Core Python: Functions and Functional Programming

by Austin Bingham and Robert Smallshire

Start Course

Bookmark Add to Channel Download Course

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

Course Overview

Course Overview

[Autogenerated] Hi, everyone. My name is Austin Bingham and welcome to my course core python functions and functional programming. I'm a founder and principal consultant at 60 North in Python. As with most programming languages, functions are a fundamental tool for managing program complexity. Understanding how to use functions fluently is a key skill on the road to mastering python, as is knowing alternative techniques for the cases where functions are insufficient for your needs. In this course, we'll look at some of the finer details of using functions as well as more general concepts that subsumed them. Some of the major topics will cover include the notion of cola bles, a generalization of functions that includes several other constructs and fighting lamb does a less powerful form of functions that don't require names, extended forms of pythons, formal argument and cooling syntax. The concept of closures whereby functions can retain access to objects that might otherwise be garbage. Collected Decorator's a powerful technique for augmenting functions without modifying the functions themselves. Tools for writing python in a functional style. Advanced use of comprehension sze, including nesting and the use of multiple inputs. By the end of this course, you'll have a deeper understanding of pythons sophisticated approach to calling functions, and you'll see that functions are actually just one aspect of a more general set of concepts in the language. Before beginning the course, you should be familiar with the basic use of functions and python, and you should be familiar with both comprehension tze and generator expressions. From here, you should feel comfortable diving into other core python language courses on robust resource in error handling, introspection, numeric types, dates and times. I hope you'll join me on this journey to learn more about functions in Python with E Corp. Python Functions and Functional Programming course at Coral site.

Function and Callables

Review of Functions

[Autogenerated] While it's critical to understand functions when programming and python knowing how to apply pythons broader, unified notion of call a bles is often the key to implementing. True, the elegant solutions to your problems in this module of core python functions and functional programming will introduce you to the flexibility, incision and expressiveness of callable objects in python. Well, briefly cover what we expect youto already know. In order to benefit from this course, we look at the concept of Call Abel's Pythons generalization of functions. We'll see that classes are culpable objects. We learn about Lambda as anonymous function like objects, and we'll see how to determine if an object is Call a ble. In this course, we'll assume that you're already familiar with many concepts related to functions, and we won't review them in any depth. In this section. We'll go over what we expect you to know already. First and fundamentally, you'll need to be familiar with the concept of free functions, thes air functions, which are to find at module or global scope. Similarly, though less critically, you should also be comfortable with methods that is functions enclosed within a class definition. You also need to be familiar with the two flavors of function arguments, positional and keyword positional arguments used in the call or associated with the formal arguments used in the definition. In order. Keyword arguments used the name of the actual argument at the call site to associate with the name of the formal argument in the definition and can be provided in any order so long as they follow any positional arguments. The choice of whether a particular argument is a positional or keyword argument is made at the coal site not at the definition a particular argument, maybe pass as a positional argument in one call, but as a keyword argument and another call. Furthermore, in the function definition, each argument may be given a default value. It's important to remember that the right hand side of these default value assignments is on

Lee evaluated once at the time the enclosing death statement is executed, which is typically when a module is first imported. As such, care must be taken when using mutable default values, which can inadvertently retain modifications between calls. Lastly, you need to be aware that just like almost everything else in python functions are first class which is to say they're objects which could be passed around just like any other object. You'll recall that the death keyword is responsible for binding a function object which contains a function definition to a function name. Here we create a simple function resolve, which is just a thin wrapper around a function from the Python Standard Library. _____ it module. Inspecting the Resolve binding shows that it refers to a function object and to invoke the function, we must use the post fixed parentheses, which are the function call operator in a very real sense than function. Objects are culpable objects in so far as we can call them if you find that you need to brush up on python basics before you start this course, you could always refer to our core python getting started course, which covers all of the prerequisites for this course. If you want to follow along with the examples yourself, you will need access to a working pipe than three system. For this course, the material represent will work in all recent versions of Python three. If you have a choice, you should probably get the most recent stable version at a minimum, you need to be able to run a python three rebel. You can, of course, use an I D if you wish, but we won't require anything beyond what comes with standard python distribution. Finally, we need to make quick note regarding terminology. In Python, many language features are implemented or controlled, using special attributes or methods on objects. Thes special methods are named with two leading and two following underscores. This has the benefit of making them visually distinct, fairly easy to remember and unlikely to collide with other names. This 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 referring to these special methods. Dunder is a portmanteau of the term double underscore, and we'll use it to refer to any method with leading and trailing. Double underscores. So, for example, when we talk about the method, underscore underscored Len Underscore underscore which, as you'll recall, is invoked by the Len function. Well, say Dunder Len, These kinds of attributes play a big role in this course, so we'll be using this convention frequently. If you'd like a book to support you as you work through the material in this course, you could check out the python journeymen in which you'll find the same material in written form by following the U. R L shown, you can obtain the book for a substantially discounted price. The Python Journeymen is the second book and our Python Craftsman Trilogy, the first book being the Python Apprentice and the third book Being the Python Master. Taken together, these contain material which correspond to our various python courses. Here on plural site. All three are available to plural site viewers at reduced prices.

Callable Instances

[Autogenerated] Occasionally we would like to have a function which maintains some state, usually behind the scenes between calls in general implementing functions for which the return value depends on the arguments to previous calls rather than just the current call is frowned upon and rightly so because of the difficulty of reasoning about and therefore testing and debugging such functions. That said, there are legitimate cases for retaining information within a function between calls, such as to implement a cashing policy to improve performance. In such cases, the values returned by the function, given particular arguments are not changed, but the time within which a result has produced may be reduced. There are several ways of implementing such state full functions in python, some of which will look at in later sections, but here will introduce the Python Special Method Dunder Call, which allows objects of our own design to become call a bles just like functions. To demonstrate will make a cashing version of our d. N s resolver in a file resolver dot p y notice that just like any other method Dunder call except self as its first parameter. Although we don't need to provide this woman you call it? Let's test this from the Python rebel. We must call the constructor of our class to create an instance object which, because we implemented the Dunder call method, could be called like a function. This is really just syntactic sugar for calling Dunder call directly, but we would never use that form in practice. Since resolve is an object of type resolver, we can retrieve its attributes to inspect the state of the cash which currently contains just a single entry. Let's make another call and see that the cash has grown. In order to convince ourselves that the cashing is working as designed, we could run some simple timing experiments using the Python Standard Library Time It multiple, which contains a handy timing function, also called time it for reasons we won't go into now, the time it function except to code snippets as strings, one of which is used to perform any set of operations and the other of which is the code for which the elapsed time will be reported. The function also accepts a number argument, which is the count of how many times the code under test will be run in our case of testing a cash. It's important that we set that toe one, since our code to be time will refer to names in the current name space that of the rebel. We must specifically import them from the rebel name space, which is called Dunder Main, into the name space used by time. It you can see here that the D. N s look up took around 1/100 of a second. Now execute the same line of code again, this time the time taken a short enough that Python reports it in scientific notation. Let's ask Python to report it without scientific notation using the strut dot format method, relying on the fact that the special underscore variable stores the previous report result. Now it's perhaps easier to see that the result has returned from the cache several 1000 times faster than before are callable. Object has been in Stan seated from a regular class, so it's perfectly possible to define other methods within that class, giving us the ability to create functions which also have methods. For example, let's add a method called clear to empty the cash, and another method called has host to query the cash for the presence of a particular host. Let's exercise our modified resolver in a fresh ripple session. First we import an instance she ate. A result are culpable. Then we can check whether a particular destination has been cashed, which it has not, although, of course, resolving that host by invoking our callable changes that result. Now. We contest that cash clearing works as expected, which, of course it does. So we see how the special Dunder call method could be used to define classes. Which one instance see it? It could be called using regular function Call syntax. This is useful when we want to function, which maintains state between calls and optionally needs to support attributes or methods to query and modify that state.

Classes Are Callable

[Autogenerated] it may not be immediately obvious, but the previous exercise also demonstrated a further type of call oval object and that is the class object. Remember that everything in python is an object, and that includes classes. We must take great care when discussing python programs, because class objects and instance, objects produced from those classes are quite different things. In fact, just as the deaf keyword binds a function definition to unnamed reference. So the class keyword binds a class definition toe unnamed reference. Let's start a new rebel session and give a practical demonstration using our resolve our class. When we asked the rebel to simply evaluate the imported name Resolver, the rebel displays a representation of the class object. This class object is itself call a ble. And of course, that is what we've been doing all along whenever we've called the constructor to create new instances, so we see that in python constructor calls are made by calling the class object. As we've seen any arguments past when the class object is called in this way will in due course, be forwarded to the dunder innit method of the class. If one has been defined in essence, the class object culpable is a factory function which, when invoked, produces new instances of that class. The internal python machinery for producing class instances is beyond the scope of this course, but we cover it thoroughly and later. Core python courses, knowing the classes are simply objects and that constructor calls are simply using class objects, is called levels. We can build interesting functions which exploit that fact. Let's write a function which produces a python sequence type, which will either be a to pull if a request in immutable sequence or a list. If we request a mutable sequence in the function, we just test a Boolean flag and bind either to pull or list, both of which are class objects. Toe R. C. L s reference using the assignment operator, which we then return. Notice that we must take care when choosing variable names, which refer to class objects not to use the class keyword as a variable name. Popular alternatives for variables referring to class objects are the abbreviation C. L s and the deliberate misspelling class with a K. Now we can use this function to produce a sequence class with the required characteristics We can then create an instance of the class by using the class object as a cola ble, in effect calling the constructor. As an aside, we'd like to introduce you to a new language feature called conditional expressions, which could evaluate one of two sub expressions depending on a boolean value, and they take the form. Result equals true value if condition else false value. This in taxes perhaps surprising with the condition being placed between the two possible result values. But it reads nicely, is English and emphasizes the true value, which is usually the most common result. We could use it to simplify our sequence class function down to a single expression, Obviating the need for the intermediate binding variable C. L s and retaining a single point of return. Conditional expressions could be used. Any place a python expression is expected, they're full syntax and somewhat tortured. History is covered in pep 308

Lambdas

[Autogenerated] Sometimes we want to be able to create a simple call oval object, usually to pass directly to a function without the bureaucratic overhead of the deaf statement and the code block it introduces. Indeed, in many cases it is not even necessary for the cola ble object to be bound to a name. If we're passing it directly to a function, an anonymous function object would suffice. This is where the Lambda Construct comes into play and used with care. It could make for some expressive and concise code. However, as with comprehension, Sze excessive use of land is conserved obvious. Kate rather than clarify code running counter to python IQ principles, which value readability so highly so, take care to deploy them wisely. If you're wondering why a technique for making callable objects from Python Expressions is named after the 11th letter of the Greek alphabet, the origins go back to a foundational work in computing science in 1936 by Alonzo Church. Predating electronic computers Even who developed the Lambda Calculus, which forms the basis of the functional programming techniques used in languages such as list Ah good example of a python function that expects a call Abel is the sorted built in for sorting it horrible Siri's, which accepts an optional key argument, which must be a culpable. For example, if we have a list of names and strings and we wish to sort them by the second name, we need to pass the gullible as the key argument assorted, which will extract the second name. To do this, we can use a lambda to produce such a function without the bother of needing to think up a name for it. We first create a list of scientists, and then we call sorted with the key argument, referring to our

Lambda. Here are Lambda, except a single argument called name and the body of the Lambda after the colon calls the stroh dot split method and returns the last element of the resulting sequence. Using negative indexing in isolation, the Lambda is just this part. Lambda is itself an expression which results in a callable object. We can see this by binding the result of the land expression toe unnamed reference. Using assignment, we can see that the resulting object is a function and that it is indeed culpable, like a function creating a cola ble function this way. Using a lambda and binding a name through assignment is equivalent to defining a regular function using death like this. This allows us to point out the differences between Lambda is and regular functions or methods. Death is a statement which defines a function and has the effect of binding it to a name. Lambda is an expression which returns a function object. Regular functions must be given a name, whereas Lambda Czar Anonymous. The argument list for functions is too limited by parentheses and separated by commas. The argument list for Lambda is is terminated by a colon and separated by commas. Lend arguments are without enclosing parentheses. Versions of python predating python three have special handling of two poles using to pull unpacking in the argument list. This confusing feature has been removed from Python three. So in Python three Code, there are never any parentheses between the Lambda keyword and the colon. After the argument list, both regular functions and Lambda Support Zero or more arguments. A zero argument function uses empty parentheses for the argument list, whereas a zero argument lambda places the colon immediately following the Lambda keyword, the body of a regular function is a block containing statements, whereas the body of a lambda is an expression to be evaluated and the Lambda body can contain only a single expression. No statements. Any return value from a regular function must be explicitly returned using the return statement. No return statement is needed or indeed allowed in the limited body. The return value will be the value of the supplied expression. Unlike regular functions, there's no simple way to document the Lambda with the DOC string. Regular functions could easily be tested using external testing tools because they could be fetched by name. Most Landis can't be tested in this way simply because they're anonymous and can't be retrieved. This points the way to a guideline. To keep your land is simple enough that they're obviously correct by inspection to determine whether an object is culpable, you can simply pass it to the built in function callable, which returns true or false. So as we've seen, regular functions are culpable. Lambda Expressions Air call a ble class objects are culpable because calling a class invokes the constructor methods air call noble and instance, objects can be made culpable by defining the Dunder call method, where as many objects such a string instances are not culpable

Summary

[Autogenerated] Let's review what we've covered in this module. We reviewed the basics of functions in Python. We saw how to use the Dunder call method to create callable instances. We investigated how that lets us associate state and methods with culpable objects. We saw that classes are themselves call noble, a point which seems somewhat obvious in retrospect, and that calling them produces new instances of the class we looked at. How to define Lambda is unnamed callable objects that can contain a single expression, and we closely examined the differences between Lambda is and functions and hopes of understanding when to use each. Finally, we saw how to use the cola ble function to determine if an object is call a bowl. Along the way, we were introduced to Python's conditional expression, and we explored the notion of classes as objects in the next module of core python functions and functional programming will learn about pythons, extended argument, syntax, a sophisticated means for controlling what kinds of arguments could be passed to a call a ble as well as its extended call syntax for passing sequences to call a bull's thanks for watching and we'll see you in the next module

Extended Argument and Call Syntax

Extended Formal Argument Syntax

[Autogenerated] in this module of core python functions and functional programming will look at pythons, extended argument, syntax for accepting arbitrary numbers of positional arguments and arbitrary keywords. We'll also look at how to specify positional Onley and keyword only arguments as a compliment to extended argument syntax. We'll also look at pythons extended call syntax, and we'll discover pitons idiom for forwarding arbitrary function arguments. We've already been using functions which support extended arguments in tax, although you may not have realized it, for example, have you wondered how it is possible for print to accept a 012 or, in fact, any number of arguments? Another example we've seen is the use of the straw dot format method, which in except arbitrary, named arguments corresponding to the format. Placeholders in the string in this section will learn how to define functions or, more generally, Cola Bols, which can accept arbitrary positional or keyword arguments. Let's start with positional arguments, drawing an example from geometry. Let's ride a function which can return the area of a two dimensional rectangle, the volume of a three dimensional cube, old or indeed, the hyper volume of an n dimensional hyper que Boyd such a function needs to accept an arbitrary number of America arguments and multiply them together. To do this, we use a special argument syntax where the argument name is prefixed by a single asterisk before we implement the computation will simply print out the value of our eggs and its type Using the built in type function noticed that the asterisk does not form a part of the argument name and the argument name we have chosen here are eggs is widely used in this case by convention, although it is by no means necessary. Colloquial e. This form is called Star our eggs. Now let's call our function a few times. We can see that our eggs this past as a to pull, which contains the function arguments. Knowing this, it's a simple matter to write code to multiply all the values in the to pull together to get the result. Redefining hyper volume and using Amore documentary name for the argument gives us this. The function works by obtaining an it aerator I over the to pull and using next to retrieve the first value, which is used to initialize a variable V in which the final volume will be accumulated. We then use a four loop to continue it oration with the same it aerator To deal with the remainder of the values. We can use the function to compute the areas of rectangles, the volumes of Q. Boyd's and hyper volumes of Hyper Que Boyd's. It also generalizes nicely down to lower dimensions to give us the length of lines. However, if called with no arguments, the function raises stop it oration, exposing an implementation detail about which clients of our function should be unaware. There are a couple of approaches to fixing this. One change could be to wrap the call to next in a try except construct and translate the exception into something more meaningful for the caller, such as the type error that is usually raised when insufficient number of arguments has passed to a function. We'll take a different approach of using a regular positional argument for the first length and the Star Arts to soak up any further length arguments using this design. The function continues to work as expected when arguments are supplied and raises a type a rare exception when insufficient arguments are given. At the same time, the revised implementation is even simpler and easier to understand than the previous version, which used innit aerator. Later in this course, we'll show you how to use the funk tools dot reduce function, which could also have been used to implement our hyper volume function. When you need to accept a variable number of arguments with a positive lower bound, you should consider this practice of using regular positional arguments for the required parameters and star arcs to deal with any extra arguments. Note that star Ours must come after any normal positional arguments and that there can only be one occurrence of star arcs within the argument list. Thes star arc syntax on Lee collects positional arguments, and a complimentary syntax is provided for handling keyword arguments. Let's look at that now.

Keyword and Positional-only Arguments

[Autogenerated] arbitrary keyword arguments could be accepted by Cola Bols that use an argument prefixed by eight double asterisk. Conventionally, this argument is called quarks or K W R. Eggs all the depending on your situation. You may care to choose a more meaningful name. Let's make a function, which returns a single HTML tags as a string. The first argument to the function will be a regular positional argument, which will accept the tag name. This will be followed by the arbitrary keyword arcs construct to which tag attributes can be passed. As before, we'll perform a simple experiment to determine how the keyword arguments are delivered. When we call the function with some suitable attributes to create an HD male image tag like this, we could see that the arguments are transferred to our keyboard arguments formal parameter as a regular python dictionary, where each key is a string bearing the actual argument name. Now we'll go ahead and implement our tag function properly, using a more descriptive name than chords. Here we iterated over the items in the Attributes dictionary, building up the results during as we go. It's worth pointing out of this stage that thestreet dot format method we call here also uses the arbitrary keyword arcs technique to allow us to pass arbitrary named arguments corresponding to our replacement fields in the format string. This example also shows that it's quite possible to combine positional arguments and keyword arguments. In fact, the overall syntax is very powerful, so long as we respect the order of the arguments we define First star arcs if present must always proceed. Star stark wards So this isn't allowed. Second, any arguments proceeding star. Our eggs are taken to be regular positional arguments, as we saw in the hyper volume example earlier. Thirdly, any regular arguments after star our eggs must be passed as mandatory keyword arguments, failure to pass them as keyword arguments results in a type error. There are situations where you will want to use mandatory keyboard arguments without accepting an arbitrary number of positional arguments. To support this python allows you to omit the name of the star arcs argument. The arguments following the asterisk can only be passed to the function as keyword arguments. But python will not allow you to pass extra positional arguments to the function, Fourthly, and finally we have the star star corks arbitrary keyword arguments which, if present, must be last in the argument list. Any attempt to define an additional formal argument after Starr Starr corks results in a syntax error, you should take particular care when combining these language features with default arguments, which have their own ordering rule specifying that mandatory arguments must be specified before optional arguments at the call site. Pep 5 70 introduced support for positional on the arguments and these air available in Python 38 and later. Whereas keyword on the arguments can only be passed as keyword arguments, positional only arguments can never be passed with the key word to specify positional only arguments. You include a forward slash and your functions argument. List all the arguments before the forward slash our positional only so you could pass them to the function positionally. Any attempt to pass them by keyword will result in a type error. You may wonder why python includes this feature, and it turns out that there are some good reasons. One reason is to provide parody in pure python code with modules implemented and see and other languages. Thes extension modules can and historically have implemented functions with positional only arguments using techniques that aren't available in pure python. An example of a function like this is range Maur. Importantly, though, this feature allows you to prevent certain argument names from becoming part of your functions. AP Eyes. By prohibiting users from using keywords for some arguments, you prevent them from creating dependencies on those names. This could be particularly useful when the argument names have no useful semantic meaning and our only to find it all because Python requires it. As with our number length example from before, before moving on, we should point out that all the features of the extended argument syntax apply equally to regular functions. Lambda and Other Koala Bols although it's fair to say that they are rarely seen in combination with Lambda in the Wild.

Extended Call Syntax

[Autogenerated] the compliment to extended formal arguments. Syntax is extended call syntax, which allows us to use it horrible. Siri's such a za to pull the populate positional arguments and any mapping type, such as a dictionary that has string keys to populate keyword arguments. Let's go back to a simple version of Print Our Eggs, which only deals with mandatory positional and star are eggs. Arguments will now create an interval. Siri's in this case a toodle, although it could be any other conforming type and apply it at the call site for Print Our Eggs, using the asterisk prefix to instruct Python to unpack the series into the positional arguments, notice that the use of the star syntax and the actual arguments does not necessarily need to correspond to the use of star in the formal argument list. In this example, the 1st 2 elements of our trouble have been unpacked into the mandatory positional arguments and the last to have been transferred into the arcs to pull. Similarly, we can use the double asterisk prefix at the call site toe. Unpack a mapping type, such as a dictionary into the keyword arguments mandatory or optional. First, we'll define a function color, which accepts three arguments. Red, green and blue. Each of these three arguments could be used as either positional or keyword arguments at the coal site. At the end of the argument list, we add star star corks to soak up any additional keyword arguments that are past. Now we create a dictionary to serve as our mapping type of key word arguments and apply it at the function call site, using the double star prefix Notice again. How There's no necessary correspondence between the use of Star Star in the actual arguments versus the use of Star Star in the formal argument list. Items in the dictionary are matched up with the arguments, and any remaining entries are bundled into the corks parameter. Before moving on, we'll remind you that the addict constructor uses the star star corks technique to permit the creation of dictionaries directly from keyword arguments. We could have used that technique to construct a dictionary K in the previous example, but we didn't want to make the example of more complex than necessary. Otherwise, we could have constructed K as K equals dicked _________ 21 green equal 68 blue equals 1 20 Alfa equals 52. One of the most common uses of star arcs and star Stark works is to use them in combination to forward all arguments of function to another function. For example, suppose we defined a function for tracing the argument and return values of other functions. We passed the function whose execution is to be traced, but that function could take any arguments whatsoever. We can use extended argument syntax to accept any arguments to are tracing function and extended call syntax. To pass this arguments to the trace function, we'll trace a call to end to demonstrate that trace can work with any function without advanced knowledge of the signature of that function.

Summary

[Autogenerated] Let's review what we've covered in this module. We saw how pythons extended argument syntax allows us to accept arbitrary numbers of positional arguments as well as arbitrary keyword arguments in our functions. We also saw how to specify keyword on Lee and positional on the arguments to functions and other cola Bols. We looked at pythons extended call syntax for applying the elements in it horribles and mapping sze to function parameters. And we saw how to combine these features for perfect argument forwarding in the next module of core python functions and functional programming will investigate local functions and the key to how they operate. Closures. Thanks for watching, and we'll see you in the next module.

Closures

Local Functions

[Autogenerated] in this module of core python functions and functional programming, We'll look at pythons support for local functions that is functions to find. Within other functions. We'll explore the enclosing name space, which comes into play when local functions air used. We'll see how local functions could be returned from functions just like other objects will investigate how Python uses the concept of closures to manage object lifetimes for local functions. We'll see how to use the non local keyword for pulling names from a closing name spaces into local functions, similar to how the global keyword works. We'll pull all of these parts together into an interesting example. As you'll recall in Python, the deaf key word is used to define new functions. Death essentially binds the body of the function toe a name in such a way that functions are simply objects. Like everything else in python, it's important to remember that death is executed at runtime, meaning that functions are defined at runtime. Up to now, almost all of the functions we've looked at have been defined at module scope or inside classes, in which case we refer to them as methods. However, Python doesn't restrict you to just defining functions in those two contexts. In fact, Python allows you to define functions inside other functions. Such functions are often referred to as local functions, since they're defined local to a specific functions scope. Let's see a quick example here we define a function sort by last letter, which sorts a list of strings by their last letter. We do this by using the sorted function and passing last letter as the key function. Last letter is to find inside sort by last later. It is a local function. Let's test it out. Just like module level function definitions, the definition of a local function happens at runtime when the deaf keyword is executed. Interestingly, this means that each call to sort by last letter results in a new definition of the function last letter that is just like any other name bound in a function body. Last letter is bound separately to a new function. Each time it's cold. We can see this for ourselves by making a small modification to sort my last letter to print the last letter object. If we run this a few times, we see that indeed, each execution of sort by less letter result in a new last letter instance, the main point here is that the death call in sort by less letter is no different from any other name binding in the function and a new function is created Each time death is executed, local functions are subject to the same scoping rules as other functions. Remember the L E G B rule for name. Look up First, the local scope is checked than any enclosing scope. Next the global scope and finally the built in scope. This means that name look up in local functions, starts with names to find in the function itself, it proceeds to the enclosing scope which in this case, is the containing function. This enclosing scope includes both the local names of the containing function as well as its parameters. Finally, of course, the global scope includes any module level name bindings. We can see this in a small example. Here we define the function inner local tow outer inner simply prints a global variable and a few bindings from outer. This example shows the essence of how the L E g B world applies to local functions. If you don't fully understand what's going on. You should play with this code on your own until it's clear. It's important to note that local functions are not members of their containing function in any way. As we've mentioned, local functions are simply local name bindings in the function body. To see this, you can try to call a local function via member access syntax. The function object Outer has no attributes named Inner inner is only defined when outer is executed, and even then, it's just a normal variable in the execution of the Functions body. So what are local functions useful for? As we've seen, they're useful for things like creating sorting key functions. It makes sense to define these close to the call site if they're one off specialized functions. So local functions are a code organization and readability aid. In this way, they're similar to land us, which, as you'll recall, our simple unnamed function objects. Local functions are more general than Landis, though, since they may contain multiple expressions and may contain statements such as import. Local functions are also useful for other, more interesting purposes. But before we could look at these will need to investigate two more concepts, returning functions from functions and closures. As we've just seen, local functions are no different from any other object created inside a functions body. New instances are created for each execution of the enclosing function. They're not somehow specially bound to the enclosing function and so forth. Like other bindings and a function, local functions can also be returned from functions. Returning a local function does not look any different from returning any other object. Let's see an example here in closing, defines local funk and returns it. Callers of enclosing combined its return value toe a name in this case LF and then call it like any other function. This ability to return functions is part of the broader notion of first class functions where functions could be passed to and returned from other functions or, more generally treated like any other piece of data. This concept can be very powerful, particularly when combined with closures, which will explore in the next section

Closures and Nested Scopes

[Autogenerated] So far, the local functions we've looked at have all been fairly boring. They're defined within another function scope, but they don't really interact with the enclosing scope. However, we did see that local functions can reference bindings, and they're in closing scope via the l E G B roll. Furthermore, we saw that local functions could be returned from their defining scope and executed in another scope. This raises an interesting question. How does a local function use bindings to objects to find in a scope that no longer exists? That is, once a local function is returned from its enclosing scope, that enclosing scope is gone along with any local objects that defined. How could the local function operate without that in closing scope? The answer is that the local function forms what's known as a closure. A closure essentially remembers the objects from the enclosing scope that the local function needs. It then keeps him alive so that when the local function is executed, they can still be used. One way to think of this is that the local function closes over the objects it needs, preventing them from being garbage collected. Python implements closures with a special attributes named Dunder Closure. If a function closes over any objects than that function has a dunder closure attributes, which maintains the necessary references to those objects, we could see that in a simple example, the Dunder closure attributes of LF indicates that LF forms a closure and we could see that the closure is referring to a single object. In this case, that object is the X variable to find in the function that defined. LF so we can see that local functions can safely use objects from their enclosing scope. But how is this really useful? Ah, very common use for closures isn't so called the function factories. These factories are functions that return other functions where the returned functions are specialized in some way based on arguments to the factory. In other words, the factory function takes some arguments. It then creates a local function which takes its own arguments but also uses the arguments passed to the factory. The combination of runtime function definition enclosures makes this possible a typical example. If this kind of factory creates a function which raises numbers to a particular power, here's how the factory looks raised. Two takes a single argument e x p, which is an exponents. It returns a function that raises its arguments to that exponents. You can see that the local function raised to e. X p refers to e x p in its implementation, and this means that python will create a closure to refer to that object. If we call raise two, we can verify that it creates this closure. We can also see that square does indeed behave as we expect, and we can create other functions the same way.

The Nonlocal Keyword

[Autogenerated] the use of local functions raises some interesting questions regarding name. Look up. We've looked in some detail at the l E G B rule, which determines how names are resolved in python when you want the values to which those names refer, However, L E G B doesn't apply When we're making new name bindings. Consider this simple example. When we assigned to message in the function local what precisely is happening in this case, we're creating a new name binding in that function scope from the name message to the string local Critically, we're not re binding either of the other message variables in the code. We can see this by instrument in the code a bit. Now we're actually calling the functions in closing and local, and we could see that neither the enclosing nor the global bindings for the name message is affected. When local assigns to the name message again, local is creating an entirely new name binding, which only applies in the context of that function. In earlier courses, we've discussed pythons. Global keyword global could be used to introduce a name binding from the global scope into another scope. So in our example if we wanted the function local to modify the global binding for message rather than creating a new one, we could use the global keyword to introduce the global message binding into local. Let's do that and see the effects first. Let's use the global keyword to introduce the module level binding of message into the function local. If we run this, we can see that the module level binding of message is indeed changed when local is called again. The global keyword should be familiar to you already. If it's not, you can always review Corp I Thin Getting started. If Global allows you to insert module level name bindings into a python function, how can you do the same for name bindings in closing scopes or in terms of our example? How could we make the function local modified the binding for message to find in the function in closing. The answer to that is that Python also provides the key word nonlocal, nonlocal inserts and name binding from an enclosing name space into the local name space. More precisely, non local searches, the enclosing name space from innermost toe outermost for the name you give it as soon as it finds a match. That binding is introduced into the scope where nonlocal was invoked. Let's modify our example again to show how the function local can be made to modify the binding of message created in the function in closing by using not local. Now, when we run this code, we see that local is indeed changing the binding and enclosing. It's important to remember that it's an error to use non local where there's no matching in closing binding. If you do, this, python will raise a syntax error. You can see this if you had a call to non local inter function local, which refers to a non existent name. When you try to execute this code, Python will complain that no such name does not exist like global nonlocal does not something you're likely to need to use a lot, but it's important to understand how to use it for those times when it really is necessary, or for when you see it used in other people's code to really drive it home. Let's create a more practical example that uses not local, and this example the make timer function returns a new function Each time you call this new function, it returns the elapsed time since the last time you called it. Here's how it looks and here's how you can use it. As you can see the first time you invoke T, it returns nothing. After that, it returns the amount of time since the last invocation. How does this work? Every time you call make timer, it creates a new local variable named Last called it, then defines a local function called Elapsed, which uses the non local keyword to insert makes timers binding of last called into its local scope. Elapsed, then uses the last called binding to keep track of the last time it was called. In other words, Elapsed uses non local to refer to a name binding, which will exist across multiple calls to elapsed. In this way, Elapsed is using non local to create a form of persistent storage. It's worth noting that each call to make timer creates a new independent binding of last called as well as a new definition of elapsed. This means that each call to make timer creates a new independent timer object, which you can verify by creating multiple timers As you can see, calls the T one have no effect on t two, and they're both keeping independent times.

Summary

[Autogenerated] Let's review what we've covered in this module. We saw how we can define local functions that is functions within other functions, and we saw how local functions necessitate the enclosing scope. We explored how local functions could be returned from other functions like any other object, and how this could be used to implement constructs like factory functions. We then looked at how Python uses closures to ensure that objects stay alive and available for the lifetime of a local function, We saw how the non local keyword lets us bind names from enclosing scopes into local functions. We used all of this to implement functions with the interesting property of maintaining state between invocations in the next module of core python functions and functional programming will explore one of the most interesting features of python, the function decorator, which allows you to alter existing callable objects by wrapping them in another call a bull. Thanks for watching and we'll see you in the next module

Function Decorators

Function Decorators

[Autogenerated] in this module of core python functions and functional programming will introduce you to the concept of function decorators. We'll see how function decorators can add new behaviors to existing functions without modifying the decorated functions. We'll learn how to implement decorators using any of the gullible types of objects and python. We'll see howto apply multiple decorators to a function, and we'll look at the semantics of how this kind of decoration works. We'll explore some standard library features for creating decorators, and we'll see a technique for creating parameter rise decorators. Now that we've looked at the concepts of local functions enclosures, we have what we need to finally look at an interesting and useful python feature called decorators. At a high level, Decorators are way to modify or enhance existing functions in a non intrusive and maintainable way. In Python, a decorator is a call Abel object that takes in a gullible and returns a culpable. If that sounds a bit abstract, it might be simpler for now to think of decorators as functions that take a function as an argument and return. Another function by the concept is a bit more general in that as well see, coupled with this definition, is a special syntax that lets you decorate functions with decorators. The syntax looks like this. This example applies. The decorator in this case named my decorator to the function named in this case, my function theat symbol is the special syntax for applying decorators two functions. So what does this actually do? When Python sees decorator application like this, it first compiles the base function, which in this case is my function. As always, this produces a new function object. Python then passes this function objects to the function. My decorator. Remember that decorators, by definition, take culpable objects as their only argument, and they're required to return a callable object as well. After calling the decorator with the original function object, Python takes the return value from the decorator and binds it to the name of the original function. The end result is that the name my function is bound to the result of calling my decorator with the function created by the deaf. My function line. In other words, decorators allow you to replace, enhanced or modify existing functions without changing those functions. Callers of the original function don't have to change their code because the decorator mechanism ensures that the same name is used for both the decorated and a neck aerated function. As with so many things in Python, a simple example is much more instructive than words. Suppose that we had some functions which returned strings, but we needed to ensure that those strings on Lee contained asking characters. We can use the ask you function to convert all non asking characters to escape sequences, So one option would be to simply modify every function to use the ask you function. This would work, but it isn't particularly scaleable or maintainable. Any change to the system would have to be made in many places, including if we decided to remove it completely. A simple solution is to create a decorator which does the work for us. This puts all of the logic in a single place. Here's the decorator. As you can see, the decorator escape. Unicode is just a normal function. It's only argument. F is the function to be decorated. The important part, really is the local function. Wrap Rap uses thes star arcs and corks idiom to accept any number of arguments and then calls F the argument to escape unit code. With these arguments, wrapped takes EFS return value, converts non asking characters to escape sequences and returns. The resulting string, in other words, wrapped, behaves just like F, except that it escapes non sq characters, which is precisely what we want. It's important to notice how escape Unicode returns rap. Remember that a decorator takes a cola bill as its argument and returns a new call level. In this case, the new gullible is rap. By using closures, rap is able to use the parameter F even after escape. Unicode has returned. Now that we have a decorator, let's create a function that might benefit from it. Are extremely simple. Function returns the name of a particular northern city. And of course, we can see that it works toe. Add Unicode, escaping to our function. We simply decorate Northern City with our escape unico decorator. Now, when we call Northern City, we see that indeed non asking characters are converted to escape sequences. This is, of course, a very simple example, but it demonstrates the most important elements of decorators. If you understand what's going on in this example, then you understand 90% of what there is to know about decorators

What Can Be a Decorator?

[Autogenerated] Now that we've seen how decorators work, let's look at our other kinds of call levels could be used as decorators. We just used to function as a decorator, and that's probably the most common form of decorator in general use, however, to other kinds of call A bulls are also used fairly commonly. The first of these is class objects. Remember that class objects are cola, ble and calling that produces a new instance of that class. So by using a class object as a decorator, you replace a decorated function with the new instance of the class because the function to be decorated will be passed, the constructor and thereby the initial izer recall, however, that the object returned by the decorator bust itself be a call a ble object, so the instance resulting from the constructor call must be Call a ble, which means it must support the dunder call method that is, we can use class objects as decorators so long as the instance, objects we get when we call the constructor are themselves. Call a ble by virtue of implementing the Dunder call method, in this example will create a class call count which keeps track of how many times it's called call counts Initialized takes a single function F and keeps it as a member at tribute. It also initialize is account attribute to zero call counts Dunder call method than increments that count each time it's called and then calls F returning. Whatever value F produces. You use this decorator much as you might expect by using at Cole Count to decorate a function. Now if we call hello a few times, we can check its call count. Great class decorators are useful for, in some sense, attaching extra dated two functions. So now we've seen how to use class objects as decorators. Another common kind of decorator is a class instance, as you might have guessed, when you use a class instance as a decorator, Python Colisee instances Dunder call method with the original function and uses. Dunder calls Return value as the new function. These kinds of decorators are useful for creating collections of decorated functions, which you can dynamically control in some way. For example, let's define a decorator, which prints some information each time the decorated function is called. But let's also make it possible to toggle this tracing feature by manipulating the decorator itself, which will implement as a class instance. First, here's a class. Remember that, unlike our previous example, the class object itself is not the decorator. Rather, instances of trace could be used as decorators, so let's create an instance of trace and decorated function with it. Now, if we call rotate list a few times, we could see that tracer is doing its job. We can now disabled, tracing simply by setting tracer dot enabled to false. And we see that the decorated function no longer prints out tracing information. The ability to use functions, class objects and class instances to create decorators gives you a lot of power and flexibility. Deciding which to use will depend a great deal upon what exactly you're trying to do. Experimentation and small examples are a great way to develop a better sense of how to design decorators

Applying Multiple Decorators

[Autogenerated] and all of the examples we've seen so far, we've used a single decorator to decorate functions. However, it's entirely possible to use more than one decorator at a time. All you need to do is list each decorator on a separate line above the function, each with its own At sign like this. When you use multiple decorators like this, they're processed in reverse order. So in this example, some function is first past the decorator three. The Culpable, Returned by Decorator three is then passed The decorator to that is a decorator to is applied to result of Decorator three in precisely the same way that it would be applied to a normal function. Finally, Decorator one is called with result of decorator to the call. Abel, returned by decorator. One is ultimately bound to the name some function. There's no extra magic going on, and the decorators involved don't need to know that they're being used with other decorators. This is part of the beauty of the decorator abstraction. As an example, let's see how we can combine to decorators. We've already seen our tracer and our Unicode escape er, first, let's see the decorators again and now let's decorate a single function with both of these. Now, when we use this to invent, names for Norwegian islands are non asking, characters will be properly escaped and the tracer will record the call. And, of course, we can disable the tracing without affecting the escaping. So far, we've only seen decorators applied to functions, but it's entirely possible to decorate methods on classes as well. In general, there's absolutely no difference in how you use decorators for methods to see this. Let's create a class version of our island maker function and use the tracer decorator on it. We can use this to cross the North Sea and make Amore British version of our island maker. As you can see, Tracer works perfectly well with methods.

Preserving Function Metadata

[Autogenerated] decorators replace a function with another call a ble object, and we've seen how this could be a powerful technique for adding functionality in a modular maintainable way. There's a subtle problem, however, with how we've used decorators so far bye naively replacing a function with another culpable. We lose important to metadata about the original function, and this can lead to confusing results in some cases. To see this, let's define an extremely simple function in the rebel. Let's look at some attributes of dysfunction. First, it has an attribute Dunder name, which is simply the name of the function as the user defined it. Similarly, it has an attribute. Dunder Doc, which is the doc string defined by the user. You may not interact with these attributes a lot directly, but they're used by tools like de buggers and ideas to display useful information about your objects. In fact, pythons built in help function uses these attributes. So far, so good. But let's see what happens when we use a decorator on our function. First, let's define a simple, no up decorator and decorate our hello function. All of a sudden. Help is a whole lot less helpful instead of telling us that hello is named hello. In reporting the expected Doc String, we're seeing information about the wrapper function used by the no op decorator. If we look at hellos, Dunder name and Dunder Doc attributes, we can see why, since we've replaced the original hello function with a new function. What Dunder name and under Doc attributes we get when we inspect Hello are those of the replacement function. This is an obvious result in retrospect, but generally not what we want. Instead, we'd like the decorated function to have its original name and doc String. Fortunately, it's very easy to get the behavior we want. We simply need to replace the Dunder name and Dunder Doc attributes of our NOAA proper function with the same attributes from the rap function. Let's update our decorator to do this. Now. When we examine our decorated function, we get the results. We want this works, but it's a bit ugly, and it would be nice if there were a more concise way of creating rapper functions, which properly inherited the appropriate attributes from the functions they wrap. Well, we're in luck. The function wraps from the funk tools package does precisely that Funk tools that Wraps is itself a decorator, which you apply to your wrapper function. The decorator takes the function to be decorated as its argument, and it does the hard work of updating the wrapper function with the rep functions attributes. Here's how that looks. If we now look at our hello function in the rebel one more time, we could see that indeed, everything is as we want As you start to develop your own decorators. It's probably best to use funk tool start raps to ensure your decorated functions continue to behave as your users expect.

Parameterized Decorators

[Autogenerated] one interesting and practical use of decorators is for validating function arguments in many situations. You want to ensure that function arguments are within a certain range or meet some other constraints. Let's create a decorator, which verifies that a given argument to a function is a non negative number. This decorator is interesting in that it takes an argument. This might appear confusing at first, but you'll see how it works if you just followed the description of decorators, which we've just finished covering. Here's how you can use this decorator to ensure that the second argument of a function is not negative. We could see that it works as expected. So how does this decorator work? First, we need to recognize that check non negative is not, in fact, a decorator at all. A decorator is a callable object that takes a callable object as an argument and returns a callable object checking out negative, takes an integer as an argument and returns a function. The nested validator function. What's going on here? The key is how we use check non negative. You'll see that at the point where we decorate, create list week actually call check non negative. In other words, the return value of Czech non negative is the actual decorator. Python takes check, not negatives, return value and passes our function. Create list to it. Indeed, if you look at the validate function itself, you'll see that it looks exactly like the other decorators we've defined in this module. Interestingly, the rap function returned by validator forms a closure over not just f the decorated function but also over index. The argument past to check, not negative. This can be a bit of a mind bender, and it's well worth spending a little extra time to make sure you really understand how this works. If you understand this example, you're well on your way to mastering python decorators. We've seen how to use and create decorators, and hopefully it's clear that decorators are a powerful tool for python programming. They're being used widely in many popular python packages, so it's very useful to be familiar with them. One word of warning, though. Like many powerful features in many programming languages, it's possible to overuse decorators, so use decorators when they are the right tool. When they improve, maintain ability at clarity and simplify your code If you find that you're using decorators just for the sake of using decorators, take a step back and think about whether they're really the right solution. Let's review what we've covered in this module. We saw how decorators can enhance, modify or even replace existing cola bols without modifying their implementation or a P I. We looked at how to apply decorators to functions. Using the at symbol. We learned that decorators are implemented as call a bles. That, except a colorable argument and return a culpable object. We explored how to create decorators with several types of culpable object functions, classes and instances. We saw how to apply multiple decorators to a call a ball and how this works by Recursive Lee. Applying the rules for applying a single decorator We looked at the funk tools dot wraps function for preserving important call a bill metadata when implementing decorators. And finally we saw how to use decorator factories to implement so called parameter rised decorators in the next module of core python functions and functional programming. We look at some of the tools Python provides us for doing functional style programming. This approach to programming can lead to some elegant and surprising solutions to certain kinds of problems. Thanks for watching, and we'll see you in the next module.

Functional-style Tools

Map

[Autogenerated] in this module of core python functions and functional programming will introduce the idea of functional style programming and python. Well, look at the built in map function for applying a gullible to each element in unendurable Siris. Well, then, look at the built in filter function for skipping the elements inimitable, which don't meet some criteria. Then we'll explore the more fundamental reduced function from the standard Library Funk Tools module, which accumulates a result by combining the elements in it terrible. Finally, we'll see how to combine these somewhat abstract tools to solve practical problem pythons. Concept of Federation inedible objects is fairly simple in abstract, not involving much more than the idea of a sequence of elements that could be accessed one at a time in order. This high level of abstraction allows us to develop tools that work on it. Terrible's at an equally high level, and Python provides you with a number of functions that serve as simple building blocks for combining and working within trebles and sophisticated ways. A lot of these ideas were originally developed in the functional programming community, so some people refer to the use of these techniques as a functional style python. Whether you think of these as a separate programming paradigm or just a s'more tools in your programming arsenal, thes functions can be very useful and are often the best way to express certain computations. The map function is probably one of the most widely recognized functional programming tools, and python at its core map has a very simple thing, given a function and a sequence of objects, it calls the functions for every element in the sequence, producing a new sequence containing the return values of the function. In other words, we map a function over a sequence to produce new values. Let's see a simple example. Suppose you wanted to find the Unicode code point for a character in a string. The map expression for that would look like this. This essentially says, for every element in the string called the Function Ord, with the element as an argument, generate a new sequence comprising the return values aboard in the same order as the input sequence. Graphically, it looks like this, So let's try this out in the ripple rather than return a list. As you might expect, we instead get a map object. The map function, it turns out, performs lazy evaluation. That is, it doesn't produce any output until it's needed. Another way to say this is that map will not call its functions or access any elements from its input source until they're actually needed. The map object returned by map is itself an aerator object, and only by iterating over it. Can you start to produce output to make this a bit more clear, let's re use our trace decorator from module three to print out a message whenever map calls its function. We won't be using Trace as a decorator, but rather we leverage the fact that we can call the Trace instance to get a call. A bill that does tracing forests as reminder. Here's how traces to find and here's how we use Trace to invoke the ord function as we map over a string. The function returned by traced under col. Will print a bit of text each time it's called so we can see how map works. It's not until we start to generate over result that we see our function executed again. This gets to the heart of lazy evaluation. Map does not call the function or eatery over the input sequence until it needs to. In order to produce output here, we're driving the process by manually iterating over the map object returned by map. More commonly, you will not manually drive the map object, but instead you illiterate over using some higher level method. For example, you could use the list constructor to read all of the elements produced by map, or you could use a four loop. The point is that maps lazy evaluation requires you to generate over its return value in order to actually produce the output sequence. Until you access the values in this sequence, they are not evaluated. So far, we've seen examples using map over single infant sequences, but in fact, map could be used with as many input sequences as your map function needs. If the function you passed the map requires two arguments, then you need to provide two infant sequences to map. If the function requires three arguments, then you need to provide three input sequences. Generally, you need to provide as many Edwin sequences as there are arguments in the map function. What map does with multiple input sequences is to take an element from each sequence and pass it as the corresponding argument to the map function to produce each output value. In other words, for each output value that map needs to produce, it takes the next element from each input sequence. It then passes these in order as the arguments to the map function and the return value from the function is the next output value from map. An example will help make this clear. First, we defined three sequences and a function that takes three arguments. Now, if we map combined over the sequences, we see that an element from each sequence is passed to combine for each output value. It's possible, of course, that the input sequences are not all the same size. In fact, some of them might be infinite sequences. Map will terminate as soon as any of the input sequences is terminated. To see this, let's modify our combined function to accept a quantity argument. Now let's pass an infinite sequence to map and see that it terminates after the finite inputs are exhausted. You may have noticed that Matt provides some of the same functionality is comprehension Sze, for example. Both of these snippets produce the same lists. Likewise, both this generator expression and this call to map produce equivalent sequences. In cases where either approach will work, there's no clear choice, which is better. Neither approach is necessarily faster than the other. And while many people find comprehension more readable, others feel that the functional style is cleaner. Your choice will have to depend on your specific situation, your audience or perhaps just your personal taste.

Filter

[Autogenerated] the next functional style tool will look at is filter. As its name implies, filters looks at each element of the sequence and filters out or removes those which don't meet some criteria, like map filter applies a function. Each element in a sequence and also like map filter, produces its results lazily, unlike map filter, only accepts a single input sequence, and the function it takes must only accept a single argument. The general form of Filter takes a function of one argument as its first parameter and a sequence as its second. It returns. An honorable object of type of filter filter applies its first argument to each element of the input sequence, and the sequence Filter returns contains Onley, those elements of the input for which the function returns. True. For example, here we use a lambda as the first argument to filter this. Lambda returns true for positive arguments and false for everything else, meaning that this cult of filter will return a sequence containing only the positive elements of its input sequence. And indeed, that's precisely what we see. Remember that the return value from Filter is a lazy adorable, so we have to use the list constructor or some other technique to force evaluation of the results you could optionally pass. None is the first argument to filter, in which case filter will filter out all input elements, which evaluate to false in a 1,000,000,000 context. For example, since zero false, the empty list and the empty string are all false e, they were removed from the output sequence. While this course is focused on Python version three, the map and filter functions represent one area where it's useful to discuss a difference between Python two and three. As we've just shown in Python three, the map and filter functions return a lazily produced sequence of values and python to however, these functions actually return lists. If you find yourself writing code in Python to, it's important to keep this in mind.

Reduce

[Autogenerated] the final functional style tool will look at is the reduced function in the funk tools Standard library module reduced could be a bit confusing at first, but it's actually quite simple. In useful, reduce repeatedly applies a function of two arguments to an interim accumulator value and each element of the series in turn, updating or accumulating the interim value each step. With the result of the cold function, the initial value of the accumulator could either be the first element of the sequence or an optional value that we supply. Ultimately, the final value of the accumulator is returned, thereby reducing the Siri's down to a single value reduces not unique to python by any means, and you may have come across it in other languages you've worked with, for example, reduces the same misfolded many functional languages. Likewise, it's equivalent to aggregate in dot net slink and accumulate in c++ is standard template library. The canonical example of reduced is the summation of a sequence and in fact, reduces a generalization of summation. Here's how that looks. Using the ad function from the Standard Library operator module, which is a regular function version of the familiar in fix plus operator. Conceptually, what's happening is something like this. To get a better idea of how reduces calling the function, we can use a function which prints out its progress. Here we see that the interim result is past as the first argument to the reducing function, and the next value in the input sequence is past. As the second, the reduced function has a few edges that are worth noting. First, if you pass an empty sequence to reduce, it will raise a type error. Second, if you pass the sequence with only one element than that element is returned from reduced without ever calling. The reducing function reduce accepts an optional argument specifying the initial value. This value is conceptually just added to the beginning of the input sequence, meaning that it will be returned if the input sequence is empty. This also means that the optional initial value serves as the first accumulator value for your reduction. The optional value is useful for example, if you can't be sure if you're in, but we'll have any values. Take care when selecting initial values, since the correct value, of course, depends on the function you're applying. For example, you use an initial value of zero for summation, but a value of one for products

Combining the Tools

[Autogenerated] you may have been wondering about the naming of these functions, and in particular, you may have been wondering if there's any relationship between pythons, map and reduce functions and the popular map produced algorithm. The answer is yes, Pythons map and reduce our very much the same as those terms and map reduce. To see this, let's write a small example that counts words across a set of documents. First, you need a function, which counts words in a document. This produces a dictionary mapping words to the number of times that word was found in the input string. For example, we can run count words over. It was the best of times. It was the worst of times. With this function, we can now map it across a collection of documents. This will result in a sequence of dictionaries with word frequencies for the individual documents. Here's how that looks. Next we need a function which takes two word count dictionaries and combines them with this. We have all the tools we need to run a proper map. Reduce. Here's how that looks so it's interesting to see a somewhat real world application of the relatively abstract notions we've presented so far in this module, and who knows? Maybe someday someone will find a way to make money using techniques like this. Let's review what we've covered in this module. We saw that map applies a function to each element in an input in terrible producing, another a terrible with the results. We inspected map and saw that it produced its results lazily rather than all up front, We looked at how we can provide multiple infinite terribles to map, so long as the culpable argument was of a correct charity. We learned how filter applies a predicate to the elements of unutterable and how it produces another interval containing just the elements for which the predicate returned. True. We then investigated funk, tool, start reduce and saw that it repeatedly applies it to argument. Call a ble to the elements of unutterable to combine its elements into a single output value. We saw that reduce will raise an exception for empty inputted trebles, but that you can provide an initial value to avoid this problem. We also saw that selecting the right initial value for reduced could be critical for getting correct results. Finally, we saw how to combine some of these tools to implement a simple but in principle, very powerful map reduce algorithm in the next module of core python functions and functional programming, we'll take another look at comprehension. Sze seeing how we can pass multiple input intervals to them and even nest of them. Thanks for watching, and we'll see you in the next module.

Multi-input and Nested Comprehension

Multi-input Comprehensions

[Autogenerated] in this module of core python functions and functional programming will expand our understanding of comprehension. Sze By seeing how they can process more than one Impenetrable at a time, we'll see how comprehension with multiple inputs are largely equivalent to nested four loops. Well, look at nested comprehension. Tze with the expression part of one comprehension comprises another comprehension, and we'll see how nested comprehension are nothing more than application of rules and syntax we already know. We'll show that multiple inputs and messing are supported by all comprehension types, including generator expressions, as you'll recall Comprehension Czar sort of shorthand syntax for creating collections and in terrible objects of various types. For example, a list comprehension creates a new list object from an existing sequence and looks like this here. We've taken arrange sequence in this case, the intruders from 0 to 9 and created a new list where each entry is two times the value of the original sequence. This new list is completely normal list, just like any other list made using any other approach. They're our comprehension sin taxes for creating dictionaries, sets and generators as well. It's lists and all of this in taxes working essentially the same way. All of the comprehension examples we've seen up to now use only a single input sequence. But comprehension is actually allow you to use as many interpret sequences as you want. Likewise, a comprehension can use as many if clauses as you need as well. For example, this comprehension uses to input ranges to create a set of points on a five by five grid, giving us a list containing the so called Cartesian product of the two input ranges. The way to read this is as a set of nested four loops where the later four clauses are nested inside the earlier four clauses and the result expression of the comprehension is executed inside the innermost or last four lube. So for this example, the corresponding nested four loop structure would look like this. The outer four loop, which binds to the X variable, corresponds to the 1st 4 clause in the comprehension. The Inter four loop, which binds to the Y variable, corresponds to the 2nd 4 close in the comprehension. The output expression in the comprehension or recreate the to pull is nested inside the innermost. For liv, the obvious benefit of the comprehension syntax is that you don't need to create a list variable and then repeatedly upend elements to it. Python takes care of that for you in a more efficient and readable manner with comprehension. Sze, as we mentioned earlier, you could have multiple if clauses in a comprehension. Along with multiple four clauses these air handled in essentially the same way as four clauses later. Clauses in the comprehension are nested inside earlier clauses. For example, consider this comprehension. This is actually fairly difficult to read and might be at the limit of the utility of comprehension Sze. So let's improve the layout of it there. That's better. This calculates a simple statement involving two variables and two if clauses by interpreting the comprehension is as a set of nested statements. The non comprehension form of this statement is like this. This could be extended to his many statements, as you want in the comprehension, though, as you can see, you might need to spread your comprehension across multiple lines to keep it readable. This last example also demonstrates an interesting property of comprehension, Sze, where later clauses can refer to variables bound in earlier clauses in this case. The last statement refers to X, which is bound in the 1st 4 claws. The four clauses and comprehension can also refer to variables bounding earlier parts of the comprehension. Consider this example, which constructs a sort of triangle of coordinates here. The 2nd 4 Clause, which binds to the Y variable, refers to the X variable to find in the 1st 4 claws. If this is confusing, just remember that you could think of this as a set of nested four lives. In this formulation, it's entirely natural for the inter for Luke to refer to the outer.

Nested Comprehensions

[Autogenerated] there's one more form of nesting incomprehension that's worth noting, although it doesn't really involve new syntax or anything beyond what we've already seen. We've been looking at the use of multiple four and if clauses in a comprehension. But it's also entirely possible to put comprehension in the output expression for a comprehension that is, each element of the collection produced by a comprehension can itself be a comprehension. For example, here we have 24 clauses, but each belongs to a different comprehension entirely. The outer Comprehension uses another comprehension to create a list for each entry in its result rather than a flat list, then this produces a list of lists. The expansion of this comprehension looks like this, and the resulting list of lists looks like this. This is similar to, but different from multi sequence comprehension. Sze. Both forms involved more than one ____________ loop, but the structures they produce are very different. Which form you choose will, of course, depend on the kind of structure you need, So it's good to know how to use both forms in our discussion of multi input NSAID comprehension. We've only just shown list comprehension in the examples However, everything we've talked about applies equally to set comprehension. Dicked comprehension is and generator comprehension. Sze, for example, you could use a set comprehension to create the set of all products of two numbers between zero and nine like this. Or you can create a generator version of the triangle coordinates we constructed earlier. Let's review what we've covered in this module. We saw how to use more than one input in terrible to any kind of comprehension, and we showed that this provides behavior similar to nested four loops, and we saw that it's perfectly possible to nest. Comprehension is inside other comprehension. Sze, well done on completing core python functions and functional programming functions, are absolutely central to python, and they support some sophisticated and sometimes subtle features. This course is giving you everything you need to know to build elegant, robust functions in python and to effectively use those written by others. Along the way, we looked at some interesting applications of functions and saw how abstractions like Call a Ble and the notion of function application can lend themselves to some interesting design opportunities. Look out for other core python courses here on plural site which build on the knowledge you've gained here and which explained the many other tools and abstractions provided by Python for building powerful programs. 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 Journeymen, the second book of 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 above all else, it's great fun to write Python software Happy programming.

Course authors

Austin Bingham

Robert Smallshire

Course info

Level	Intermediate
Rating	★★★★★
My rating	★★★★★
Duration	1h 18m
Released	3 Mar 2020

Share course