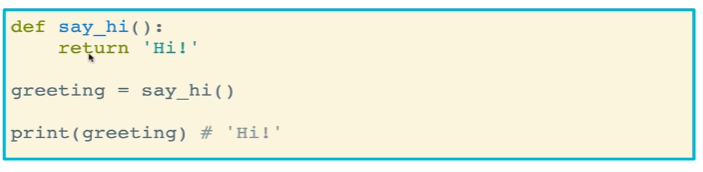
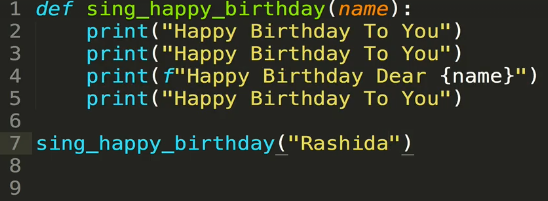
* What is a function?
  + A **function** is a process for executing a task
  + A bunch of lines of code wrapped up into a package that we can open and use anytime
  + It can accept an input and return an output. The output returned can and often does depend on the input
  + Useful to executing similar (but not necessarily identical) procedures over and over again
  + Many objects have built-in functions, called methods
    - Some methods require inputs, and some do not
* Why use functions?
  + They help you stay **DRY** – **D**on’t **R**epeat **Y**ourself
    - It helps you avoid having to rewrite code over and over again
  + Helps you “abstract away” code for other users so that users and other codes don’t have to worry about it
    - Imagine having to write the “print()” function every time you wanted to print something
* Function Structure
  + Defining a function begins with the keyword **def** followed by the name of the function and a set of parentheses and a colon
  + A function is called by entering the function name and parentheses, passing in any variables as needed (more on this later)
  + Defining a function does NOT call it. It simply defines it



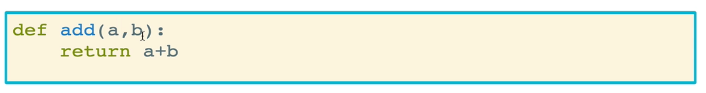
* The **Return** Keyword
  + The return keyword is used to return the value of a function that is called, which is oftentimes assigned to a variable
    - The returned value must be explicitly assigned to the **return** keyword. The return keyword will output whatever value is placed after the keyword
    - Tuples can be used to return multiple values (more on this later)
  + The return keyword exits the function. Any lines of code within the function that come after the return keyword is not executed (unless return is part of an if statement and is passed over)
  + The return keyword pops the function off the **call stack**
    - A call stack is a “to-do” list of what Python must do when executing your code
    - Learn more at 1



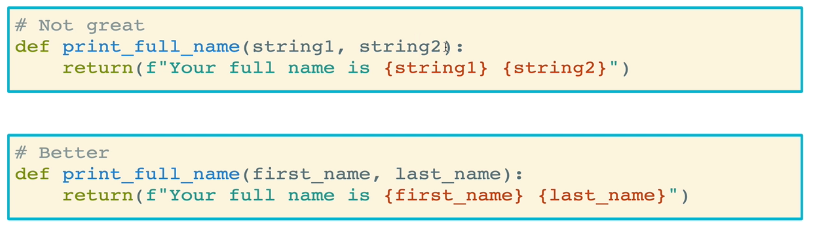
* Functions take inputs in the form of **parameters**, which in turn can affect how the function works
  + You can call a parameter anything you want, it’s just like naming a variable
    - Just make sure you abide by the same rules as when defining variables
  + These parameters can then be referenced within the function itself
  + Parameters exist only inside the functions that they are used in. They do not exist outside of the context of the function



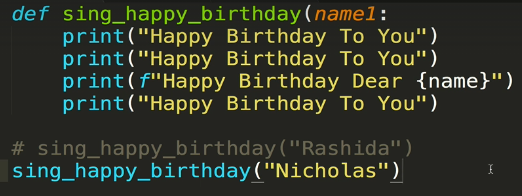
* Functions can take multiple parameters
  + They are declared and separated by commas



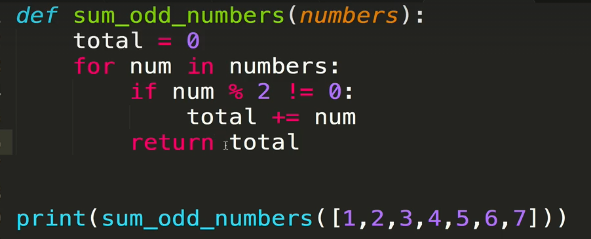
* + It is often useful to use parameter names that are semantic and make sense for what the function is doing



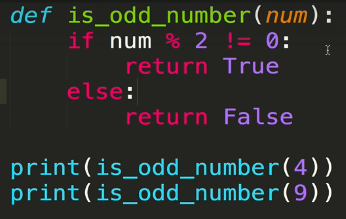
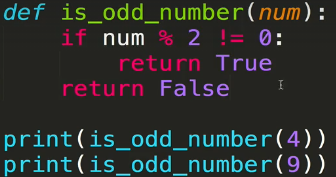
* What is the difference between **parameters** and **arguments**?
  + The two are often used interchangeably
  + A **parameter** is a variable within a method definition
  + When a method is called, the **arguments** are the data that are passed into the method’s parameters
  + In the example below, *name* is a parameter and *Nicholas* is an argument



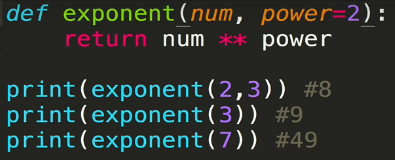
* Common mistakes with the return keyword
  + Placing it incorrectly within a loop, as a result of an indentation error
    - In the example below, the loop only runs through the first number. Since *return* is within the for loop, the function will exit after running just once.
    - To fix it, you need to un-indent the return keyword and bring it out of the for loop



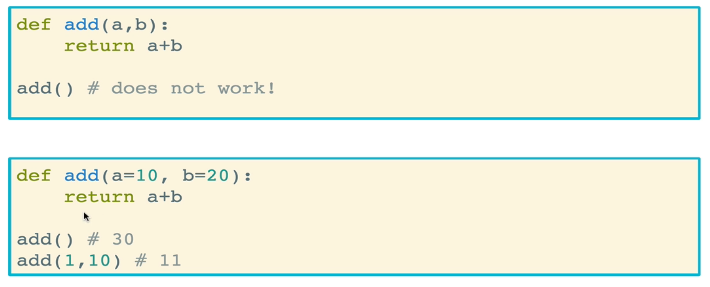
* + “Unclean” code
    - The code below is functional and works correctly. But the *else* clause is actually unnecessary. This is because if the number is odd, the function will return True and exit the function, never getting to the else statement. If the number is even, it will fail the first conditional check and all code within it, and go to the next section in the function. We simply need to write “return False”

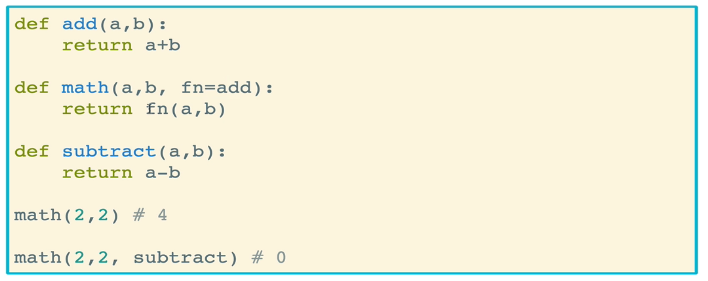
* + Another example of the unnecessary else
* **Default parameters** give parameters a default value when that value is not passed as an argument to the function call
  + By default, if a parameter is not given a default value, it must be passed in (else an error is thrown)
  + This is easy: simply add an equal sign and a default value for that parameter



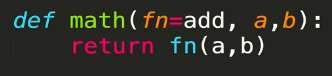
* + Another example



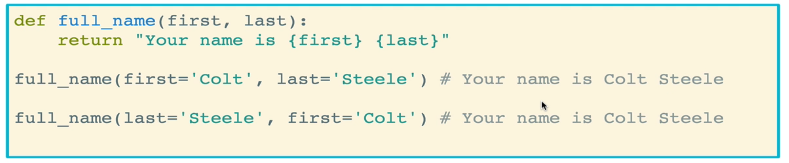
* + Default parameters are useful because:
    - They allow you to be more defensive, and help you avoid your code throwing errors
    - Avoids errors with incorrect parameters
    - It makes your parameters more readable
  + Default parameters can be lists, dictionaries, strings, Booleans, and even other functions!



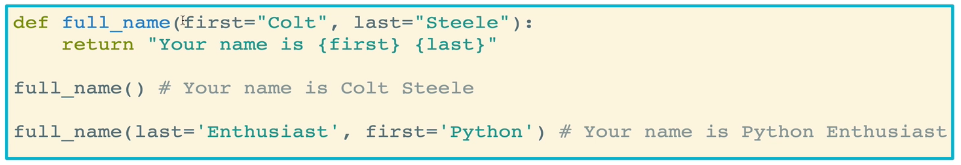
* + An important note is that when defining functions and parameters, the following must be observed:
    - Parameters with default values must be defined *after* parameters that do not have default values
    - Alternatively, all parameters must have default values
    - This is because when passing in arguments, they are assigned to parameters in the exact order that those parameters are defined in the function
    - The code below where the first parameter is a default function will not work



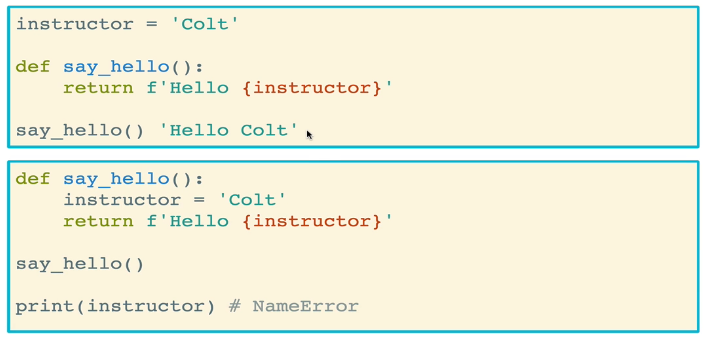
* **Keyword arguments** are arguments that specify the parameter that it is being passed to by using an equal sign (“=”)
  + When using keyword arguments, you can pass arguments in any order. It does not matter if you pass them in a different order from that in which the parameters are declared in the function definition
  + Of course, this only works if you actually know the names of the parameters
  + It also makes function calls more clear. There is no ambiguity which argument is being assigned to which parameter
  + It will be very useful when passing in dictionaries and unpacking a dictionary’s values



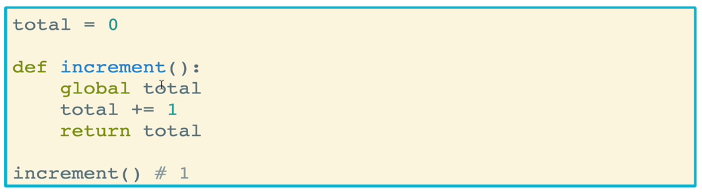
* + Keyword arguments are different from default parameters
    - Default parameters are assigned when *defining* a function and using an “=” to set up the parameters and their default values
    - Keyword arguments are assigned when *calling* a function and using an “=” to assign the argument to a parameter



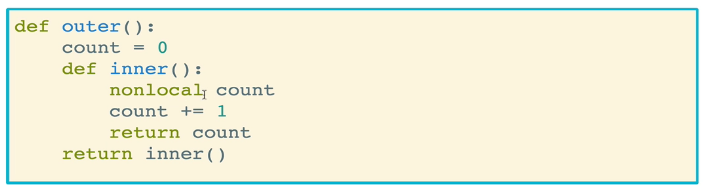
* **Scope** refers to the fact that variables and properties in Python are not always available in every part of an application or code. There are rules governing where variables can be accessed
  + In any function, whenever we define a variable within a function, it is only available within that function. In other words, they are scoped to that function and are not available outside of it
  + Global variables that are defined outside of a function are accessible by any function



* The **global** scope
  + Any variable defined that is not in a function is considered a global variable
  + If you try to *manipulate* a global variable inside a function, however, you MUST use the **global keyword** to tell Python to look for the global variable. If you do not, a reference assignment error will be thrown
  + Note that you can *access* and *use* a global variable without the variable keyword. But you cannot manipulate it



* The nonlocal keyword
  + This allows you to modify a parent’s variables in a child (nested) function
  + A parent variable is NOT the same as a global variable; that variable is defined within the parent function and not in the global scope



* How to **document** functions
  + You can use triple double quotes (“”” “””) to assign a description to a function, usually what a function does
  + You can access the description using the *function\_name.\_\_doc\_\_* command,