Python
Lecture 5

Functions

Chapter 1

Introduction to Functions

In programming, as we start to write bigger and more complex programs, one thing we will start to notice is we will often have to repeat the same set of steps in many different places in our program.

Let's imagine we were building an application to help people plan trips! When using a trip planning application we can say a simple procedure could look like this:

```
1. Establish your origin and destination
2. Calculate the distance/route
3. Return the best route to the user
```

We will perform these three steps every time users have to travel between two points using our trip application. In our programs, we could rewrite the same procedures over and over (and over) for each time we want to travel, but there's a better way! Python gives us a useful concept called *functions*.

Functions are a convenient way to group our code into reusable blocks. A function contains a sequence of steps that can be performed repeatedly throughout a program without having to repeat the process of writing the same code again.

In this lesson, we are going to explore the idea of a function by slowly building out a Python program for our trip planning steps!

At the end of this lesson, you'll know how to:

- Write a function and return values from it.
- Allow functions to take custom input.
- Experiment with how functions access our other python code.

And much more!

Why Functions?

Let's come back to the trip planning application we just discussed in the previous exercise. The steps we talked about for our program were:

```
1. Establish an origin and destination
2. Calculate the distance/route
3. Return the best route
```

If we were to convert our steps into Python code, a very simple version that plans a trip between two popular New York tourist destinations might look like this:

```
print("Setting the Empire State Building as the starting point and
Times Square as our destination.")

print("Calculating the total distance between our points.")
```

```
print("The best route is by train and will take approximately 10 minutes.")
```
Anytime we want to go between these two points we would need to run these three print statements (for now we can assume the best route and time will stay the same).

If our program now had 100 new people trying to find the best directions between the Empire State Building and Times Square, we would need to run each of our three print statements 100 times!

Now, if you're thinking about using a loop here, your intuition would be totally right! Unfortunately, we won't be always traveling between the same two locations which means a loop won't be as effective when we want to customize a trip. We will address this in the upcoming sections!

For now, let's gain an appreciation for functions.

Defining a Function

A *function* consists of many parts, so let's first get familiar with its core - a function definition.

Here's an example of a function definition:

```
def function_name():
  # functions tasks go here
```
There are some key components we want to note here:

- 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 (where we have our python comment).

Notice we've indented our `# function tasks go here` comment. Like loops and conditionals, code inside a function must be indented to show that they are part of the function.

Here is an example of a function that greets a user for our trip planning application:

```
def class_welcome():
  print("Welcome to Jo's Python class!")
  print("The best programming class in the world.")
```

Note: Pasting this code into the editor and clicking ***Run*** will result in an empty output terminal. The `print()` statements within the function will not execute since our function hasn't been used. We will explore this further in the next exercise; for now, let's practice defining a function.

Example

```
def directions_to_timesSq():
  print("Walk 4 mins to 34th St Herald Square train station")
  print("Take the Northbound N, Q, R, or W train 1 stop")
  print("Get off the Times Square 42nd Street stop")
```

Calling a Function

Now that we've practiced defining a function, let's learn about calling a function to execute the code within its body.

The process of executing the code inside the body of a function is known as calling it (This is also known as "executing a function"). To call a function in Python, type out its name followed by parentheses `( )`.

Let's revisit our `directions_to_timesSq()` function :

```
def directions_to_timesSq():
  print("Walk 4 mins to 34th St Herald Square train station.")
  print("Take the Northbound N, Q, R, or W train 1 stop.")
  print("Get off the Times Square 42nd Street stop.")
```

To call our function, we must type out the function's name followed by a pair of parentheses and no indentation:

```
directions_to_timesSq()
```

Calling the function will execute the `print` statements within the body (from the top statement to the bottom statement) and result in the following output:

```
Walk 4 mins to 34th St Herald Square train station.
Take the Northbound N, Q, R, or W train 1 stop.
Get off the Times Square 42nd Street stop.
```

Note that you can only call a function *after* it has been defined in your code.

Now it's your turn to call a function!

Example

```
ef directions_to_timesSq():
  print("Walk 4 mins to 34th St Herald Square train station.")
  print("Take the Northbound N, Q, R, or W train 1 stop.")
  print("Get off the Times Square 42nd Street stop.")
  print("Take lots of pictures!")
directions_to_timesSq()
```

Whitespace & Execution Flow

Consider our welcome function for our trip planning application:

```
def class_welcome():
  print("Welcome to Jo's Python class!")
  print("The best programming class in the world.")
```

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 tells the computer what is part of a function and what is not part of that function.

If we wanted to write another statement outside of `class_welcome()`, we would have to unindent the new line:

```
def class_welcome():
  # Indented code is part of the function body
  print("Welcome to Jo's Python class!")
  print("The best programming class in the world.")

# Unindented code below is not part of the function body
print("Woah, this really is the best programming class in the world! He is a great tutor! *wink*")

class_welcome()
```

Our `class_welcome()` function steps will not print `Woah, this really is the best programming class in the world! He is a great tutor! *wink*` on our function call. The `print()` statement was unindented to show it was not a part of the function body but rather a separate statement.

We would see the following output from this program:

```
Woah, this really is the best programming class in the world! He is
a great tutor! *wink*
Welcome to Jo’s Python class!
The best programming class in the world.
```

Lastly, 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.

`Woah, this really is the best programming class in the world! He is a great tutor! *wink*` was printed before the `print()` statements from the function `class_welcome()`.

Even though our function was defined before our lone `print()` statement, we didn’t call our function until after.

Let’s play around with indentation and the flow of execution!

Example

```
print("Checking the weather for you!")

def weather_check():
  print("Looks great outside! Enjoy your trip.")

print("False Alarm, the weather changed! There is a thunderstorm approaching.
Cancel your plans and stay inside.")

weather_check()
```

Parameters & Arguments

Let’s return to our `class_welcome()` function one more time! Let’s modify our function to give a welcome that is a bit more detailed.

```
def class_welcome():
  print("Welcome to Jo’s Python class!")
  print("The best programming class in the world.")

class_welcome()
```

This will output:

```
Welcome to Jo’s Python class!
The best programming class in the world.
```

Our function does a really good job of welcoming anyone who is traveling to Times Square but a really poor job if they are going anywhere else. 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 a function that defines a single parameter:

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

In the context of our `class_welcome()` function, it would like this:

```
def class_welcome(destination):
  print("Welcome to class!")
  print("Looks like you're going for " + destination + " today.")
```

In the above example, we define a single parameter called `destination` 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 destination that it can apply to any statements in the function body.

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

```
class_welcome("Python class")
```

This would output:

```
Welcome to class!
Looks like you're going to Python class today.
```

To summarize, here is a quick breakdown of the distinction between a parameter and an argument:

- The parameter is the name defined in the parenthesis of the function and can be used in the function body.
- The argument is the data that is passed in when we call the function and assigned to the parameter name.

  Let's write a function with parameters and call the function with an argument to see it all in action!

Example

```
def generate_trip_instructions(location):
  print("Looks like you are planning a trip to visit " + location)
  print("You can use the public subway system to get to " + location)
```

```
generate_trip_instructions("Grand Central Station")
```

Multiple Parameters

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
```

When we call our function, we will need to provide arguments for each of the parameters we assigned in our function definition.

```
# Calling my_function
my_function(argument1, argument2)
```

For example take our trip application's `trip_welcome()` function that has two parameters:

```
def trip_welcome(origin, destination):
  print("Welcome to Tripcademy")
  print("Looks like you are traveling from " + origin)
  print("And you are heading to " + destination)
```

Our two parameters in this function are `origin` and `destination`. 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 (more on this in the next exercise!).

Our function call could look like:

```
trip_welcome("Prospect Park", "Atlantic Terminal")
```

In this call, the argument value of `"Prospect Park"` is assigned to be the `origin` parameter, and the argument value of`"Atlantic Terminal"` is assigned to the `destination` parameter.

The output would be:

```
Welcome to Tripcademy
Looks like you are traveling from Prospect Park
And you are heading to Atlantic Terminal
```

Let's practice writing and calling a multiple parameter function!

Example

```
def calculate_expenses(plane_ticket_price, car_rental_rate, hotel_rate, trip_t
ime):
  car_rental_total = car_rental_rate * trip_time
  hotel_total = hotel_rate * trip_time - 10
  print(car_rental_total + hotel_total + plane_ticket_price)

calculate_expenses(200, 100, 100, 5)
```

Types of Arguments

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.

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

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.

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

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

Here 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:

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

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

Let's practice using these different types of arguments!

Example

```
def trip_planner(first_destination, second_destination, final_destination="Lag
os"):
  print("Here is what your trip will look like!")
  print("First, we will stop in " + first_destination + " then " + second_dest
ination + ", and lastly " + final_destination)

trip_planner("Port Harcourt", "Calabar")
```

Variable Access

As we expand our programs with more functions, we might start to ponder, where exactly do we have access to our variables? To examine this, let's revisit a modified version of the first function we built out together:

```
def trip_welcome(destination):
  print(" Looks like you're going to the " + destination + " today.
")
```

What if we wanted to access the variable `destination` outside of the function? Could we use it? Take a second to think about what the following program will output, then check the result below!

```
def trip_welcome(destination):
  print(" Looks like you're going to the " + destination + " today.
")
```

```
print(destination)
```
Output Results

```
NameError: name 'destination' is not defined
```
If we try to run this code, we will get a NameError, telling us that `destination` is not defined. The variable `destination` has only been defined inside the space of a function, so it does not exist outside the function.

We call the part of a program where `destination` can be accessed its *scope*.
The *scope* of `destination` is only inside the `trip_welcome()`.

Take a look at another example:

```
budget = 1000

# Here we are using a default value for our parameter of
`destination`
def trip_welcome(destination="California"):
    print(" Looks like you're going to " + destination)
    print(" Your budget for this trip is " + str(budget))

print(budget)
trip_welcome()
```
Our output would be:

```
1000
Looks like you're going to California
Your budget for this trip is 1000
```
Here we are able to access the `budget` both inside the `trip_welcome` function as well as our `print()` statement. If a variable lives outside of any function it can be accessed anywhere in the file.

We will be exploring the concept of *scope* more after this entire lesson but for now, let's play around!

Note: Working with multiple functions can be a bit overwhelming at first. Don't hesitate to use hints or even look at the solution code if you get stuck.

Example

```
# This function will print a hardcoded count of how many locations we have.
favorite_locations = "Paris, Norway, Iceland"
def print_count_locations():
    print("There are 3 locations")

# This function will print the favorite locations
def show_favorite_locations():
  print("Your favorite locations are: " + favorite_locations)

print_count_locations()
show_favorite_locations()
```

Returns

At this point, our functions have been using `print()` to help us visualize the output in our interpreter. Functions can also *return* a value to the program so that this value can be modified or used later. We use the Python keyword `return` to do this.

Here's an example of a program that will `return` a converted currency for a given location a user may want to visit in our trip planner application.

```
def calculate_exchange_usd(us_dollars, exchange_rate):
  return us_dollars * exchange_rate

new_zealand_exchange = calculate_exchange_usd(100, 1.4)

print("100 dollars in US currency would give you "
+ str(new_zealand_exchange) + " New Zealand dollars")
```

This would output:

```
100 dollars in US currency would give you 140 New Zealand dollars
```

Saving our values returned from a function like we did with `new_zealand_exchange` allows us to reuse the value (in the form of a variable) throughout the rest of the program.

When there is a result from a function that is stored in a variable, it is called a *returned function value*.

Let's try to `return` some data in the exercises!

Example

```
current_budget = 3500.75

def print_remaining_budget(budget):
  print("Your remaining budget is: $" + str(budget))

print_remaining_budget(current_budget)

# Write your code below:
def deduct_expense(budget, expense):
  return budget - expense

shirt_expense = 9
new_budget_after_shirt = deduct_expense(current_budget, shirt_expense)

print_remaining_budget(new_budget_after_shirt)
```

Multiple Returns

Sometimes we may want to return more than one value from a function. We can return several values by separating them with a comma. Take a look at this example of a function that allows users in our travel application to check the upcoming week's weather (starting on Monday):

```
weather_data = ['Sunny', 'Sunny', 'Cloudy', 'Raining', 'Snowing']

def threeday_weather_report(weather):
  first_day = " Tomorrow the weather will be " + weather[0]
  second_day = " The following day it will be " + weather[1]
  third_day = " Two days from now it will be " + weather[2]
  return first_day, second_day, third_day
```

This function takes in a set of data in the form of a list for the upcoming week's weather. We can get our returned function values by assigning them to variables when we call the function:

```
monday, tuesday, wednesday = threeday_weather_report(weather_data)

print(monday)
print(tuesday)
print(wednesday)
```

This will print:

```
Tomorrow the weather will be Sunny
The following day it will be Sunny
Two days from now it will be Cloudy
```

Let's practice using multiple returns by `return`ing to our previous code example.

Example

```
def top_tourist_locations_naija():
  first = "Lagos"
  second = "Calabar"
  third = "Port Harcourt"

  return first, second, third

most_popular1, most_popular2, most_popular3 = top_tourist_locations_naija()

print(most_popular1)
print(most_popular2)
print(most_popular3)
```

Class Practice

1. Write a function to determine age to visit a website

```
def website_welcome(name, age):
    if age >= 18:
        print("Welcome " + name + "!")
        print("You are very old: " + str(age))

    else:
        print("You are not welcome, you are too young!")
        years_left = 18 - age
        print("Wait another " + str(years_left) + " years")
Website_welcome("Jo", 16)
```

2. Write a function to print a list of even numbers

```
def list_numbers(n):
    i = 0
    new_list = []
    while i <= n:
        if i % 2 == 0:
            new_list.append(i)
        i += 1
    return new_list

x = list_numbers(5)
print(x)
```

Python Lecture 5 Project

1. Write a function to check if a number ‘n’ is in a list ‘lst’.

```
def is_in_list(n, lst):
    # write code here
    return
```

Here is what the output should look like:

```
> is_in_list(5, [1, 2, 3, 4])
False

> is_in_list(5, [1, 5, 6])
True
```

2. Write a function to reverse a list. Do not use the reverse method. Use your knowledge of loops to reverse the list.

```
def reverse_list(my_list):
    # write code here
    return
```

Here is what the output should look like:

```
> reverse_list([3, 5, 7, 8, 1, 4, 5])
[5, 4, 1, 8, 7, 5, 3]

> reverse_list([5, 10, 6, 14, 100, 15, 400])
[400, 15, 100, 14, 6, 10, 5]
```

3. Write a function to determine how much of tips you received while working at Jo’s 5-star restaurant and say “thank you” for the people who tipped you 500 or more.

```
def receive_tips(tips):
    # write code here
    return
```

Here is what the output should look like:

```
> receive_tips([1000, 500, 100, 400, 6000, 1000, 400])
Thanks for ₦1000
Thanks for ₦500
Thanks for ₦6000
Thanks for ₦1000
9400

> receive_tips([300, 100, 700, 400, 500, 1000])
Thanks for ₦700
Thanks for ₦500
```

```
Thanks for ₦1000
3000
```