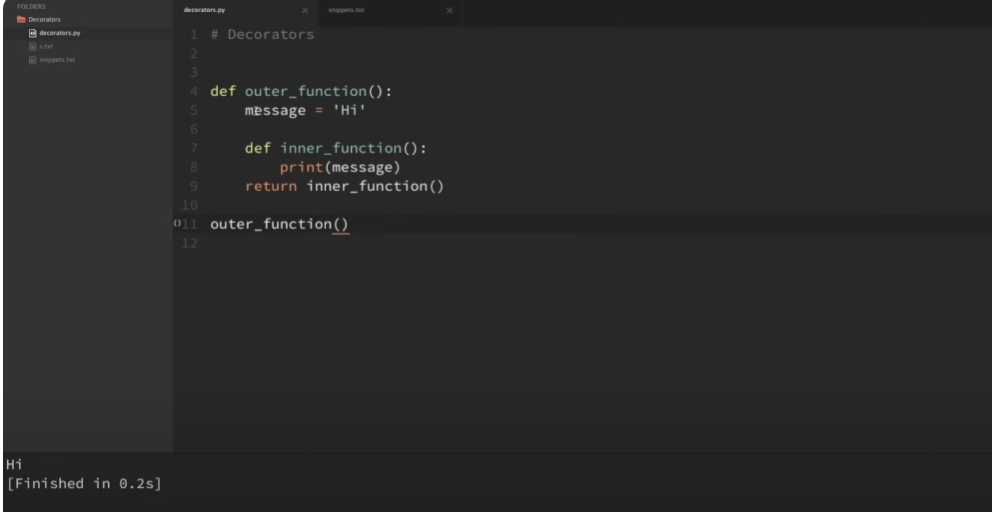
Hey everybody, how's it going? In this video, we're going to be learning about Python decorators. We're going to be learning how decorators work and also when you would want to use them. So decorators are a slightly more advanced topic, but we're going to walk through a few different examples so that we can understand exactly what's going on. Now, if you haven't watched my videos on closures or first class functions, then you should probably give those videos a watch because they will explain a lot of the underlying concepts that I'm not going to spend a lot of time on here. and I'll put a link to those videos in the comment section below. But if you haven't seen those videos, then let me just give a really quick summary.

So first class functions. First class functions allow us to treat functions like any other object. So for example, we can pass functions as arguments to another function, we can return functions, and we can assign functions to variables. Now closures allow us to take advantage of first class functions and return an inner function that remembers and has access to variables local to the scope in which they were created.

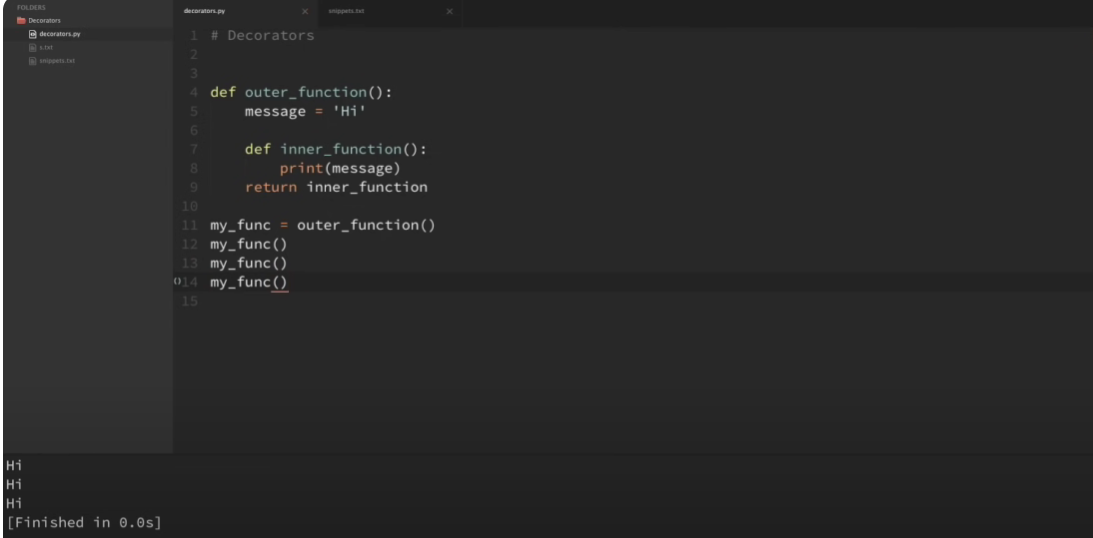
Now, if both of those definitions sound a little confusing, then let's go ahead and take a look at a quick example here. And again, I'm not gonna go as in-depth as I do in those videos, but let's just do a quick recap. So we have an outer function here that doesn't take any parameters. And within our outer function, we have a local variable called message.

Then we are creating an inner function within the outer function. Now the message variable wasn't created within the inner function, but the inner function does have access to it. And this is what we call a free variable. Now all our inner function does is print this message. Then we are executing our inner function and returning the result. So now if I execute this outer function here and run this code, what it does is it comes in here to this outer function sets this message variable, creates this inner function, then it executes this inner function and returns the result. So we can see when the inner function is executed that it printed out hi, and that was the value that we had assigned to our message variable.

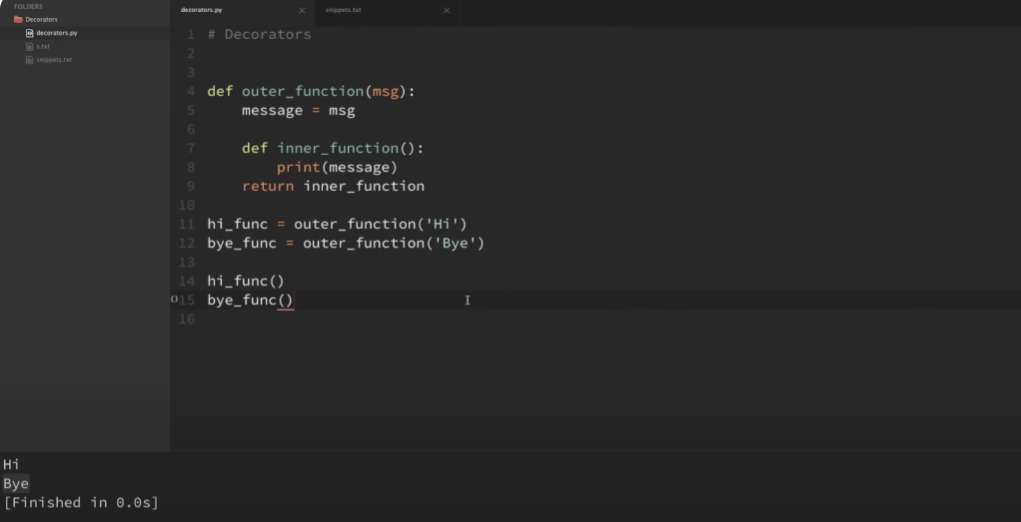


Now instead of executing the inner function and return it, let's instead return the function without executing it, and we can do that by taking away the parentheses here. So now when we execute this outer function, what it's going to do is it's going to return this inner function waiting to be executed. So in order to illustrate this, I'm going to actually set this equal to a variable here called my func. So now my func is going to be equal to our inner function waiting to be executed. So now I can execute this just like any other function. So I'll execute it a couple of times in a row here. And let me go ahead and run that. So you can see that it prints out our message variable multiple times.

So that's what a closure is because it remembers our message variable even after the outer function has finished executing.

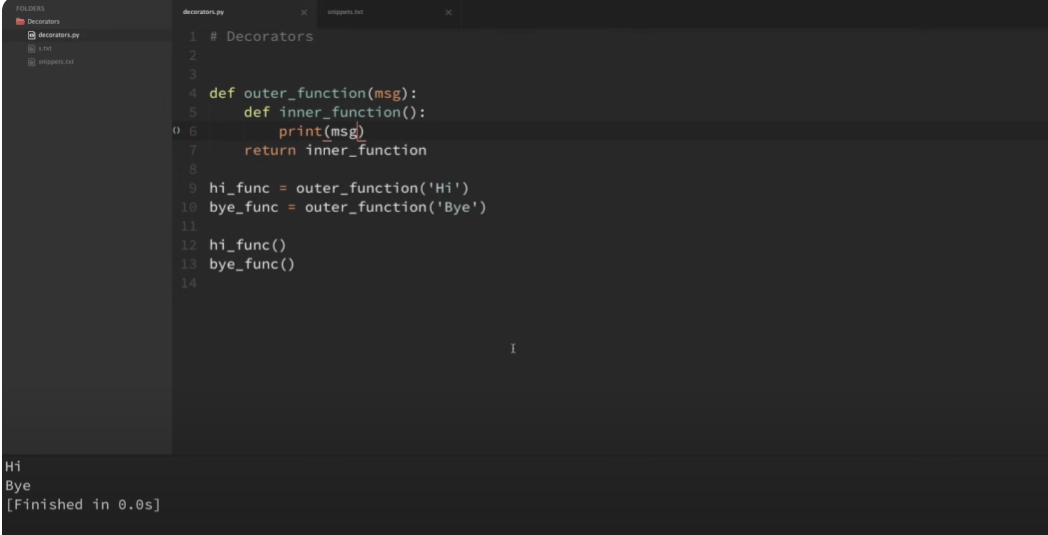


So we can see this in more detail if we pass in some arguments to our outer function. So I'm going to have our outer function take in a variable here called msg and instead of hard coding in high here to this message variable, I'm just going to set it equal to that argument and save that. So now that our outer function takes in some arguments, now I can create two different variables here. I'm gonna call one high function and I'll call another one here by and then I will pass in two messages here. I'll pass in high to the high function and by to the by function and I can run that. So now again, each of these variables is equal to our inner function ready to be executed. and when we execute it, it's going to print out the specific message that we have passed in. So let me execute both of those and you can see that the high function prints out high and the by function prints out by. So you can see that each of those variables has a unique message variable based on what we passed in to its outer function.



So now if we look at this message variable here, really all it's doing is getting the value of this MSG argument. So really what we could do is just cut out the middleman and remove that. And instead of printing out, setting message and printing that out, we can just come down here and directly print out that message there.

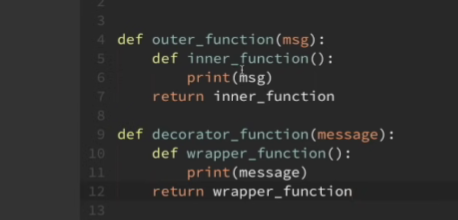
And if I run that, you can see that that still works also. Okay, so that is a quick recap of closures. And I go into closures more in depth in the actual closures video. But now let's move on to decorators.



So, decorators are very similar to what we've already done here. Okay, so what is a decorator? A decorator is just a function that takes another function as an argument, adds some kind of functionality, and then returns another function. All of this without altering the source code of the original function that you passed in.

So, to show an example of this, let me grab some code here from my snippets that's almost identical to the example that we just looked at here. Now actually this is identical to the example that we just looked at,

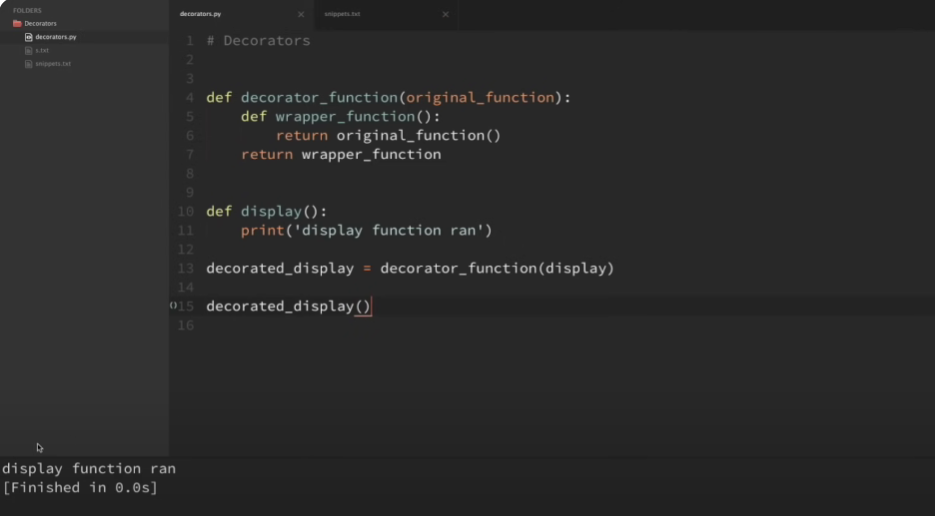
except we've changed the naming of our outer function to decorator function, and we've changed the name of our inner function to wrapper function. So now I'm gonna go ahead and delete this previous example that we've already looked at.



Okay, now we've already seen what this does. It returns this wrapper function that is waiting to be executed, and when it's executed, it just prints out our message. Okay, now what if instead of printing a message that we passed in, we instead execute a function that we passed in. And that's what a decorator does. So let's change our code so that it does this. So instead of a message, we're going to accept a function as an argument. And I'm just going to call this original function.

And within our wrapper function, instead of just printing out our message, I'm going to execute this original function and then return that. Okay, so this is a decorator. It's about as simple of a decorator as you can get. So now, let me grab some more code from my snippets here so that we can decorate a function and see exactly what this looks like. So let me grab this here. I'm also gonna cut this out. And I'm just gonna replace all of this with this new code. Okay, so I have an extremely simple function here called display. And when we execute this display function, all it does is print out that the display function ran. Now I have a variable here called decorated display, which passes in our display function to our decorator function.

And just as a reminder, our decorator function here returns this wrapper that is waiting to be executed. So now this decorated display variable is actually equal to our wrapper function that is waiting to be executed. And when it's executed, it just executes the original function that we passed in and returns it. So now I can run this decorated display variable just like any other function. So if I run that, then you can see that it says that the display function ran.

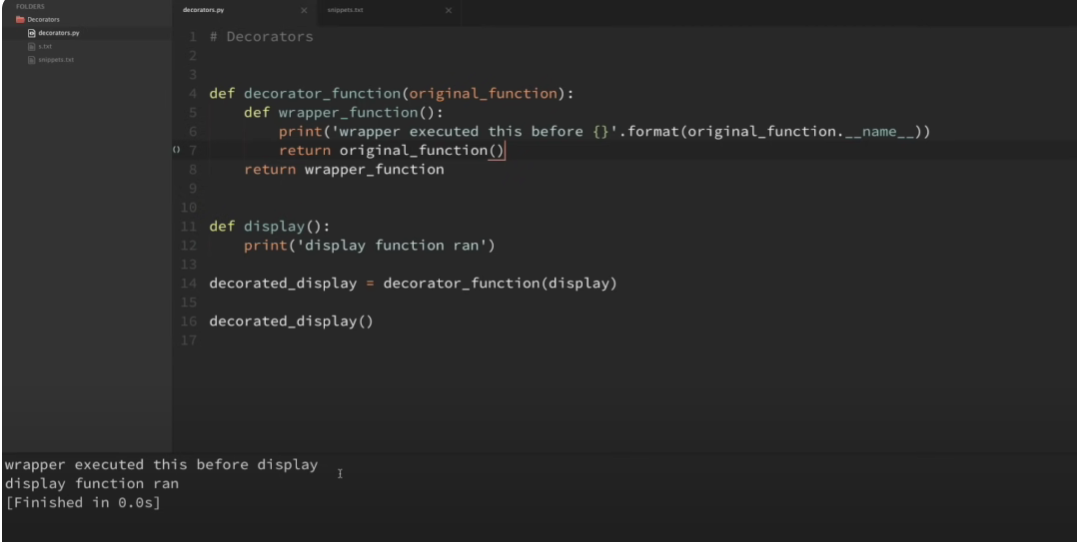


So just one more quick walkthrough here. What we did is that we set this decorated display variable equal to our decorator function and we passed in our display function here. and when we passed in our display function, it's equal to our original function here within our decorator, and then we create this wrapper function, and we return that wrapper function waiting to be executed, and when it is executed, it executes the original function, which in this case is display, and just returns that.

So when we run this variable here, it's actually executing our wrapper function, which then executes our display function and prints out this display function ran. So I know with these decorators, sometimes it can be hard to keep track of the wrappers and the original functions and things like that, but hopefully as the video goes on, we'll get more and more comfortable with this.

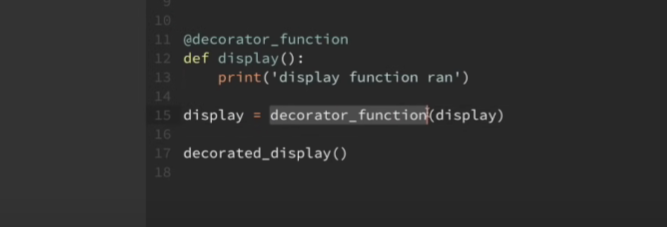
Okay, so now that we have a basic decorator example here, why would we want to do something like this? So decorating our functions allows us to easily add functionality to our existing functions by adding that functionality inside of our wrapper. So for example here, without modifying our original display function in any way, I can come inside of our wrapper and add any kind of code that I want.

So for example here, I'm just going to write a print statement here that says that our wrapper executed this before, and then I will pass in the name of our original function. So let me grab the original function here, and I'm just going to pass in the name. And I'm gonna need a placeholder here for that function name. Okay, so now if I execute this code, we can see that it executed our wrapper message before it executed our original display function and printed out that message.



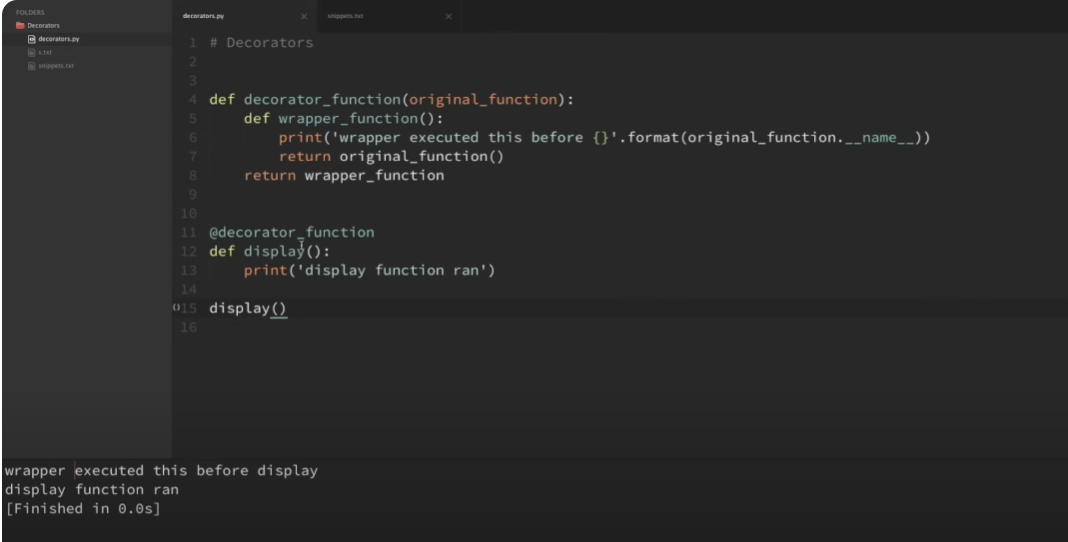
So, so far I've been using this syntax in order to create our decorators. Most of you may have come to this video expecting to see something that looks something more like this now This is how we usually see decorators in Python.

But this is the same thing as setting our original function equal to the function wrapped within our decorator. So having this syntax here would be the same thing as saying that I want my display function equal to this decorator function with a our original function passed in. So now anytime if I use this syntax here, anytime I run this function, then it will have that new functionality added on.



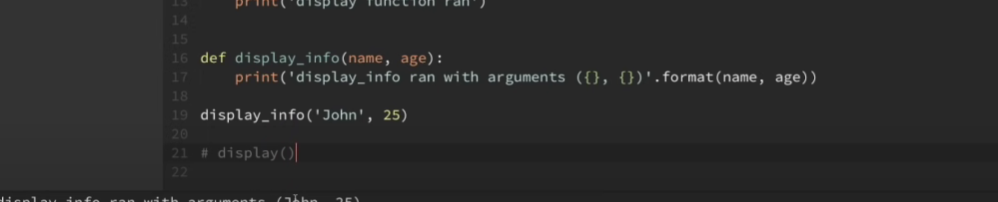
So if I just run this display function now, then you can see that it still has that wrapper code added to our original function. So even though those are functionally the same, I think that this syntax here is a little bit easier to read. especially whenever we chain multiple decorators together and we'll look a little bit at that later on.

So again, just one more time, if I put this decorator symbol above the function like this, then it is exactly the same as saying that I want my display function now equal to this decorator function with this display function passed in like this. So this line and this line are exactly the same thing. So now for the rest of the video, instead of using this syntax here, I'm just gonna use it the way that we'll normally see it within Python and just use this syntax here. So now I can just run our original functions here with the decorator applied and it applies the additional functionality that we were hoping for.



Okay, so using our decorator here, this wouldn't work if our original function took in any arguments.

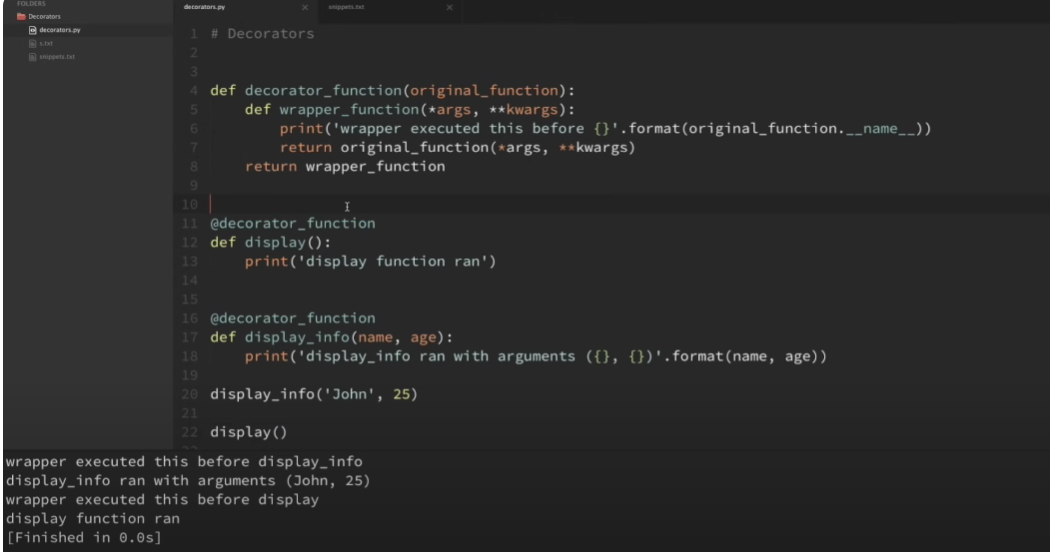
So for example, let me grab another function here from my snippets called display info. So what this display info function does is it just takes in a name and an age and it simply prints out that information. So if I comment out this display function down here and execute this, then you can see that it just prints out that display info ran with the arguments John and 25.



Okay, so now what if I wanted to decorate both this display function and this display info function with the same decorator? So if I take this decorator here and try to apply it to this display info function, if I run this now, then you can see that we get an error. And the error down here says that the wrapper function takes zero positional arguments, but that two were given.

So what we need is we need to be able to pass any number of positional or keyword arguments to our wrapper and have it execute our original function with those arguments. Now the way that we can do this is by adding star args and star star quargs to our wrapper function. So if I come up here to our wrapper function and type in star args and star star quargs, and then I also want to pass these into our original function. Now you can actually call args and quargs anything that you want, but by convention in Python, these are usually the names that people go with. And what these do is they allow us to accept any arbitrary number of positional or keyword arguments for our functions.

So now if I save this and I run it, you can see that it works with both of our functions now.

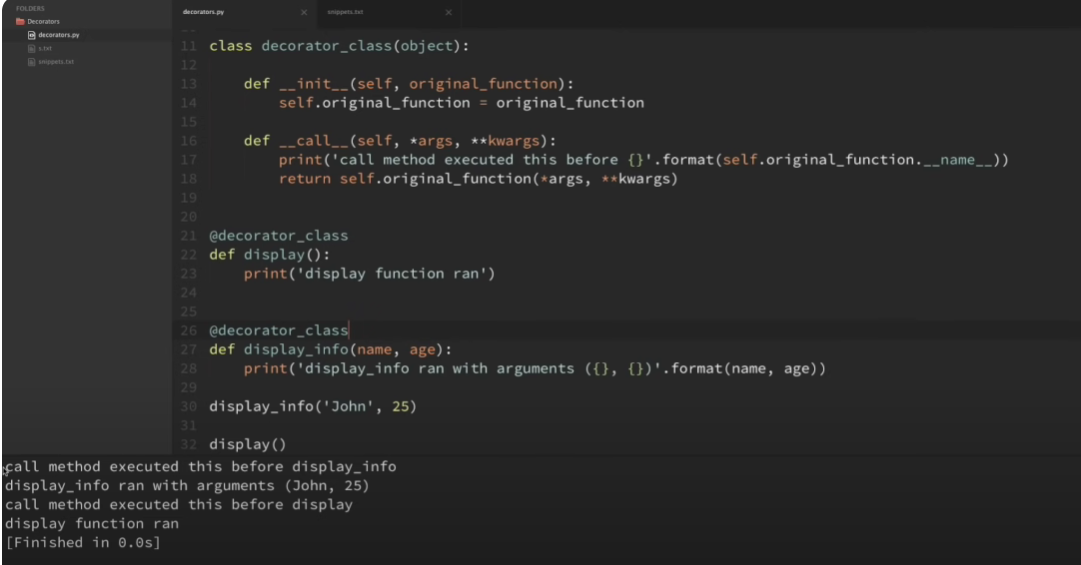


And actually let me go ahead and uncomment this just to make sure that it still works with both. And you can see that it does work with both of our functions. Okay, so now that we have these function decorators working with multiple examples, let me also show you another syntax that you might see from time to time. So some people like to use classes as decorators instead of using functions as decorators. So let's go ahead and take a look at an example of that. So let's just take this example from above here, this decorator function. and let's turn this into a class decorator that has the exact same functionality.

So I'm just gonna create a new class here, and instead of decorator function, I'm gonna call this decorator class. And I'm just gonna do an object there. Okay, now when we passed in our original function to our decorator function, we did that using an argument. So how are we gonna pass that original function into our decorator class? And the way that we're gonna do that is with an init method. So let's do an init method here, and we're gonna pass in self for the instance, and then we'll also just pass in the original function here. And then within the init method, I'm gonna do a self dot original function equals original function. Now what that's gonna do is it's gonna tie our function with the instance of this class.

Okay, and lastly, how do we mimic the functionality here with the wrapper adding some functionality to our original function? And the way that we're gonna do that within our class is we're gonna implement this call method, and the call method is another double underscore method, so it's double underscore call, and we're gonna pass in self, and we're also gonna pass in these args and quarks, because this is going to behave just like our wrapper function behaved. Okay, so now I'm just going to grab this code from the wrapper function up here and paste this in. And instead of saying that the wrapper executed this before, I'm just gonna say the call method executed this. Now we're using an instance now, so everywhere that I see these original functions, I have to do a self.original function and save that.

So now if I use this decorator class instead of my decorator function to decorate these functions down here, then hopefully whenever I run this, we'll get the same result, except now we're using the class to decorate. So you can see here it says that the call method executed this before display info when we ran the display info function here, and the call method executed this print statement before the display when we ran the display function there. So we got the exact same result, but only using the class as the decorator instead of the function. So I hope that's not too confusing, but I wanted to show both ways that this is implemented because some people like to use classes instead of functions for their decorators.



So, I wanted you to be aware that that was an option. But for the rest of the tutorial, I'm going to continue using the function decorators because Those are the ones that I see more often, and they're also the ones that I think that you'll run into more often. So, I'm going to go ahead and delete this class decorator here, but I just wanted you to know that that was a possibility, and you should look into them because they do have some extra functionality, and some people find them easier to work with than the functions.

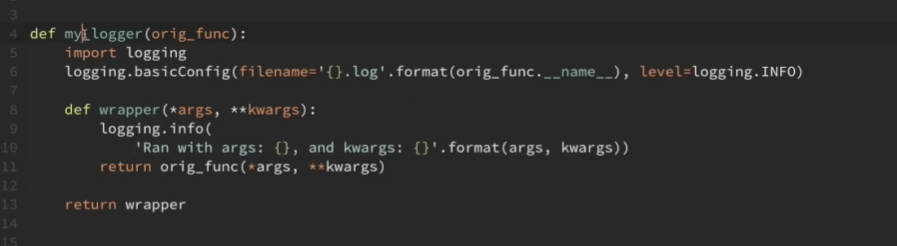
Okay, so let's go ahead and rerun this, and now we have our decorator functions working with our original functions here again. Okay, so now we have a basic idea for how decorators work and how we can use them to add some functionality to our functions.

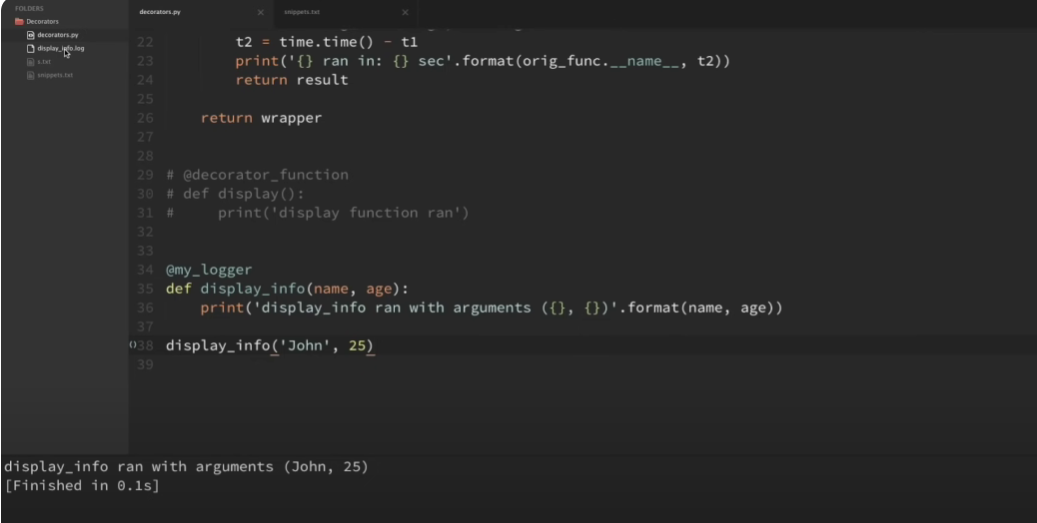
But where a lot of people get tripped up is when they try to think of some practical examples for decorators. So let's go ahead and look at a couple of practical examples and maybe they'll give you some ideas for how you can use decorators within your projects. So let me go to my snippets code here and grab some of these practical examples here. So I'm just going to go ahead and grab these and paste them in over here. And actually just to clear up some room, I'm actually going to go ahead and just replace our old example here with the new ones that I'm adding in. Okay, so probably one of the more common use cases for decorators in Python is logging. So let's say that I want to keep track of how many times a specific function is run and what arguments were passed to that function.

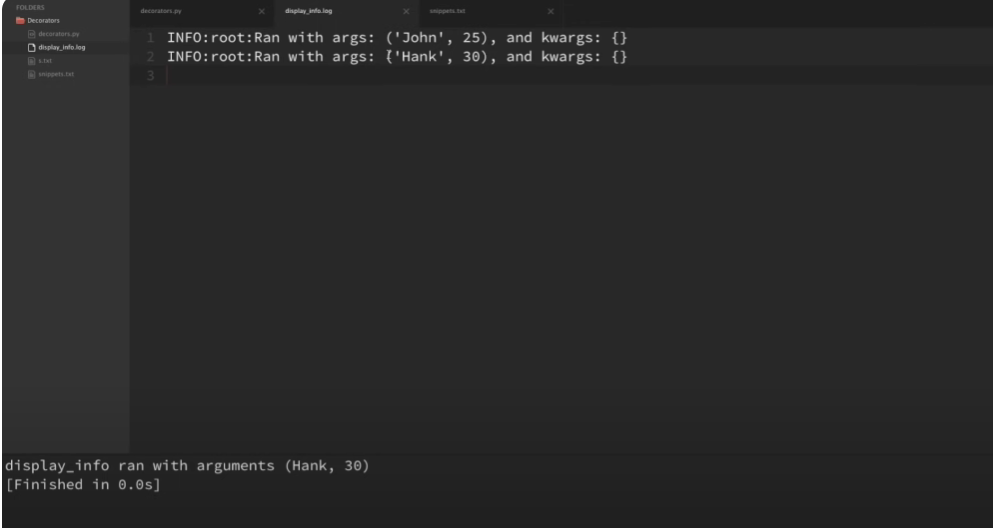
So let's kind of walk through this my logger decorator here and see exactly what's going on. So you can see that we are passing in our original function and then we are importing the logging module and setting up a log file that matches the name of our original function. wrapper function that takes in the arguments and keyword arguments. And within this wrapper function, it runs this logging.info and logs that we ran the function and also logs out the arguments. And then before leaving the wrapper function, we run our original function with the arguments and keywords and then we return that result. And then lastly, we are returning our wrapper function which allows us to run all of this with the added functionality. Okay, so let's actually look at this being run with an example.

So I'm going to go ahead and copy this decorator here, and I'm going to come down here to our display info function. And for now, I'm just going to copy that out and save that. And now I'm going to run this display info function that has been decorated with the logger. So if I run this, you can see that it created this display info log up here. So now if I open up this log you can see that it says that it ran with these arguments and no keyword arguments. So I can go in here and change these arguments here and rerun that and now if I open back up the log again, then you can see that it passed in the new arguments.

So now you can reuse this decorator anytime you want to add that logging functionality to any new function. So you can imagine how repetitive and error-prone it would be if you wanted to add that functionality to multiple functions and tried to manually add in that logging code within each individual function. The decorator allows us to maintain our added functionality in one location and easily apply it anywhere that we want within our code base.



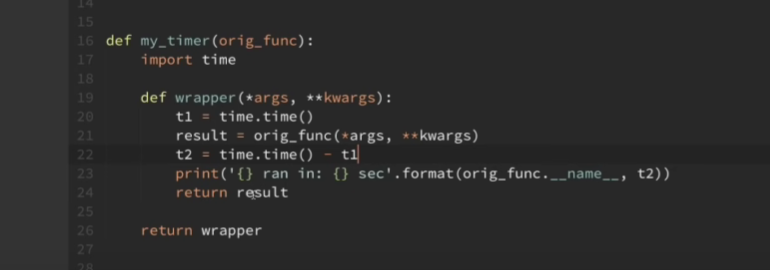


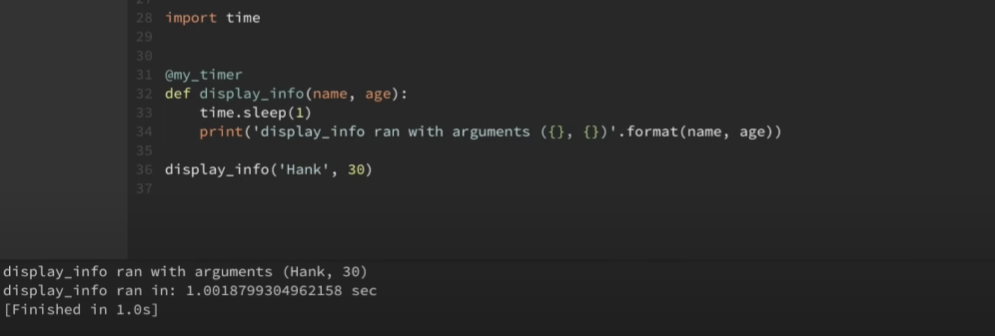


Okay, now another easy practical example that people sometimes use decorators for is for timing how long functions run. So let me go ahead and walk through this timing function here. Now you can see that this is pretty similar. We're passing in our original function and we are importing the time module and within our wrapper function we set the beginning time as T1.

Now this is a little different here. We're running the original function and setting it equal to our result. We're not returning the results yet because we want to get the end time and we're doing that here at t2. So this t2 takes the current time and subtracts the beginning time, and then we'll know how long it took to run that function. And then at the end here, we are printing out that our function ran and the time that it took, and then finally we are returning the results of the original function. And then again, last thing, we return that unexecuted wrapper, which allows all of that functionality to be added to our original function. Okay, so now let's apply this decorator to the same example that we used before.

So I'm gonna do my timer there. And actually, just so that this takes a little bit of time, I'm gonna import time here, and on the first line of this function, I'm gonna do a time.sleep of one, just so it takes one second to run that function. Okay, so now if I run this, then you can see that it printed out that we ran our display info function with these arguments, and then also our wrapper printed out that our display info ran in one second.





Okay, now I know that this video is getting a little long, but let's go ahead and look at one more example to hopefully knock these points home. What we're gonna do here is we're going to chain some decorators together.

So for this example, I'm just gonna use the two simple decorators that I have here, my logger. So what if I wanted to apply both of these decorators to one function? Well, it's just as easy as stacking the decorators one on top of each other. So if I do my timer there, then I can also add my logger there. But the way this currently is, if I run this now, then we're gonna get some unexpected results. So you can see here in our console that it says that our function ran with these arguments, but then it says that the wrapper function ran in one second, and that's not what we wanted. We wanted it to print out that display info ran in one second.

So why exactly is it calling this wrapper? If we look at our log file, we can see that it did what we expected. So maybe if we switch the order of these, then it will run how we expect. So now when we try that by switching the order, Now you can see that it runs correctly in the console, but now if we look at our log files, that it didn't log to this display info log, but instead it created this wrapper.log and logged it within that file. So let's step through this line by line to see exactly what's going on here. Okay, so remember here, if I take away this comment out that mylogger for a second, if we remember here, this syntax of putting the decorator

above this function is the same thing as saying display info equals my timer and then passing in display info to my timer. So if these two lines, this line here and this line here are the same, then what does it what is the stacked version of this equivalent to? The stacked version is actually equivalent to just to keep chaining these together. So now we can do my logger and pass in all of that into my logger. So the lower ones here on the stack get executed first and then the higher ones. Okay, so I know that this can look a little confusing, but let's go ahead and just step through this. So we're passing our original display info function into my timer. So my timer uses this

wrapper here to add the timing functionality to our original function prints out the function name and the time that it took prints that out to the console and here at the end our original function is replaced with this wrapper when we return the wrapper function so to look at what is returned at that point Let's look at the name of the function that is only decorated with one of these decorators. So for now I'm going to take away my logger here and save that and I'm going to comment out these lines here for now. So now if I am to print out this display info name and save that and run it, you can see that the name of that function is wrapper.

And it's because we're passing in this original display info function into my timer, and my timer is returning wrapper. Now this wasn't a big deal for our previous examples, but now we want to use that returned wrapper function for something else. So now we are passing that returned wrapper function into our logger also. So whenever I undo this code here, back to where we're passing this in to my logger. Now we can imagine that all of this here is equal to that wrapper function, and we're passing that wrapper function in to my logger. So when we go look at the logger, this original function here, this original function parameter, is no longer equal to our display info function, but instead it's equal to the wrapper function that was returned by the timer.

So that is why when we created the log file, that it created wrapper.log instead of displayinfo.log like we expected. And the same thing goes whenever we switch the order of those decorators. Whenever we saw that it printed out that the wrapper executed in a certain amount of time with the timer. Okay, so how do we fix something like this? It's always a good idea to preserve the information of our original function whenever we use decorators. And we can preserve that information by using the FuncTools module and the wraps decorator. So I'm going to go ahead at the top of this file, I'm going to go ahead and import this. So it is from FuncTools import wraps. Now I know this is going to be a little confusing because we're going to be using a decorator inside of a decorator, but it's really not that bad after you get used to it.

All we have to do is decorate all of our wrappers with the wraps decorator. So what I mean by that is I'm just going to go ahead and copy this and every wrapper for our decorators, we can just do this wraps decorator and we want that to wrap the original function. So now I'm also going to place this and decorate the wrapper function within timer also and save that. So now if I come back down here and I rerun the single decorated example like I did before, so I'm going to comment all of those out. So now if I do this again and print the display info name like I did before, this is when we got wrapper that printed out. Now you can see that it prints display info like we would expect.

And this extra stuff here is just because I forgot to comment out this line here. But now that this is the display info like we would expect it to be, Now we should be able to go back and run our stacked decorators without any problem. So I'm going to go ahead and uncomment out those. And we don't need this line here anymore because it's the same as this syntax here. So I'm going to go ahead and remove that and remove the printed name. And just to make sure that this is working, I'm going to go ahead and change a couple of these variables here and go ahead and run that. Okay, so now we can see that Everything was printed out correctly to the console.

It has all the correct names there. If I open up displayinfo.log, then it actually logged the correct arguments there. So we are using both of these decorators on a single function like we were trying to do. Okay, so I think that that's going to do it for this video. I really hope that this video helped in better understanding how decorators work and also how you might use them. So you're going to run into decorators all the time when working with Python and also when you're working with third-party libraries. So they're used a lot for class properties. They're also used in routing for some web frameworks and all kinds of stuff. So hopefully after this video you'll be able to realize a little more of what's going on behind the scenes when you see these decorators and other codebases.

Now the most confusing thing about decorators is trying to keep track of wrappers and outer functions and original functions and all that kind of stuff. But hopefully if you are a little confused then you can just watch through the video a couple of times and hopefully it'll start to fall in place. But if anybody does have any questions just feel free to ask in the comment section below and I'll do my best to answer those. Be sure to subscribe for future videos and thank you all for watching.



