**WEEK 01 – INTRO TO PYTHON**

- **Python** is a high-level, interpreted programming language that is widely used for web development, data analysis, artificial intelligence, and scientific computing. It is a versatile language that is easy to learn and can be used for a wide range of tasks.

- **Guido van Rossum** is a Dutch programmer and the creator of the Python programming language. He started working on Python in **1989**, and the first version was released in **1991**.

- Python, which is named after the **Monty Python comedy group.**

- It is known for its simplicity, readability and easy to learn.

**APPLICATION FOR PYTHON**

**Web development**: Python has a number of popular web frameworks such as Flask and Django that make it easy to build web applications. It can also be used to write scripts to automate web scraping and data extraction.

**Scientific computing and data analysis**: Python has a number of powerful libraries for data analysis and visualization, such as NumPy, pandas, and Matplotlib. These libraries make it easy to work with large sets of data and perform complex calculations.

**Machine learning and artificial intelligence**: Python has become a popular choice for developing machine learning and AI models, thanks to libraries such as scikit-learn, TensorFlow, and Keras.

**Automation and scripting**: Python is often used for automating repetitive tasks and writing scripts to automate workflows. It can be used to control and automate software, such as Blender, GIMP, and Autodesk Maya, and even hardware like Raspberry Pi, Arduino.

**Game development**: Python is also used to develop games using libraries like Pygame and PyOpenGL.

**Networking**: Python has a number of libraries for working with network protocols, such as Scapy and Twisted.

**PYTHON ROAD MAP**

- **Variables:** Used to store values in a program. In Python, you can use any name as a variable name, but it should not start with a number or special characters.

**Data Types:** Python supports various data types such as integers, floating-point numbers, strings, and Boolean values.

**Operators:** Python supports various operators such as arithmetic, comparison, and logical operators

**Conditional Statements:** Used to control the flow of a program based on certain conditions. Python supports if, elif, and else statements.

**Loops:** Used to execute a block of code multiple times. Python supports for and while loops.

**Functions:** Used to group a set of related code together and can be reused throughout the program. Python supports the use of functions with the 'def' keyword.

**List and Tuples:** These are both built-in data types that allow you to store multiple values in a single variable. Lists are enclosed in square brackets [] and are **mutable**, meaning you can add, remove or change values within the list. Tuples are enclosed in parentheses () and are **immutable**, meaning you can't change the values once they are assigned.

**Dictionaries:** Like lists and tuples, but instead of using indexing to access the values, you use keys. Dictionaries are enclosed in curly braces {} and keys are unique within the dictionary.

**Modules and Libraries:** Python has many built-in modules and libraries that provide additional functionality. For example, the math module provides mathematical functions, while the `os` module provides functions for interacting with the operating system. You can also install third-party libraries using package managers like pip.

**Exception Handling:** a way to handle errors that occur during the execution of a program. Python uses try-except blocks to handle exceptions

**Object-Oriented Programming (OOP):** Python supports OOP, which is a programming paradigm that focuses on creating objects, which have methods and properties, and are instances of classes. This allows for code reuse and organization.

**Decorators:** are a way to add functionality to existing functions without modifying the original code. They are defined using the @ symbol followed by the name of the function that will be used as a decorator. They can be used to add logging, timing, or other types of functionality to functions.

**Context Manager:** An object that defines the methods \_\_enter\_\_() and \_\_exit\_\_() that are used to set up and tear down resources.

**WEEK 02 – Python Data Structures/Collection**

**Python List**

* Used to store multiple items in a single variable.
* One of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
* Created using square brackets **[]**
* It is also possible to use the **list()** constructor when creating a new list.
* List can include different data types
* A list is **ordered**, **changeable**, and **allows duplicate** members.

**List Items**

* Ordered, changeable, and allow duplicate values.
* Indexed, the first item has index [0], the second item has index [1] etc.

**Ordered List**

* The items **have a defined order**, and that **order will not change**.
* **If you add** new items to a list, the **new items will be placed at the end of the list**.

**Changeable List**

* Meaning that **we can change, add, and remove items in a list** after it has been created.

**List Duplicate**

* Since lists are indexed, **lists can have items with the same value**

**List Length**

* To determine how many items a list has, use the **len()** function

**Python Tuples**

* Used to store multiple items in a single variable.
* Used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
* A collection which is **ordered and unchangeable.**
* Written with **round brackets ().**
* To create a tuple with only one item, you must add a **comma** **after** the **item**, otherwise Python will not recognize it as a tuple.
* Tuple items can be of any data type
* A tuple can also contain different data types
* It is also possible to use the **tuple()** constructor to make a tuple.
* The **remove()** method removes the specified **item**.

**Tuple Items**

* **Ordered, unchangeable, but allow duplicate values.**
* Indexed, the first item has index [0], the second item has index [1] etc.

**Tuple Length**

* To determine how many items a tuple has, use the **len()** function

**Python Set**

* Used to store multiple items in a single variable.
* Used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
* A collection which is **unordered, unchangeable\*, and unindexed.**
* Sets are written with **curly brackets {}.**
* **Duplicates Not Allowed in Sets**
* **Set items are** **unchangeable, but you can remove items and add new items.**
* Once a set is created, you cannot change its items, but you can remove items and add new items.
* It is also possible to use the **set()** constructor to make a set.

**Set Items**

* A set can contain different data types

**Set Length**

* To determine how many items a set has, use the **len()** function

**Python Dictionaries**

* Used to store data values in key:value pairs.
* A collection which is ordered, changeable and do not allow duplicates.
* Written with curly **brackets {}** and have **keys and values.**
* This is the same as the JSON format.
* **As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.**
* If dictionaries are ordered, it means that the items have a defined order, and that order will not change.
* If Unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
* The values in dictionary items can be of any data type.

**Dictionary Example**

* Print the "brand" value of the dictionary
* An item in the dictionary can be selected by having the dictionary with **square brackets []** with a specific item inside quotations **“item”.**

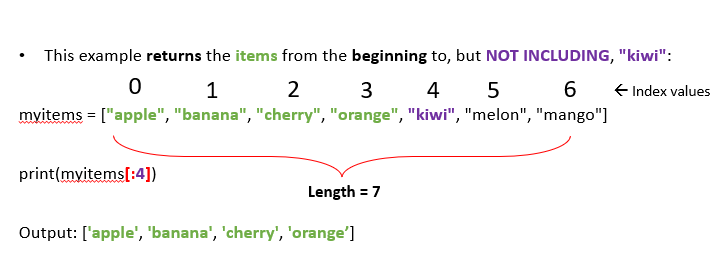
**Dictionary Length**

* To determine how many items a dictionary has, use the **len()** function

**Data Manipulation with Python**

**Accessing List and Tuples**

* List and Tuple items are indexed, and you can access them by referring to the index number inside square brackets **[]**
* You can specify a range of indexes by specifying where to start and where to end the range.
* When specifying a range, the return value will be a new list with the specified items.
* The search will start at **index 2 (included)** and end at **index 5 (not included)**.
* Remember that the first item has index 0.



* The items within the list and tuples can be accessed using the Negative Indexing method.
* You can also specify negative indexes if you want to start the search from the end of the list
* Checking **items** within the list or tuple for sanity checking.
* Using an **if** statement is a practical method.
* To determine if a specified item is present **in** a list use the in keyword:

**Accessing Set Items**

* You **CANNOT ACCESS** items in a **SET** by **referring to an index or a key.**
* But **you can loop** through the set items using a for loop or ask if a specified value is present in a set, by using the in keyword.

**Accessing Dictionary Items**

* You can access the items of a dictionary by referring to its **key name**, inside square brackets **[]**:
* Another method to access the key values can be done by using the **get()** method or function will return a list of all the keys in the dictionary.
* To access the keys, using the **keys()** method or function will return a list of all the keys in the dictionary.

**Changing Items in a List**

* To **change** the value of a **specific item**, refer to the **index number** using square brackets []:
* Change the **second value** by replacing it with **two** **new values**

**Changing/Updating items in a Tuple**

* Once a tuple is created, you cannot change its items! However, there is work around. Therefore, we refer to it as **UPDATING** rather than **CHANGING.**
* Once a tuple is created, you **cannot change its values**.
* Tuples are **unchangeable**, or **immutable** as it also is called.
* But there is a workaround. You can **convert** the tuple into a **list**, **change** the **list**, and **convert** the **list** back into a tuple.

**Changing Items in a Dictionary**

* You can **change** the value of a specific item by **referring** to its **key** **name**:
* **Updating** an **item** to the dictionary is done by using a **new** **index key** and using the .update() method

**Adding Items in a List**

* To add an item to the end of the list, use the **append()** method

**Adding Items in a Tuples**

* tuples are **unchangeable**, or **immutable** as it also is called

**Adding Items in a Dictionary**

* **Adding** an **item** to the dictionary is done by using a **new** **index key** and assigning a **value** to it:

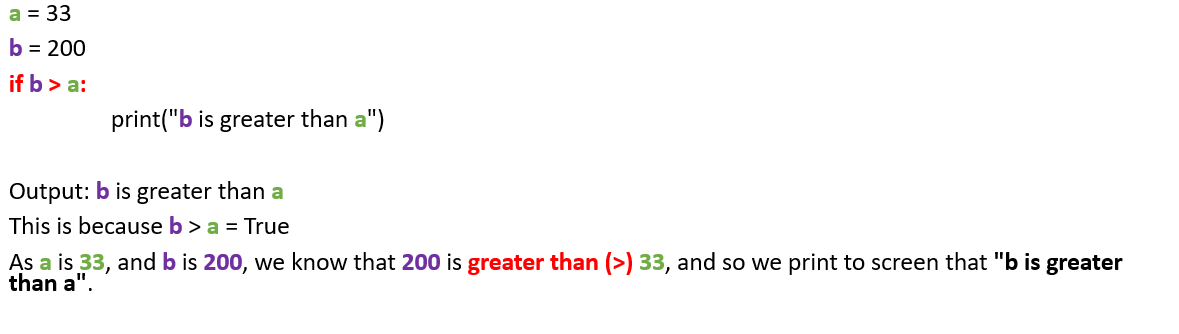
**Adding Items in a Set**

* Once a set is created, you **cannot** **change** its **items**, **BUT** you **CAN** **ADD** **new** **items**.
* To add one item to a set, use the **add()** method.

**WEEK 03 - Conditions and Loops with Python**

**If Else**

* In this example we use two variables, **a** and **b**, which are used as part of the **if statement** to test whether **b** is **greater than (>) a**.

****

**Elif**

* The **elif** keyword is pythons' way of saying "if the previous conditions were not true, then try this condition".

**Else**

* The **else** keyword catches anything which isn't caught by the preceding conditions.

**Short Hand If**

* if you have only one statement to execute, you can put it on the same line as the if statement.

**Short Hand If ... Else**

* If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

**And**

* The and keyword is a logical operator, and is used to combine conditional statements:

**Or**

* The or keyword is a logical operator, and is used to combine conditional statements:

**Nested If**

* You can have if statements inside if statements, this is called nested if statements.

**The Pass Statement**

* if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

**The Break Statement**

* With the break statement we can stop the loop even if the while condition is true

**The Continue Statement**

* With the continue statement we can stop the current iteration, and continue with the next:

**The Break Statement**

* With the break statement we can stop the loop before it has looped through all the items:

**The Else Statement**

* With the else statement we can run a block of code once when the condition no longer is true:

**For loop**

* Use for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
* Less like the for keyword in other programming languages and works more like an iterator method as found in other object-orientated programming languages.
* it can execute a set of statements, once for each item in a list, tuple, set etc.

**While loop**

* With the while loop we can execute a set of statements as long as a condition is true.

**Iterables**

* An object which can be looped over or iterated over with the help of a for loop.
* Familiar examples of iterables include lists, tuples, and strings - any such sequence data types can be iterated over in a for-loop.
* Such include a string.

**The range() Function**

* To loop through a set of code a specified number of times, we can use the range() function,
* The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

**Python Zip**

* The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
* If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

**Python Enumerate**

* The enumerate() function takes a collection (e.g. a tuple) and returns it as an enumerate object.
* The enumerate() function adds a counter as the key of the enumerate object.

**Week 04 – Python Functions**

**User-Defined Functions**

* A way to organize and reuse code.
* Creates a block of code once and use it multiple times in your program.
* Defined using the "def" keyword, and they can take in parameters and return values.

**Function Scope**

* Another important concept related to functions is the idea of a "scope". Variables defined inside a function are only accessible within that function and are not available outside of it.

**Arguments**

* Values passed to a function when it is called. These values are then assigned to the function's parameters, which are the names used within the function to refer to the received values.
  + Types of Arguments:
    - Positional Arguments
    - Keyword Arguments
    - Variable-length Arguments

**Positional Arguments**

* the function "my\_function" takes three positional arguments "a", "b", and "c". When we call the function with the values 1, 2, and 3, the function adds them together and returns the result, which is 6.

**Key-word Arguments**

* the function "my\_function" takes three keyword arguments "a", "b", and "c". When we call the function with the keyword arguments c=1, a=2, and b=3, the function adds them together and returns the result, which is 6.

**Default Value Arguments**

* In Python, you can define a function with default values for its arguments. This means that if the user doesn't supply a value for a particular argument when calling the function, it will automatically be set to the default value specified in the function definition.

**Variable Length Arguments**

* the function "my\_function" takes a variable-length tuple of arguments "\*args". When we call the function with the values 1, 2, 3, 4, and 5, the function adds them together using the built-in "sum" function and returns the result, which is 15

**\*args and \*\*kwargs**

* \*args and \*\*kwargs are used as arguments if there is doubt in the number arguments to be passed in a function.
* It is worth noting that is the “\*” (args) and “\*\*” (kwargs) rather than args or kwargs.
* In general, \*args and \*\*kwargs are most useful when you don't know in advance how many arguments or keyword arguments you need to pass to a function, or when you want to make your code more flexible.
* They are particularly useful when you are writing functions that work with different types of data, or when you are writing functions that need to be reused in different contexts.

**\*Args**

* Use \*args when you need to accept an arbitrary number of non-keyword arguments. This is useful when you don't know in advance how many arguments you need to pass to a function, or when you want to make your code more flexible

**\*Kwargs**

* Use \*\*kwargs when you need to accept an arbitrary number of keyword arguments. This is useful when you want to provide default values for some of the arguments, or when you want to make your code more flexible.

**Built-in Functions**

* A pre-defined function that is available in the global namespace of the interpreter. These functions are always available to use in any Python program without the need for any additional imports or installations.

**map()**

* Executes a specified function for each item in an iterable. The item is sent to the function as a parameter.
* Such elements of an iterable include a list, tuple, or string and returns a new iterable object with the results.
* This built-in function allows you to process and transform all the items in an iterable without using an explicit for loop
* function is a function that takes one argument and returns a value.
* iterable is an iterable object that contains the values to be processed by the function.

**filter()**

* Filters the elements of an iterable based on a given function and returns an iterator containing only the elements for which the function returns True.
* function is a function that takes one argument and returns a boolean value (True or False).
* This built-in function allows you to process an iterable and extract those items that satisfy a given condition
* iterable is an iterable object that contains the values to be filtered.

**reduce()**

* Applies a given function to the elements of an iterable in a cumulative way and returns a single value that represents the "reduced" result of the operation.
* function is a function that takes two arguments and returns a value.
* iterable is an iterable object that contains the values to be reduced.
* initial is an optional initial value to start the accumulation. If not provided, the first element of the iterable is used as the initial value.
* In Python 2, reduce() was a built-in function and did not require importing from the functools module. However, in Python 3, reduce() was moved to the functools module and is no longer a built-in function.

**sorted()**

* Takes an iterable as input and returns a new sorted list of the elements in the iterable. The sorted() function can be used with any iterable data type such as lists, tuples, or strings.
* iterable is the iterable to be sorted.
* key is an optional function that is used to determine the sorting order of each element in the iterable. If provided, the function should take an element of the iterable as input and return a value that will be used as the sorting key for that element.
* reverse is an optional boolean flag that specifies whether the elements should be sorted in reverse order.

**Lambda Functions**

* A lambda function (also called an "**anonymous function**" or a "**function literal**") is a small, one-line function that is defined without a name.
* Lambda functions are often used when you need to define a simple function that you will only use once, or when you want to pass a function as an argument to another function.
* Lambda functions are defined using the **lambda** keyword, followed by the function's arguments and a colon, and then the expression that should be evaluated and returned by the function.

**Lambda Advantages**

* Provides a **cleaner** code
* **Performs similarly** to a function
* **Preserves namespaces** and **reduces code pollution**
* **Improve the functionalities of built-in functions** and **make them more expressive** (expressive in programming means concisely readable and meaningful)
* **Fas**t and **requires less overhead** than a user-defined function
* Provides **a dedicated function**

**Lambda Disadvantages**

* Non-re-usable
* Has no exact identity
* Cannot include statements or multiple expressions
* Can make code more difficult to debug if not used properly

**Higher-order functions with lambda**

* A higher-order function is a function that takes one or more functions as arguments, or that returns a function as its result.
* In the context of lambda functions, a higher-order function is a function that takes one or more lambda functions as arguments, or that returns a lambda function as its result.
* Higher-order functions and lambda functions can be very powerful tools for writing concise and expressive code in Python.
* They allow you to create small, single-purpose functions that can be combined and reused in many different contexts, and they can help you write code that is easier to read, write, and maintain.

**Week 05 - Recursion**

**Recursion**

* Recursion is the process of defining something in terms of itself.
* A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

**Python Recursive Function**

* In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

**Pitfall**

* The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

**Advantages of Recursion**

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* Sequence generation is easier with recursion than using some nested iteration.

**Disadvantages of Recursion**

* Sometimes the logic behind recursion is hard to follow through.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
* Recursive functions are hard to debug.

**Uses and implementations of Recursion**

* Fibonacci Number Sequence
* Binary Tree Traversal
* Tower of Hanoi
* Calculation Exponents
* Computing GCD
* Permutations and Combinations
* Maze Solving
* Fractals

**Week 06 – Exception**

**Difference between Syntax error and Exception**

* **Syntax Error**: As the name suggests this error is caused by the wrong syntax in the code. **It leads to the termination of the program.**
* **Exceptions** are raised when the program is syntactically correct, but the code resulted in an error.
* This error does not stop the execution of the program, however, it changes the normal flow of the program.
* In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

**Try and Except Statement – Catching Exceptions**

* Try and except statements are used to catch and handle exceptions in Python.
* Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

**Catching Specific Exception**

* A try statement can have more than one except clause, to specify handlers for different exceptions.
* Note that at most one handler will be executed. For example, we can add IndexError in the code.

**Try with Else Clause**

* In python, you can also use the else clause on the try-except block which must be present after all the except clauses.
* The code enters the else block only if the try clause does not raise an exception.

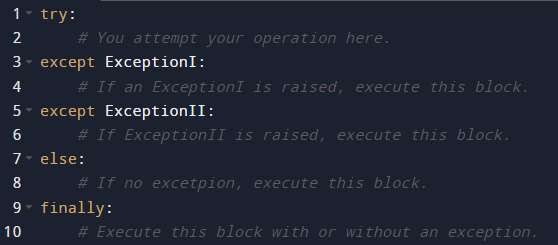
**The “Finally”, Keyword in Python**

* Python provides a keyword finally, which is always executed after the try and except blocks.
* The final block always executes after normal termination of try block or after try block terminates due to some exception.

**Raising Exception**

* The raise statement allows the programmer to force a specific exception to occur.
* The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).
* The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

**Summary**

****