Lesson 6: Functions

Functions

Many times while programming, we want to sort our code into small tasks that can be used many times. *Functions* contain code that completes a task and can be called several times. But what is a function exactly?

Inside math, functions are often written as $f(x)$. For example:

$$f(x) = 2x + 1$$

But what's really happening here? Our function, f, is taking in an input, x, and spitting something back out. We can see this if we look at multiple values for x:

Input, x	Inside the function ($2x + 1$)	Output, $f(x)$
0	$2(0) + 1$	1
1	$2(1) + 1$	3
2	$2(2) + 1$	5
3	$2(3) + 1$	7
4	$2(4) + 1$	9

In programming, functions are very similar. They can take in inputs and output something else. Let's create a function like our mathematical $f(x)$ from above. To start a function we use the ***def*** keyword:

```
def f(x):
    return 2*x + 1
```

1. Here, we define a function, ***f***, that takes in a parameter (input), **x**
2. Notice the colon(**:**) after the function definition.
3. Notice how we use the return keyword to return our output
    a. **NOTE:** We are NOT printing anything to the command line here. We are simply returning an output when the function is called—more on that in a second.

Okay, but all we've done so far is define a function. In order to actually use it, we need to *call* it. So how do we do that?

In order to call a function, we simply need to repeat its name by putting in inputs.

```
def f(x):
   return 2*x + 1

print(f(3))
```

1. Here, we call our function f by inputting 3 as x
    a. Our function sets the parameter x = 3 and then returns 2*x + 1
2. We then print the function output to the command line

We can also pass in variables as parameters or use variables to store function outputs.

```
my_variable = 3  # store the value 3 into my_variable
y = f(my_variable)  # store the return value of f(my_variable) in y
print(y)  # print the variable y to the command line
```

Functional Parameters

Functions can take multiple parameters (inputs). We're not limited to using just one.

Example 1

Write a function that takes in two values and prints their sum. Then, call it.

```
def sum(x, y):
    return x + y

print(sum(1, 2))
```

We can also use no parameters, meaning we don't take in any input, but still produce some output.

Example 2

Write a function that takes returns "Good morning"

```
def morning():
    return "Good Morning"

print(morning())
```

Exercise 1

a. Write a function to return the maximum of two numbers.
b. Can you do it with three?

Exercise 2

Write a function that checks if an integer is even or odd, and returns what kind of number it is in string form.

Hint: use the % operator to get the remainder when you divide two numbers. For example, **5 % 2** will be equal to **1** because **5 / 2 = 2 R 1**

Exercise 3

Write a function that takes in a year, and returns *True* if it is a leap year or *False* if it is not.

Void vs Value-Returning Functions

The functions we've been writing produce outputs. However, some functions are only meant to do tasks. For example, we might want to make a function that prints something to the command line. These types of functions are called **void** functions.

Example 4

Write a void function that prints the word "Hi!". Then, call it.

```
def hi():
    print("Hi!")

hi()
```

Exercise 4

Can we change our sum function to print to the command line instead of returning the sum?

Function Styling

While this isn't completely necessary, it's good to get into the habit of styling your functions to be neat and readable.

Rule 1: Have a descriptive name. Someone reading your code should know exactly what a function does

by looking at the name.

```
def sum(x, y):
    return x + y
```

1. By looking at the name of this function, we can tell that it is going to return the sum of two numbers.
2. Name is short and concise

What if my function name is too long?
If you can't describe your function with a concise name, chances are your function does too much, and can be broken into multiple functions. However, there are some cases where this is unavoidable. In these cases, comments are especially important.

Case: Axis-Aligned Bounding Box Queries
The axis aligned bounding box query is a collision algorithm used in games, particle research, etc. If we wanted to write a function that completed an axis-aligned bounding box query, it would have a seriously long name:

```
def axis_aligned_bounding_box_query(objects):
    # query code
```

Then, every time we called this function, we'd have to type out the entire name:

```
axis_aligned_bounding_box_query([])
axis_aligned_bounding_box_query([object1, object2, object3])
```

It's kind of annoying, right?

Instead, many implementations of axis-aligned bounding box queries use the acronym AABB

```
def AABBQuery(objects):
    """Completes an Axis-Aligned Bounding Box query on the given objects
    to detect collision

    Args
    ----
        Objects: list of objects
```

```
            Objects in space to detect collision

    Returns
    -------
        Objects: list of list of objects
            Objects colliding
    """
    # code

AABBQuery([])
```

We can now tell exactly what the AABBQuery does and don't have to type out a long function name every time we call it. Much nicer, right?

Rule 2: Explicit types. In Python, types are implicit, but they are still there. This can lead to confusion with some functions. To combat this, we should explicitly say which types our function is dealing with and what type it returns.

```
def sum(x: int, y: int) -> int:
    return x + y
```

For void functions, simply show that it returns None:

```
def prints_sum(x: int, y: int) -> None:
    print(x + y)
```