card printing function. What to do First, The card printer, which is just the module level function. We first grade the primary output string from the boarding card. The new python feature here is the use of line continuation backslash characters, which allow us to split a long statement over several lines. This is used together with implicit, shrinking cat nation of adjacent strings to produce one long string with no line breaks. We then measure the length of the output line and build some banners and borders around it. Finally, we can. Captain ate the lines together using the joint method called on a new line separator before printing the whole card, followed by a blank line. Note that the card printer doesn't know anything about flights or aircraft. It's very loosely coupled. You could probably easily envisage in HTML card printer that has the same interface to the flight class. We had a new method. Make boarding cards, which accepts the card printer. This calls the implementation detail underscore passenger seats. Method sorts the results and loops over the resulting passenger seat. Two poles for each of these, then calls the card printer that was passed in the Underscore passenger seats. Method is, in fact, a generator function, which searches all seats for occupants, yielding the passenger in the seat number as they're found, it first determines the number of rows and seat letters in each row. Then it loops over the road numbers and for each row loops over the seat letters. It finds the passenger in that seat, and if the passenger is not none yields the passenger and seat information. Now, if you run this in the rebel, we could see that the new boarding card printing system works. We import the consul card printer and make flight functions, make a new flight object and then print the boarding cards with the consul card printer function.

Polymorphism and Duck Typing

[Autogenerated] Polymorphism is a programming language feature which allows us to use objects of different types through a uniform interface. The concept of polymorphism applies to functions and more complex objects. We've just seen an example of polymorphism with the card printing example. The make boarding card method didn't need to know about an actual or, as we say, concrete card printing type on Lee. The abstract details of its interface, essentially just the order of its arguments. Replacing our consul card printer with the putative 80 Mel card printer would exercise polymorphism, polymorphism and python is achieved through duck typing. Duck typing is in turn named after the duck test attributed to James Whitcomb 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 wherein Objects Fitness for a particular uses only determined it run Time is the cornerstone of pythons object system. This is in contrast to statically type languages where compiler determines if an object can be used, and in particular it means that an object suitability is not based on inheritance, hierarchies based classes or anything except the attributes an object has at the time of use again. This is in contrast of languages such as Java, which depend on nominal sub typing through inheritance from based classes and interfaces. We'll talk more about inheritance in the context of python shortly. Let's return to our aircraft class. The design of this class is somewhat flawed in that objects in san seated using it depend on being supplied with the seating configuration that matches the aircraft model, which for the purposes of this exercise, we assume is fixed for model. Better and simpler, perhaps, is to get rid of the aircraft class entirely and make separate classes for each specific model of aircraft with a fixed seating configuration. Here's an Airbus A 3 19 and Boeing 777 thes two aircraft classes have no explicit relationship to each other or to our original air craft class, beyond having identical interfaces to each other. With the exception of the initial Isar, which now takes fewer arguments. As such, we can use these new types in place of each other. Let's change our make flight method into make flights to use them. It will first construct flight be a 758 with an Airbus A 3 19 and allocate some seats. Two passengers. It will that construct Flight a F 72 with the Boeing 777 and seats and passengers there finally returned both flights as a to pull the different types of aircraft. Both worked fine when used with flight because they both quack like ducks or fly like planes or something. We import everything from air travel and then make the flights binding them Center names F and G. We could see that F has model Airbus A 319 and G is a Boeing 777 F S 127 seats available, while the much larger Boeing has 491 seats available. We can't, of course, relocate passengers on the flights and make boarding cards. As before. Duck typing and polymorphism is very important in python. In fact, it's the basis for the collection protocols we discussed, such as it aerator miserable and sequence

Inheritance and Implementation Sharing

[Autogenerated] inheritance is a mechanism whereby one class, the sub class, could be derived from another class, the base class allowing us to make behavior more specific in the sub class and nominally typed languages such as Java. Class based inheritance is the means by which runtime polymorphism is achieved. Not so in python as we've just demonstrated. The fact that no python method calls or attributes lookups are bound to actual objects until the point at which they're called a quality known as late binding means. We can attempt polymorphism with any object, and it will succeed if the object fits. Although inheritance and python can be used to facilitate polymorphism after all, Dr classes will have the same interfaces based classes. Inheritance and python is mostly useful for sharing implementation between classes. As usual, this will make much more sense with an example. We would like our aircraft classes Airbus A 3 19 in Boeing 777 to provide a way of returning the total number of seats well at a method called numb seats to both classes. To do this, the implementation could be identical in both cases, since it could be calculated from the seating plan Unfortunately, we now have duplicate code across two classes, and as we add more aircraft types, the code duplication will only worsen. The solution will look out here is to extract the common elements of Airbus A 3 19 in building 777 into a base class from which both aircraft types will derive. Let's recreate the class aircraft this time with the goal of using it as a base class. It contains just the method we want to inherit into the derived class numb seats. This class isn't usable on its own because it depends on a method called seating plan, which isn't available in aircraft. Any attempt to use aircraft stand alone will fail. Here we construct an aircraft and ask for its number of seats. Since this aircraft instance doesn't have a seating plan method, we get an attribute error. The class is abstract and so far as it's never useful. To instance, he ate it alone. So how can we use aircraft meaningful E. That's where derived classes come in. We specify inheritance and python using parenthesis containing the base class name immediately after the class name in the class statement. Here's the Airbus class now inheriting from aircraft, and this is the Boeing class. Let's exercise them at the rebel now. Our Airbus A 3 19 objects have a numb seats method. As two instances of Boeing 777 we could see that both aircraft subclass is inherited. The NUM seats method, which now works as expected because the cult a seating plan to find on both sub classes, is successfully resolved on the self object at runtime. Now that we have the base aircraft class, we can hoist other common functionality into it. In this case, initialize ER and registration methods are identical between the two subtypes. Now the derived classes only contained 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

[Autogenerated] classes are an important tool for any python programmer, and this module introduced the essentials of how classes work in Python, while there are certainly more to know about classes and python much more. In fact, for many Python users, this may be all that you need to know to get your work done. Indeed, if you follow our advice to reserve classes for when other simpler techniques fall short, you may be surprised at how infrequently you really need them at all. In this module, we learned that all types and python have a class classes to find 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, and python classes are defined using the class keyword faulted by the class name, which is in Camel case, Instances of the class are created by calling the class as if it were a function instance. Methods are functions to find inside the class, which should accept an object instance called self, as the first parameter methods are called. Using the instance dot methods. Syntax, which is syntactic sugar for passing the instance as the formal self argument to the method. An optional special initialize ER method called Dunder. And it could be provided which is used to configure the self object at creation time. If present. The constructor calls the Dunder innit method Dunder, and it is not the constructor. The object has been constructed by the time the initial izer is called. Arguments past the constructor are forwarded to the initial izer 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 and variance should be established in the initial Isar. If THEAN variance can't be established, raise exceptions to signal failure. Methods have doc strings just like regular functions. Classes can have doc strings. Even within an object method. Calls must be proceeded with self. You can have as many classes and functions and module as you wish related classes and global functions are usually grouped together this way. Polymorphism and python is achieved through duck typing, where attributes and methods are only resolved. That point of views late binding polymorphism and python does not require shared based classes or named interfaces. Class, inheritance and python is primarily useful for sharing implementation rather than being necessary for polymorphism. All methods are inherited, including special methods like the initial izer. Along the way, we found that strings support slicing because the implement thes sequence protocol following the law of the Meter can reduce couple ING weaken nest comprehension sze. It can sometimes be useful to discard the current item into comprehension, using a \_\_\_\_\_ reference conventionally the underscore. When dealing with one based collections, it's often easier just to waste one list entry. Don't feel compelled to use classes when a simple function will suffice. Functions are also objects. Complex comprehension. Zor generator expressions could be split over multiple lines to aid readability. Statements can be split over multiple lines using the backslash line. Continuation character used this feature sparingly and only when it improves readability. Object oriented design where one object tells another information could be more loosely coupled than those were one object queries. Another tell. Don't ask. In the next module of core python getting started, we'll take a look at how to work with file I owe in python. Both were working with text as well as binary data. As part of this will introduce you to the with block pythons approach to managing resources like files. Thanks for watching and we'll see you in the next module.

File IO and Resource Managements

Overview

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

Opening Files

[Autogenerated] toe open a file in Python we call the built in open function. This takes a number of arguments, but the most commonly used. Our file the path to the file, and this is required mode. Read, write, penned and binary or text. This is optional, but we recommend always specifying it for clarity, explicit is better than implicit. Encoding. If the file contains encoded text data, which encoding to use, it's often a good idea to specify this. If you don't specify it, Python will choose the default encoding for you at the file system level files contain on Lee a series of bites Python distinguishes between files opened in binary and text mode, even when the underlying operating system doesn't files opened and binding remote return and manipulate their contents as bites. Objects without any decoding binary mode reflects the raw data in the file. Ah, file opened in text mode treats its contents as if it contains text strings of TheStreet. Type the raw bites having been first decoded, using a platform dependent encoding or using the specified encoding if given by default. Text mode files also engage support for pythons, a universal new lines this causes translation between a portable single new line character in our program. Strings slash end in a platform dependent new line representation of the raw bytes stored in the file system, for example. Carriage return new line slash our slash in on Windows. Getting the encoding right is crucial for correctly interpreting the contents of a text file, which is why we labor the point. If you don't specify in encoding, Python will use the default from sister dot get default Encoding. Ours is utf eight, but there's no guarantee that the default, including on your system, is the same as the default, including on another system with which he was to exchange files. It's better for all concerned to make a conscious decision about the text. Two bites and coding. You can get a list of supported Texan coatings in the python documentation

Writing Text

[Autogenerated] Let's start by writing some texts to a file will be explicit about using the utf eight encoding. Because we have no way of knowing what your default and coding is, we'll use keyword arguments to make things clearer. Still well assigned to F. The result of calling open the first argument is the file name. The next argument is a string contending letters with different meanings. In this case, W means right and T means text. All mod strings should consist of a read, write or penned mode. One of the proceeding should be combined with a selector for text or binary mode, so typical mode strings might be W B right binary or 80 upend text. Although both parts of the mod code support defaults, we recommend being explicit for the sake of readability. The exact type of the object returned by open depends on how the file was opened. Dynamic typing in action. But for our purposes, it's sufficient to know that the object returned is a file like object, and as such, we can expect it to support certain attributes and methods. We've shown previously how we can request help for modules and methods and types But in fact, we can request help on instances to this makes sense when you remember that everything is an object. Let's call help on our file F browsing through the help, we can see that F supports a method right? Quit the help with Q and continue at the rebel. We'll write the string. What are the roots that clutch to our file? The call returns the number of code points or characters written to the file a few more lines. What branches grow at 19 code points and out of this stony rubbish with 27 code points. Note that it's the caller's responsibility to provide new line characters when they're needed. There's no right line method on python file objects. When we finished writing, we should remember to close the file by calling the clothes method. Note that when we close the file, all the contents become visible to other programs. Closing files is important if you now exit the rebel and look at your file system on UNIX with L s Dash l. You should see the wasteland dot T X T file with 78 bites on Windows. You could do the same with D I r. Here you should see wasteland dot t x t with 79 bites, one more than on UNIX. Because pythons universal new line behavior for files has translated the line ending to your platforms native endings, the number returned by the right method is the number of code points or characters in the string passed, too right, not the number of bites written to the file after Encoding and Universal New Line Translation. In general, when working with text files, you cannot sum the quantities returned by right to determine the length of the file and bites.

Reading Text

[Autogenerated] to read the file back. We use open again, but pass a different mode. String open wasteland dot t X t mode. Artie Encoding utf eight. In this case, we used the arty mode for read text. If we know how many bytes to read, or if you want to read the whole file, we can use read. Looking back through our rebel, we can see that the first rite was 32 characters long. We can read that back with the call to the reed method in text mode. The Reed method accepts the number of characters to read from the file, not the number of bites. The coal returns, the text and advances the file pointer to the end of what was red. The return type is struck because we opened a file in text mode to read all the remaining data. In the file, we can call read without any argument, giving us parts of two lines in one string. Note the new Line character in the middle at the end of the file. Further calls to read, return and empty string. Normally, when we have finished reading a file, we would close it. But for the purposes of this exercise will keep the file open and move the file point or back to the beginning of the file using the secret method with a zero offset from the start of the file. The return value is the new file pointer position. As a quick aside, it's important to note that for text mode, files seek cannot be used to move to an arbitrary offset, in particular when seeking from the beginning of the file, as we do here, the only legal values for offset are zero, and any value returned by the files tell method. Any other values were result in undefined behavior. Using read for Texas. Quite awkward and thankfully, Python provides better tools for reading text files line by line. The first of these is the reed line function. The first call to read line reads to the first new line. The second call reads to the end of the file, the returned lines of terminated by a single new line character. If there is one present in the file, the last line here does not terminate with a new line because there's no new line sequence. At the end of the file, you shouldn't rely on the string returned by read line being terminated by a new line. Again, the universal New Line support will have translated to slash in from whatever the platform, native new Line sequences. Once we reach the end of the file, further calls to read, line return and empty string. Let's rewind again with G dot seek zero sometimes, and 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 read lines method. This is particularly useful if parsing the file involves hopping backwards and forwards between lines. It's much easier to do this with a list of lines and then with file streams of characters. This time, we'll close the file before moving on.

Appending Text

[Autogenerated] sometimes we'd liketo appended to an existing file. We could do that by opening the file with Mode A, which opens the file for writing a pending 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. Open wasteland dot t x t mode 80 Encoding utf eight. Although there's no right line method in python, there is a right lines method which writes an interval Siri's of strings to a stream. If you want line endings on your strings, you must provide them yourself. This seems odd at first, but it preserves symmetry with read lines while also giving us the flexibility for using right lines to write any honorable Siri's of strings to a file. Son of man, you cannot say or guess for, you know only a heap of broken images where the sun beats noticed that only three lines are completed here. I say completed because the file we repenting to doesn't end with a new line

Iterating over Files

[Autogenerated] the culmination of these increasingly sophisticated text file reading tools. It's the fact that file objects support the generator protocol, with the generation yielding the next line in the file. This means that they could be used in four loops and any other place where innit aerator can be used. At this point, we'll take the opportunity to create a python module called files dot p y. We can call this directly from the system command line, passing the name of our text file Python three files dot pie wasteland dot t x t. The double line spacing occurs because each line of the poem is terminated by a new line and then print ads its own. To fix that, we could use the strip method to remove the white space from the end of each line prior to printing. Instead, we'll use the right method of these standard out stream. This is exactly the same right method we used to write to the file and can be used because the stream is a file like object. We get hold of a reference to the standard out stream from the cyst module. Replace print line with sys dot standard out dot right line rerunning, we get the poem printed without extra blank lines. Now, alas, it's time to move on from one of the most important poems of the 20th century to get to grips with context managers.

Closing Files with Finally

[Autogenerated] for the next demonstration, we're going to need a day to file containing some numbers. We'll write a sequence of numbers called recommends sequence to a text file with one number per line. Bracamonte sequence itself isn't important to this exercise. We just needed a way of generating numeric data, so we won't be explaining the sequence. Generator. Feel free to experiment, though. The module contains a generator for yielding the recommended embers and a function which writes the start of the sequence to file using the right lines method. A generator expression is used to comfort each number two a string and add a new line. Better tools dot i slice is used to truncate the otherwise infinite sequence. We'll write. The 1st 1000 recommend numbers to have filed by executing the module, passing the file name in Siri's length as command line arguments heightened. Three. Recommend a P Y. Recommend dot dat 1000 will not take a complimentary module. Siri's dot p y, which reads this data file back in. We simply read one line at a time from the open file, stripped the new line with the call to the strip string method and converted to an integer running it from the command line. Everything works as expected. Now let's deliberately create an exceptional situation open recommend at that in a text editor and replace one of the numbers with something that isn't a string of fight integer. Save the file and rerun. Thean Constructor raises a value error, which is Unhand Aled. And so the program terminates with Stack trace. One problem here is that our F dot close call was never executed. To fix that, we can insert a try. Finally, block, put a try before the open call, inventing everything up to the close. Cole then put a finally before the close call in denting that as well. Now the file will always be closed. Doing so opens up the opportunity for another re factoring. We can replace the four loop with the list comprehension and return this list directly returned open racket into flynt dot strip for line in F close bracket. Even in this situation, close will be called. The finally block is called. However, the tribe block is exited

With-blocks

[Autogenerated] up to now are examples of all followed a pattern. Open a file work with the file closed the file. The clothes is important because it informs the underlying operating system that you're done working with the file. If you don't close the 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 the clothes, we want a mechanism that enforces that and make sure that it happens even if we forget this need for resource cleanup is common enough that python implements a specific control flow structure called with blocks to support it with blocks can be used with any object which supports the context manager protocol. And this includes the file objects returned by open, exploiting the context manager nature of the file object and using the with block our read Siri's function can become open the file with the with block and return the list comprehension. We no longer need to call close explicitly because the with construct will call it for us win, and by whatever means execution exits the block. Now we can go back and modify our recommends. Siri's writing program to use a with block, too. This again removes the need for the explicit clothes.

Moment of Zen

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

Binary Files

[Autogenerated] to demonstrate handling of binary files. We need an interesting binary data format. The BMP file format contains device independent bit maps, and it's simple enough that we can make a BMP file writer from scratch. In this session, the code will be placed in a module b m p dot p y. In a straightforward from a file handling point of view for simplicity's sake, we have decided to deal with eight bit grayscale images, which have the nice property that they're one bite per pixel. The right grayscale function accepts to arguments the file name and a collection of pixels. As the doctoring points out, this collection should be a sequence of Inderal. Siri's of integers. A lists of lists of objects would do just fine. Each end is a pixel value from 0 to 2 55 Each inner list is a row of pixels from left to right, and the outer list is a list of pixel rose from top to bottom. The first thing we do is figure out the size of the image by counting the number of rose to give the height and the number of items in the zero throw to get the width, we assume, but don't check that All Rose have the same length in production code. That's a check we'd want to make next. We opened the file for right. Invite every mode using the W B mode string we don't specify and encoding that makes no sense for roll 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 bite sequence B M toe identified as a BMP file. We used the right method, and because the file was opened in binary mode, we must pass a bites object. The next four bites should hold a 32 bit into JER containing the file size. We don't know that yet. We could have computed it in advance, but we'll take a different approach of writing a placeholder value for now, then returning to this point later to fill in the details. To be able to come back to this point, we used the tail 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 the variable, which will act as a sort of bookmark. We write 40 bites as the placeholder, using escape syntax to specify the zeros. The next two pairs of bites are unused, so we just write a zero bites of them, too. The next four bites are for another 32 bit indigent, which should contain the offset and bites 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 four bite placeholder. We'll return here shortly when we know more. The next section is called The Image Header. The first thing we have to do is write the length of the image as a 32 bit integer. In our case, the header will always be 40 bites long. We just hardwire that as a hex. A decimal noticed that the BMP format is little Indian. The least significant bite is written first. The next four bites are the image with as a little Indian 32 bit integer, we call a module scope implementation detail function here called Underscore in 32 2 Bites, which converts an Inter object into a bites object containing exactly four bites. We then use the same function again to deal with the image height. The remainder of the header is essentially fixed for eight bit grayscale images, and the details aren't important here, except to note that the whole header does, in fact, total 40 bites. Each pixel in an eight bit BMP image is an index into a color table with 256 entries. Each. Entry is a four bite be gr color for grayscale images. We need to write 256 4 bite gray values on a linear scale. This snippet is fertile ground for experimentation, and a natural enhancement to this function would be to be able to supply this pallet separately as a function argument. At last, we're ready to write the pixel data, but before we do, we make a note of the current file pointer offset using tell as this was one of the locations we need to go back and fill in later. Writing the pixel data itself is straightforward enough. We used the reversed built in function to flip the order of the Rose BMP images are written bottom to top for each row. We simply passed the edible Siri's of editors to the Bites constructor. If any of the integers air out of the range 022 55 the constructor will raise a value error. Each row of pixel data in a BMP file must be a multiple of four bites, long, irrespective of image with. To achieve this, we compute the number of padding bites required by subtracting. The road length module is four from four. This value is used with repetition. Operator apply to a single zero bite to produce a bites object containing 012 or three bites. We write this to the file to terminate each row after the pixel data. We're at the end of the file. We undertook to record this offset value earlier, so we record the current position using tell into an end of file bookmark variable. Now we can return and to fill our promises by replacing the placeholder offsets we recorded with the real values. First, the file length. To do this, we seek back to these size bookmark. We remembered back near the beginning of the file and write the size stored in E O f bookmark as a little Indian, 32 bit integer using our underscore into 32 2 bites function. Finally, we seek to the pixel data offset placeholder bookmarked by pixel offset bookmark and write the 32 bit into your 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 UN buffered rights to the file system.

Bitwise Operators

[Autogenerated] dealing with binary files often requires pulling apart or assembling data at the bite level. This is exactly what our underscore in 32 2 bites 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 bit wise shift and bit wise and operators to extract individual bites from the integer value. Note that bit wise and uses the ampersand symbol to distinguish it from the logical and which is the spelled out word. And the double arrow is the right shift operator, which shifts the binary representation of the indigent right by the specified number of bits. The routine shifts. He entered your 12 and three bites to the right before extracting the least significant bite with the bit wise. And at each position, The four resulting integers are used to construct a tupelo, which is then passed to the Bites constructor to produce a four bites sequence

Pixel Data

[Autogenerated] in order to generate a BMP image file, we're going to need some pixel data. We've provided a simple module, fractal dot p Y, which produces pixel values from the iconic Mandel brought set Fractal. We're not going to explain the fractal generation coded detail, still less the math behind it. But the code is simple enough and doesn't rely on any python features we haven't encountered previously. The key takeaway is that the mantle brought function uses nested list Comprehension is to produce a list of lists of integers in the range of 022 55 representing an image of the fractal, the integer value. Each point is produced by the Mandal function. Let's fire up a ripple and used the fractal and BNP modules together. First, we use the mantle brought function to produce an image of 448 by 256 pixels. You'll get best results using images with the aspect ratio of 7 to 4. This last call may take a second or so. Our fractal generator is simple rather than efficient. We can take a look at the return to data structure, a list of lists of integers just as we were promised. Let's write those pixel values to a BNP file import bnp bmp dot right grayscale Mandel dot bmp pixels. Find the file and open it in an image viewer, for example, by opening it in your Web browser.

Reading Binary Data

[Autogenerated] we're not going to write a full blown BMP reader, although that would be an interesting exercise. We'll just make a simple function to determine the image dimension in pixels from a BMP file. We'll add the code into B m p dot p y. 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 for the 1st 2 magic bites that we expect in a BMP file. If they're not present, we raise a value error, which will, of course, caused the context manager to close the file. Looking back at our BMP writer, we can determine that the image dimensions are stored exactly 18 bites from the beginning of the file. We seek to that location and use the reed method to read two chunks of four bites each for the 2 32 bit integers, which represent the dimensions. Because we open the file and binary mode, read returns of bites object. We passed each of these two bites objects to another implementation detailed called Underscore bites to into 32 which assembles them back into an integer. The two inventors, representing image, width and height are returned. Is the to pull the underscore bites TN 30 to function uses bit wise, left shift and bit wise. Or which is the vertical bar or pipe Simple, together with indexing of the bites object to reassemble the integer note that indexing into a bites object returns an integer Let's use it. Bmp dot dimensions. Mandel dot b m p.

File-like Objects

[Autogenerated] there is a notion in python of file like objects. This isn't as for Melissa, specific protocol like sequence protocol is for the two pull like objects. But thanks to the polymorphism afforded by duck typing, it works well in practice. The reason is not closely specified is that different types of data streams and devices have many different capabilities, expectations and behaviors. So in fact, defining a set of protocols to model them would be quite complex without actually gaining us much in practice. Other than a smug sense of theoretical achievement, this is where the E A F p 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 Sikh method doesn't exist or does exist but doesn't behave as you expect, you might say if it looks like a file and reads like a file that it's a file. We've actually seen this already. The objects return to us when we opened files in text and binary mode, or actually of different types. Although both with definite file like behavior. There are other types in the Python Standard library, which implement file like behavior. We saw one of them in action back of the beginning of the course used to retrieve data from the U. R L on the Internet. Let's exploit this polymorphism across file like objects by writing a function to count the number of words per line into file and return that information as a list. We'll call it words per line. Accepting a single argument f L o. It'll return a list comprehension that texted length of line dot split for each line in the argument. Now we'll open a regular text file containing the fragment of T. S. Eliot's masterpiece we created earlier with open wasteland dot t x t as real file. And then we'll pass real file to our new function and display the results. The actual type of Rio file is underscore io text io wrapper, which is an internal python implementation detail. Well, now do the same. Using a file like object representing a web resource referred to by a girl. First we import you are all open from you're a lib dot request. We then open the resource using neural Open. Http 60 north dot com slash c slash t dot text As Web file. Count the words per line with words per line of Web file and display the results in this case, the type of what vial is. Http dot client dot http response. Quite a different thing from what we saw before, however, since both are filed like objects are function can work with both. There's nothing magical about file like objects. It's just a convenient and fair the informal description for a set of expectations we can place on an object which are exploited through duck typing.

Context Managers

[Autogenerated] the with statement construct can be used with any type of object which implements the context Manager protocol. We're not going to show you how to implement that in this course, but we will show you a simple way to make your own classes usable with the context manager. Using the code in fridge dot p y, we'll import raid into the rebel and go on the rampage from Fridge Import Raid. Now we raid the fridge for bacon. Importantly, we remembered to close the door so the food will be preserved until our next raid. However, what if we raid the fridge for deep fried pizza? This time we were interrupted by a health warning and didn't get around to closing the door. We can fix that by using a function called Closing in the Python Standard Library Context. Lib module. After importing the function, we wrap our refrigerator. Reiter, constructor coal in a call to closing, which wraps are object in a context manager that always calls the clothes method on the wrapped object before exiting. We use this object to initialize a with block. Now we execute a raid for spam. We see that the clothes is called twice are explicit. Culture closes unnecessary. So let's fix that up by removing the explicit call too close a more sophisticated implementation. We checked that the door was already closed and ignore other requests. Now, if we ill advisedly raid for deep fried pizza, we see that even though the health warning was triggered, the door was still closed for us to buy the context manager.

Summary

[Autogenerated] as you've seen, Python makes it easy to work with files or indeed, any file like object at many different levels from Beit level manipulation. The high level lines of encoded text and python provides excellent tools for doing the book. Keeping associated with files and other kinds of resource is use diligently. The tools and techniques we've covered here will help ensure the use files efficiently, safely and with predictable results in this module. We've learned that files are open using the built in open function, which accepts a file mode to control, read, write, upend behavior and whether the fire let's be treated as raw, binary or encoded text data protects data. You should specify a text encoding text files do with string objects and perform universal new Line translation and string encoding finery. Files do with bites objects with no new line translation or encoding. When writing files, it's our responsibility to provide new line characters for line breaks. Files should always be closed after use files provide various line oriented methods for reading and are also it aerators, which yield line by line files or context managers, and the with statement can be used with context managers to ensure that cleanup operations such as closing files are performed. The notion of file like objects is loosely defined but very useful in practice. Exercise E a. F p. To make the most of them context managers aren't restricted to file like objects. We can use tools in the context of standard library module, such as the closing rapper, to create our own context managers. Along the way, we found that helped convey used on instances of objects, not just types. Pythons supports bit wise operators and or and left and right shift well done on completing core python getting started. It took, you know, small amount of effort to get here, but we're confident that your hard work will pay dividends. When you start to use what you've learned. Python is a big language, and this course can really only hope to lay a foundation upon which you can start to build. We hope we've opened enough doors so that you can confidently and rapidly expand your python knowledge. Look out for other core python courses here on plural site, which build on the knowledge you've gained here, which explained the many other tools and abstractions provided by Python for managing complexity. Remember to check out our Python Craftsman Book series, which covers these topics in written form. Specifically, you'll find these topics covered in The Python Apprentice, the first book in the trilogy. We'll be back with more content for the ever growing Python language in Library. Please remember, though, that the most important characteristic of Python is that it's great fun to write Python software Happy programming.