Set 12: Functions Part 1

Skill 12.1: Explain the purpose of functions

Skill 12.1 Concepts

Functions are an essential part of the Python programming language. We have already used many of Python's built-in functions in this class. For example, the *print()* function.

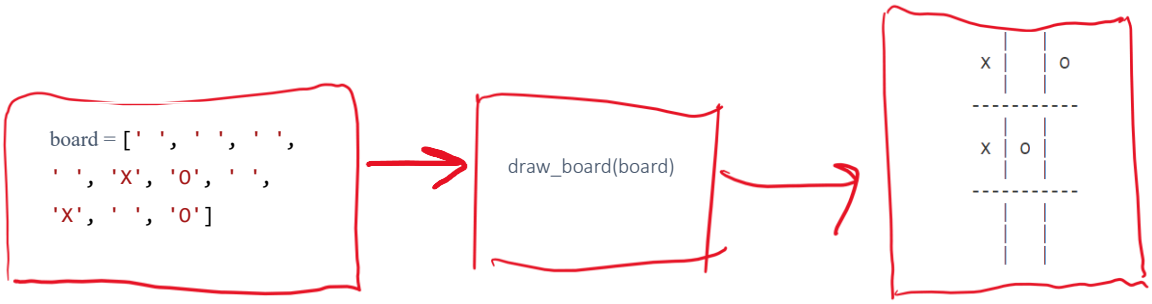
So far, all the Python we have been writing has been a series of commands. Consider the following code for drawing a tic-tac-toe board on the screen,

```
game_board = [""] + [" "]*9
print('   |   | ')
print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
print('   |   | ')
print('-----------')
print('   |   | ')
print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
print('   |   | ')
print('-----------')
print('   |   | ')
print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
```

That is a lot of code for such a simple task! For an actual tic-tac-toe program, we would have to repeat the above code after each player's turn.

In programming, we often use code to perform a specific task multiple times. Instead of rewriting the same code, we can group a block of code together and associate it with one task, then we can reuse that block of code whenever we need to perform the task again. We achieve this by creating a function. A function is a reusable block of code that groups together a sequence of statements to perform a specific task.

It is possible to implement a function without understanding the underlying complexity. For example, once the *draw_board* function is defined, it should be possible to draw many boards without understanding how it takes place. This process is illustrated below.



understanding how this works is NOT necessary once it is defined

Skill 12.1 Exercise 1

Skill 12.2: Define a function

Skill 12.2 Concepts

A function in Python consists of many parts. Below is an example of a function that prints "Hello!".

```
def say_hello():
    print("Hello!")
```

Below are the key components,

- The def keyword indicates the beginning of a function (also known as a function header). The function header is followed by a name in *snake_case* format that describes the task the function performs. It's best practice to give your functions a descriptive yet concise name.
- Following the function name is a pair of parenthesis ( ) that can hold input values known as parameters (more on parameters later in the lesson!). In this example function, we have no parameters.
- A colon : to mark the end of the function header.
- Lastly, we have one or more valid python statements that make up the function body. These must be indented to show that they are part of the function

When the code below is executed, nothing is printed. This is because the function has not been used.

Code	Output
`def say_hello():` `  print("Hello!")`	

Skill 12.2 Exercise 1

Skill 12.3: Call a function

Skill 12.3 Concepts

As we saw above, a function declaration binds a function identifier to a block of code.

However, a function declaration does not ask the code inside the function body to run, it just declares the existence of the function. The code inside a function body runs, or executes, only when the function is called. To call a function in your code, you type the function name followed by parentheses.

Code	Output

`def greet_world():`
`  print("Hello World!")`
 	
`greet_world()`
`greet_world()`
`greet_world()`
`greet_world()`
	Hello World!
	Hello World!
	Hello World!
	Hello World!

Calls greet_World 4 times

In the example above, the function call, *greet_world()*, executes the function body, or all of the statements indented below the colon. Once a function is defined, we can call it as many times as needed.

Skill 12.3 Exercise 1

Skill 12.4: Explore execution flow

Skill 12.4 Concepts

Consider the function below,

```
def class_welcome():
  print("Welcome to Python!")
  print("Get to work, Sid!")
```

The print statements all run together when *class_welcome()* is called. This is because they have the same base level of indentation (2 spaces).

In Python, the amount of whitespace in front of the statement tells the computer what is part of the function and what is not part of the function. We saw this convention with *if* statements and loops as well.

Consider the example below,

Code	Output

def class_welcome():
  print("Welcome to Python!")
  print("Get to work, Sid!")
print("Cody, sit!")
class_welcome()
 	
The code in the function is not executed until it is called.
	Cody, sit!
	Welcome to Python!
	Get to work, Sid!

In the example above, note that the execution of a program always begins on the first line. The code is then executed one line at a time from top to bottom. This is known as *execution flow* and is the order a program in python executes code.

Cody sit! was printed before the *print()* statements in the function *class_welcome().* Even though our function was defined before this print statement.

The *print()* statements in the function were printed after *Cody sit!* Because we didn't call our function until after this statement.

Skill 12.4 Exercise 1

Skill 12.5: Pass data to functions with parameters

Skill 12.5 Concepts

Let's return to our *class_welcome()* function one more time! We've modified it a bit by putting the print("Cody, sit!") command in the body of the function.

Code	Output

def class_welcome():
  print("Welcome to Python!")
  print("Get to work, Sid!")
  print("Cody, sit!")
class_welcome()
	Welcome to Python!
	Get to work, Sid!
	Cody, sit!

Our function does a really good job of telling Sid to "get to work" and Cody to "sit". But doesn't work if we want to tell someone else to get to work or sit. In order for us to make our function a bit more dynamic, we are going to use the concept of function parameters.

Function parameters allow our function to accept data as an input value. We list the parameters a function takes as input between the parentheses of a function ( ).

Here is the format of a function that defines a single parameter,

```
def my_function(single_parameter):
  # some code
```

In the context of our *class_welcome()* function, it would look like this,

```
def class_welcome(name1):
  print("Welcome to Python!")
  print("Get to work, " + name1)
  print("Cody, sit!")
```

In the above example, we define a single parameter called *name1* and apply it in our function body in the second print statement. We are telling our function it should expect some data passed in for *name1* that it can apply in the function body.

But how do we actually use this parameter? Our parameter of *name1* is used by passing in an *argument* to the function when we call it,

Code	Output

`def class_welcome(name1):`
`  print("Welcome to Python!")`
`  print("Get to work, " + name1)`
`  print("Cody, sit!")`
`class_welcome("Ayush")`
The argument "Ayush" is assigned to the parameter "name1"
	Welcome to Python!
	Get to work, Ayush
	Cody, sit!

Using a single parameter is useful but functions let us use as many parameters as we want! That way, we can pass in more than one input to our functions.

We can write a function that takes in more than one parameter by using commas,

```
def my_function(parameter1, parameter2, parameter3):
  # Some code
```

In the context of our *class_welcome()* function, it would look like this,

```
def class_welcome(name1, name2):
  print("Welcome to Python!")
  print("Get to work, " + name1)
  print(name2 + " sit!")
```

Our two parameters in this function are name1 and name2. In order to properly call our function, we need to pass argument values for both of them.

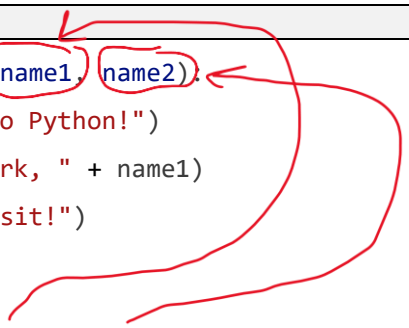
The ordering of your parameters is important as their position will map to the position of the arguments and will determine their assigned value in the function body.

Our function call could look like,

Code

```
def class_welcome(name1, name2):
  print("Welcome to Python!")
  print("Get to work, " + name1)
  print(name2 + " sit!")


class_welcome("Ayush", "Reid")
```



The argument “Ayush” is assigned to the parameter “name1”
The argument “Reid” is assigned to the parameter “name2”

Output

```
Welcome to Python!
Get to work, Ayush
Reid sit!
```

Skill 12.5 Exercise 1

Skill 12.6: Apply different argument types

Skill 12.6 Concepts

In Python, there are 3 different types of arguments we can give a function.

- Positional arguments: arguments that can be called by their position in the function definition.
- Keyword arguments: arguments that can be called by their name.
- Default arguments: arguments that are given default values.

Positional Arguments are arguments we have already been using! Their assignments depend on their positions in the function call. Let’s look at a function called *calculate_taxi_price()* that allows our users to see how much a taxi would cost to their destination 🚕.

```
def calculate_taxi_price(miles_to_travel, rate, discount):
  print(miles_to_travel * rate - discount )
```

In this function*, miles_to_travel* is *positioned* as the first parameter, *rate* is positioned as the second parameter, and *discount* is the third. When we call our function, the position of the arguments will be mapped to the positions defined in the function declaration.

Code

```
def calculate_taxi_price(miles_to_travel, rate, discount):
  print(miles_to_travel * rate - discount )


# 100 is miles_to_travel
# 10 is rate
# 5 is discount
calculate_taxi_price(100, 10, 5)
```

Output

```
995
```

Alternatively, we can use *Keyword Arguments* where we explicitly refer to what each argument is assigned to in the function call. Notice in the code example below that the arguments do not follow the same order as defined in the function declaration.

Code

```
def calculate_taxi_price(miles_to_travel, rate, discount):
  print(miles_to_travel * rate - discount )


calculate_taxi_price(rate=0.5, discount=10, miles_to_travel=100)
```

Output

```
40.0
```

Lastly, sometimes we want to give our function parameters default values. We can provide a default value to a parameter by using the assignment operator (=). This will happen in the function declaration rather than the function call.

Below is an example where the discount argument in our *calculate_taxi_price* function will always have a default value of 10,

```
def calculate_taxi_price(miles_to_travel, rate, discount = 10):
  print(miles_to_travel * rate - discount )
```

When using a default argument, we can either choose to call the function without providing a value for a discount (and thus our function will use the default value assigned) or overwrite the default argument by providing our own,

Code

```
def calculate_taxi_price(miles_to_travel, rate, discount = 10):
  print(miles_to_travel * rate - discount )


# Using the default value of 10 for discount.
calculate_taxi_price(100, 0.5)


# Overwriting the default value of 10 with 20
calculate_taxi_price(100, 0.5, 20)
```

Output

```
40.0
30.0
```

Skill 12.6 Exercise 1