# **Python: Decorators**

### **Decorators in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Have you ever seen this weird at symbol above function definitions in Python programs and wondered what it meant? Well these are decorators and you'll find them sprinkled around any Python code that is of significant size. Decorators can be tricky to understand initially but we'll look at loads of examples and see how they're used in production code. Change of topic. Now I find the Fibonacci sequence fascinating. It's all over nature from the arrangement of leaves on a stem to the number of petals of flowers. So what better way to learn about decorators than to combine it with all things Fibonacci? Hi, I'm Jonathan Fernandes, and I work in data science, machine learning, and AI for consultancy. And I use Python an awful lot. Come join me and let's look at decorators together.

## **Question 1 of 1**

Which option is an example of a decorator in Python code?

=property

* def propagate\_exceptions(self):

#property

* def propagate\_exceptions(self):

@property

* def propagate\_exceptions(self):
* Correct

&property

* def propagate\_exceptions(self):

### **Working with functions in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- Now, because decorators are mainly applied to functions, and classes in Python, it's important that we have a good understanding of how these behave. So I'm going to use visual studio as my editor, so you can use whatever you like. Now, I'm going to just open a new terminal. So, terminal. I'm going to just increase the size of my screen, and I'm going to head over to the Python ripple. Let's create a simple function called "Print fib" that prints out the string, Fibonacci. So, def printfib. And I'm going to include a doc string, so this is the line after the function definition that tells you what the function does. And then we want to just do a print Fibonacci. Now, if you try and use the function without the parentheses, Python won't complain, but it won't do anything useful either. And when we call the printfib function, we get the string, Fibonacci. Now lets do two further checks. First, let's just confirm what type of function pfib is. So, I do a type printfib. This confirms that functions are objects, like everything else in Python. And if we look to see what the function returns, we can see that since there's no return statement, the printfib function returns a nontype. So let's modify the function printfib. So this time, instead of just printing Fibonacci, we return the string, Fibonacci. So def printfib. And we modify our doc string. So. And we now have the function returning the string Fibonacci. So if I go ahead and call the printfib function, printfib, and I look at the type, and you can see that this time around, a string has been returned. The Fibonacci sequence is a sequence of numbers, such that any number, except for the first and second, is the sum of the previous two. So, here's the sequence. So, it's one, one, two, three, five, eight, 13, 21, and so on. Now, we can easily rewrite the Fibonacci sequence with the following mathematical definition. And we define the first two terms as being one. Now, we're not going to do anything fancy, we're just going to take that definition, and create a function called fib, that takes n as input a number, and returns the number from the Fibonacci sequence. So, I'm just going to clear my screen. Just going to enter the Python ripple again. So, Python, and then def fib(n), return the nth number in the Fibonacci sequence. And so if n is less than two, you'll want to return n. Else. And then just following that mathematical definition of Fibonacci. Return fib(n-1) + fib(n-2). And that will sum up the previous two terms. Now let's make sure this is working correctly. So if we input the first couple of values, so I expect to get a one for fib(1), and for fib(2). And let's go ahead and try the eighth value, and so I expect to get a 21 for the eighth number in the sequence. So we got our Fibonacci sequence working. Now in Python, functions are first class objects. This means that the functions can be passed around and used as arguments, just like any other objects. So these are strings, integers, floats, and so on. So you can call a function from another function. And you've probably already done this when using the Berton help function. So I can get some documentation on the fib function that we've just created. So I do a help, and then I passen, as an argument, the fib function. And what's returned is the doc string. So this was a quick review of functions. Next, let's look at working with functions within functions.

### **Functions within functions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] Now, one of the reasons why we're interested with working with functions within functions is because this is the basis for decorators. So let's kick things off with an example, let's say I have a function called fib\_three. And within this function, I create a, another function called get\_three, which returns the three Fibonacci numbers. So let's define fib\_three. So fib\_three accepts three arguments, a b and c, which are the three Fibonacci numbers. And then we have the function get\_three within fib\_three. So, and this one returns, the three Fibonacci numbers. So return a, b and c. Now functions are actually objects in Python. I can pass them around. I can store them in variables. I can also create the function, get\_three and return the function back as a result. So let's do that. So return get\_three. Now from Python's perspective, there's no difference between doing this and returning an integer or a string or a tuple. So let's call it fib\_three with the first three numbers of the Fibonacci sequence. Now that's not surprising because what has been returned is the reference to fib\_three. I can assign that to a variable. So let's assign that to variable f and I can go ahead and call f. Now that's interesting because the get\_three function captures the surrounding environment. So although we don't have any arguments in the get three function, we're able to return the three arguments used in the fib\_three function. The reason that this works is that the inner function get\_three, captures those variables and kind of takes them along with the function using closures. So we're now in a position to take a look at decorators in the next video.

>>>>Function inside function is also possible

## **Question 1 of 2**

Functions are actually \_\_\_\_\_ in Python.

* arrays
* objects  
  Correct
* classes
* variables

## **Question 2 of 2**

Which code, when executed, has the function return the string Fibonacci and looks like the code below?

<class 'str'>

def printfib():

"Print out Fibonacci"

print ("Fibonacci")

* type(printfib())

def printfib():

("Return Fibonacci")

print `Fibonacci`

* type(printfib())

def printfib():

'''Return Fibonacci'''

return 'Fibonacci'

* type(printfib())
* Correct

def printfib():

```Print out Fibonacci```

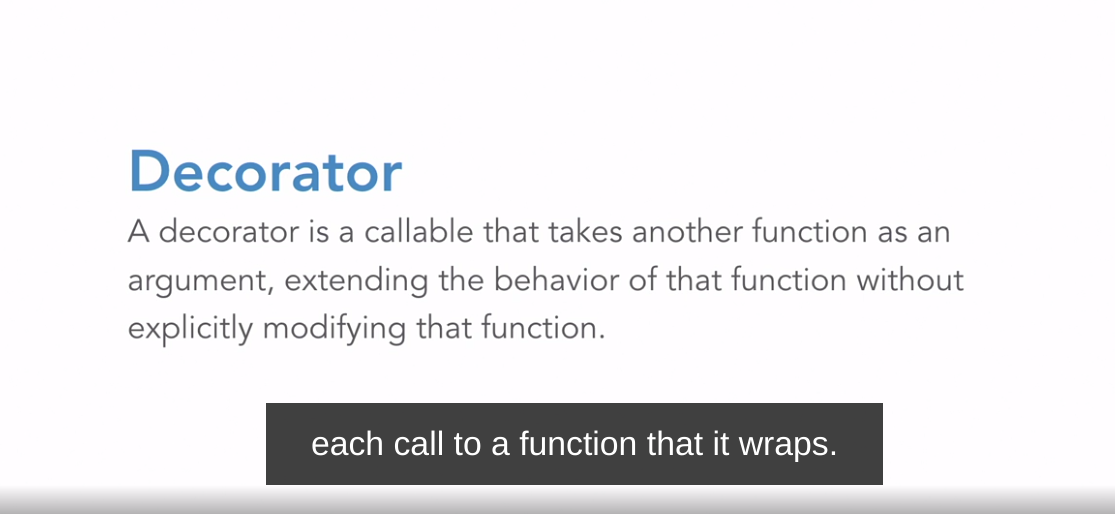
print `String`

* type(printfib())

### **What are decorators?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now that we've reviewed functions in general let's look at decorators. A decorator is a callable that takes another function as an argument, extending the behavior of that function without explicitly modifying that function. So a decorator has the ability to run additional code before and after each call to a function that it wraps. This means that decorators can access and modify input arguments and return values. So why do we use decorators with functions? I mean if I want to change a function why don't I just go ahead edit the function? Well let me give you an example to illustrate. Now let's say I have a project. And say I want to do the same operation to all of the functions like adding log into the function. Now this is a big project, so I've got something like 50 different functions. I could go ahead and edit each of the functions. So I go into each function, paste some code around logging that function, and then move on to the next function. **Alternatively, I could use decorators. The advantage of a decorator is that it will change the behavior of that function without permanently modifying it. So let's say later on I decide that I don't want to be logging the functions it's easier to remove this than going into each function and removing some lines of code.** Let's go back to our simple print Fib example. So go ahead and open the file my decorator.py. As you can see this already has the function pfib that we created earlier. Now let's create another function that we'll call my decorator above it. So, def my\_decorator, and we'll take in as an argumentative function. And so this is our decorator function. The original pfib function only returns Fibonacci in low case. This time round, we'll return Fibonacci but in capitals with a dash or a hyphen between each letter. So I'm going to create my wrapper function, so wrapper. And we say this returns a string. So I'm going to type FIBONACCI, and I'm going to put the hyphen between each of the letters. And let me copy that string and let's return Fibonacci that way. So return, and our string F-I-B-O-N-A-C-C-I. And at the end of this, we're going to return a wrapper function. So return wrapper. Now let's print out the pfib function so that we can see the output. So print pfib. Now let's confirm that if you call pfib you'll just get the output Fibonacci. So python, and my file, just my decorator.py and you can see we've got the output Fibonacci. Now if we go ahead and define the following, so we say, pfib = my\_decorator. And we provide as an argument pfib. What do you think the output will be? Well let's run the function and take a look. Pfib now points to the inner function called wrapper within my\_decorator. Now you may think, "wow, when did we do that?" When you call the my\_decorator function with pfib as an argument, what's being returned is the wrapper. And this assigned to pfib. Now Python allows you to use decorators with the simplest syntax using the add symbol. So adding a my\_decorator before the pfib function definition is just another way of applying a decorator to a function. So let's go ahead and replace our current definition with the @my\_decorator syntax before the function pfib. So I'm going to remove that and use the alternate syntax @my\_decorator, and let's confirm that we get the same output. So decorators are powerful because they can replace the decorated function with a different one. But let's confirm this by printing out the reference to pfib. So I'm going to replace my call to the pfib function with just printing out pfib. And you can see that it has a reference to the the wrapper function in my\_decorator. So let's complete this section with a summary of what a decorators typically look like. So on the first line we've got the function definition called my\_decorator, taking in as an argument the function func. We then an inner function which I've called wrapper. Now in wrapper you normally do something before calling the function. Whatever is returned from calling the function func is stored in result. And you might want to do something after calling the function. And finally we return the inner function, wrapper. In the Fibonacci example we looked at it just so happened that we didn't have to do anything before or after calling the function.



### **Challenge: Beautify your output**

Selecting transcript lines in this section will navigate to timestamp in the video

(pleasant electronic music) - [Instructor] Go ahead and grab the make\_posh Python file. We'll use this as our starting point. You can see that there's already a skeleton decorator function written for you. Now, got ahead and create the contents of the decorator called make\_posh so that the Fibonacci, well, looks posh. The expected output is as shown. This should take you about five minutes.

### **Solution: Beautify your output**

Selecting transcript lines in this section will navigate to timestamp in the video

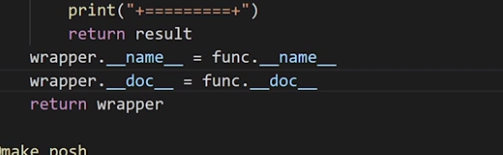
(bright music) - [Instructor] Let's use the decorator template as our starting point. Now one possible solution is to print out the top half of the box then call the PFIB function and the finally print out the bottom half of the box. So let's modify our wrapper to do just that. Let's just experiment to try and figure out how we print out the top half of the box. Now Fibonacci has nine characters so we need nine dashes and then we need a symbol on either side. So print, a cross, and then nine dashes. And another cross. We then need the borders. So that's the pipe symbol and then nine spaces. And the pipe symbol again. We then want to call our functions, so result equals func. And finally let's print out the bottom half of the box. So we need the same border so I'm just going to copy the line from above. And we need the bottom row. And we need to use equals signs instead of the dashes so we need nine equals signs. So we return the result and we're going to return the wrapper. So we've decorated our PFIB function with make\_posh and we should be done. Let's check. So Python make\_posh. And there you have it. The expected output.

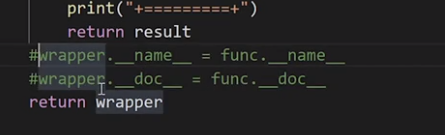
### **Challenges with debugging**

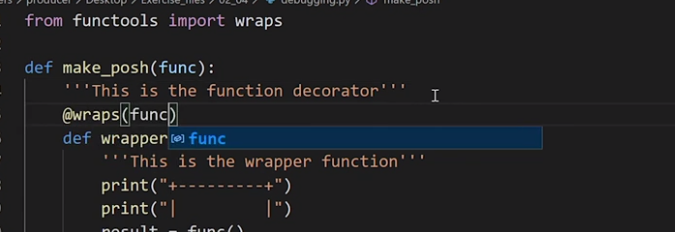
Selecting transcript lines in this section will navigate to timestamp in the video

- We have a problem. By replacing a function with another callable, we've lost important metadata. Let me show you with an example, and then we'll look at a couple of solutions. So head over to the file called debugging.py, and we'll use that as our starting point. You will probably recognize one of the functions from our make posh challenge question. So let's head over to the REPL. So I'm in the same folder as my debugging.py file. And so from debugging import my two functions, so make\_posh and printfib. Now let's print out printfib's function name and doc string. So that's just printfib and dunder name and printfib and dunder doc. So, so far, so good. You've got the name printfib and we've got the docstring for printfib. Now many editors and the help function uses the docstring. So for example if I typed help and printfib, I get the docstring for the printfib function. So, so far, so good. Now let's see what happens when we decorate the printfib function. So I'm going to get out of the REPL, and I'm going to decorate the printfib function with make\_posh. So that's make\_posh, I'm going to save my file. Just going to clear my screen here. So let's head back to the REPL. So, python, and **I just press up arrow to import make\_posh and printfib again. Lets check what we get for printfib's name and docstring. Printfib, name. So this time we're getting the details of the wrapper function. If I go ahead and do a printfib dunder doc, I'm getting the docstring for the wrapper function. Now, this is because the decorator function make\_posh, what is being returned is the wrapper. This is an untended consequence, because clearly we want the name and the docstring for our printfib function. There's a simple fix for this. Let's just assign the function name and the docstring to our wrapper. So if I quit the REPL, and just clear my screen. Let's head back to our function and let's assign the function name to our wrapper. So for wrapper dunder name we want to be using the function name. So that's function, and you do exactly the same thing for doc. I'll go and head** back into my REPL. I'm going to import the files, so make\_posh and printfib. This time if I type printfib name I get printfib. And printfib doc gives me the docstring for printfib. Now, **Python provides a cleaner solution using wraps from the functools module. So let's go ahead and grab wraps from functools. I'm just going to comment out these two lines. So, from** functools import, and lets use the wraps module. And we will use wraps here, with the function as the argument. So what's happening here is that wraps is changing the decorator function's name and docstring to its own. So, if I get out of the REPL, clear my screen. Let me just make sure I've saved my file. So then, just a couple of up arrows. So we grab the make\_posh and printfib. If I do a printfib name, and I'm getting the printfib docstring as expected. And if you're wondering, yes, wraps itself is a decorator.









### **Challenge: HTML styling with decorators**

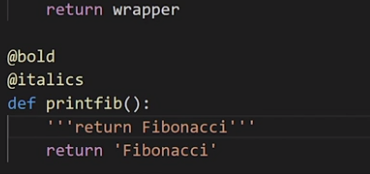
Selecting transcript lines in this section will navigate to timestamp in the video

(bright chime) - [Instructor] Now I'm guessing you've come across HTML before. If you haven't, it's relatively straightforward. If you want some text to be bold then you put the text between bold tags which is a b in between angle brackets. The first one marks the start of bold and the second tag with an additional slash mark besides the b marks the end. It's very similar for italics except this time we use the I open and close tag. Go ahead and grab the challenge\_html Python file. We'll use this as our starting point. Using the print fib function that we've seen before, create two decorators, bold and italic. The output should be Fibonacci in between the bold and italic tags and make sure you get the correct order. So that's Fibonacci is first in italics and then you make the italicized text bold. You've also learned how to preserve the original name and doc string of the original function. So include that in your solution. This should take you between five to 10 minutes.

### **Solution: HTML styling with decorators**

Selecting transcript lines in this section will navigate to timestamp in the video

(bright upbeat music) - [Narrator] Now because we want to ensure that we preserve the original name and doc string, let's use wraps from the functools module. From functools, import wraps. And let's go ahead and create a bold decorator. def bold, which we'll take in as an argument, the func. And so this is a bold decorator. And we want to use the wraps decorator with the function as an argument. So wraps. And now let's go ahead and create our wrapper. Define wrapper. That returns the HTML bold tags. Now since the Printfib function returns a string, we can concatenate the bold tags to the function calls. You can say result equals bold and we can concatenate Printfib return value to bold tags. Return result and you want to return the wrapper. let's go ahead and decorate our print function with bold. And let's call our functions. We want to be able to see the result, Put that within a print. So Printfib, then we should find that our Fibonacci string is decorated with the bold HTML tags. Python, challenge HTML. And we've got Fibonacci in between bold tags. Since this works let's copy the bolt decorator across and make the changes for the italics decorator. def bold. And we can just reuse this code, Just change the names of the function. italics, this is the italics decorator, return HTML, italics tags, and italics are I rather than B. We can go ahead and check that everything's working with italics. And we can see that that's working with italic. Let's now go the whole hog. We want Fibonacci to be enclosed first by italics, and then the result enclosed by bold. We put the italics decorator above the function and the bold above that. And we should be done. Let's check this is working. And there you have the outputs that we want. It's important that you understand, that the order of the decorators is really important. Go ahead and try and swap the decorators above the print function. That means, bold is going to be first applied to Printfib, and then italics. You'll see that the HTML tags are applied in a different order.



## **Question 1 of 5**

When does a decorator have the ability to run additional code to a function that it wraps?

* within the call itself  
  Incorrect
* before and after each call  
  Correct
* after each call
* before each call  
  Incorrect

## **Question 2 of 5**

Why do you use decorators with functions?

* They will automatically convert a list to a string with the Join() module.
* They will change the behavior of the function without permanently modifying it.  
  Correct
* They will change the behavior of the function and permanently modify it.
* They will automatically convert numbers to strings using the str() method.

## **Question 3 of 5**

If you want your Fibonacci string to first be enclosed by italics and then enclosed by bold, which answer correctly shows the order in which decorators should be, above the print fib function?

@bold (Apply Second)

@italics (Apply First)

* printfib():

defprint():

@italics(1)

@italics (2)

* print

@italics

@bold

* def printfib():
* Correct

@bold

@italics

* def printfib():

## **Question 4 of 5**

Using the printfib function, what should your output show if you decorated your Fibonacci string with bold HTML tags?

* <b=Fibonacci=b>
* <b>Fibonacci</b>
* Correct
* <b text='Fibonacci'/b>
* <b FIBONACCI /b>

## **Question 5 of 5**

Using a decorator called @make\_posh, how would you modify the input of pfib() using

@make\_posh

def pfib():

```printout Fibonacci```

return `Fibonacci`

so the outcome looks like the following?

+- - - - - - - - -+

| |

Fibonacci

def make\_posh(func):

def wrapper():

print(`+---------+`)

print(`| |`)

result = func()

return result

* return wrapper
* Correct

def my\_decorator(func):

def wrapper(make\_posh):

print(+---------+)

print(| |)

result = func()

return Fibonacci

* return wrapper

def make\_posh(wrapper):

def wrapper(Fibonacci):

print("+---------+")

print("| |")

result = func()

return result

* return wrapper

def make\_posh(pfib):

def wrapper():

print("+---------+")

print("| |")

result = Fibonacci return result

* return wrapper

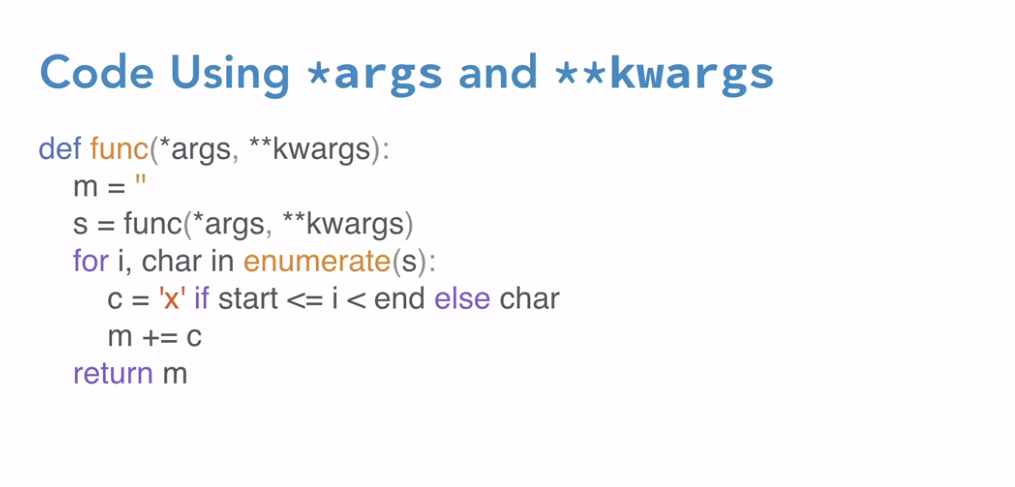
### **Functions with \*args and \*\*kwargs**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] You've probably already come across functions with single star, normally star-args and double star arguments, normally double star kwargs or double star keyword arguments. So why do you need them? If you want to pass multiple arguments or keyword arguments to a function, then single star arguments and double star keyword arguments are your go to. Now let's say I want to print out the numbers of the Fibonacci Sequence and I'd use them as input arguments. So let's head over to the Repl. So now that I'm in the Python Repl, I'm going to call this function pfib and just you don't confuse it with print fib that we used earlier in the course. So let's define pfib. **So define pfib as A, B and C and we want to print out these three arguments. Now what I'm going to provide are the first three numbers of the Fibonacci sequence. So one, one and two and we should be able to see these three arguments printed out. The problem that I have with this function is that it will only accept three arguments. What if I need to enter more or less than three numbers? This is when I can use single args or star args so I can re-create the pfib file and what I want to ensure that there's always at least one number that is sent out as an argument. So I'm going to call that A and I use star args for the rest. So I didn't want to print out that first document A and then I wanted to print out args. So this time round, if I go back to pfib and I enter four arguments, so that's one, one, two and then the next number in sequence is three, you can see that the optional arguments are stored in a tuple. So if I only have a single argument, then nothing would be stored in the tuple and so an empty tuple is returned.** So the mistake I made here was to not enter an argument. So I've got to enter at least one argument here and you can see that an empty tuple is returned. Now, keyword arguments have a very similar option. You might have seen them being used in functions. Again, let's ensure that there's always at least one number that is being used as an argument and let's re-define pfib. So define pfib A and then double star kwargs and I'm going to print out that required argument A and then I want to print out kwargs. So this time round, I'm going to use an abbreviation of their position in the Fibonacci sequence as the keyword argument. So se for second, th for third and so on. So pfib one, the second number is one the third is two, the fourth fo is three and finally, fi for the fifth. T**his time notice that the keyword arguments aren't stored and tuples but rather than dictionaries. Similarly, if I only have a single value as an argument, then I'll have that argument printed out along with an empty dictionary. Now you've probably seen a function that uses both star and the double star together. This is a function that takes any combination of arguments so you can call it with no arguments,** you can call it with positional arguments, you can call it with some mix of positional and keyword arguments, it doesn't matter. So that's redefined pfib. So before I do this, I'm just going to clear my screen just to make this a little bit clearer. So, I'm going to define pfib as args and kwargs and then I'm going to print out args and kwargs. Now I can use a combination of arguments and keyword arguments, so I can provide only positional arguments. So pfib, one, one, two, and three, I can provide only keyword arguments. So pfib, fi for first, se for second, th for third and so on, or I can provide a mix of positional and keyword arguments. So fo for the fourth and the fi for the fifth and you can see that it separates the arguments out into tuples and dictionaries. So if I call it with no arguments, I get an empty tuple and dictionary. You can also create a wrapper function that serves as a pass through. So I define my wrapper function, star args and double star kwargs. So what we'll do in the wrapper function is to unpack the arguments for args. So I want to print out so I want to unpack the arguments for orgs and then finally we'll call pfib. So what I'm going to do is I'm just going to print out a note so that I know where I am. So I'm saying when leaving wrapper and then we call pfib and pass those arguments through. So that star args and double star kwargs. So if you look at the top of your screen, you've got the definition for pfib, which prints out the args and the kwargs and then at the bottom of the screen, we've got the wrapper function that prints out the star args, print the message leaving wrapper and then we pass those arguments through to pfib. So let's go ahead and call this rapper with a mix of single star and double star arguments. So wrapper one, one and the third element is going to be two and as you can see, the first line is where we unpacked the arguments and the wrapper function and lines two and three are when we print out the values in pfib and the arguments are separated out into tuples and dictionaries. So let's go ahead and update our decorator template to include args and kwargs. So we use wraps from functools to preserve the doc string and the name of the function. The decorator accepts a function as an argument and the wrapper function now accepts args and kwargs. These are then passed through to the function and we return the wrapper at the end of the decorator as before. So this is a helpful decorator template that you'll be able to use at work.

\*\*kwargs == take and outputs dictionary

\*args take as tuple outputs tuple

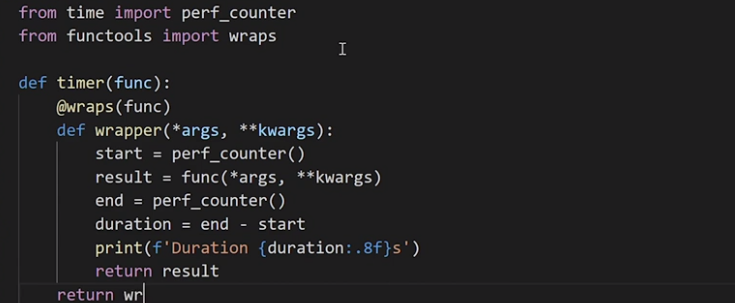


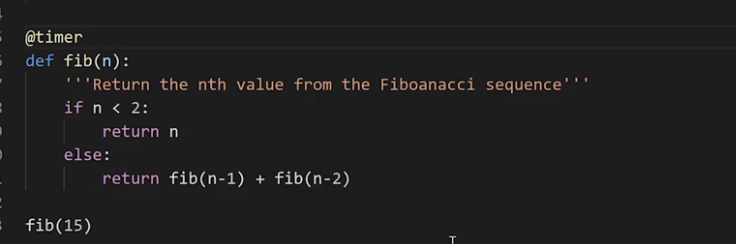


### **Decorators with \*args and \*\*kwargs**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Very quick review of star args and double star kwargs in the previous video. Pfib with star args and double star kwargs. And you want to just print out the values of these arguments, so print args and print kwargs. Now you can also create a wrapper function that serves as pass through. So let's define our wrapper, which takes in these arguments, star args and double star kwargs. And so what we'll do in this wrapper function is to unpack the arguments for args. And then we'll call the pfib function. So let's just print out a note, so that we know that we're in the wrapper function, and let's unpack the arguments for args. So, print star args. And then we want to call pfib, which will print out all of the arguments. So we pass through the arguments from the wrapper function across to pfib. So pfib, star args, and double star kwargs. So we've got both our pfib function defined at the top, and then our wrapper function below that. So let's go ahead and call this wrapper with a mix of single star and double star arguments. So wrapper, and then we'll pass in the Fibonacci numbers, so one, one, and then the third number is two. Then you can see that the first line stays that way in the wrapper function. The second line unpacks the arguments in the wrapper function. Lines three and four are when we print out those values in pfib and the arguments are separated out into tuples and dictionaries. So let's go ahead and update our decorator template to include args and kwargs. So we use the wraps from functools to preserve the doc string and the name of the functions. The decorator accepts a function as an argument. And this time, the wrapper function now accepts args and kwargs. These are then passed through to the function and we return the wrapper at the end of the decorator as before.

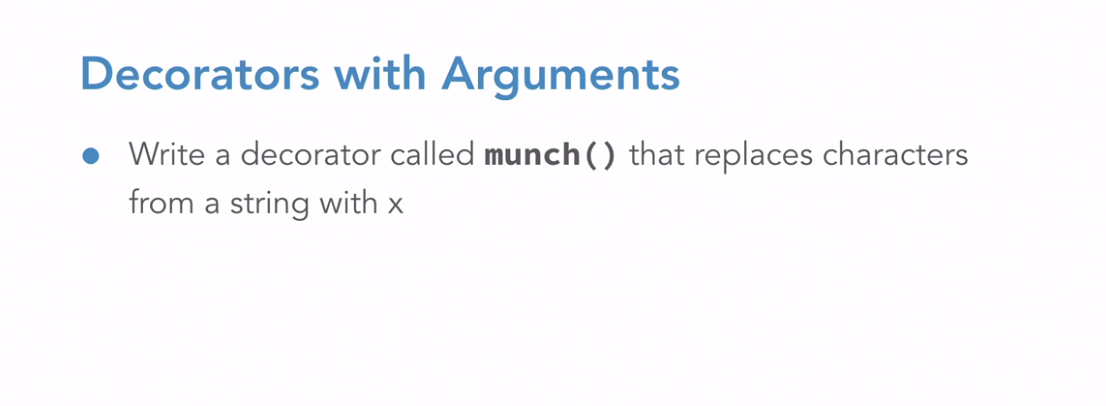


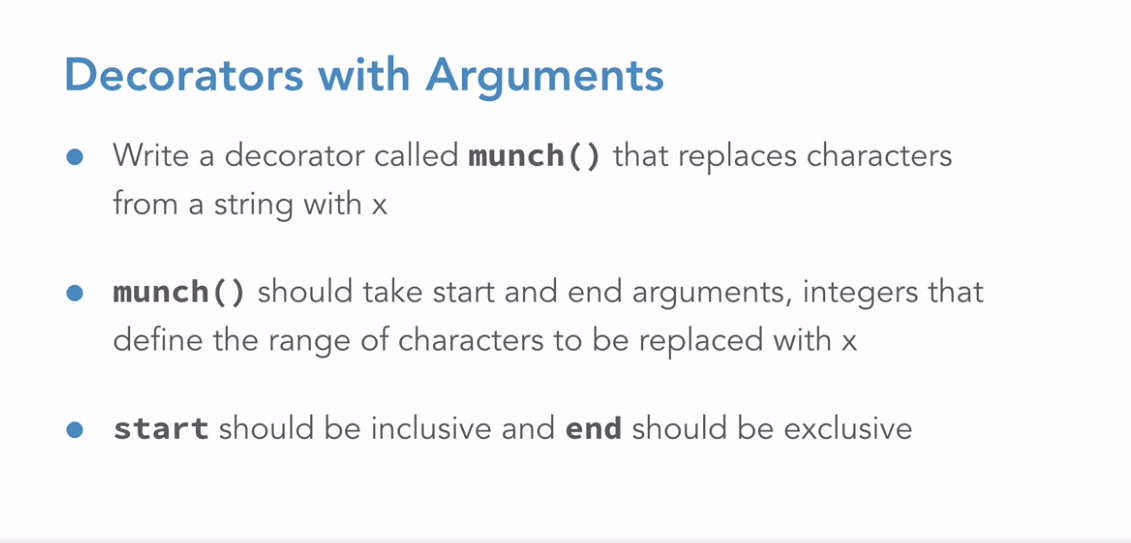


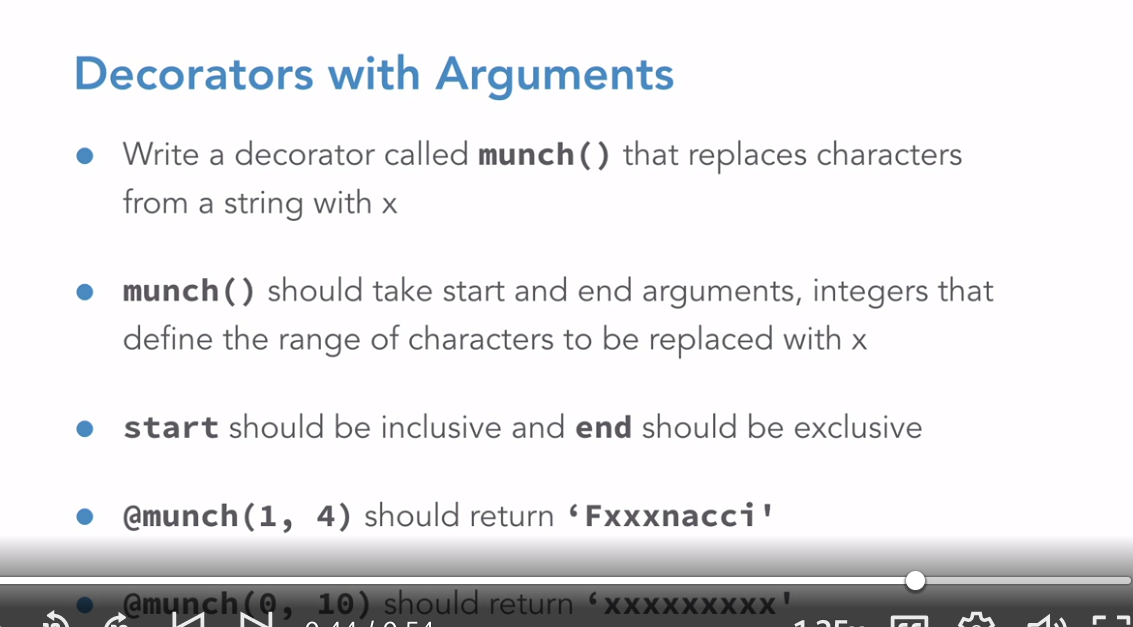
### **Challenge: Decorators with arguments**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Instructor] So for this challenge write a decorator called munch that replaces the characters from the string Fibonacci with X so it munches away at the string. Hence the name. It takes the start and end arguments. Both of these need to be integers that defines the range of characters to be replaced with X. So the start and the end work like the Python in-built function, range. So start is inclusive and end is exclusive. So for example, if we pass arguments one and four, we'll get the following output. And with the arguments zero and 10, we'll just get nine Xs. And that's because we're replacing or munching away at all the nine characters of the string Fibonacci. Use munch.py as your starting point. This challenge should take you about 10 minutes.



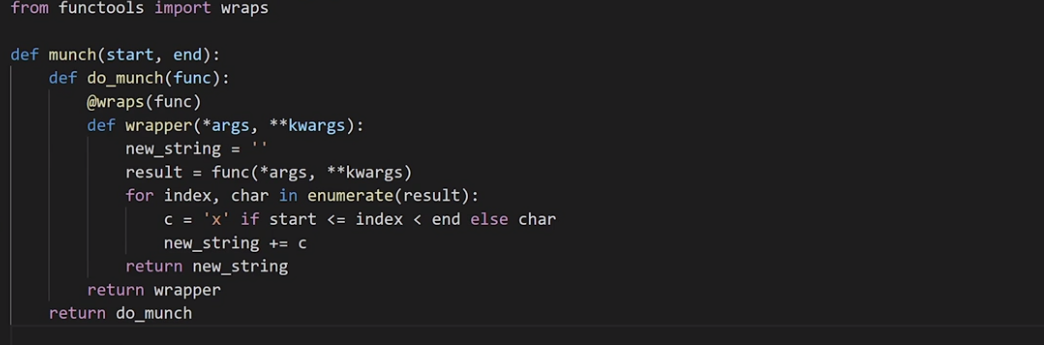




### **Solution: Decorators with arguments**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Instructor] So let's use our decorator template as our starting point. I'm going to head over to our munch.py file and let's call our decorator munch for now, so def munch. We know that it accepts a function as an argument. We then want to use the wraps decorator to ensure that we have the correct doc stream and name for our function, so perhaps that takes in as an argument a function and let's go ahead and create our wrapper. Now although we don't have any arguments in our Fibonacci function, let's just use \*args and \*\*kwargs, if this changes in the future. So define our wrapper. And so you want to do something before our call to the function, you then want to call the function and then potentially do something after this. So before we call our function, we probably want to create a string and we'll be modifying the string and returning this, so let's call this string new string. So, new string and let's make it an empty string. Now I'm using a string, but this could also be a list. So let's call our function. So function \*args and pass in the \*args and \*\*kwargs. And we want to store the value that's been returned in result. So when we call fib, the string Fibonacci is returned. And so result will have the string Fibonacci. So the next thing we want to do is to probably work our way through the Fibonacci string. And one way to do that is to use the enumerate function. Now enumerate will return a tupple. So where the first part is the index, and the second part is the character. So for index and char for character in enumerate, and we want to work our way through the string Fibonacci. So we say the value of the current character and let's call that C is going to be an x if it's between the start and end value. If our index is between the start and end values. So start less than an equal to index, which is less than the end. Otherwise, it's going to be the current character from the result, and so that would be what we have from our core value. And we want to add this character at a time to the new string. So new string plus equals the value of C. Now the question is, what is this start and end value? And where do we get it from? 'Cause we aren't going to get it from fib. So the \*args and the \*\*kwargs are useless to us. This means that there must be a level or a function above our current decorator, where we could capture Start and End values. So let's call this outer function Munch, because this is the name of the decorator above fib and the function Munch must have two arguments. So this is the start and the end. This means what we created previously as a decorator must be an inner function. And let's rename our old function munched to say do\_much. And now all we need to do is to return the new string, returned the wrapper, and do\_much. So return new string, return wrapper and return do\_munch. So let's see how we do with the current arguments of one and four. So I'm just going to save this file and run it. So Python munch.py. And you can see that we get the expected output. Now let's change the arguments from one and four to zero and 10. Now since Fibonacci has nine characters, we should just get back nine x's. So save the file. I'm just going to clear my screen and let's run the function again. And since Fibonacci has nine characters, we should get back the nine x's. And we have exactly that output. And so that's the end of this challenge.



## **Question 1 of 5**

Which time module function in Python can you use to get an approximation for how long in seconds a function takes?

* the time.ns function
* the thread\_time\_ns function
* the clock\_settime function
* the perf\_counter function  
  Correct

## **Question 2 of 5**

How many arguments must the munch() decorator have?

* It must have one argument.
* It must have three arguments.
* It does not matter.
* It must have two arguments.  
  Correct

## **Question 3 of 5**

Using the munch() decorator and the word Fibonacci, what would be returned if the following was called?

@munch (1, 6)

* xibonxcci
* Fxxxxxxci
* Incorrect
* Fxxxxxcci
* Correct
* xibonxcci

## **Question 4 of 5**

After you define your wrapper as displayed below, which code snippet correctly shows the wrapper being called with a mix of single star and double star arguments?

>>> def pfib(\*args, \*\*kwargs):

print(args)

print(kwargs)

>>>def wrapper(\*args, \*\*kwargs):

print("In wrapper:")

print(\*args)

print(\*args, \*\*kwargs)

>>> wrapper(1, 2, th=3)

wrapper:

1 2 3

(1, 1, 2)

* {`th`: 3}

>>> wrapper(1, 1, th=2)

In wrapper:

1 2

(1, 2)

* {`th`: 2}

>>> wrapper(1, 1, th=2)

In wrapper:

1 1

(1, 1)

* {`th`: 2}
* Correct

>>> wrapper(1, 1, 2)

wrapper:

1

(1, 2)

* {`th`: 2}

## **Question 5 of 5**

After defining your pfib as displayed below, which answer correctly shows a mix of positional and keyword arguments?

def pfib(\*args, \*\*kwargs):

print(args)

print (kwargs)

>>> pfib(1, 1, 2, 3)

(1, 1, 2, 3)

* { }

>>>pfib(1, 1, 2, 3)

1

* (1, 2, 3)

>>>pfib(fi=1, se=1, th=2) ( )

* {`fi`: 1, `se`: 1, `th`: 2}

>>> pfib(1, 1, 2, fo=3, fi=5)

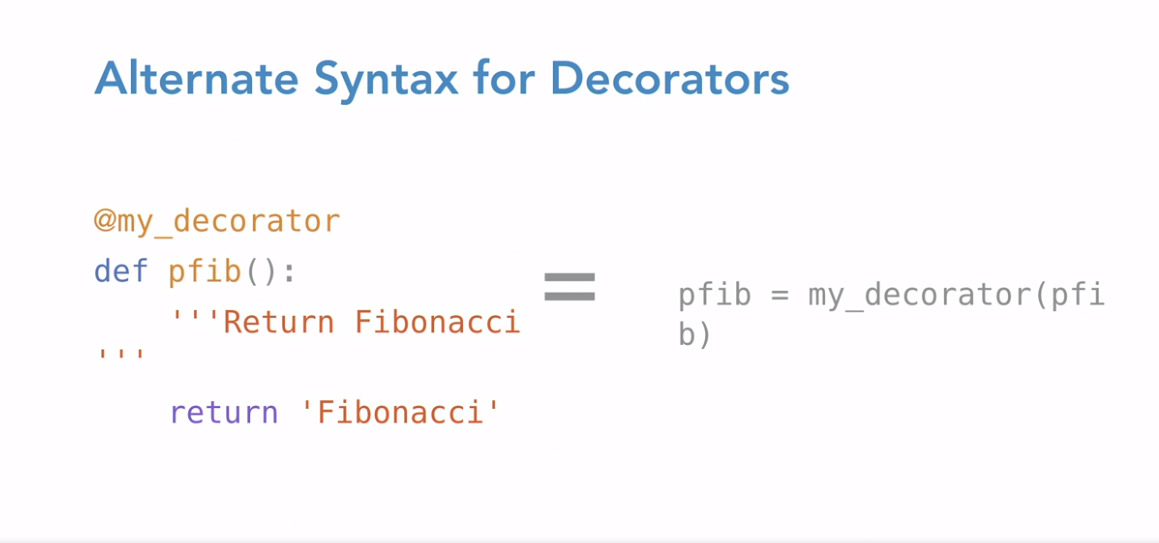
(1, 1, 2)

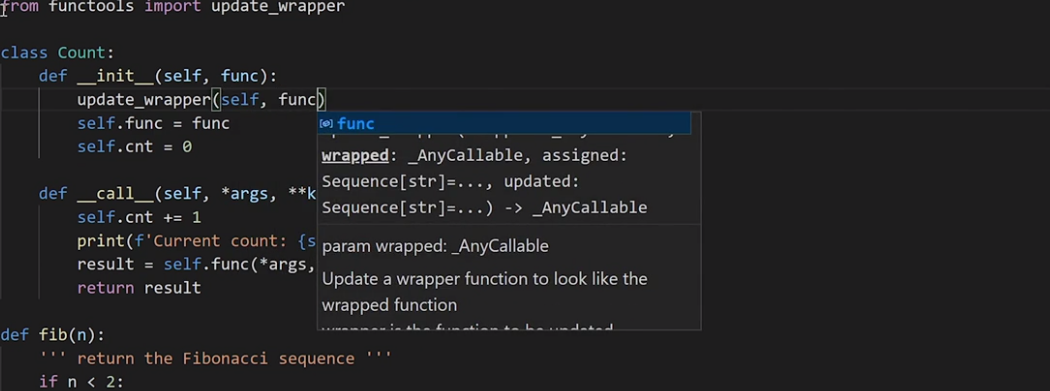
* {`fo`: 3, `fi`: 5}
* Correct

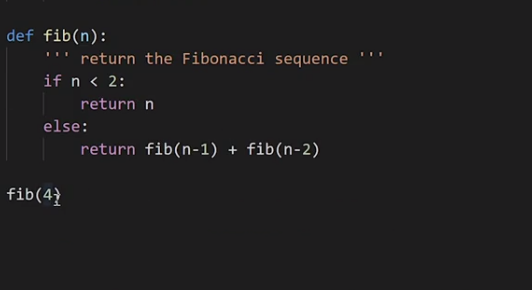
### **Classes and decorators in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] We've used decorators with functions. Let's now look at the other place where you work with decorators and that's classes. The best way to understand decorators with classes is to use an example. So let's create a class that counts the number of times, we call the Fibonacci function when determining the number in the sequence. So that's how many times do we call the Fibonacci function to calculate the fourth number in the sequence for example. Let's use the count.pi module as our starting point. So let's start off with our Fibonacci method. With the decorator syntax that we've used all along, the @my decorator syntax is just a cleaner way of decorating the function Pfib using my decorator and then assigning that to Pfib. So this means that if my decorator is a class, it needs to take the function as an argument in its init method. So let's create our class count with the function and our count. So class count and we have our init method. So that's def done the init, we have self and then we taken as an argument our function. So self.func is the function and we want to set the count to zero. Now this class also needs to be callable so that we can use it instead of a decorated function. **Now for a class to be callable, you implement the special Dunder call method. This means the Dunder call method is run each time you try to call an instance of the count class. And in this way, we can increment the count value. So let's implement the Dunder call. So that's def Dunder call. And because this is behaving like a decorated function, we're going to accept args and kwargs based on the function that we will call. So we want to increment our count, and then we want to print out our current count. So I'm just going to use an f string. So the current count is self.count.** I then want to call the function and store it in results. So self.func and we pass through the arguments and we return the result. **The Dunder call method will be called instead of the decorated function. It basically performs the same task as the wrapper function that we've seen all along. Now if you remember when using functions, we used wraps from the func tools module. Things are a little different with classes. We'll use the update wrapper from the func tools module. So from func tools, import update wrapper. So let's go ahead and update our Dunder init method. So update wrapper and this time we want to provide our function as an argument. So self func,** and the purpose of the update wrapper is to ensure that we have the correct doc string and name of the function. So now let's see how many function calls there are to calculate the fourth element. So we've got a call to calculate default element at the bottom of our file. So let me just save that file. And let's run it. Now there's no point creating a decorator if you're not going to use it. So let's run this code again. But this time, let's decorate our fifth function with count. Count and let's go ahead and run that code again. And you can see that there are nine calls to the Fibonacci function to calculate the fourth number in the sequence. Let's take a look at this as a graphical representation of each of these function calls. So in order to determine the fourth number of the Fibonacci sequence, we need to determine the values for the third and second, and similarly for the second, we need to determine what the values are for the first and so on. So you can see that in total, there are nine calls to the functions to determine the fourth element. Now it gets worse. **There are 15 calls to determine the fifth element. And let's see how many calls there are to determine the 10th number in the sequence. So I'm going to replace the number four with ten. And let's rerun this function. And you can see that there are a whopping 177 calls in order to determine the 10th sequence of the Fibonacci function. This is exactly why when timing the functions earlier, the larger the number of the Fibonacci sequence, the longer it took. So you can see that we're repeating many of the calculations that we're making. So in the next video, we look to see how we can optimize this.**



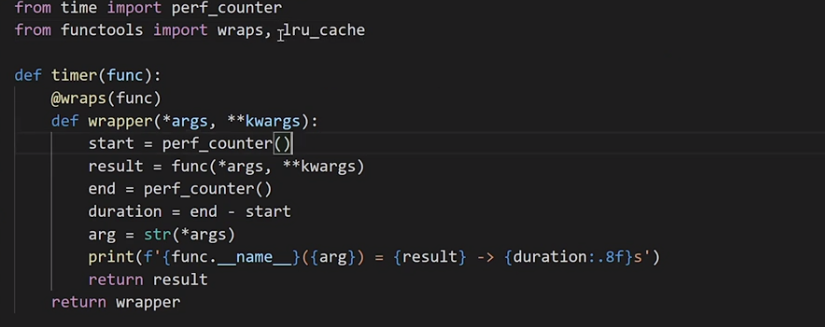


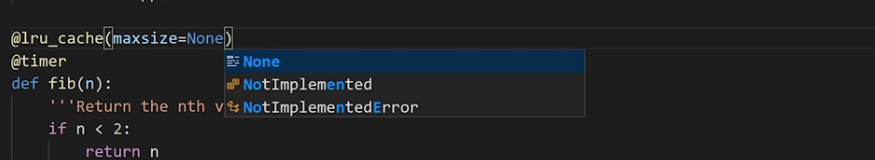


### **Use decorators as a cache**

Selecting transcript lines in this section will navigate to timestamp in the video

- What we found when trying to calculate the values of the Fibonacci sequence using a recursive function, is that several calculations are recomputed. So how could we optimize this process? **Well, one way to go about it is to store the results of each computation when they're completed so that you can use them again. This technique is known as memoization. Think of the word memorization without the letter R. Python also has a built in decorator for memorizing functions. This is LRU cache from functools. So go ahead and grab the cache.py file, and let's use LRU cache. S**o let's go ahead and decorate our fib function. So at LRU cache, and let's set the MAX SIZE argument to none. This indicates that there's no limit to the size of the cache. Now let's go ahead and try the Fibonacci sequence. And let's get an idea of how fast this is. So we're running this for fib 18. So this is two calculated the 18th number in the Fibonacci sequence. So Python, cache. And you can see that this took 0.00624 seconds. Now let's see how fast this code is without using the cache. So let's comment out the LRU cache decorator. I'm going to save that file. And let's rerun this. And you can see that this is 3.76 seconds. So if I just head over to the Python ripple, 3.76 divided by 0.00624, which was the time when using the cache function. So using a cache makes a significant difference in the performance of our Fibonacci functio**n. Now, you might be interested to know that the order of the decorators does matter. So if you have the timer decorator close to the function, like this, then we are just timing the runtime of our function fib.** Now if we want to swap the decorators around, so that LRU cache is closer to the function call, so I move LRU cache closer to the fit function, now, we're timing the runtime of our function fib as before, plus, the time it takes to do the cache lookup. Now, there isn't a huge difference in between these two. They're way quicker than not using a cache, but it's important that we understand what we're doing. So here, we only want to time the runtime of our function. So we want the timer decorator, to be closer to our fib function definition.





Thats it

## **Question 1 of 2**

If you want to time the runtime of your function, which code snippet correctly shows this?

@lru\_cache(maxsize=None)

@timer

'''Return the nth value from the Fibonacci sequence'''

if n< 2:

return n

else:

return fib(n-1) + fib(n-2)

* fib(18)
* Correct

@timer

@cache\_size

("Return the nth value from the Fibonacci sequence")

if n< 2:

return n

else:

return fib(n-1) + fib(n-2)

* fib(18)

@lru\_cache(maxsize=None)

@data\_handler

'Return the nth value from the Fibonacci sequence'

if n< 2:

return n

else:

return fib(n-1) + fib(n-2)

* fib(18)

@time\_wrapper(18)

@timer

"Return the nth value from the Fibonacci sequence"

if n< 2:

return n

else:

return fib(n-1) + fib(n-2)

* fib(18)

## **Question 2 of 2**

For a class to be callable, which special call method must be implemented?

* the class method
* the static method
* the dunder call method  
  Correct
* the instance method

### **Install Flask**

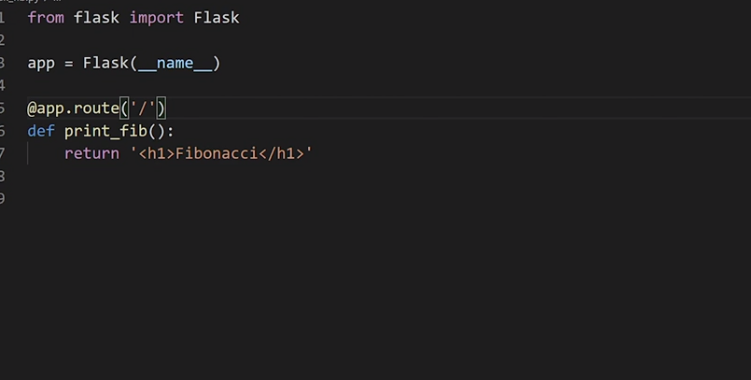
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's look at a couple of examples of decorators in real production code. Flask is a web micro framework that uses decorators. So let's go ahead and install Flask. So I'm going to head over to my terminal, and I'm going to create a directory called flask and move into that directory. So now that you have the application directory created, it's time to install flask. Now a good way to do this is to use a virtual environment, which is a copy of the Python interpreter into which you can install packages privately, and this doesn't affect the global Python interpreter installed in your system. So virtual environments are great because they prevent version conflicts in your systems Python interpreter. So let's go ahead and set up our virtual environment. So in Windows, this is just python with a m flag and virtual environment and I'm just going to call my virtual environment venv. So if we take a look at our directory, you can see that the virtual environment has been created there. So the next thing we want to do is to activate our virtual environment. And so we do this in Windows by typing V-E-N-V, we head over into the scripts directory. And next we call activate. So you can see that our prompt has changed. And we have a V-E-N-V at the start of our prompt. So that means we're in the V-E-N-V virtual environment. So we can go ahead and install Flask here. So pip install flask. And just to confirm that everything's working as it should, let's head over to our Python interpreter so and we want to import flask. So we can confirm that everything has been installed successfully. I'm going to be using a Docker container to show you what you need to do in Unix. So I'm going to be using Ubuntu. So I've created my flask directory, and I've moved into the directory. Now ignore the fact that I'm root. Okay, this isn't a production system. We're just using this to demonstrate some key features on a Unix platform. So go ahead and install the Python virtual environment. So Python three with the m flag, and this time we type source, then bin, and then activate. So you can see again that we're in the virtual environment because of the change in the prompt. So let's go ahead and install Flask. So pip install flask. And then just like we did in Windows, I head into the Python interpreter or the REPL say import flask and we can confirm that everything has been installed successfully.

### **Run Flask**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now in the flask directory, go ahead and create a file called flask underscore fib. I'm going to save this as... And now with just five lines of code, we'll be able to set up a web service. So let's import flask from flask. So from flask... import flask. Now the next thing we want to do is to pass the main module or package of the application, to the flask class constructor. So this is usually dunder name. So we say, app equals flask, and we pass in dunder name. Let's create a function print underscore fib, that will print out the string Fibonacci on our web browser, but as an HTML heading. So def print fib. And we want to return the string Fibonacci, but we'll use a HTML heading. Now when you send a request to a web server from your web browser, this request is forwarded to the flask application instance. So the flask instance needs to know what code it needs to run, for each of the URL requested. And so there needs to be a mapping, between the URL and the associated Python function. This is called a root or a route, depending on where you live in the world. And we define the root in a flask application, through the app dot route decorator. So in this case, the decorator tells flask, which URL will trigger the print fib function. So we add a decorator above our function. So app dot route. So this means when you go to the main page on the server, you will see Fibonacci in large letters, as this is returned by the print fib function. So the flask application, needs to know what code it needs to run, for each URL requested. So in Windows, go ahead and set the flask underscore app environment variable. So I'm in my virtual environment. So set flask underscore app, and I provide the name of my Python file. So that's flask underscore fib dot py. Let me just do a quick check on the directory to make sure that that file is in this directory. So I can see that flask underscore fib is in my directory. And then the next thing I need to do, is to run the flask application. So flask run. Now you can see at the last line, it says running on HTTP, 127.0.0.1, and on port 5000. So if we copy this URL location, and head over to our browser, we can see Fibonacci as our heading in large letters. So in Unix, set the environment variable. So we are in our Python virtual environment, in the flask directory. I've got a file called flask underscore fib in this directory. So you want to export, flask app. And we provide the name of our Python file, so that's flask underscore fib. And then we can go ahead, and run the flask application. Now in my Docker image, I need to also set to other environment variables. So I'm going to go ahead and export... LC ALL, and export LANG. And if I go ahead and run flask again, so let's flask run. You can see that we have exactly the same prompt as we did in Windows. And you can go ahead and go to your browser, and you'll see the same output that we saw in Windows.



### **The route function in Flask**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] So let's look at routes in a bit more detail because that's where the decorator function is, so if we want to add another page besides the one that we just created, we could create a page and the associated Python function. So let's create a Python function called, list\_fib, which just has the first couple of Fibonacci numbers and just to make things more interesting let's list these numbers as HTML bold. So I create my Python function called, list\_fib and I want to return these in bold text. One, one, two, three, five, eight, 13, 21 and so on. And now let's add a route to this function and let's just call it fib. So this means that if we enter fib after the root directory in our browser we expect to get a list of the first few Fibonacci numbers, So I add my app.route decorator, so and I provide as an argument /fib. So I'm going to save my file, I'm going to go head and cancel my web server, so that's control + C. So we're going to bring the web server again by typing Flask run. So I'm just going to add fib at the end of the URL and we can see the workings of our new function. So let's take a look at the source code for the Flask application. Let's do a search for on getup and Flask. I'm going to select source, I'm going to select the app.py file and if you scroll down, you can see the definition for the Flask class. So let's now look for the definition of the route function. So lets do a search for def route because this is the function that we've been using all along. Now the doc strings says, a decorator is used to register a view function for a given URL rule. Now within route, we've got a function called decorator because that's exactly what it does. The add underscore, URL underscore rule on line one, two, three, six is the function that does all the hard work and adds the URL to the map containing all of the rules. So the rule that the route function plays is to provide a convenient decorative for uses of the Flask function. So Flask is a great example of decorator in the real world.

## **Question 1 of 3**

What is the role of the route function?

* to provide a convenient decorator for the uses of the Flask function  
  Correct
* to provide a web framework for the uses of the route function
* to return an updated dictionary of the current local symbol table using the locals function
* to return the largest item in an iterable using the max function

## **Question 2 of 3**

When you send a request to a web server from your web browser, where is this request forwarded?

* to the main page on the server  
  Incorrect
* to the flask application instance  
  Correct
* to the fib function returning the flask instance
* to the time module instance

## **Question 3 of 3**

After setting up and activating your virtual environment called venv in windows using python -m venv venv, which command do you enter to install Flask?

* install flask venv
* pip install flask
* Correct
* flask install flask
* pip flask install

### **Experiment with decorators in Python**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now that you've got a pretty good overview of decorators, I'm hoping that you want to experiment with them. Here are a couple of built-in functions in the standard Python library that use decorators. So we've got the **property decorator**, we've got the **static method decorator,** and we've got the **class method decorator.** Go ahead and try and see if you can make sense of how you use them from the documentation. We've also taken a look at the flask module. Take a look at the code for the flask module. In particular, app.py. How have decorators been used here? Well, I hope you found this course helpful. Thanks for watching and I'd love to hear back from you and to connect via LinkedIn. Thank you and Soli Deo Gloria.