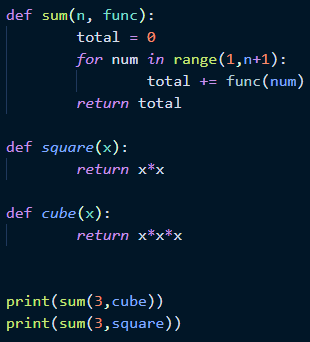
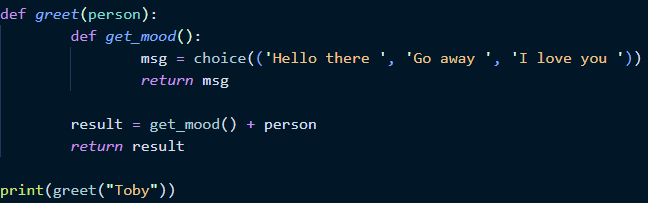
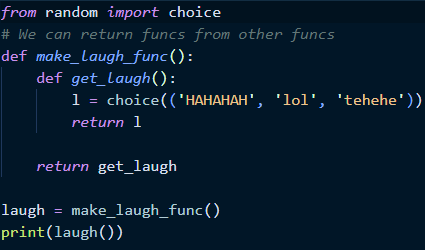
* Recall that a **higher order** function is a function that either returns a function, or accepts a function as an argument
  + In the example below, the sum() function accepts a number and some other function that you want to execute. Here, the function is square(). When called with a number and func, the sum() function will iterate over a range from 0 to the end number and sum the squares of each of the numbers in the range
  + We can also do cubes as a function



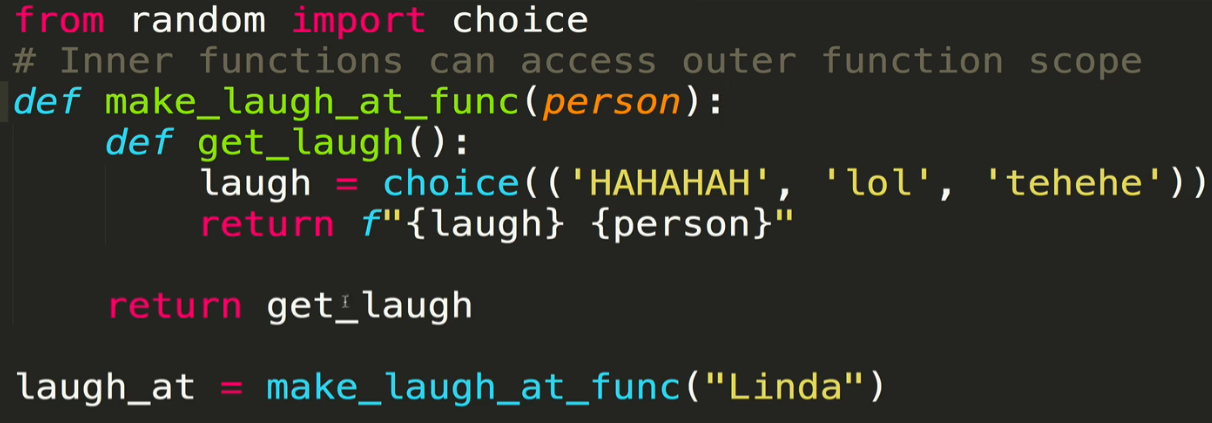
* + Example of **nested functions**: The greet() function accepts a person’s name as an argument, and within greet() is another function called get\_mood() that is called to return a randomly-selected message. That message is then concatenated with the person’s name
    - Note that the random.choice method is needed here



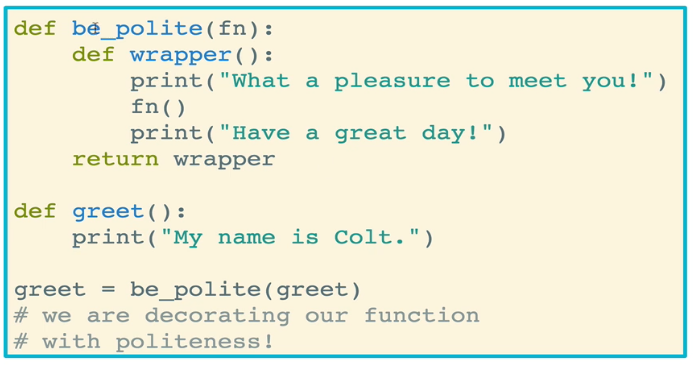
* + Example of **returning a function**: calling the make\_laugh\_func() function returns the get\_laugh() function



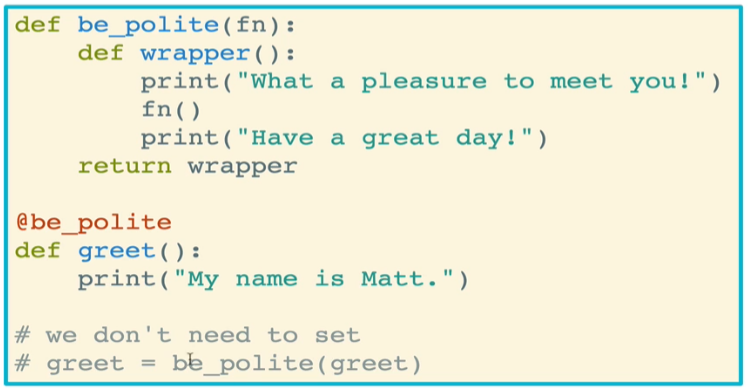
* + Similar example, except the make\_laugh\_at\_func() call accepts a person’s name as an argument. That argument is then accessible to any function within the original function (remember scope!), as the example below shows



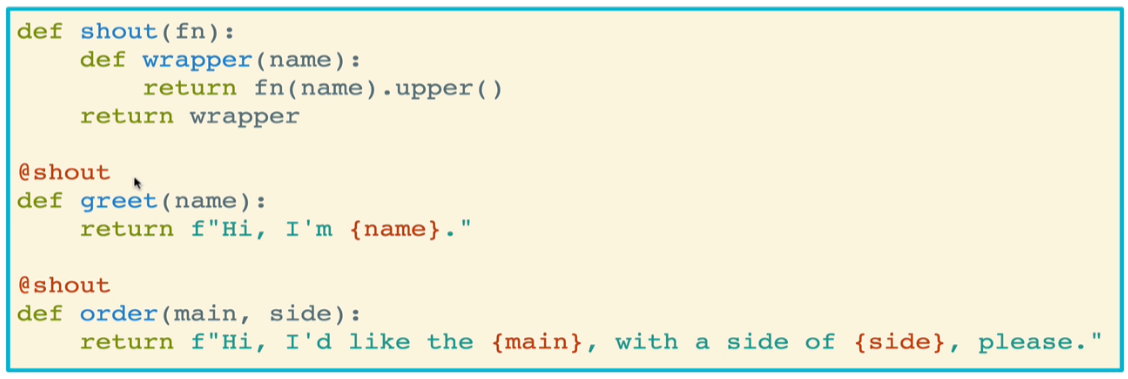
* A **decorator function** is just a function that wraps other functions to enhance their behavior
  + They are an example of higher-order functions
  + They have their own syntax, utilizing the “@” symbol
  + Simple decorator example: In the function below, the greet() function is “decorated” by the be\_polite() function. The two functions as defined are completely independent of each other. However by passing greet() into be\_polite() and calling greet() within that function, we essentially decorate the greet() function with politeness
    - Note that below it would be better to store the be\_polite(greet) function call as some other variable name to avoid confusion



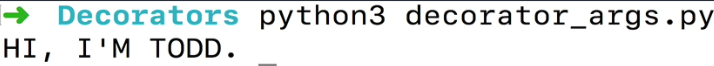
* **Syntactic sugar** allows us to make decorators more easily
  + Below we see that the functions definitions are no different from the previous example, except that the greet() function has *@be\_polite* defined indicated prior to its definition
  + This means we do not need to explicitly pass the greet() function into the be\_polite() function, Python will take care of that for us



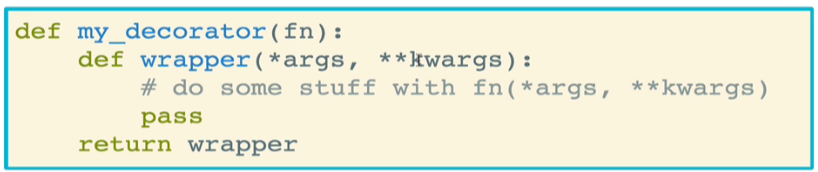
* Decorated functions oftentimes need to accept arguments
  + Example: The shout() function returns a wrapper function that executes the .upper() method on whatever function was passed into shout(). The decorated functions are greet() and order(), both of which accept one or more arguments

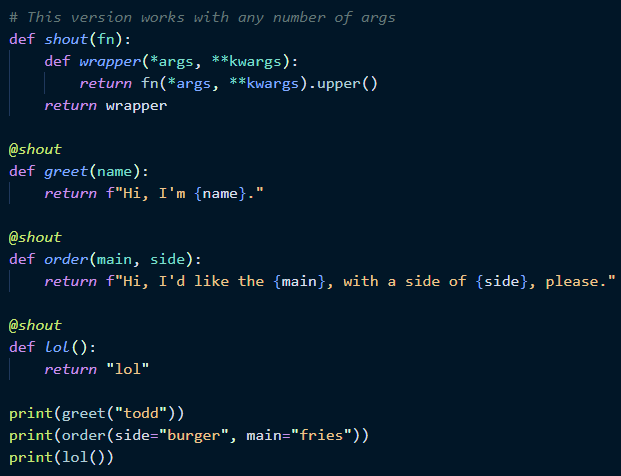


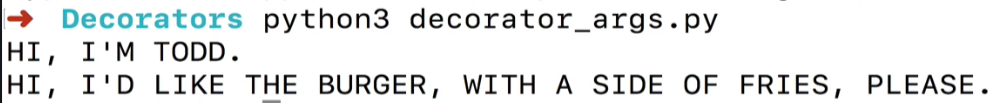
* + - When calling greet(“todd”), the result is as follows:



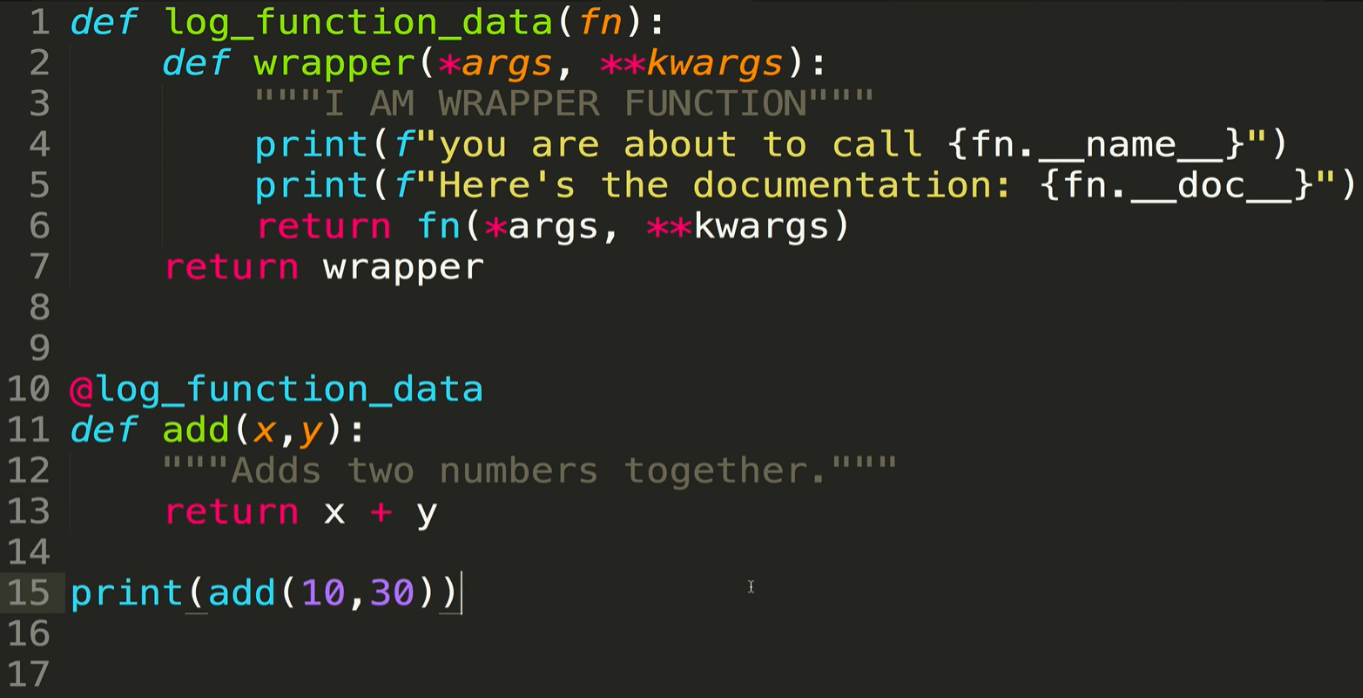
* + However we encounter an issue when calling order(“burger”, “fries”) because the wrapper function takes only 1 argument, whereas order() has two arguments. This is handled by using \*args and \*\*kwargs in both the definition of the decorator function (shout() in this case) and the wrapper function call. This is standard: wrapper functions that are returned by the decorator will typically accept an unlimited number of arguments

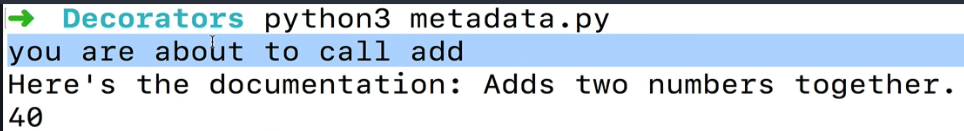




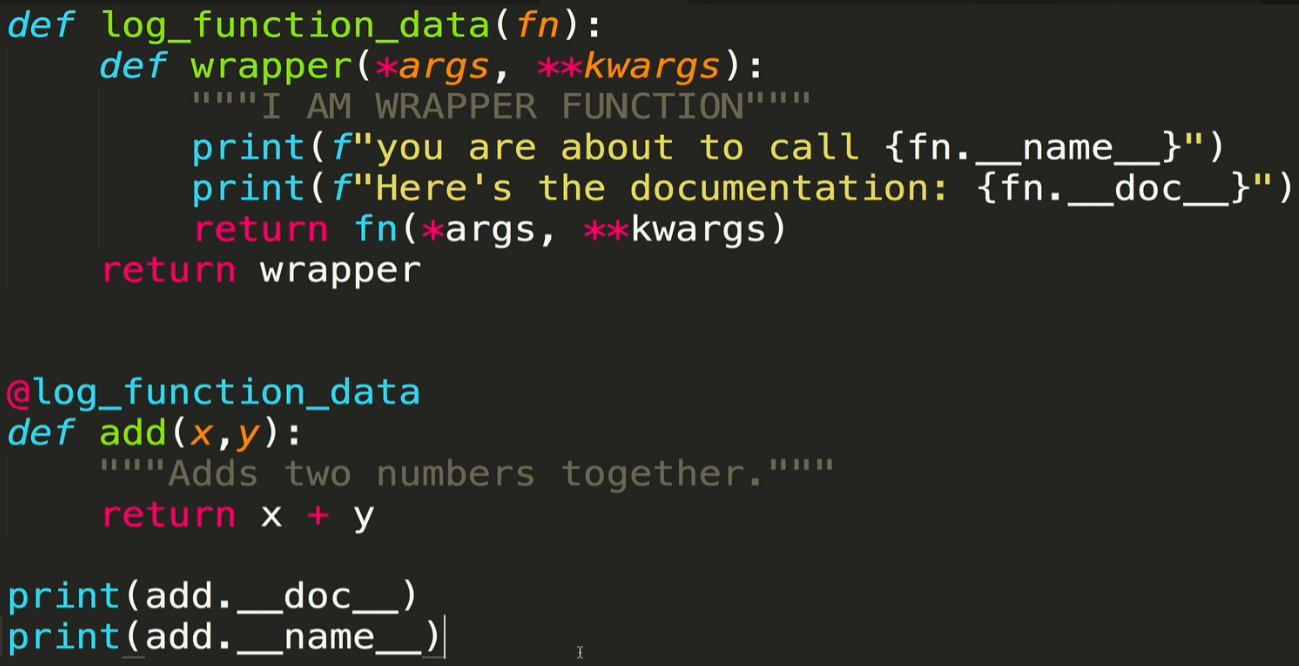


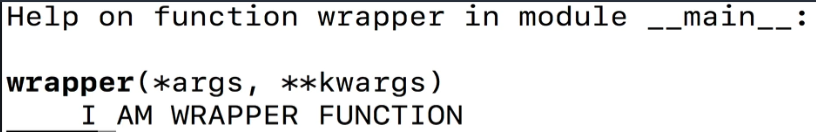
* **Metadata storage** presents an issue in decorator functions
  + Consider the example below, which as written is fully functional and should cause no problems

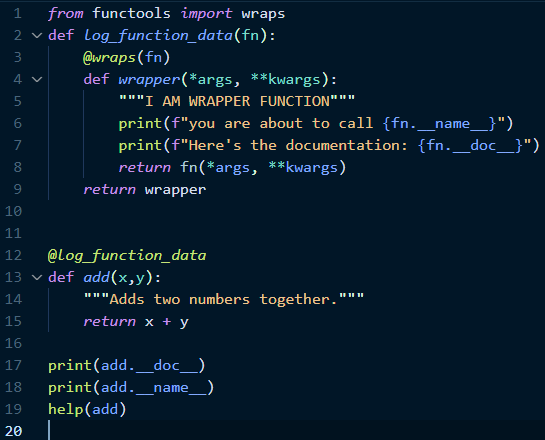




* + The problem arises when you try to access the metadata, as follows. When running help() on the decorated function (add() in this case), we get the wrapper function information as a response instead of the information for add(). That is, the metadata for add() is buried under the wrapper function

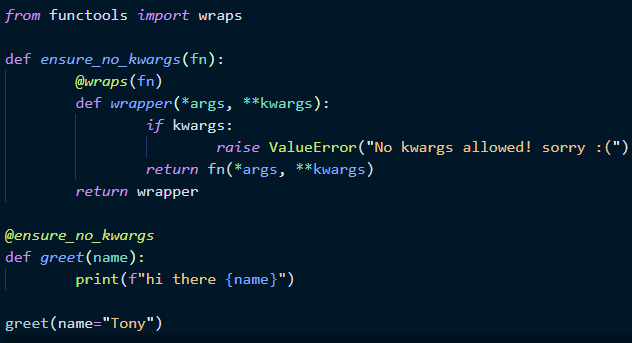


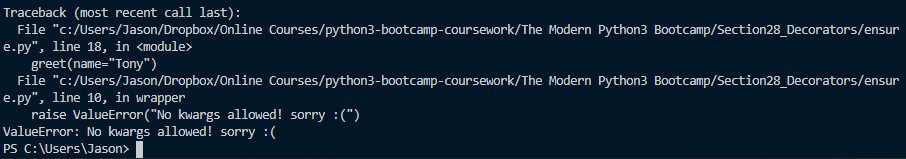


* + We solve this conundrum by using the **functools** module, which includes a method called **wraps.** Wraps is used as a wrapper function around your other wrapper function, and it ensures that the attributes of the decorated function are not lost
    - Simply add the @wraps() decorator which must include the decorated function as an argument. Place this right before your wrapper function definition
    - With that fix in place, we get the correct doc string and the correct help

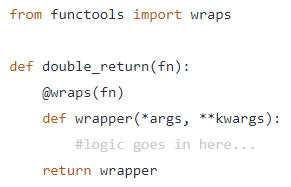


* Decorators can be used to enforce restrictions on arguments
  + In the example below, we use a decorator function to ensure that no keyword arguments are passed into the decorated function greet()
  + A decorated function will only be invoked if it is returned to the wrapper

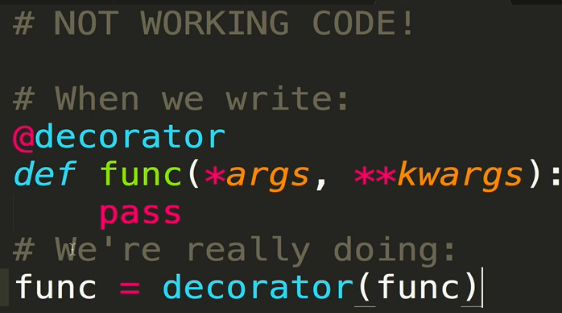




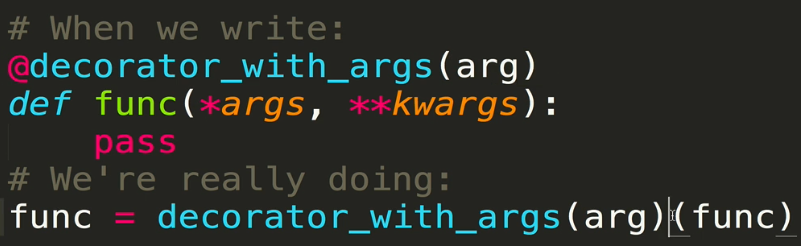
* Decorator boilerplate code below will be used in most instances in which decorator functions are being used



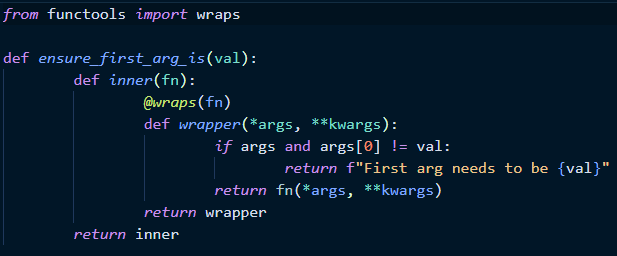
* Sometimes you will need to customize a decorator function be providing specific arguments to it. This can be done!
  + Remember that when we use the syntactic sugar for our @decorator\_function call, we are really calling that decorated function on our decorated function



* + In order to add an argument to this, we actually need to do an extra step and call the decorator function with an argument. Essentially, we need another “layer” of functions.



* + Here’s how it works:
    - You first define your decorator function, which is called with some “value” that you are trying to specify, enforce, or otherwise decorate the other function with
    - Next, within the decorator function, you define the inner() function (it can be named anything, doesn’t have to be “inner()”). This inner() function accepts the decorated function as the argument
    - Finally, within the inner() function, we go ahead and define the wrapper function() which accepts arguments from the decorated function (per usual) and performs the check
    - We then have to return the decorated function to the wrapper, then return the wrapper to the inner function, and finally return the inner function to the decorator function
    - For this specific example, the wrapper function will verify that there are any arguments at all, and whether the first argument is equal to the value that we want to verify against



* + Full example which checks that a fav\_foods() function is called with the first argument being “burrito”

