Python Session 3

After going through the content, please take a minute to respond to this feedback survey of our reference manuals.
Feel free to use the outline on the left side of the document to jump to different sections.
Functions are usually a difficult concept to understand. If you have any questions, feel free to come to the office hours hosted by Sahas and Scott!

Functions

What are functions?

Functions are pre-writable code that can **improve readability** and **be reused**. Algorithms can be written **once** in a function and be **reused** as many times without rewriting it again. **`print()`** and **`input()`** are some of the pre-written functions that we've used so far.

Function Declaration and Call

All functions need to be **declared** before we can call them. Declarations include the **`def`** keyword and may contain zero to as many parameters between the parentheses, following this format: **`def <func_name>(<parameters>):`**
Definitions are written after the function name is declared. In code, functions are placed **before** you use them.

```
def foo():              # This is how we declare a function
    print("We in!")     # This is where we define a function (notice the indent)
```

Functions that you define can be **called** by simply writing the function name with parentheses following it.

```
# using foo() defined in previous example ^^^^
foo()
```

```
We in!
```

Functions can also be placed in a list and indexed to call. These are called **function pointers**. This is not often used, but is good to know in case the situation ever comes.

```
def foo(x):
    print(x)
def bar():
    print("Hello!")

f_list = [foo, bar]
f_list[0](20)
f_list[1]()
```

```
20
Hello!
```

Function Parameters, Arguments, Returns

Functions may have **parameters** that can be set when you declare a function. Parameters are like variables and are created between the parentheses. Multiple parameters are separated by commas. Functions with parameters require **arguments**, or values/variables, to be passed in when they are called.

```
def foo(s, num):                # foo() requires two parameters: s and num
    print(str(num) + ":", s)

phrase = "The quick brown fox jumps over the lazy dog"
foo(phrase, 9)                  # wee pass in a string and number to use foo()
```

```
9: The quick brown fox jumps over the lazy dog
```

Be aware that not passing in arguments in parameterized functions **will give an error**… with an exception. **Optional or default parameters** do not require an argument to be passed in for it! These parameters can be created by assigning these parameters with a **default value** that the function can use if no arguments are passed in. Python only allows for optional parameters to be placed **after all the required parameters**.

```
def foo(s, num=0):              # num is now an optional parameter, placed at the end
    print(str(num) + ":", s)

phrase = "The quick brown fox jumps over the lazy dog"
foo(phrase)                     # not passing in a value for num is allowed now
```

```
0: The quick brown fox jumps over the lazy dog
```

Python reads and matches arguments from left to right. However, sometimes, we may want to pass in an argument for a specific parameter. We can do this with **keyword or named arguments**. This can be done by assigning a specific parameter with the argument when calling it. Arguments for optional parameters must be placed **after all non-specific arguments**.

```
def foo(w, x, y="y", z="z"):          # y and z are both optional parameters
    print(w,x,y,z)

foo(3, 2, z="K")      # assigns parameter z with "K", skipping the arg for param y
```

```
3 2 y K
```

Functions usually **return** a literal or variable value, but it is not required. This can be done by using the `return` keyword. Just having `return` on its own will exit the function immediately. A value can be put after the `return` to return that value while exiting the function. This returned value can then be used in an algorithm or stored in a variable or immediately used in more functions to form complex one-liners.
Returns can be placed anywhere in the function and there can be any amount of them throughout the function.

```
def power(x, y):
    pow = x
    for i in range(1, y):
        pow *= x
    return pow

print(power(2, 4))    # immediately used the return of power() in print()
```

```
16
```

⚠ **Be aware that *list* parameters are pass-by-reference. This means that even though the variable names are different, the original list themselves will be modified when changed in the function!**

Main Function

We mentioned in session 1's Extras that Python's main function is different from other programming language's main function. This is because it's a scripting language and doesn't require a main function. However, we can still create a "**`main()`**" function by adding something like this in your code:

```
if __name__ == "__main__":
    # code here
```

Note that code not in this "**`main()`**" function will behave as normal, and any code before it will run before we get to it, unless it was written in a **`utils.py`**.

Extras

Naming Scheme Review

With the introduction of parameters, it may be a good idea to revisit naming schemes. **Parameter names are the same as variables**; there are certain rules they must follow to be valid. These rules are:

- Cannot start as a number
- Numbers, characters, and underscores only
- Can start or end with underscores, or can be only made of underscores (usually for throw-away return var)

One-liners

The nice thing about functions is that when they return things, you can immediately use it in another function. This allows us to create **one-liners** that can simplify the code a little more.
One-liners use this property to fit and use functions all in one line to achieve a certain goal that either makes it **more readable** *or* **more memory efficient** (which in our case, just means less variable allocation).

```
print(int(input("Enter Number: ")) + 15)
```

```
Enter Number: 5
20
```