PYTHON

**Module 1**: An Introduction to Python • What can Python do? • Why Python? • Good to know • Python Syntax compared to other programming languages • Python Install

**Module 2**: Beginning Python Basics • The print statement • Comments • Python Data Structures & Data Types • String Operations in Python • Simple Input & Output • Simple Output Formatting • Operators in python

**Module 3**: Python Program Flow • Indentation • The If statement and its’ related statement • An example with if and it’s related statement • The while loop • The for loop • The range statement • Break &Continue • Assert • Examples for looping

**Module 4**: Functions& Modules • Create your own functions • Functions Parameters • Variable Arguments • Scope of a Function • Function Documentations • Lambda Functions& map • n Exercise with functions • Create a Module • Standard Modules

**Module 5**: Exceptions Handling • Errors • Exception handling with try • handling Multiple Exceptions • Writing your own Exception

**Module 6**: File Handling • File handling Modes • Reading Files • Writing& Appending to Files • Handling File Exceptions • The with statement

**Module 7**: Classes In Python • New Style Classes • Creating Classes • Instance Methods • Inheritance • Polymorphism • Exception Classes & Custom Exceptions

**Module 8**: Generators and iterators • Iterators • Generators • The Functions any and all • With Statement • Data Compression

**Module 9**: Data Structures • List Comprehensions • Nested List Comprehensions • Dictionary Comprehensions • Functions • Default Parameters • Variable Arguments • Specialized Sorts

**Module 10**: Collections • namedtuple() • deque • ChainMap • Counter • OrderedDict • defaultdict • UserDict • UserList • UserString

**Module 11**: Writing GUIs in Python (Tkinter) • Introduction • Components and Events • An Example GUI • The root Component • Adding a Button • Entry Widgets • Text Widgets • Check buttons

**Module 12**: Python SQL Database Access • Introduction • Installation • DB Connection • Creating DB Table • INSERT, READ, UPDATE, DELETE operations • COMMIT & ROLLBACK operation • handling Errors

**print statement**:

The print statement is a built-in function in Python that allows you to display information on the screen. Its primary purpose is to output text, variables, or expressions to the console or standard output device, typically the terminal or command prompt.

Ex: print("Hello, World!")

**Comments**:

Comments in Python are textual annotations within the code that are ignored during program execution. it serve as notes or explanations to make the code more understandable for us, including other programmers who might read or work on the same code in the future.

Ex: # write the code

print("Hello, World!") # This is another comment

**Python Data Structures & Data Types:**

**Data Structures**:

Lists: Ordered collections of items that are mutable (modifiable)

my\_list = [1, 2, 3, 4, 5]

Tuples: Similar to lists but immutable (cannot be changed after creation)

my\_tuple = (1, 2, 3)

Sets: Unordered collections of unique elements.

my\_set = {1, 2, 3}

Dictionaries: Collections of key-value pairs, where each value is associated with a unique key.

my\_dict = {'name': 'Alice', 'age': 30}

Strings: Sequences of characters

my\_string = "Hello, World!"

**Data Types:**

**Numeric Types**:

int: Represents integer numbers, such as 5, -10, or 1000.

float: Represents floating-point numbers, such as 3.14, -0.001, or 2.5e-3.

complex: Represents complex numbers, with a real and imaginary part, such as 3 + 2j or -1 - 4j.

**Sequence Types**:

str: Represents strings of characters, such as "hello", 'Python', or "123".

list: Represents ordered collections of items, which can be of different types, such as [1, 2, 3], ['a', 'b', 'c'], or [1, 'hello', True].

tuple: Similar to lists but immutable (cannot be modified), such as (1, 2, 3) or ('apple', 'banana', 'orange').

**Mapping Type**:

dict: Represents key-value pairs, where each value is associated with a unique key, such as {'name': 'Alice', 'age': 30, 'city': 'New York'}.

**Set Types**:

set: Represents unordered collections of unique elements, such as {1, 2, 3} or {'apple', 'banana', 'orange'}.

frozenset: Similar to sets but immutable.

Boolean Type:

bool: Represents boolean values, True or False.

**String operations in Python:**

String operations in Python involve various techniques for manipulating text data

Concatenation: Combining strings together.

String Formatting: Creating formatted strings. it as filling in blanks in a template

Substring: Extracting a part of a string.

String Length: Finding out how many characters are in a string. It's like counting the number of letters in a word.

String Comparison: Comparing strings to check if they are equal, not equal, or one comes before/after the other alphabetically.

str1 = "Hello"

str2 = "World" o/p: Hello World

result = str1 + " " + str2

print(result).

**Introduction:**

In the world of data analysis, machine learning, and software development, Python has emerged as a go-to programming language due to its simplicity.Whether you're working on a small script or a large-scale application, understanding the performance characteristics of your code is crucial. This is where concepts like time and space complexity come into play.

**What is Time Complexity in Python?**

Time complexity in Python refers to the amount of time an algorithm takes to complete as a function of the size of its input. It's often expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's runtime. For example, an algorithm with time complexity O(1) executes in constant time, regardless of the size of the input, while an algorithm with time complexity O(n^2) grows quadratically with the input size.

**Analyzing Time Complexity in Python?**

Analyzing the time complexity of a Python algorithm involves counting the number of operations it performs as a function of the input size. This analysis considers worst-case, best-case, and average-case scenarios. While Big O notation gives a high-level understanding of an algorithm's efficiency, it's essential to delve deeper into its behavior under different conditions.

**What is Space Complexity in Python?**

Space complexity in Python measures the amount of memory an algorithm requires to execute. Like time complexity, it's also expressed using Big O notation. Some algorithms may have excellent time complexity but be memory-intensive, leading to high space complexity. Understanding both time and space complexity is crucial for designing efficient Python algorithms.

**Real-World Applications in Python?**

Time and space complexity concepts find applications across various Python domains,Like from data analysis to web development. In data analysis, optimizing algorithms for time and space complexity can lead to faster processing of large datasets. In web development, efficient algorithms are crucial for handling user requests and delivering responsive web applications.

**Best Practices in Python**

Enhancing time and space complexity in Python often involves leveraging appropriate data structures and algorithmic strategies/Techniques.

For example, Utilizing built-in data structures like dictionaries and sets can streamline operations and conserve memory. Moreover, steering clear of nested loops and unnecessary memory allocations aids in minimizing both time and space complexity overheads.

**Conclusion**

In essence, comprehensing time and space complexity in Python lays the foundation for writing efficient code. By analyzing algorithm efficiency, Python developers can build faster,more scalable software solutions. Whether you're a data scientist, web developer, or machine learning enthusiast, mastering these concepts will undoubtedly enhance your problem-solving abilities and contribute to the development of robust, high-performance Python solutions.

**2.\_\_init\_() :**

In Python, the \_\_init\_() method is like a special function that gets called automatically whenever we create a new object (instance) of a class. It's commonly known as the constructor. The purpose of \_\_init\_\_() is to initialize the object's attributes or perform any necessary setup to get the object ready to use.

Ex: Let's say we have a class called Car, and we want each car object to have attributes like color, make, and model. we can use the \_\_init\_\_() method to set these attributes when a new car object is created.

**\_\_main\_\_:**

In Python, \_\_main\_\_ is a special module name that indicates the main entry point of a Python program. When you run a Python script directly, the code within the script's main block, often identified by if \_\_name\_\_ == "\_\_main\_\_":, is executed.

This is particularly useful when we want certain code to run only if the script is executed directly, not when it's imported as a module in another script.

3. **Data Types:**

**Numeric Types**:

int: Represents integer numbers, such as 5, -10, or 1000.

float: Represents floating-point numbers, such as 3.14, -0.001, or 2.5e-3.

complex: Represents complex numbers, with a real and imaginary part, such as 3 + 2j or -1 - 4j.

**Sequence Types**:

str: Represents strings of characters, such as "hello", 'Python', or "123".

list: Represents ordered collections of items, which can be of different types, such as [1, 2, 3], ['a', 'b', 'c'], or [1, 'hello', True].

tuple: Similar to lists but immutable (cannot be modified), such as (1, 2, 3) or ('apple', 'banana', 'orange').

**Mapping Type**:

dict: Represents key-value pairs, where each value is associated with a unique key, such as {'name': 'Alice', 'age': 30, 'city': 'New York'}.

**Set Types**:

set: Represents unordered collections of unique elements, such as {1, 2, 3} or {'apple', 'banana', 'orange'}.

frozenset: Similar to sets but immutable.

Boolean Type:

bool: Represents boolean values, True or False.

**4.Variables in Python:**

Variables in Python are like containers that hold information. When you create a variable, you're giving a name to something so you can easily refer to it later.

* Integer
* Float
* String
* Boolean
* List
* Tuple
* Dictionary
* Set

**5. Operators in Python:**

Operators in Python are like tools that help us work with data in our code. They allow us to perform different kinds of tasks, like doing math, comparing values, and combining data together.

1. Arithmetic Operators: for basic maths: + , - , \*, / , % .

2. Comparison Operators:compare values: >, <, ==, !=, >=, <=.

3. Logical Operators:combine conditions: and, or, not.

4. Assignment Operators:assign values: =, +=, -=, \*=, /=, %=, \*\*=, //=.

5. Identity Operators:check if objects are the same: is, is not.

6. Membership Operators:check if a value is in a sequence: in, not in.

7. Bitwise Operators:perform operations on binary numbers: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift).

**6.Type Conversions:**

Type conversion in Python involves transforming the data type of a value to another type. It's essential for manipulating data effectively within Python programs. There are two types:

**Implicit Conversion**: Automatically occurs in certain operations, such as arithmetic operations between different types.

**Explicit Conversion (Type Casting**): Manually performed using functions like int(), float(), str(), etc., allowing developers to control the data type conversion process.

**Data Structures:**

In Python, data structures are fundamental constructs used for organizing, storing, and manipulating data efficiently. Here's a brief overview of some common data structures:

**Number**: Represents numerical data, including integers, floating-point numbers, and complex numbers.

**String**: A sequence of characters used for representing text data. Strings are immutable.

**Lists:** Ordered collections of items that can be of any data type. Lists are mutable and support various operations like appending and slicing.

**Tuples**: Similar to lists but immutable, tuples are used for storing collections of data that should not be modified.

**Sets:** Unordered collections of unique elements, useful for tasks like removing duplicates and performing set operations.

**Dictionaries:** Collections of key-value pairs, allowing efficient lookup and insertion based on keys. Dictionaries are mutable.

**List Comprehensions**:

List comprehensions in Python provide a concise and elegant way to create lists. They allow you to generate a new list by applying an expression to each item in an existing iterable like a list, tuple, or string. The syntax is compact and readable, making it easy to write and understand.

List comprehensions are versatile and can be nested or combined with conditional expressions to create complex lists efficiently. They are widely used in Python for tasks like filtering, mapping, and transforming data in a concise and readable manner.

expression: The expression to evaluate for each item.

item: The variable representing each item in the iterable.

iterable: The existing collection of items.

**Input/Output statements :**

Input and output (I/O) statements are essential for interacting with users and handling data in Python.

Standard Input/Output: Use input() for user input and print() for output to the console.

File I/O: Employ open() to handle file operations, including reading and writing data.

Formatted Output: Utilize formatted strings (f-strings) or the str.format() method for structured output.

Command Line Arguments: Access command-line arguments via sys.argv for input from the terminal.

**Conditional Statements**: Use if, elif, and else to execute different blocks of code based on conditions.

**Looping**: Employ for and while loops to execute code repeatedly, iterating over sequences or as long as a condition is true, respectively.

**Break & Continue**: break exits a loop prematurely, while continue skips the rest of the current iteration and moves to the next.

**Range**: The range() function generates a sequence of numbers commonly used with loops, defining start, stop, and step values.

**Functions and Methods**

**Functions:**

Functions in Python are blocks of reusable code that perform a specific task. They are defined using the def keyword followed by the function name, parameters , and a colon. The body of the function is then indented and contains the code to be executed when the function is called. Functions can optionally return a value using the return statement.Functions in Python are versatile and can be defined anywhere in the code. They can accept arguments, perform operations, and optionally return values**.**

For example: The function called add\_numbers that takes two numbers, adds them together, and gives us the result. Whenever we need to add numbers, we can just call this function and get the answer.

**Methods:**

Methods are similar to functions but are associated with objects. They are functions that are defined within a class and operate on objects of that class. In Python, methods are accessed using dot notation, where the method is called on an object.Methods can access and modify the state of an object, and they can also perform operations specific to that object's type. For example, methods of a string object can manipulate strings, while methods of a list object can manipulate lists.

For example: We have a dog object named my\_dog, we can tell it to bark() by writing my\_dog.bark().

**Difference**:

One significant difference between functions and methods is that functions can be defined anywhere in our code, while methods are always defined within a class. This reflects the idea that methods are closely associated with specific types of objects, while functions are more general-purpose.The main difference between functions and methods lies in their association with objects. Functions are standalone blocks of code that can be defined anywhere and operate independently of objects. Methods, on the other hand, are functions defined within a class and are associated with objects of that class.In summary, functions are more general-purpose and can be used independently, while methods are specific to the objects they operate on and are called using dot notation on those objects.

**Conclusion:**

In essence, functions and methods are indispensable tools in Python, offering distinct capabilities for organizing and executing code.

**Errors and exception handling**

Errors and exception handling in Python are mechanisms for dealing with unexpected situations that arise during program execution.

**Errors** occur when something goes wrong in your code, such as a syntax error or a runtime error like division by zero.

**Exceptions** are errors that occur during program execution but can be handled by the programmer. Common exceptions include ZeroDivisionError, TypeError, and FileNotFoundError.To handle exceptions, Python provides a try statement, which allows you to wrap code that may raise an exception. If an exception occurs, you can catch and handle it using except blocks. Additionally, we can use finally blocks to execute cleanup code, regardless of whether an exception occurred or not.

By using exception handling,we can gracefully handle errors and prevent our program from crashing, improving its robustness and reliability.

**OOPS**

Object-Oriented Programming (OOP) in Python is a programming paradigm that emphasizes the use of objects and classes to structure code and model real-world entities.

Objects are instances of classes, which act as blueprints for creating objects. A class defines the properties (attributes) and behaviors (methods) of objects.

**Class:** A class is a blueprint or template for creating objects. It defines the attributes and methods that objects of that class will have.

**Object:** An object is an instance of a class. It is a specific realization of the class, with its own unique attributes and behaviors.

**Encapsulation:** Bundling data (attributes) and methods that operate on that data within a single unit (class), preventing direct access to internal data from outside the class.

**Inheritance:** Allows a class (subclass) to inherit attributes and methods from another class (superclass), promoting code reuse and hierarchical organization of classes.

**Polymorphism:** Refers to the ability of objects to take on different forms or behaviors based on the context. In Python, polymorphism is achieved through method overriding and method overloading**.**

**FILE OPERATIONS**

File operations in Python involve reading from and writing to files.

First, we open a file using the open() function, specifying the file path and mode.

Then, we can read from the file using methods like read(), readline(), or readlines(), or write to it using the write() method.

Finally, remember to close the file using the close() method.

Python's with statement provides a more concise and safe way to work with files by automatically closing them when done. Understanding file operations is essential for handling external data in Python programs.

**Modules and Packages**

Modules: Python files containing Python code, typically with functions, classes, and variables. Modules can be imported and used in other Python scripts.

Packages: Collections of related modules that provide functionality for specific tasks or domains. They often include modules, sub-packages, and sometimes extension modules written in other languages.

**WHY?**

Modules and packages allow us to organize and reuse code effectively. They help in modularizing your codebase, making it more manageable, maintainable, and scalable.Instead of writing everything from scratch, you can leverage existing modules and libraries to save time and effort.

**HOW?**

**Creating Modules**: We write Python code in a .py file and save it with a meaningful name. now we can then import the module into our scripts using the import statement.

**Using Package**: Install packages using package managers like pip, and then import modules from those packages as needed.

**Built-in Methods:**

Built-in methods are functions that are part of the Python language itself and are available for use without needing to import any additional modules.

**Examples**:

print(): Used to output data to the console.

len(): Returns the length of a sequence (e.g., list, tuple, string).

type(): Returns the type of an object.

range(): Generates a sequence of numbers.

open(): Opens a file.

input(): Reads input from the user.

sum(): Calculates the sum of elements in a sequence.

**Types of file handling**

File handling in programming refers to the process of working with files, which are collections of data stored on a computer's disk. File handling typically involves operations such as reading, writing, and closing files**.**

**Reading Files:** This involves opening an existing file to extract data from it. Python provides functions like ‘open()’ to open files in read mode, read() to read the entire content, ‘readline()’ to read one line at a time, and ‘readlines()’ to read all lines into a list.

**Writing to Files:** Python allows you to create new files or overwrite existing ones with fresh data. You can use ‘open()’ with write mode ('w') to create a new file or truncate an existing one. Then, you can use ‘write()’ to add content to the file.

**Appending to Files**: Similar to writing, appending involves adding new data to the end of an existing file without erasing its previous contents. You can achieve this by opening the file in append mode ('a') and using ‘write()’ to add content at the end**.**