Intermediate python - Functions - 1

One should look for what is and not what he thinks should be. (Albert Einstein)

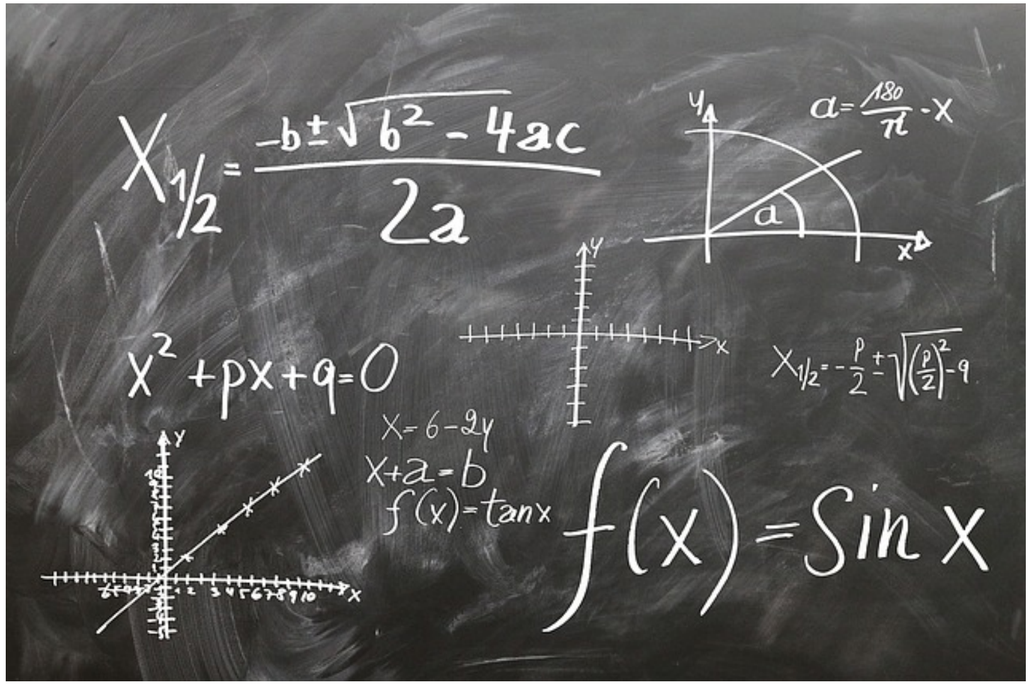
Functions: Topic introduction

In this part of the course, we will cover the following concepts:

- Function definition and use cases
- Function implementation in Python

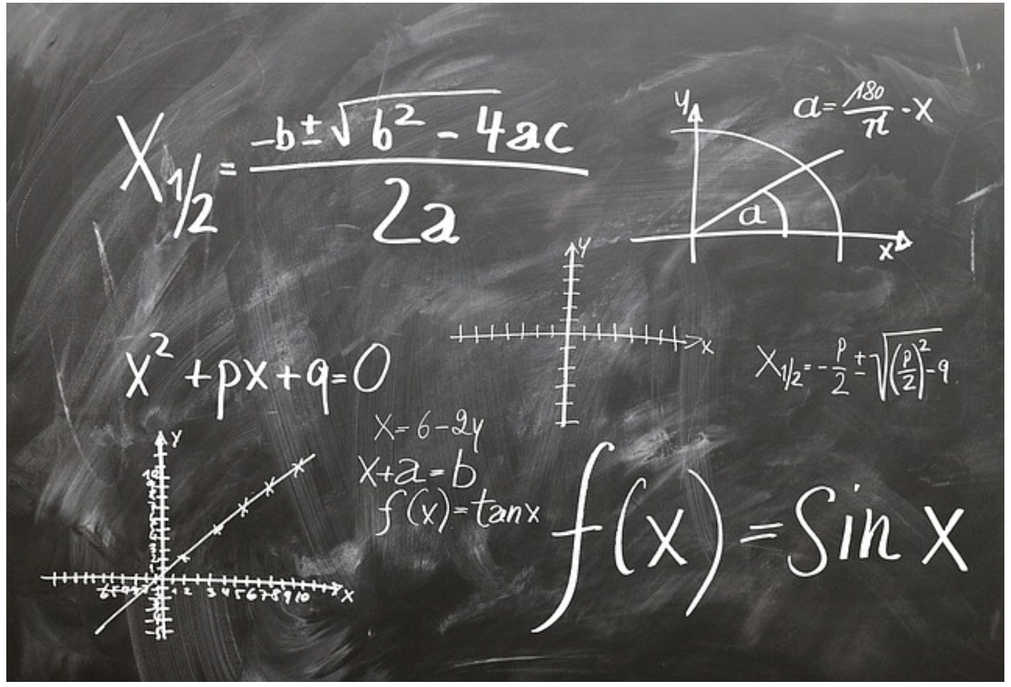
Chat question

- What do you see in the image below? What does it make you think of?
- Share your thoughts in the chat or out loud



Chat question

- When you hear the word **function**, you might imagine mathematical symbols
- What is a function in Python programming?



Module completion checklist

Objective	Complete
Identify the use cases and types of functions in Python	
Implement functions in Python	

Types of functions in Python

There are **two** basic types of functions in Python:

- Built-in functions:
  - These are part of the Python language that have already been built and are available for end-users
  - Example: `print()`, `len()`, `abs()`
- User-defined functions:
  - These are functions that a user can create according to a specific use case or functionality
- We will now see how we can write our own functions in Python

How functions apply to programming

- In programming, functions serve a very similar purpose as in mathematics
- They help us **abstract** from operations on actual numbers and let us **define** those operations through a set of variables
- The **logic** of operations does not change, but the **value** of variables changes
- This makes it easy to reuse such blocks and only pass new values to those variables

Module completion checklist

Objective	Complete
Identify the use cases and types of functions in Python	✔
Implement functions in Python	

User-defined functions in Python

- Functions can:
  - Have `0` or more arguments
  - Perform `1` or more actions
  - Return some value(s) or no value(s)
- The simplest possible function will have no arguments and will not return any value

- We can **define** a function, using a `def` construct that includes:
  - a `name` for function
  - an `action` or `actions` to perform when the function is triggered

```
# Define a function that prints the value of `Pi`.
def PrintPi():      #<- function name
    print(3.14)     #<- action to perform
```

- To "trigger" the function, we can **call** it by its name

```
PrintPi()
```

```
3.14
```

Functions in Python with one or more arguments

- Most of the time, you will create or use a function that has at least `1` **argument**
- When we need to define a function with argument(s), the `def` construct includes:
  - a `name` for function
  - at least 1 argument in `parentheses`
  - an `action` or `actions` to perform when the function is triggered

```
# Define a function that prints the value of `Pi`
# and takes a number of decimal points to which we want to round the number.
def PrintPi(num_decimals):                #<- function name + argument(s)
    pi = 3.14159265359                    #<- action to perform
    rounded_pi = round(pi, num_decimals) #<- action to perform
    print(rounded_pi)                     #<- action to perform
```

```
# Print value of pi rounded to 4 decimal points.
PrintPi(4)
```

```
3.1416
```

Functions in Python that return a value

- Functions can also **return value(s)** in addition to some `actions` they perform
- When we need to define a function with or without argument(s) that returns value(s), we use a `def` construct that includes
  - A `name` for function
  - At least `0` or more `arguments` in parentheses
  - An `action` or `actions` to perform when the function is triggered
  - A `return` value or values

```
# Define a function that prints the value of `Pi`,
# takes a number of decimal points to which we want to round the number,
# and returns the value back to us.
def GetPi(num_decimals):                    #<- function name + argument(s)
    pi = 3.14159265359                      #<- action to perform
    rounded_pi = round(pi, num_decimals)    #<- action to perform
    return rounded_pi                       #<- value to return
```

Functions in Python that return a value (cont'd)

- Let's print the value that `GetPi()` returns

```
# Return a value of pi rounded to 4 decimal points.
print(GetPi(4))
```

```
3.1416
```

- A function that returns a value can also be assigned to a **variable**
- Now the value returned and assigned to a variable can be **re-used** throughout the code

```
# Return a value of pi rounded to 4 decimal points.
# Assign it to a variable.
pi_4 = GetPi(4)
print(pi_4)
```

```
3.1416
```

Functions in Python: general structure

- Here is a general outline of a function that takes arguments and returns something in Python
- The arguments, actions, and returned values are all up to you to define based on what you are trying to achieve

```
def FunctionName(argument1, argument2, ...):
    action1
    action2
    ...
    return something
```

Functions in Python: MakeFullName

- Let's define a function that concatenates the *first name* and *last name* into a **full name**

```
# Define a function that concatenates
# first and last names.
def MakeFullName(first_name, last_name):
    full_name = first_name + ' ' + last_name
    return full_name
```

- Here are the components of the function definition broken down:
  - `def` defines a function
  - `MakeFullName` is the name of our function
  - The two arguments it takes are specified in the parentheses: `first_name` and `last_name`
  - The `return` statement controls what gets returned when someone uses the function (i.e. `full_name`)

Functions in Python: calling a function

- After we have defined a function, we need to test it
- Running a function in programming is usually known as **calling** a function
- Let's call `MakeFullName()` - to do that, we need to call the function by its name and pass any necessary values we want for the arguments of the function

- Let's print out the function results

```
# Call the function.
print(MakeFullName("Harry", "Potter"))
```

```
Harry Potter
```

- If the function returns a value (like a string with full name in this case), we can assign the output of that function to a variable

```
# Call the function and save
# output to variable.
output_name = MakeFullName("Harry", "Potter")
print(output_name)
```

```
Harry Potter
```

Functions in Python: EvaluateCarPrice

- Let's define another function that evaluates the price of a car and gives us the action to do based on the price
- Here are the components of the function definition broken down:
  - `EvaluateCarPrice` is the name of our function
  - The argument it takes is specified in the parentheses: `price`
  - The `return` statement controls what gets returned when someone uses the function (i.e. `action`)

```
def EvaluateCarPrice(price):
    if price > 38000:
        action = "Leave the dealership immediately, this is a rip off!"
    elif price > 22000 and price <= 38000 :
        action = "Take the car and go celebrate, you can afford it!"
    else:
        action = "Leave the dealership immediately, this is a scam!"
    return action
```

Functions in Python: calling a function

```
# Let's set the price of a car to $45,000.
price = 45000
action1 = EvaluateCarPrice(price)
print(action1)
```

```
Leave the dealership immediately, this is a rip off!
```

```
# Let's set the price of a car to $32,000.
price = 32000
action2 = EvaluateCarPrice(price)
print(action2)
```

```
Take the car and go celebrate, you can afford it!
```

```
# Let's set the price of a car to $5,000.
price = 5000
action3 = EvaluateCarPrice(price)
print(action3)
```

```
Leave the dealership immediately, this is a scam!
```

Functions that return two or more values

- Remember that one function can return multiple values
- To demonstrate, let's define another function that evaluates monthly sales data by profit
- Our data will take the form of a dictionary with `month: amount` key-value pairs
  - `MostProfitableMonth` is the name of our function
  - The argument it takes is a dictionary `sales_data`
  - It returns a **tuple** of the two values: `biggest_month` and `biggest_amt` for the month with the most profit

```
def MostProfitableMonth(sales_data):

    biggest_amt = 0
    biggest_month = None

    for month, amt in sales_data.items():
        if amt >= biggest_amt:
            biggest_amt = amt
            biggest_month = month

    return (biggest_month, biggest_amt)
```

Question

What is the data type of `sales_data.items()`?

a. tuple

b. list

c. array

Functions that return two or more values (cont'd)

```
# Let's define a dictionary with sales data.
year_sales = {'January': 1045, 'February': 1008, 'March': 1025,
                    'April': 1080, 'May': 1100, 'June': 1050, 'July': 1050,
                    'August': 950, 'September': 1010, 'October': 1500,
                    'November': 1450, 'December': 1380}
```

- To **save the output** of a function that returns 2 or more values, assign the function call output to the same number of variables as there are outputs in the function in the same order

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales)
print("Best month:", best_sales_month)
```

```
Best month: October
```

```
print("Best amount:", best_sales_amt)
```

```
Best amount: 1500
```

Passing a list as an argument

- We can also create a function to operate on a list

```
# Define a function that returns the length of all character strings in a list.
def calculate_lengths(list_of_strings):
  # Define an empty list to store the result.
  lengths = []
  # Use a for loop to access items in the input list.
  for item in list_of_strings:
    lengths.append(len(item))

  return lengths
```

```
result = calculate_lengths(["Monday", "Tuesday", "Saturday"])
print(result)
```

```
[6, 7, 8]
```

Functions that have default arguments

- Let's modify another `MostProfitableMonth` and add another argument
  - This argument will have a **default** value
  - Given a default value, we are not required to provide the other argument
- Let's set the second argument to be `verbose`, which in programming usually signals to print a message with a returned value
- Let's set the default value to `True`

```
def MostProfitableMonth(sales_data, verbose = True):

    biggest_amt = 0
    biggest_month = None

    for month, amt in sales_data.items():
        if amt >= biggest_amt:
            biggest_amt = amt
            biggest_month = month

    if verbose:
        print('The best sales month was', biggest_month, 'with amount sold equal to', biggest_amt)

    return (biggest_month, biggest_amt)
```

Functions that have default arguments (cont'd)

- Let's call the function `MostProfitableMonth()` and provide only the first (required) argument

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales)
```

```
The best sales month was October with amount sold equal to 1500
```

- Let's call the function `MostProfitableMonth()` and set the `verbose` argument to `False`

```
# Assign output of function to variables in correct order.
best_sales_month, best_sales_amt = MostProfitableMonth(year_sales, False)
```

- The output is still the same, but the "verbose" message is no longer printed, so if we need to see it, we would have to do it manually

```
print("Best month:", best_sales_month, "Best amount:", best_sales_amt)
```

```
Best month: October Best amount: 1500
```

Anonymous functions: lambda

- **Anonymous** functions, also known as **lambda** functions, are small **nameless** functions
- They work the same way as conventional functions, but are used as shorthand within other pieces of code
- They are often used as a part or as an argument in other functions

Anonymous functions: syntax

- To write an anonymous function in Python, we use the `lambda` command followed by an **argument** and a `colon :`
- The expression that follows represents an **action** we would like to perform using those arguments

```
print((lambda v: v + " the 5th of November!")("Remember"))
```

```
Remember the 5th of November!
```

- As we can see above, the lambda function does not need to be tied to any name and can be directly called on the argument `"Remember"`

Anonymous functions with multiple arguments

- We can also assign a lambda function to a name for reusability

```
remember = lambda v: v + " the 5th of November!"
print(remember("Remember"))
```

```
Remember the 5th of November!
```

- Anonymous functions can take several arguments separated by a comma

```
y = lambda a, b: a + b
print(y(765, -987))
```

```
-222
```

Function within another function

- In order to make different parts of our code interact with each other, we can also call a function from within another function

```
# Define a function to check if a number is even
or odd.
def even_odd(num):
  if num%2 == 0:            #<- % is the Modulo
operator.
    result = "Even"
  else:
    result = "Odd"
  return result
```

```
# Define a function to print if each element of a
list is even or odd.
def even_odd_list(numbers):
  for num in numbers:
    result = even_odd(num)    #<- Call function on
num an assign its result
    print(num, ": ", result)
```

```
even_odd_list([22,41,16,13])
```

```
22 :  Even
41 :  Odd
16 :  Even
13 :  Odd
```

Knowledge check

Link: ***https://forms.gle/EBJAx1EUsGJ2xMNF9***

Exercise

You are now ready to try Tasks 1-5 in the Exercise for this topic

Module completion checklist

Objective	Complete
Identify the use cases and types of functions in Python	✔
Implement functions in Python	✔

Summary

In this module, we:

- Defined functions and examined their use cases
- Identified and implemented different types of functions

Congratulations on completing this module!