Unit 3-1: Functions in Python

* What are Functions?
  + Functions are extremely powerful tools. They define a process that runs logic. They tell your computer to do something. Here, we have a function called "make\_a\_barnyard\_noise" that prints the word "moo" to your screen:
    - def make\_a\_barnyard\_noise():
    - print("moo")
  + That's right! That little piece of code will make your computer moo.
  + Let's dive into world of endless possibilities functions provide.
* Defining Functions in Python
  + In Python, functions are defined using the def syntax shown below.
    - def myfunction():
    - # code...
  + Following def, we have the function's name with parentheses and a colon. This is just the beginning of the function. We will also need to include some code to indicate what the function will do.
* Function Arguments
  + Any required arguments are placed inside of the function's parentheses. In the modified myfunction() below, we are now defining the function to take one input, labeled var1.
  + With this function we are creating a new variable, var2, which is simply var1 plus 2. We use the return keyword to specify that the function will output var2. Note that the function code is indented after def. This is now a complete (albeit simple) function.
    - def myfunction(var1):
    - var2 = var1 + 2
    - return var2
  + If the return statement is not specified in the function definition, the function will return None. Once it reaches a return statement, the function will terminate. Functions may contain multiple return statements, sometimes embedded in if-else statements, or loops. It's important to remember that the function will terminate when it first hits any valid return statement.
* Calling Functions
  + We can now use the function. Below we're calling the function, inputting a 3 for var1 and assigning the output to a variable called five\_var.
    - def myfunction(var1):
    - var2 = var1 + 2
    - return var2
    - five\_var = myfunction(3)
  + When we call the function with an input of 3, the integer is assigned as var1 inside the function. Then, the internal code of the function is run, creating the var2 variable and returning it as the output. The output of the function in this case is the integer 5, which we assign to five\_var.
  + If you print the output of the function, you can see that it is 5.
    - print(myfunction(3))
    - > 5
    - print(five\_var)
    - > 5
  + Every time you run the function, the code for the specified process is run on the input and an output is generated.
* Reuseability:
  + Functions are essential because they allow you to define reusable processes. Let's look at a useful example. Imagine that we didn't have the string function upper built into Python, so we wrote our own. The custom function below converts a string from lowercase to uppercase.
    - def uppercase(input\_str):
    - output\_str = ''
    - case\_switches = {
    - 'a':'A', 'b':'B', 'c':'C', 'd':'D', 'e':'E', 'f':'F',
    - 'g':'G', 'h':'H', 'i':'I', 'j':'J', 'k':'K', 'l':'L',
    - 'm':'M', 'n':'N', 'o':'O', 'p':'P', 'q':'Q', 'r':'R',
    - 's':'S', 't':'T', 'u':'U', 'v':'V', 'w':'W', 'x':'X',
    - 'y':'Y', 'z':'Z'
    - }
    - for char in input\_str:
    - if char in case\_switches.keys():
    - output\_str = output\_str + case\_switches[char]
    - else:
    - output\_str = output\_str + char
    - return output\_str
  + To understand how the function works, recall that:
    - You can iterate through strings in Python, and each element will be one of the characters.
    - You can concatenate strings by adding them together.
  + The function takes a string as its input then builds a new string, output\_str, by iterating through all of the characters of the original string and replacing any lowercase letters with uppercase ones. These swaps are defined as key-value pairs in the case\_switches dictionary. Finally, a new string is returned by the function.
  + We can use the function to convert a string to uppercase.
    - print(uppercase('abcd'))
    - > 'ABCD'
    - print(uppercase('123abc'))
    - > '123ABC'
    - a\_string = 'hello'
    - upper\_string = uppercase(a\_string)
    - print(upper\_string)
    - > 'HELLO'
  + If we find ourselves performing this conversion repeatedly, defining its process in a function can save us a great deal of time and effort. It'd be tedious to rewrite the necessary code every time we wanted to convert a string.
* Multiple Arguments
  + So far, our functions have only taken one argument, but you can define functions to take any number of arguments (or none at all). For example, we could define a function that adds numbers together:
    - def adder(number1, number2):
    - return number1 + number2
    - print(adder(4, 9))
    - > 13
* Keyword Arguments
  + Python functions can also take arguments that have a default value, known as keyword arguments. These are specified in the function definition like variable assignments. The function below is designed to "clean" strings by removing certain characters and then converting the strings to floats.
    - def strs\_to\_floats(list\_of\_strs, remove\_chars=['$',',']):
    - float\_list = []
    - for string in list\_of\_strs:
    - for char in remove\_chars:
    - string = string.replace(char, '')
    - float\_list.append(float(string))
    - return float\_list
    - prices = ['$25.22', '$50.12', '$1,235.01']
    - print(strs\_to\_floats(prices))
    - > [25.22, 50.12, 1235.01]
  + Notice that, when we call the function on the list of string prices, we only provide the list of strings as an input. Because we do not specify anything for remove\_chars, it defaults to ['$',','], and the function removes the dollar signs and commas.
  + If we wanted the output in terms of cents, we could specify to only remove the dollar signs by manually passing in a value for the remove\_chars keyword argument.
    - print(strs\_to\_floats(prices, remove\_chars=['$',',','.']))
    - > [2522., 5012., 123501.]
  + Our list of characters to remove contains the decimal point, too. So, the function removes the decimal points prior to converting the strings to floats, and the output changes.
  + Functions can take multiple keyword arguments, just like they can take multiple arguments.
  + You can define a function as taking any combination of arguments and keyword arguments, **but note that keyword arguments must always come after normal arguments!**
    - def func1():
    - # code...
    - def func2(arg1, arg2, arg3, kwarg1=1, kwarg2=True, kwarg3=[1,2,3]):
    - # code...
    - def func3(kwarg1=True, kwarg2=False):
    - # code...
  + Remember that arguments must be provided when the function is called, but keyword arguments are not required, as there is a default specified. You could, for example, call func3() above without inputs, as all of the arguments are keyword arguments.
* Nested Functions
  + Functions can call other functions within them, making it easy to organize your code into neat, clear pieces. Below are manual implementations of the sum(), mean(), and variance() functions for a list of numbers. The perform\_calculation() function takes a list of numbers, along with an optional operation specifier for which function we want to run.
    - def sum(numbers):
    - total = 0.
    - for n in numbers:
    - total += n
    - return total
    - def mean(numbers):
    - return sum(numbers) / len(numbers)
    - def variance(numbers):
    - sq\_mean\_deviations = []
    - num\_mean = mean(numbers)
    - for n in numbers:
    - sq\_mean\_deviations.append((n - num\_mean)\*\*2)
    - return mean(sq\_mean\_deviations)
    - def perform\_calculation(numbers, operation='sum'):
    - if operation == 'sum':
    - return sum(numbers)
    - elif operation == 'mean':
    - return mean(numbers)
    - elif operation == 'variance':
    - return variance(numbers)
    - else:
    - return None
  + The mean() function uses the sum() function internally, and the variance() function uses the mean() function. The perform\_calculation() function is able to call any of these three functions. In addition, our code is more compact and clear than if we had coded the math operations from scratch every time.
* Passing Functions into Functions
  + Any Python object can be passed in as a function argument — even functions themselves! For example, we could change the perform\_calculation() function to take a function instead of a string for the operation keyword argument.
  + def perform\_calculation(numbers, operation=sum):
  + return operation(numbers)
  + Now, the operation variable is assigned as the function we want to perform on our numbers, and sum() is our default function. To modify the function to perform on our numbers, all we need to do is specify the alternate function.
    - perform\_calculation(numbers, operation=mean)
    - perform\_calculation(numbers, operation=variance)
  + Notice here that we are not calling the function, as there are no parentheses following mean or variance when we pass it in to the operation keyword argument. Without parentheses, we are referring to the function object itself.
* \*args
  + You may be wondering if it is possible to define a function that can take an arbitrary number of arguments or keyword arguments. For example, if there was no way to know in advance exactly how many arguments or keyword arguments will be supplied to the function.
  + What if we wanted our adder() function from earlier to add together any amount of numbers supplied as arguments? What if we wanted the function below to work?
    - adder(1, 2)
    - > 3
    - adder(4, 5, 6, 5)
    - > 20
    - It's possible to define functions to work in this way. Instead of writing out each argument name, we can put \*args in the function definition.
    - def adder(\*args):
    - total = 0
    - print('args:', args)
    - for arg in args:
    - total += arg
    - return total
  + \*args becomes a tuple containing the arguments supplied to the function when it's called. We then iterate through the \*args tuple and add the values to the total. By printing \*args in our function, we can see what happens.
    - adder(1, 2, 3)
    - > args: (1, 2, 3)
    - > 6
    - adder(7, 33)
    - > args: (7, 33)
    - > 40
* \*\*kwargs
  + You can also specify an arbitrary number of keyword arguments using the \*\*kwargs placeholder. Similar to how \*args evaluates to a tuple of values (args) inside the function, \*\*kwargs evaluates to kwargs, a dictionary of key-value pairs.
  + To demonstrate this, we can write a function that simply prints out the dictionary:
    - def kwarg\_printer(\*\*kwargs):
    - print(kwargs)
    - kwarg\_printer(var=10, other\_var='a string')
    - > {'var':10, 'other\_var':'a string'}
  + You can easily create a function that can take any number of arguments and keyword arguments by including both \*args and \*\*kwargs in the definition:
    - def flexible\_function(\*args, \*\*kwargs):
    - # code...
* The Power of Functions
  + Functions add an incredible amount of power and flexibility to your code. They're the building blocks for all large, complex programs and processes.
  + You'll consistently use functions in your Python code.
  + Practice breaking your code up into functions to make it cleaner, clearer, and more concise whenever possible.
* Additional Resources
* If you're interested in gaining more practice with this topic, check out the following resources:
  + [Codecademy: Learn Python:](https://www.codecademy.com/learn/python) "Section 4, Functions" (particularly Function Syntax).
  + [DataCamp: Intro to Python for Data Science:](https://www.datacamp.com/courses/intro-to-python-for-data-science) "Section 3, Functions and Packages."
  + [DataCamp: Python Data Science Toolbox (Part 1):](https://www.datacamp.com/courses/python-data-science-toolbox-part-1) Section 1, "Writing Your Own Functions."
  + Bonus! For an advanced challenge, see [DataCamp: Python Data Science Toolbox (Part 1)](https://www.datacamp.com/courses/python-data-science-toolbox-part-1), and check out "Section 2, Default Arguments, Variable-Length Arguments and Scope."