



Python 3 Decorators

Course Overview

Course Overview

Hi everyone. My name is Mateo, and welcome to my course about Python decorators. In this course, I will talk about what decorators are and how you can use them to decorate your functions. I will first talk about higher order functions, closures, and non-local variables so that you can see how decorator functions are implemented in practice. After implementing custom decorators, I will talk about decorators with arguments and how to apply multiple decorators on a single function. Finally, I will also cover how you can use classes as decorators and how to decorate classes, instead of functions. Some of the major topics that we will cover include implementing decorators, preserving function's metadata, decorators with arguments, applying multiple decorators, and class decorators. By the end of this course, you will know more about what the decorators are, how they work and how you can create your own decorators to extend the functionality of a function. Before beginning the course, you should be familiar with the fundamentals of the Python language, specifically with functions, and it would be great if you already have some experience with object-oriented programming. I hope you'll join me on this journey to learn more about Python decorators with the Python Decorators course, at Pluralsight.

Working with Higher Order Functions and Closures

Introduction and Prerequisites

Hi, everyone. My name is Mateo, and welcome to the course about Python decorators. In this course, we will take a closer look into what decorators are and how you can create your own. But before we start, let's first take a look at the version of Python that I will be using to demonstrate the capabilities of decorators. This course was created using the displayed version of the Python language, and this slide shows the versions of Python for which the information in this course applies to. If your version matches one of these, you will have no problem following along with the demonstrations. Now, let's talk about prerequisites. This is an advanced level course, so you should already have an intermediate-level knowledge of Python. So if you're a beginner, this course is probably not for you. You should check out other beginner Python



courses on Pluralsight and then get back to this one. I want to single out that you have to be familiar with functions because functions are an important building block for learning how decorators operate. There's a course on Pluralsight about Python functions, so you can check out that one if you need a refresher. Now, let's talk about the outcomes. What can you expect to know once you finish this course? This course will show you what decorators are, how they work, and how you can create your own. Aside from some special cases, decorators are mostly created by a library or framework developers because they want to offer additional abstractions that application developers can use to make their life easier. However, even if you're not going to create a lot of your own decorators, it is important to know what decorators are and how they work because you will most likely use a lot of them in your own applications. In this course, we will first talk about what decorators are, then how to stack them on top of each other, how to define decorators with arguments, and we will also discuss class decorators and how you can decorate classes. All of the slides and code from the demonstrations is available for you to download in the Exercise Files tab on the course's page. If you have any questions, please ask them in the Discussion tab and not in the feedback form because I cannot give you an answer to your anonymous feedback. And that's it. Welcome to the course. Let's start learning about decorators.

Using Functions as First Class Objects

Since everything in Python is an object, functions are also objects. If you want to know more about this, you can check out my course about object-oriented programming in Python. We can say that functions in Python are first class citizens or first class objects, but what does that mean exactly? That means that anything that you can do with objects like strings or integers, you can also do with the function object. Here, I defined two simple functions for adding and subtracting two numbers. When you run this program, the Python will turn these two function definitions into function objects. The identifier or the function's name is like a variable that we can use to refer to that function object in memory. Since functions are first class objects, I can assign this function to another variable just like any other object. Notice that I assigned the function object itself by referring to the function's name. I didn't use parentheses to call the function. The plus variable is now a new reference to the same function object in memory, so the plus is pointing to this add function object. That means that we can now use this reference to call the function because a reference is just like another name for the same function. I will also print out these two variables so that you can see that these two variables are pointing to the same function object. And since functions are just objects, we can also place them in a collection like a list. I can use index 1 to get the subtract function and call it with some arbitrary parameters. Let's run the script to see what we get. As you can see, the plus variable was able to access the add function and call it with some



arguments. You can also see that both the add and the plus identifier point to the same function object in memory. The memory address is the same, so this is the same object. Finally, the list element at the index 1 refers to the subtract object so I was able to use it indirectly. Okay. So we can assign function objects to variables and place them in lists, but since function objects are just objects, that means that we can also pass them as arguments to other functions. The calculator function will take in a function object as its first parameter. It will then call that function with the other two parameters, a and b. So I will call this calculator function two times and pass the add and subtract function object as a first argument. This first calculator function call will set the operation parameter to reference the add function in memory. It will then use it to add these two numbers together. I will run the script to prove that it works, and as you can see, the correct operations were applied based on the function we passed into the calculator function. Functions that take in other functions or return other functions are known as higher-order functions. In this example, the calculator function is a higher-order function because it has a function parameter. However, higher-order functions are also the ones that return function objects, which is not what we did here. Here, I returned the result value from the function call. So join me in the next lesson to see how we can return function objects.

Returning Inner Functions from Higher Order Functions

Let me refresh your memory about functions local scope. I'm, again, calling this simple add function from the previous lesson. Each one of these function calls will create its own local scope. The numbers that you pass in as arguments will be copied over to the functions parameters. Just like any variables that you define inside of the functions body, these parameters are only accessible inside of the functions local scope. For example, I can print the value of the parameter a inside of the function's body. And if I run the script, the expected output should be 2 and 5, and it is. However, if I try to print this same variable outside of the function scope, I will get an error. The name a is not defined because I tried to access it from the global scope. Also, even if I was somehow able to refer to this object, as soon as the function is done with evaluation, the values from the functions local variables are released from memory by garbage collection. I just wanted to make sure that we are all on the same page. In this script, I defined the function inside of another function. The inner function definition is inside of the local scope of the outer function. Remember that functions are just objects, so when you place this definition here, Python will create a new function object in memory. When you define a function inside of another function, that function is known as inner function, so I named it accordingly. In Python, you know that you can use variable names to access string or integer objects in memory. Well, in the same way, you can use function names which are also just identifiers to access function objects in memory. So if a function definition is like creating a variable, that means that this function belongs to the local scope of the outer function. I'm calling this



function from the enclosing local scope of the outer function. So once I call the outer function, the inner function should be defined and evaluated. And if I run the script, you can see that the Inner function was successfully evaluated. However, what if I try to call this inner function from the global scope? I will get a name error just like the one from the previous script. The name inner is not defined because the function inner is defined inside of the local scope of another function. However, unlike with basic objects, there is a way to access this function outside of the local scope. And to achieve that, I can return the inner function object, instead of calling it. Then I can assign this return function to some globally available variable, which I will simply name func. Remember the example from the previous lesson where I assigned the add function to the plus variable. The plus was just a reference to the function object which means that this func variable is also just a reference to the Inner function object. I'm then using that reference to call the Inner function and printing that function object to the terminal. This is another example of the higher-order function because the outer function is returning a function object. As you can see, the inner function was successfully called through the global variable func, and here is the proof that the global function is just a reference to the inner function defined inside of the local scope of the outer function, but how is this possible? We know that as soon as this outer function is done, all of its local objects will be released from memory and the same goes for its function objects, but we were able to call this function after the local scope was long gone. Well as it turns out, Python will keep objects in memory as long as there are references to that specific object. Since the func variable is a reference to the inner function, that function will stay in memory even though its enclosing scope is gone. So now you know that we can use higher-order functions to define and return new function objects. This gets us one step closer to understanding decorators.

Retaining Nonlocal State with Closures

For this demonstration, I created another higher-order function which also returns an inner function; however, this one is a little bit more interesting. The power function takes in an exponent as an argument and returns an inner function. The inner function has a base argument and returns that base raised to the power of the exponent. I created two instances of this function and assigned them to these two identifiers. The power of two function will reference an inner function object with the exponent set to 2, and the power of 3 will use the exponent 3 because that number is passed to the power function. Finally, I can use these two new functions and exponentiate number 5. And if I run this, you can see that the results are correct, so the functions work. However, if you take a closer look at the definition of the inner function, you will see that, in theory, this should not work. To show you what I mean, let's discuss what happens when the power function is called for the first time. This call will pass the value 2 to the exponent parameter, and then the inner function is defined inside of the local scope of the power



function. The exponent is not a local variable of the inner function because it's not defined inside of its body or as its parameter. However, as you already know, if some variable is not present in the current scope, Python will look for it outside in the enclosing scope. So since this variable is a parameter of the power function, that means that the inner function can access it from the power's local scope. Once the function object is defined, the power function will return it and this will create an outside reference to that object. Because of the previous lesson, we know that this global reference will keep the inner function alive even after the local scope of the power function is finished. But if the local scope is gone, then how can this function object still access the value of the exponent variable? It makes sense that it can access it while the function is being defined, but when we call this function here, the local scope of the power function that created this function doesn't exist anymore. Well as it turns out, Python is specially prepared for this situation and it creates something known as nonlocal variables. Exponent is the so-called free nonlocal variable, and it's kept in memory even after the local scope of the enclosing function is gone. It is named nonlocal because it's not defined inside of the function's local scope, but in its enclosing scope. A function that utilizes nonlocal variables is known as a closure because it encloses the needed nonlocal variables together with the function object. A closure is in a way hiding the nonlocal variables from the global scope and other functions because nonlocal variables are only available from inside of the closure and nowhere else. The closure will package a tuple of so-called cell variables with the function object. Each cell variable is basically a reference to the nonlocal variable. In this example, we only access the exponent from the outside scope, so we only have one cell variable. So this exponent variable is just an indirect reference to the exponent variable from the outer scope. Once the local scope of the power function is garbage collected, its local variables are destroyed, but the exponent variable will still stay in memory because of the cell variable from the inner function. And just like we saw with the returned function objects, Python will not deallocate variables which are still referenced by some other part of the code. And now that I covered closures, we are finally ready to talk about decorators.

Decorating Functions

Now that we are familiar with closures and higher-order functions, we can talk about how decorators are implemented. In general, a decorator is a higher-order function which takes in another function as an argument and returns a closure. To demonstrate this, I created a simple function that prints a greeting message to all of my interns. In this made up scenario, I'm a boss that regularly communicates with his interns. And this function is just one simple generic message that I often have to use for each new batch of interns, so I made my job easier by automating this. This generic message introduces them to a new job, but let's imagine that I will also need a special function for other messages like introducing them to their



workplace or promoting them to a new position and so on. Since these messages will mostly be sent by email, I need to follow a certain email etiquette which means prepending the message with something like dear interns and ending the message with regards, your new boss. I could add this additional text here inside of the function, but then I would have to do it for every new function that produces a different message. This is bad for maintainability because if you decide to change how you write your emails in the future, you would have to make that modification in each function. Instead of doing that, you can create a decorator function to keep the separation of concerns in your code. This is a decorator that will make my messages email friendly. Like I first stated, a decorator is a higher-order function that takes in a function that it needs to decorate. The point of the decorator function is to create a new inner function which will add some additional behavior. As you can see, the inner function will first print Dear interns, and then it will actually call the original function from the decorators parameter list. This func function will be one of the functions that we talked about, the ones that produce generic messages for interns. After that, the inner function will close out the email with these two strings. So this inner function is actually calling the decorated function, but it also does something before and after it. Since this looks like the function is wrapping the original function with some additional behavior, some programmers refer to it as a wrapper function, hence the name I assigned. And when I say the original function, I mean this function from the parameter, the function that we want to decorate. Finally, like I said in the beginning, the decorator function will usually return a closure, but do you know why this wrapper function is a closure? It's a closure because it calls this func function which is not defined in its own local scope. This makes the func function a nonlocal variable, so the wrapper function is, therefore, a closure. Now, let's see how I can use this new decorator. I will first pass my greeting_message function to the decorator function. Decorator always takes in a function that it needs to decorate. A new instance of the wrapper closure will be returned, so I will reassign the greeting_message identifier to be the reference to that new function object. In the first lesson, I told you that you can think of function names as variable names that point to function objects. So this variable name will no longer be referring to this function object. It will, instead, refer to the wrapper closure returned from the decorator. The decorator function returns a new function that wraps around original functions behavior, and that's why we call them decorator functions because they decorate other functions with additional functionality. Of course, we could have assigned the decorated function to some other name, but the point of decorators is to use the same function with additional behavior without actually modifying the function itself or keeping track of some other version of this function. Let's see how it works. And as you can see, the function was successfully decorated. The function call looks like I'm calling the original function directly, but I'm, instead, calling the decorated version of this function. So to sum up, the greeting_message is now referring to the wrapper closure returned from the decorator. This wrapper closure is calling the original function from the decorator's



parameter. And finally, this parameter is nothing but a reference to the original function object. We will take a closer look at decorators in the next module, but I cannot finish this one without mentioning the syntactic sugar for decorating functions. Instead of manually reassigning this function name to a decorated function, Python offers us a special syntax. You can simply place the decorator name above the function's definition, but you have to prepend it with the @ sign. This syntax is equivalent to doing this, but it looks nicer and it allows you to immediately see which decorators are applied to the given function so that you don't have to search for that in some other part of the code.

Summary

It's time for the summary. In this module, we talked about functions as first class objects. That means that functions in Python can be used like any other object. You can assign them to different variables, store them in collections, or pass them to other functions. The functions that take another function as an argument or return a function are known as higher-order functions. The functions that you define inside of another function's body can be available outside of that outer function, but they can only be available if you return them from the higher-order function and make a new reference to that place in memory. Variables that you use from the inner functions in closing scope are known as free nonlocal variables. This is because they are not local to the function itself, but they are local to the enclosing scope. And functions that use nonlocal variables are known as closures because they enclose the function itself together with the extended scope of nonlocal variables. And finally, we talked about decorator functions which extend the functionality of other functions. They do that by taking in the function they want to decorate as an argument and call that function inside of another inner function. That inner function is a closure that needs to be returned from the decorator, and the usual further steps are to assign this new closure to the identifier of the original function or to use the decorator syntax on the original functions definition. And, that's it for this module. In the next module, we will dig deeper into decorators and see some examples of where the decorators are actually used in practice. Thanks for watching, and if you have any questions, ask them in the discussion.

Implementing Function Decorators

Digging Deeper into Decorators

Hi everyone. My name is Mateo, and welcome back to the course about Python Decorators. In the previous module, I showed you how you can utilize higher-order functions and closures to create decorators. In this module, we will dig



deep into decorators themselves and see why we even use them in Python. Here's the script you saw in the previous lesson which implements a simple email decorator. As I already mentioned, this syntax can be used to apply a decorator to a specific function. When Python interpreter sees this, it will first pass the original function itself to the decorator, and then it will reassign the original function's name to the function returned by the decorator. So, this is what the decorator syntax actually does, but we can also do this on our own. I will use the decorator syntax for the rest of the course, but in this lesson, I want to be more explicit, so I'll leave this statement so that we know what's going on. Now, let's see what Python does when we provide it with this code. The first thing it sees is the email decorator definition. This definition will create a new function object in memory which you can refer to with the email decorator label. The same thing will happen for the greeting_message function definition. A new function object will be allocated in memory. After that, we are calling the decorator by passing the original function as a first argument. Just like in many other languages, all of the function calls will be managed by a structure known as the call stack. Call stack keeps track of all active subroutines or functions in a program together with their return addresses local variables and function parameters. A function call will push a new frame on the call stack. A frame holds all of the necessary information required to evaluate a certain function. One part of the frame contains all of the arguments we passed in to the function. The func parameter will point to the greeting_message function object because that's the function we passed in as our first argument. Inside of the function definition, the first thing we see is the definition of the inner wrapper function. Just like we saw before, this definition will allocate a new function object in memory. Notice that the wrapper identifier is not defined inside of the global scope. The name of the function and the function itself belong to the local scope of the decorator function. Python will also analyze what's going on inside of the function definition, and it will see that I'm trying to access a variable which is not local to the wrapper function scope. This means that the func name in this context is a nonlocal variable, which also means that this function itself is a closure. So Python will automatically add a closure attribute to the function object itself. This attribute is a tuple of cell variables I mentioned in the previous module. Since we only have one nonlocal variable, the tuple will contain only one cell object which will point to the greeting_message function object. So the wrapper function is indirectly pointing to the original greeting_message function through the closure attribute. Once the decorator function returns, we are right back at this first assignment. The greeting_message identifier is here being reassigned from the original greeting_message function to the function object returned by the decorator. So the greeting_message is not pointing to the same function in the global scope anymore, it is now pointing to the returned closure function. And since the decorator call is over, its frame will be popped off the call stack along with all of the local variables. However, the wrapper function will still remain in memory because the greeting_message identifier is pointing to it from the global scope. Finally, I'm calling the newly-decorated greeting_message function. This



will, of course, create a new function frame on the stack, but which function are we actually calling here? The greeting message now points to the closure object returned from the decorator call, and the original greeting_message object is now referenced by a nonlocal variable from inside of the new function. Now, let me show you how you can check that what I said is actually true. I'm back in the code editor with the old email decorator script, but instead of just running the new function, I will show you the contents of the closure attribute. Let's run the script to see the output. The first output here is the closure tuple, and as expected, it only has one cell object. I am accessing that cell object directly by using the index notation. If you want to see the actual object that the cell is referring to, you need to use the cell contents attribute on the cell itself. You can see that the name of the function object is still greeting_message, which is not very helpful because this name now points to a different function in the global scope. However, we can take a look at the memory address of this original function object. Here, I use the id function to display the memory address of the newly-decorated greeting_message function. The most important thing to notice here is that this address is not the same as the address of the original greeting_message function, which proves that these two functions are indeed different objects in memory.

Decorating Functions with Arguments

Hopefully, now you can wrap your head around how decorators work in practice. Now, let's talk about decorating functions with arguments. The greeting_message function didn't define any parameters, so I modified it slightly to include this x parameter. So instead of welcoming interns to a new job, I can also welcome them to something else like to a desk, for example, which is what I passed in here as an argument. Now, let me run this program again to see if it still works, and it seems that we get a type error which says that our wrapper function local to the email decorator takes 0 positional arguments, but we provided one argument. The error reminds us that this function call is actually calling this function which doesn't have any parameters. The solution could be defining a parameter for this wrapper function and then passing it down to the nonlocal function call. And this would work, however, since decorators are usually meant to be reusable, we should be more flexible when handling arguments. So instead of defining specific amount of arguments, we can use the special args and kwargs parameters. I will assume that you are already familiar with how these parameters work. Basically, by using these special unpacking operators, you are instructing the function to take any number of arguments and store them in these parameters. All of the positional arguments will be stored inside of the args tuple, and all of the keyword arguments will be stored inside of the kwargs dictionary. So for example, when I just pass the desk string, this string will be placed inside of the args tuple. And now, we can use the same syntax on the nonlocal function as well. When you use these operators in arguments, instead of parameters, they do



the opposite of what they did with the parameters. In other words, the operators will unpack all of the values from the tuple and the dictionary and pass them as arguments to the func function. So if we just pass the desk, the function call will look like this. As you can see, I'm here basically taking all of the arguments passed to the closure, and I'm forwarding them to the original function that we want to decorate. If I run the script again, you can see that the function is working as expected. Defining decorated functions like this with the variable number of arguments is a common practice because we usually want to make our decorators reusable as possible. That being said, you are not required to do this, and maybe in some situations, you actually want to restrict the usage of your decorators to a specific type of function with a specific number of parameters.

Retaining Function's Original Metadata

The ability of an object to analyze its own attributes at runtime is a powerful feature which you can utilize in Python. This ability is known as introspection, and a lot of tools and libraries use this feature to work with objects metadata. This metadata that I'm referring to is, for example, the function's name or its docstring. Since we didn't define any docstrings in our example, I took the liberty of adding a short description to the function. And here, I decided to print out some of these introspection attributes. The `__name__` attribute should return the function's original name. The `__doc__` should return the documentation string that belongs to this function, and that's the first comment in the function's body. So what do you think will be the output when I run this script? Let's see. As you can see, the output is not what I expected. Imagine if you use this decorated function in a large code base and you don't know that the function is being decorated at all. This could definitely lead to some issues due to the loss of metadata, especially if you use it in combination with some external tools and libraries. To prevent this from happening, you have to always be aware that your decorators will change the metadata of the original function. Actually, that's not even what's happening here. The metadata of the original function is still intact. The problem is that the name of the original function is now pointing to this wrapper closure. That's why we get this output because the name we are trying to get is the name of this inner function, and since this function doesn't have a docstring, the second output is none. The solution for this might be to update all of the wrappers metadata to the metadata of the original function which we can get from the func parameter here. And that would work, but we would then have to repeat this code for every custom decorator we make because every decorator would need to do this before returning the newly-decorated function. So, which feature of Python do you know of that we can use to add the same exact code to more functions? And of course, the answer is decorators because remember, decorators can be used to decorate the function to add some additional functionality without explicitly modifying the function's body. However, instead of creating this decorator ourselves, we can use the built-



in Python decorator which was specifically created for this purpose. The decorator's name is `wraps`, and you can import it from the `func tools` module. Python language developers were aware of this metadata problem and that's why they created this function which we can utilize whenever we create a new decorator. To apply this decorator, you need to use the decorator syntax on the wrapper closure inside of the decorator function. You also need to provide a reference to the original function as an argument of the decorator itself. We will talk about decorators with arguments in the next module. For now, you can just memorize that you need to use the `wraps` function like this, and that's all you need to do. Now, let's save the script and try to run it again. Finally, we get the result that we expected to get on the first try. The whole point of decorators is that they are not intrusive. They should just add some additional functionality to the function while leaving the original function as it is. That is why the decorated function is automatically assigned to the name of the original function because we don't want to or need to know how the decorator works. We just need it to work so that we can continue using the original function with that additional functionality. And this abstraction should also include the awareness of metadata. So whenever you create the decorators, you should use the `wraps` function to keep the metadata of the original function. As you can see, the usage of the `wraps` decorator is straightforward and it works without us even knowing how, but of course, since this is a course about decorators, we will talk more about this in the next module.

Commonly Used Decorators in Python

Aside from the decorators that you make yourself, the most common decorators that you are going to use are from other libraries like web frameworks, for example. However, there are some decorators in the Python standard library that you will most likely use, especially if you are doing some object-oriented programming. In this course, I will just glance over them. If you want to know more, you can check out my course about object-oriented programming. The first decorator I want to mention is the `classmethod` decorator. Usually, every method you define inside of the class is known as an instance method and it has to receive the required first parameter which references an instance of that class. However, there are some special cases when you want to call a method directly from the class itself, and that's where the `classmethod` decorator comes in. This decorator will turn the method into a classmethod and automatically pass it the class itself as the first parameter. There's also a legacy static method decorator that has similar functionality, but it's not used as much. Another decorator that you will probably use in your custom classes is the `property` decorator. Properties are Python's alternative to getters and setters. Instead of allowing the application code to access all of the instance attributes directly, you can create property methods by using the `property` decorator. So for example, here, I'm trying to get the value from the `x` attribute of the `c` instance. To me, it



looks like I'm accessing a plain simple attribute, but Python will actually invoke this property-decorated function and return the value from the hidden attribute. The same will happen when I try to set a new value for the x attribute. Instead of setting this value directly, this setter method will be invoked and it will first check if the new value is lesser than 5. So as you can see, this allows me to implement some additional validation without hurting the experience of the developer who uses my custom class. Notice that the decorator for the setter method is a little bit weird because we are somehow using the setter attribute of some x decorator. In this lesson, I'm just glancing over the examples, but we will get back to this specific decorator in the last module of the course. Finally, the last decorator that you might want to use is the LRU cache decorator. This decorator is available from the func tools module, which is the same module that contains the wrap decorator we used for managing functions metadata. LRU stands for least recently used cache, which is basically a special caching algorithm for optimizing functions that are computationally expensive and called repeatedly with the same arguments. For example, here I wrote a function that calculates the Fibonacci number using our recursive approach. This function will always return the same result for the same number in the argument, so the Fibonacci number for number 100 will always be the same. So if someone already calculated the Fibonacci number of 100 in the same program, I don't want to calculate it again to loosen performance. If I try to call this function again with the same argument, instead of wasting CPU power and calculating the same result again, the LRU cache will return the result of the last computation. Notice that this decorator also requires me to provide a maxsize argument. As I said, we will talk about decorators with arguments in the next module. So these are all of the decorators from the standard library that I wanted to mention, the class method, the property, LRU cache, and the wraps decorator. In the following modules, I will also mention some examples from the popular libraries and the web frameworks.

Implementing Your Own Decorators

Now that we are familiar with how decorators work, let's implement some common examples of custom decorators. For example, there are a lot of libraries that you can use to implement a logging system, but what if I don't want to use a lot of dependencies for my simple project? Instead, I can use the Python's logging module to create a simple logging decorator. Here, I just globally configured the logging system to always use the INFO level of alert. This is not really important. The point of this decorator is to log a function call. And of course, I'm using the wraps function to keep the metadata of the decorated function. The wrapper function will simply use the logging module to output the name of the decorated function together with the arguments we provided to the function call. And here's something we didn't do before. After the logging is done, I'm actually calling the original function and returning the result. Until now, we



never actually returned anything from the inner function because all of the examples I implemented didn't need to return anything. My email decorator would return none. However, a good practice in most decorators is to return the value from the original function call, even if it's just none because you might change your mind in the future and then your decorated functions will not be able to return anything. And here is the simple add function I decided to decorate with my logging decorator. As you can see, the add function will actually return something, and if I didn't return the result in here, the decorated function would simply return none, which is not what we want. Finally, I'm here calling the function and printing the result to the terminal. Let's see if this works, and it does. The 2 plus 3 are indeed 5, and above that, you can see the logging output provided by my logging decorator. Now, let's take a look at another example. In this example, I'm utilizing the built-in time module to calculate how much time did the function need to finish running? First, I'm storing the time before I run the function. Then, I call the function, and after that, I calculate the `elapsed_time`. I can then output the time to the terminal. Finally, like in the previous example, I will return the result of the function call. To try out this decorator, I decorated a `slow_function` which emulates a function that has to do some long-running calculations. Let's run this now. As you can see, the function took a little bit longer than 1 second, which is what we expected. So if you don't want to include third-party libraries in your simple prototype applications, you can just make your own decorators and reuse them on different kinds of functions. Ultimately, you are the one who should decide if decorators belong to your workflow or not, but even if you don't create your own, it is always good to know how decorators work because toolmakers often use them in libraries and frameworks.

Summary

It's time for the module summary. In this module, we took a closer look at how decorators work and how to decorate functions that have arguments. You would usually use the `args` and `kwargs` special parameters to handle a variable number of arguments so that your decorators work on different kinds of functions. I also mentioned that you can use the `wraps` decorator from the `func tools` package to preserve the metadata of the original function. The original function's name is tied to the closure returned by the decorator, so we need to update the metadata of that closure to keep the abstraction intact. I mentioned some commonly-used decorators from the standard library like the `wraps` and `LRU cache` decorators from the `func tools` module. And you'll also probably encounter decorators related to classes like the `class method decorator` and the `property decorator`. If you don't want to add a lot of dependencies to our projects, you can create your own simple decorators for purposes like timing performance or logging. Of course, this is not the only use case for decorators, and we will cover more examples in the following modules. And that's it for this module. In the next module, I will cover some advanced decorator workflows such as decorators with



arguments and how to apply more decorators on a single function.

Using Advanced Decorator Workflows

Handling Arguments with Decorator Factories

Hi everyone. My name is Mateo, and welcome back to the course about Python decorators. In the previous module, I showed you some examples of decorators from the standard library. The decorators from the `func tools` package, the `wraps` decorator, and the `LRU cache` decorator required me to pass an argument directly to the decorator itself. So the first thing that I'm going to cover in this module is how to create decorators with arguments. As a reminder, here is a simple implementation of our regular decorator without the arguments. I already covered how this works and what this decorator syntax actually represents. Now, let's see decorators with arguments. If you want to add parameters to your decorator, you will need to add an additional layer on top of the decorator function itself. When you use the same decorator syntax and you include an argument, Python will resolve this syntax in a slightly different way. The syntax will actually be equal to this statement. Let's first see how this decorator is implemented. Notice that I named the decorator `decorator_factory`. This is not an official term that you will see in Python documentation, but it is technically correct. So why did I call it the `decorator_factory`? When you create decorators with arguments, the outermost function that you define should actually define a decorator function and then return it. That's why I called it a `decorator_factory` because the function literally defines and returns a new decorator function. In the first module of this course, I defined the decorator as a higher-order function that takes in a function object as an argument and returns a closure, and that's exactly what this decorator function does, but in this case, the decorator itself is defined as an inner function of the `decorator_factory`. There's nothing special about this decorator. It just defines a closure with some decorator functionality, and it evaluates the original function from inside of the closure. However, notice that I'm using the `some_arg` variable inside of the wrapper function. This is the parameter from the `decorator_factory` which means that the wrapper will have to create another nonlocal variable to store a reference to the subject in memory. So the only difference between decorators with and without arguments is that decorators without arguments are directly applied to the function they need to decorate, while decorators with arguments first need to be returned from the `decorator_factory`, and that's the reason why the decorator syntax results to a different kind of statement here. If it simply worked in the same way as before, then the function we want to decorate would be passed in as a first argument of the `decorator_factory`, which makes no sense because the point of the `decorator_factory` is to take in some additional arguments. So the first thing that needs to happen is the resolving of the `decorator_factory`. The



`decorator_factory` will define a decorator and return it. And then we're left with the old familiar decorator that takes in a function object, and you know the rest of the story. In this lesson, I use the term `decorator_factory` so that you can distinguish between a function that defines a decorator and the `decorator_function` itself. But to keep the abstraction consistent, you should name your `decorator_factory` with the same name that you would give to the `decorator_function` itself. The users of your decorator don't need to know about your decorator implementation. To keep things simple, you can give the same name to the actual decorator, but with the addition of an underscore in front of the name. In this way, you will still be able to see how everything works without breaking the abstraction.

Implementing Decorators with Arguments

Now that we know how decorators with arguments work in theory, let's see it in practice. I modified my old email decorator to include the `fromwho` argument so that we can modify the sender of the email. Instead of the boss, I want the team leader to welcome all of the new interns. The outermost `email_decorator` function is a decorator factory. This function defines and returns the old email decorator we used in the previous lessons. The only difference is that I am now referencing this nonlocal variable from inside of the wrapper function. So the decorator factory will return a decorator which in turn will be applied to the `greeting_message` function. Let's see if this works. As you can see, the interns are now greeted by a team leader. Great.

Stacking Multiple Decorators on a Single Function

Until now, we only applied a single decorator to a function. However, there is nothing really stopping you from decorating a function with multiple decorators. To do that, you can simply stack decorators on top of each other while using the familiar decorator syntax. When you stack multiple decorators on a single function, the syntax will resolve to a statement that looks like this. The decorator on the top will take in the next decorator call as an argument. It doesn't matter how many decorators you stack on top of each other, the top decorator will always take the next decorator as an argument until it gets to the last one. So let's quickly see how this works. `Decorator2` will take in the function object as usual and return our wrapper closure. This closure object will then be passed as an argument to the `decorator1`. The `decorator1` will create another closure which will also run the received closure from `decorator2`. And finally, the second closure will be returned and assigned to the original function's name. So, when you finally call this new decorated function, you are actually calling this closure from `decorator1`, and this closure is also calling the closure from the `decorator2`. And finally, this second closure will call the original function, but in this case, the



original function doesn't do anything. So keep in mind that when you stack multiple decorators, the top decorator functionality will be evaluated first. The order of evaluation goes from top to bottom. This might be intuitive for some of you, but it's really important that you keep that in mind because I'll show you a real use case in the next lesson. Now, let's see how this would work with my email decorator. I defined another decorator and named it `company_info`. This decorator simply prints the company's contact info at the end of the email. And now, I'll stack these two decorators on top of the `greeting_message` function. Let's run this. And as you can see, the company info is placed at the end of the email. Again, the order in which you place the decorators is important because the `company_info` decorator will first run the decorated function before printing the information about a company. If you reverse the order of these decorators, then this company information would be printed after the greeting message. As an exercise, you can try to guess the exact output and see if you got it right.

Custom Decorators in Flask

Now that we covered the decorators with arguments and how to apply multiple decorators, I will show you a real use case for this concept in a Flask web framework. This is not a course about web development, so I will not show you how to use Flask. However, I will show you an example of custom decorators I created for authentication and authorization. If you want to know more, you can check out my course about Flask users sessions and authentication. This is the core of every Flask application, the route decorator. The route decorator is a decorator with an argument, and this argument is a URL path that will be resolved by the decorated function. So when someone visits the landing page of my website, which is just a slash, I want to return this HTML. This is a great example of how frameworks and libraries make use of decorators to provide you with a simple programming interface. Now, let's see my custom decorators for authentication and authorization. How these decorators actually work is not important. I'm here mainly interested in showing you how I utilize my knowledge about decorators to implement this additional functionality. The `login_required` decorator is used for authentication purposes. As you can see, I placed it under the route decorator. Remember, when you stack decorators on top of each other, the order is important. The route decorator is actually the one that receives the HTTP requests from the client, so that decorator needs to be resolved first. After the request is received, I will first check if the user is authenticated before running the logout function. This is because only logged in users should be able to log out. The inner function will first check if the current site visitor is anonymous. If it is, that means that the visitor is not logged in. In that case, I will just provide them with this message and redirect them to the login page. However, if the user is not anonymous, I will run the `nonlocal` function, which in this case is a logout function. And now, let's take a look at the



second decorator. This one I created for authorization. Notice that this `role_required` decorator takes in an argument, so this decorator is actually a decorator factory. In this case, I used three decorators. The first one receives an HTTP request for creating a new gig. The second one checks if the user is logged in. And finally, the third one checks if the role of the logged in user is an employer because only employers can create gigs. So the `role_required` is a decorator factory which receives a role as a single argument. This inner function is an actual decorator. The closure will first check if the logged in user has the correct role. If it doesn't, I will redirect the user to the home page because that user is not authorized for this action. However, if the user has the correct role, I will run the `create` function to create a new gig. So as you can see, the knowledge about decorators didn't only help me to understand how Flask decorators are implemented, but it also allowed me to modify the framework's behavior by creating my own decorators.

Summary

It's time for the module summary. In this module, we talked about decorators with arguments. Decorators with arguments are implemented just like regular decorators, but with an additional function layer over the actual decorator. I call this function a decorator factory because the function that actually receives the argument is the one that returns a decorator function. You can use multiple decorators on a single function. The order of decorators is important because the topmost decorator will always be evaluated first. I showed you how web frameworks like Flask use decorators to implement utilities for developers. The main function that the whole framework revolves around is a route decorator which is used for handling incoming HTTP requests. In the last module of the course, I will talk about classes and how they relate to decorators in Python. If you have any questions, don't hesitate to ask them in the discussion.

Decorating Classes and Class Decorators

Using Classes as Decorators

Hi everyone. My name is Mateo, and welcome back to the course about Python decorators. In this module, I will talk about how the concept of decorators relates to classes and instances. Before watching this module, it is really important that you are familiar with object-oriented programming in Python. If you don't know object-oriented programming or you need a refresher, check out my course about [Classes and Object-oriented Programming in Python](#). So the decorators are just specific kind of functions that we can use with the following decorator syntax. In this example, I'm using a decorator with an argument which



we learned about in the previous module. So how can we use classes to implement this exact decorator? I will use my own custom decorator class, instead of a decorator factory. The decorator's argument, `some_arg`, will be received by the `init` function of the `DecoratorClass`, and that value will be stored inside of the instance attribute of the same name. If you don't remember, the `init` function is automatically called in this situation when we want to instantiate a new `DecoratorClass` instance. This object is now a new `DecoratorClass` instance which includes the `some_arg` attribute, and because of the initialization, the value stored inside of the attribute is this value string. The task of the `decorator_function` will be delegated to the `call` instance method of the `DecoratorClass`. Since this method will act as a decorator, it will need to take in a function object and return a new closure. However, in this case, the accessing of the `some_arg` value is different. Since the `call` function is not defined inside of the local scope of the decorator factory, we cannot access it directly as a nonlocal variable, but this is not a problem since we can directly access it from the instance attribute itself. This is because the instance is always passed in automatically to the instance method through this `self` parameter. And that's it. That's how you can use classes to create decorators. However, the decorating, in this case, seems really verbose and complicated. That's because I wanted to show you how it works, but of course, this can be written in a much better way. The whole point of this magic `_call_` method is to define what will happen if someone decides to call the instance itself. So, we don't need to explicitly call this `call` method. We can simply just call the instance itself like a function and that will actually trigger the `call` method. And now, we are left with this statement which should look familiar. Calling the decorator with the original function and reassigning the returned closure to the original function's name is exactly what the decorator syntax does. So instead of this statement, we can simply use the decorator syntax with the instance. And this looks much better, but why do we even need to assign this instance to a variable? This `DecoratorClass` call will produce a new instance, but instead of assigning this instance to a variable, we can simply use this new instance directly as a decorator. As you can see, using a class, instead of a function, to create a new decorator is not much more verbose. All you need to define is the `init` function and the `call` function, but of course, you can add some additional functionality if you want. To solidify this concept, I recreated my email decorator, but this time, I used the `EmailDecorator` class. The `fromWho` argument is assigned to the instance attribute of the same name. The `call` function plays the role of the decorator function. The only difference here is that I have to access this decorator argument from the instance itself. And that's it. That's all I need to do to turn this decorator into a class. The application of this new decorator is easy. I can use it in the same way as the function decorator. If I use lowercase letters to name this class, the user wouldn't even know that this decorator is a class. Now, let's run this to see if everything works, and it does. We are presented with exactly the same output. If you have any questions about this, please ask them in the discussion.



Property is a Class Decorator

Now that we know about class decorators, I want to talk about the property decorator from the standard library. I hope that you're familiar with object-oriented programming in Python because this lesson will require some knowledge from that area. The point of this lesson is not to teach you how to implement your own property decorator, although no one is stopping you from doing that. The reason I wanted to include this exercise is so that you can train your mind to see through the decorator syntax and think about how decorators are actually implemented. So to start off, I defined this `MyClass` class, which is a simple class that defines a single instance attribute `x`. The underscore in front of the name signifies that this is meant to be a non-public attribute, and I don't want users to access this attribute directly. This is where the concept of managed attributes, also known as properties, come in. To define properties for your classes, you can use the built-in property class. And yes, you heard me right, this property is a class, it's not a function. Don't be confused by the lowercase naming. The standard, in this case, is avoided in favor of providing developers with an abstraction. Since the property class is used like a function, Python language creators decided that we don't need to explicitly know that this is a class, but this is a class, and this is what the signature for this class looks like. These are all of the initialization parameters that we can provide to the property init function. However, notice that they're also optional, so we usually just provide the first two. By calling the property class like this, we are actually creating a new property object, which is not unusual, it's a new instance. However, it is important to note that the property class implements a descriptor protocol, which means that its objects are descriptor objects. A descriptor object is an object that implements these two dunder methods, `get` and `set`. And this is exactly what the property class does, but it also defines this `delete` method. These methods define what happens when you decide to get a value or set a new value to this object. Let me show you what I mean in practice. I defined these two methods which in other languages would be considered as a getter and setter method for the private `x` attribute. The getter method retrieves the value from the attribute, and the setter method sets a new value to that same attribute. Usually, you would include some kind of validation in the setter method to check if the new value is appropriate, but I will keep things simple. The point of these methods is that the user should use them to manage the `x` attribute, instead of accessing the attribute directly. So as you can see, I'm passing these two methods as the first two arguments of the property initialization. The property class will take your methods and attach them to the appropriate descriptor methods. So the `get_x` method will be used to resolve the `get` function, and the `set_x` method will be used to resolve the `set` function. Your methods define behavior for the descriptor methods in this new object `x`. Since the `x` object is defined in the class's scope, that means that the `x` is a class attribute, it belongs to the class, instead of to some specific class



instance. Down here, I instantiated a new instance of my class, and then I used it to set the new x value and print that value to the terminal. The new instance doesn't have an attribute with the name x, it only has this `_x` attribute, which I defined inside of the init function. When instances don't have attributes that you are trying to access, they will take a look if the class has them. This is known as an attribute lookup chain. If the attribute doesn't exist in the instance, the next step is to look for it in a class which created the instance. So here, I'm trying to access the value of the class attribute x. This expression will trigger the descriptor method `get`, and as you know, the property class assigned your own `get_x` function to that method. So the expression will actually resolve to this. The `get` descriptor method from the x class attribute will be called with two arguments, the instance itself and the class. The instance will then be passed into the `get_x` method as this `self` parameter, and then it will simply return the value of the hidden x attribute. So by accessing this x attribute, we get the value from the non-public x attribute, and something similar will happen to this statement. Here, I'm trying to set the new value to the x object. This will trigger the set descriptor method. And of course, this descriptor method will pass the instance and the new value to your `set_x` function. This function will then simply set the new value to the non-public x attribute. As you can see, by using this property object, we created a layer above the non-public x attribute. So when users try to do something with the attribute x, they're actually calling the getter and setter functions from your class. Since all of these property parameters are optional, you don't have to define a setter function. A property can also be created with just one getter function. If you want to add the setter function after the object is already created, you can use the setter method from the property object x. The setter method will connect your `set_x` method to the set descriptor method, and it will return a new object which includes both the getter and the setter method. So yes, this is the same thing we did before, but in two statements, instead of one. But this is useful because now we can turn these statements into decorators. Since the property class can take in a function as an argument, I can use it as a decorator over the getter function. I will also rename the getter function to just x. And as you know, this syntax will resolve to a statement that looks like this. Since the decorator syntax will resolve to this exact statement, there is no need for this statement to exist anymore. And we can do the same with this other statement. I can use the setter attribute from the new x object to decorate the setter function. And if I rename the function to x as well, this decorator syntax will resolve to this statement. So this statement is also not needed anymore. And here's what we are left with. This is what all of the tutorials online will show you when you search for creating properties in Python. Python language developers use the power of decorators to create this compact interface for the programmers. In the older versions of Python, you had to directly use this property class to create properties, but as you can see, this syntax is much more understandable and much more intuitive.



Decorating Classes

Now that we covered classes as decorators, let's talk about decorating classes. So I'm not talking about using classes as decorators. I'm talking about using decorators to decorate classes, instead of functions. We know that this decorator syntax will resolve to a statement that looks like this. Well, it turns out that this syntax doesn't care if the decorator is working with a function object or with some other type of an object like a class object. This syntax will pass the class to the decorator and assign the returned value to the same class name just like it does with the function object. The whole purpose of decorating classes is usually something known as monkey patching. The term monkey patching refers to dynamically adding attributes to an object at runtime. As you can see, this class doesn't define any methods. The `add_speech` decorator should dynamically add a new `speak` method to this class, so the `add_speech` decorator will take in a class as an argument. And all I want to do here is to simply assign a new `speak` method to that class. To keep things simple, I just defined an anonymous lambda that prints the name of the class to the terminal. Remember, methods are just class attributes, so instead of defining this method directly inside of the class, I'm using this decorator to append it while the program is running. To see if this works, I will instantiate a new instance object from that class and call this `speak` method. Let's run this now. As you can see, there is a type error which says that the `NoneType` object is not callable. Check out the code and try to guess why this error happened. You can stop the video if you want to. And, did you figure it out? Well, it turns out that I forgot to return the class from the decorator. The decorator doesn't return anything, and you know that functions that don't return anything will simply return `none`. So the `SomeClass` variable will now point to `none`, instead of a class object. And here, I'm trying to call `None` object to create a new instance which will result in an error. To fix this, I need to return the class itself from the decorator. And now if I run the script again, you can see that the instance successfully called the `speak` method. Of course, this decorator doesn't make much sense because I could have just defined the `speak` method inside of the class. And if I needed this method in more classes, I could have used inheritance, instead of a decorator. The truth is, I don't have a good custom example for this situation that you can use in a simple script like this one. If you were a toolmaker of some kind and decided to create some kind of a library, this concept might be useful for registering classes to some kind of a global registry. However, the real reason I showed you how to decorate classes is because there are some decorators for classes in the standard library, and I'm going to show them in the next lesson.

Decorators from the Standard Library

In this lesson, I will quickly glance over two decorators for classes from the



standard library. The first one is the `total_ordering` decorator from the `func tools` module, the same module that implements the `wraps` and the `LRU cache` decorators. Some of the custom classes that you create will need to implement a special behavior for when you're trying to compare class instances by using comparison operators. These two dunder methods will define what happens when you try to check if the two instances of this class are equal or which one is lesser than the other. It turns out that all of the other comparison operators can be inferred from these two. So instead of defining the rest of comparison methods yourself, you can simply decorate your class with the `total_ordering` decorator. This decorator will take a look at your existing methods and dynamically attach all of the other comparison methods to your class. This is really useful in certain situations, but you should use it with caution because it does have certain drawbacks. The other decorator, which is getting more popular in recent years, is the `dataclass` decorator. The `dataclass` decorator will dynamically turn your regular class into a data class. If you want to know more about this, check out my course about object-oriented programming. But in essence, a lot of classes you make will have the same boilerplate code. You will probably define the `init` function, which will take in some arguments, and you will assign them to the instance attributes of the same name. And you'll also need some other dunder methods like `repr` for turning your class into a string. Instead of defining all of this yourself, you can simply define all of the required instance attributes as class attributes. Then you can simply decorate your class with the `dataclass` decorator, and this decorator will generate all of the boilerplate code from your class attributes. So as you can see, the decorators for classes also have their purpose in certain scenarios.

Summary

It's time for the last summary of the course. In this module, we talked about class decorators. Classes can also be used to create decorators, you just need to implement the `init` function and the `call` function. The calling of the class instance will serve the same purpose as the regular decorator function. Property decorator from the standard library is actually a class. This class returns a descriptor object and it lets you define the descriptor methods of this new object. The decorator syntax can also be used on classes. A decorator for classes will take in a class as an argument and return that same class, but the returned class will usually have some new dynamically-added methods. For example, the `data class` decorator from the standard library will dynamically attach boilerplate methods to your custom class. And that's all I wanted to show you in this course. Thank you for watching, and if you have any questions, please ask them in the discussion. If you leave your questions as a feedback, you will not get an answer because I cannot answer to an anonymous feedback.