Module 7 – Functions

Instructions

1) If not already open, from the Anaconda Navigator Home screen, launch Sypder

2) In Spyder, go to "File", then "Open", then navigate to the location where you saved: "7_Functions.py", and select it to open the script file.

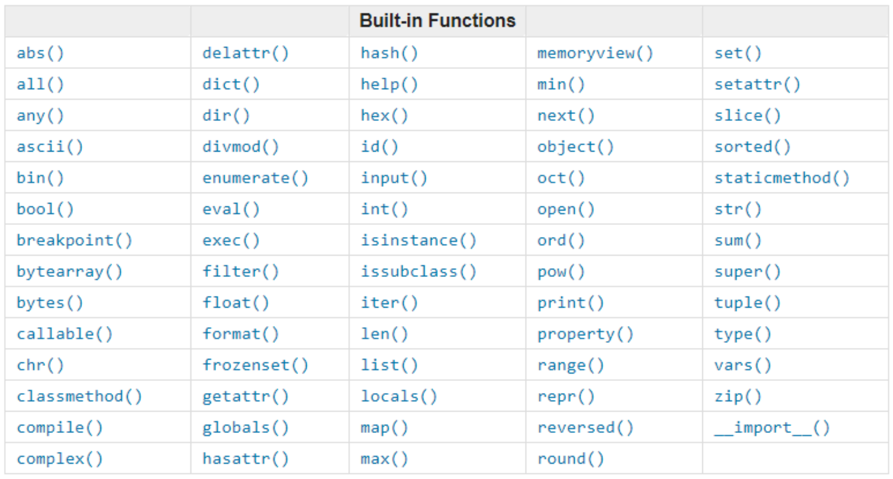
3) Introduction to Functions

So far, we've worked quite a bit with some of Python's built-in functions. Examples include: **print (), type (), float (),** and **int()**, among others. Sometimes, though, we want a block of code that executes something over and over again for our own use that doesn't exist either in Python's library or in any other module. Simply speaking, they are re-usable blocks of code that we call when needed (per Line 7 in the commented section of the script).

Here are some general tips about functions before we get started

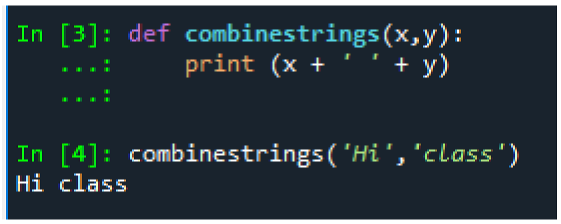
- Treat built-in function names as reserved words, so avoid using them as names to define your function
- **def(**inition**)**:
  - In Python a function is some reusable code that takes arguments(s) as input, executes, and then (can) return a result or results
  - Is called into being using the **def** reserved word
  - Structure is: function name, parentheses, and arguments in an expression
  - Try and see if a built-in function exists, but if not, it's time to make your own.

Here are the major built-in functions of Python. Explore and use them! But don't call your functions any of these names:



		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Next, let's take a look at the basic architecture of a function. I've created a simple function called **combinestrings**, which is not a name of one of Python's existing functions, nor is this is in your script file.



```
In [3]: def combinestrings(x,y):
   ...:     print (x + ' ' + y)
   ...:

In [4]: combinestrings('Hi','class')
Hi class
```

In purple text above is the function **definition** (def). This is required in order to define a function. The blue text is the **function's name** (**combinestring**s, in my case). In white (x,y) are two **parameters.** A parameter is a *variable* used in the function definition. They are a "handle" (think of them as placeholders) that allows the code in the function to access **arguments** that you enter when you actually call the function later. Not all functions need parameters, but the more useful ones do. To be clear, **defining** and **creating** (naming) the function *does not* execute the body of the function – you have to call it (with required arguments, if any) later, as I've done in the example above.

When I actually called the **combinestrings** function, I enter two arguments. An **argument** is a value passed into the function when it's called by the programmer. In my case above, these are ('Hi', and 'class') in green text. Arguments direct the function to do different kinds of work when called at different times. They are placed in parentheses after the name of the function. If you have them, the number and order of arguments and parameters must match.

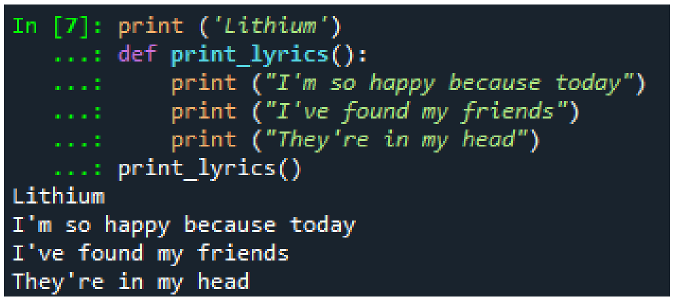
As you saw with loops and conditional statement, a colon is required after the parenthesis that hold the parameters, and indentation is required for all subsequent lines within the function.

Finally, the most useful functions include a **return** statement at the end of the function execution that returns/sends back the result of the function. Mine above does not, but you will soon see some examples of ones that do.

4) Working with functions

First review Lines 17-24 to get a sense of a simple function. This one does not have any parameters, and so, no arguments are included when calling the function on Line 23. **Run Lines 18-22** together first. Then, enter print_lyrics() in the input on the console. (Or, simply run Line 23). Review the output:

Calling the function executes what's inside, which in this case, are three print statements that follow the initial print statement that is sitting on Line 18.



```
In [7]: print ('Lithium')
   ...: def print_lyrics():
   ...:     print ("I'm so happy because today")
   ...:     print ("I've found my friends")
   ...:     print ("They're in my head")
   ...: print_lyrics()
Lithium
I'm so happy because today
I've found my friends
They're in my head
```

Let's move to a more robust example. **Run Lines 31 and 32 together**, which each are separate lists of integers, stored in two separate values.

Take a look at lines 34-37 before you run anything. First, answer these prompts, using the introduction above if you need it:

*Which are the **parameters**? Which are the **arguments**?*

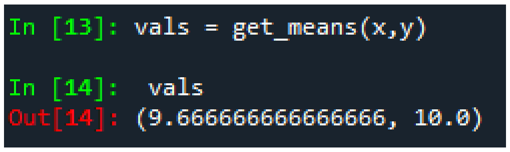
One thing that users that are familiar with R but new to Python may find odd is that there is no built-in mean() function. You might have caught that in the Table above. There are modules you can import that provide access to this function, but for now, let's build our own as an example since it's a pretty straightforward kind of thing to develop as a function.

I define the function and name it "get_means" on Line 34. Then I include two **parameters**, since this function is designed to evaluate two separate means. Geographers, of course, record and store locations of physical (or abstract) phenomena on the Earth's surface using x,y coordinate systems. So, this function will report the **mean center** of the x and y values for each list of those respective coordinates, defined in the lists on lines 31 and 32. *Sidenote: there are built in functions in ArcGIS and QGIS that do this.* Within the function on lines 35 and 36, I set two variables that will only be used in this function. These store the input of the sum of whatever argument is passed into the 'xvals' parameter location (first) divided the length (len) of that argument, and does the same for whatever argument is passed into the 'yvals' parameter location (second). Then, the values associated with the two new variables I created in the function are **returned** on Line 37.

So, after this explanation, **run Lines 34-37 together**. Notice nothing is reported in the output of the console.

Note that I set a new variable to store the returned values of the function. You don't have to do this (as in, I could have just typed get_means(x,y) to get the output). But, I might want to store those output mean values for later use, which is pretty common with functions.

After running 39 or entering **vals = get_means(x,y)** in the console, notice I still don't see the returned output. To finally see it, enter **vals** in the console:



```
In [13]: vals = get_means(x,y)

In [14]:  vals
Out[14]: (9.666666666666666, 10.0)
```

Notice, though, that we have a fair number of probably unnecessary digits stored after the decimal place, at least for reporting purposes. We know how to fix that now, though, using our formatting operators from previous modules, which the print statement does. It also references the **index positions** of the reported outputs that were returned. As in, the mean value of x is reported first, so is at position 0. The mean value of y is reported second, so is at position 1. These are stored, by the way, in a data type that we haven't encountered yet: a tuple.

I could call the **get_means** function again in the code anytime I want, so long as I pass two arguments into the function that need means calculated for each one.

Activity: can you create a function that reports a cumulative sum of the values in x and y, respectively, and returns them?

5) Another function example

Lines 45-62 present a more complex function. At this point, it's not important to understand the full nuts and bolts of what's happening in the function. It's using operations that apply to data objects we haven't seen yet. But it does show that you can embed definite loops in functions (and of course, conditional statements, if you wanted to), which helps to provide even greater functionality. This example uses the same lists of values stored in x and y on Lines 31 and 32 to operate that were used in the get_means function. It also requires the math module to get access to the .sqrt() function.

This function reports the standard distance, which is a companion metric often reported along with a mean center in geographic analysis. **Run Lines 45-62 together** to see the full output, or run subcomponents that you're interested in if you're curious.

This completes the structured part of Module 7. Feel free to experiment some more with setting variables and values, working with print statements, or anything else you are curious about.