Introduction to Python Theory:

1. Introduction to python and its features (simple, high-level, interpreted language).

❖ Introduction:

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development.
- Python supports modules and packages, which encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.
- Designed to be easy to learn and master:
  - Clean and clear syntax
  - Very few keywords
- Highly portable:
  - Runs almost anywhere - high end servers and workstations, down to windows CE
  - Uses machine independent byte-code
- Extensible:
  - Designed to be extensible using C/C++
  - allowing access to many external libraries

❖ Features of Python

- Clean syntax plus high-level data types
  - Leads to fast coding
- Uses white-space to delimit blocks
  - Variables do not need declaration
  - Although not a type-less language

❖ Python Productivity

- Reduced development time
  - Code is 2-10x shorter than C, C++, Java
- Improved program maintenance
  - Code is extremely readable
- Less training
  - Language is very easy to learn

2. History and evolution of Python

❖ History

- Python's history began in the late 1980s with Guido van Rossum, who started its development as a hobby project in 1989 at CWI in the Netherlands.
- It was officially released in February 1991 as version 0.9.0.
- Guido van Rossum, inspired by the ABC language, began developing Python as a successor to it.
- He aimed to create a language that was easy to read, understand, and use for general-purpose applications.
- The name "Python" was inspired by the British comedy troupe Monty Python's Flying Circus.
- Python 0.9.0, released in 1991, featured exception handling, functions, and modules, which were key building blocks for future development.

❖ Evolution

- Python 1.0 (1994):
  - Introduced features like exception handling, lambda functions, and the map, filter, and reduce functions.
- Python 2.0 (2000):
  - Added list comprehensions, a garbage collection system capable of collecting reference cycles, and other significant enhancements.
- Python 3.0 (2008):
  - A major, backwards-incompatible release that aimed to fix inconsistencies and remove redundant constructs from Python 2. Many of its features were backported to Python 2.6 and 2.7.
- Ongoing Development:
  - Python continues to evolve with new releases, adding new features, improving performance, and enhancing the developer experience.
- The latest stable major release of Python is 3.13, which was released on October 7, 2024. While 3.12 is the previous major release, 3.13 is currently the newest.

3. Advantages of using Python over other programming languages.

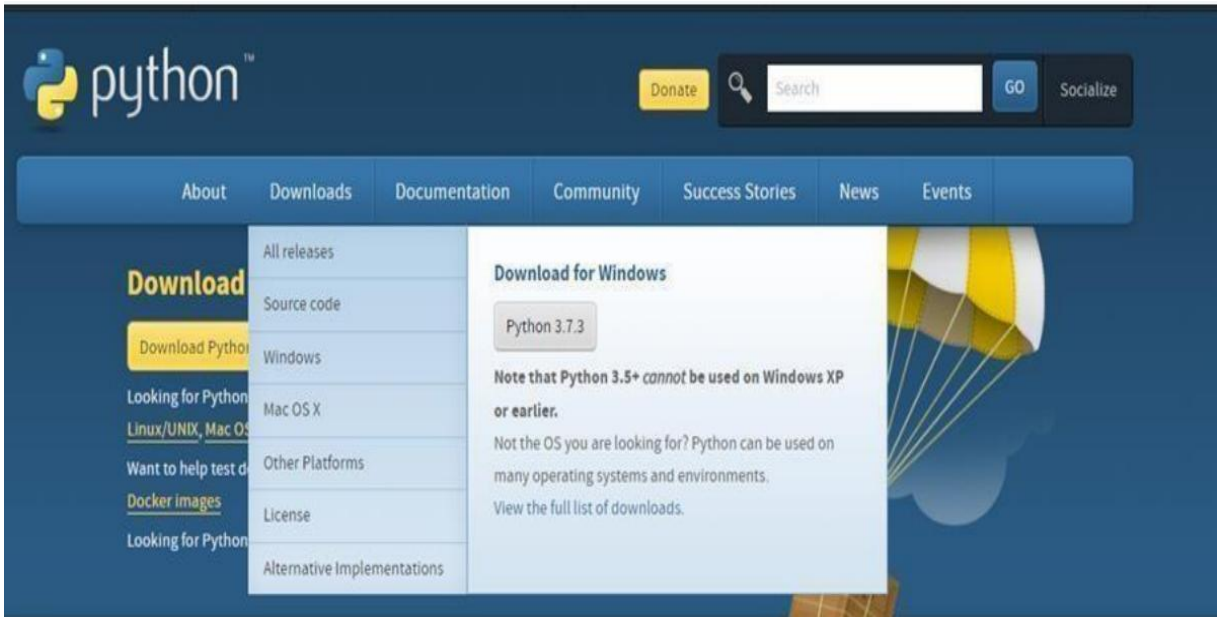
- Simplicity and Readability:
  - Python's syntax is designed to be easy to read and understand, resembling plain English
  - This makes it a great choice for beginners and allows for faster development and maintenance of code.

- Versatility:
    - Python can be used for a wide range of applications, including web development data science, machine learning, automation and more.
- Extensive Libraries and Frameworks:
    - Python has a vast ecosystem of libraries and frameworks, such as NumPy for numerical computing, Pandas for data manipulation, and TensorFlow for machine learning simplifying development across various domains.
- Cross-Platform Compatibility:
    - Python code can run on various operating systems, making it highly portable and versatile.
- Strong Community Support:
    - Python has a large and active community of developers who provide support, tutorials and resources.
- Rapid Development and Prototyping:
    - Python's simplicity and ease of use allow for faster development and prototyping.
- Open Source and Free:
    - Python is an open-source language, meaning it's free to use, distribute, and modify.
- Scalability and Performance:
    - Python can be scaled to handle large projects and complex applications.
- Ease of Learning and Teaching:
    - Python's simplicity and readability make it an excellent choice for beginners.

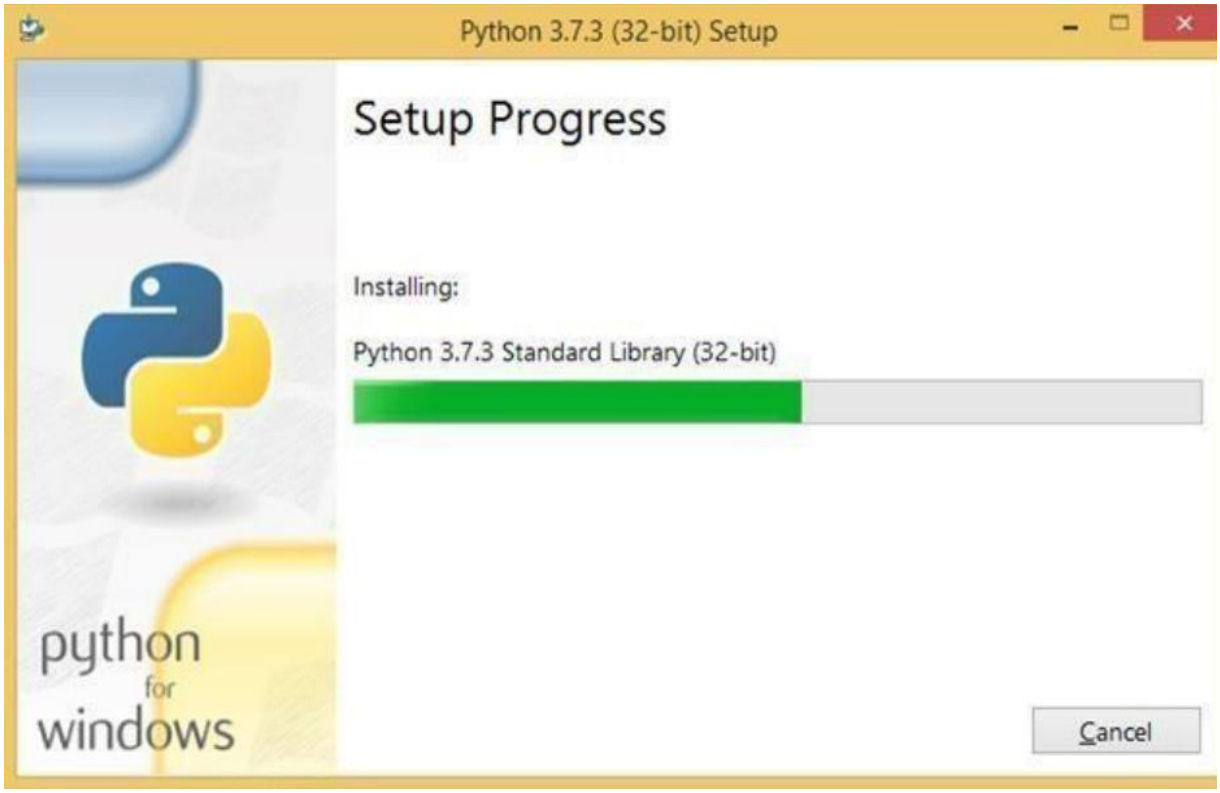
4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

- Step-1:

- Step-2:


python
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Download
All releases
Source code
Windows
Mac OS X
Other Platforms
License
Alternative Implementations
Download for Windows
Python 3.7.3
Note that Python 3.5+ cannot be used on Windows XP or earlier.
Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.
Looking for Python
Linux/UNIX, Mac OS
Want to help test d
Docker images
Looking for Python

- Step-3:


Python 3.7.3 (32-bit) Setup
Setup Progress
Installing:
Python 3.7.3 Standard Library (32-bit)
python
for
windows
Cancel

5. Writing and executing your first Python program.

```
1   #Write a Python program that prints "Hello, World!".
2   print("Hello, World!")
```

```
PS D:\Tops Python> & C:/Users/admin/AppData/Local/Programs/Python/Python313/python.exe
ts/program 1.py"
Hello, World!
PS D:\Tops Python>
```

Programming Style:

1. Understanding Python's PEP 8 guidelines.

- Python maintains a strict way of order and format of scripting Making it easy for others to read code is always a good idea, and adopting a nice coding style helps tremendously for that.
- For Python, PEP 8 has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style.
- **Use 4-space indentation and no tabs:** The 4-space rule is not always mandatory and can be overruled for continuation line.
- **Use docstrings:** There are both single and multi-line docstrings that can be used in Python.
- **Wrap lines so that they don't exceed 79 characters:** The Python standard library is conservative and requires limiting lines to 79 characters. The lines can be wrapped using parenthesis, brackets, and braces.
- **Use of regular and updated comments are valuable to both the coders and users**: If a comment is a full sentence, its first word should be capitalized. In block comments, there are more than one paragraphs and each sentence must

end with a period. Block comments and inline comments can be written followed by a single '#'.

- **Use of trailing commas:** This is not mandatory except while making a tuple.
- **Use Python's default UTF-8 or ASCII encodings and not any fancy encodings**, if it is meant for international environment.
- **Use spaces around operators and after commas, but not directly inside bracketing constructs:**
- **Naming Conventions:** There are few naming conventions that should be followed in order to make the program less complex and more readable.

2. Indentation, comments, and naming conventions in Python.

❖ Indentation

- Python uses indentation to define code blocks, unlike other languages that use braces or keywords. Indentation is crucial for code structure and readability.
  - Consistent Spacing
  - Block Definition
  - Error

❖ Comments

- Comments are used to explain the code and are ignored by the Python interpreter.
  - Single-line comments
  - Block comments
  - Docstrings

❖ Naming Conventions

- Python follows specific naming conventions to improve code readability and consistency**.**
  - Variables, Functions, and Modules
  - Classes
  - Constants
  - Private attributes
  - Avoid
  - Descriptive Names

3. Writing readable and maintainable code.

- **Guidelines:**
  - Use meaningful variable name
  - Follow PEP 8 Guidelines
  - Limit line length
  - Proper indentation
  - Add comments widely
  - Avoid long function
  - Modular Design

Core Python Concepts

1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

- Integers (int):
  - Represent whole numbers, positive or negative, without any decimal point
  - Examples: -3, 0, 5, 1000.
  - There is no limit to the size of integers in Python.
- Floats (float):
  - Represent numbers with decimal points, also known as floating-point numbers.
  - Examples: -2.5, 0.0, 3.14, 10.001.
  - They can also be expressed using scientific notation (e.g., 1.2e3 for 1200).
- Strings (str):
  - Represent sequences of characters, used to store text.
  - Enclosed in single quotes ('hello'), double quotes ("world"), or triple quotes ("'long string"').
  - Strings are immutable, meaning they cannot be changed after creation.
- Lists (list):
  - Ordered collections of items, which can be of different data types.
  - Mutable, meaning elements can be added, removed, or changed.
  - Defined using square brackets [1, 2, 'three'].

- Tuples (tuple):
  - Ordered collections of items, similar to lists.
  - Immutable, meaning they cannot be changed after creation.
  - Defined using parentheses (1, 2, 'three').
- Dictionaries (dict):
  - Collections of key-value pairs.
  - Keys must be unique and immutable data types (like integers, strings, or tuples).
  - Values can be of any data type.
  - Defined using curly braces {key1: value1, key2: value2}.
- Sets (set):
  - Unordered collections of unique items.
  - Mutable, meaning elements can be added or removed.
  - Duplicate elements are not allowed.
  - Defined using curly braces {1, 2, 3} or the set () constructor.

2. Python variables and memory allocation.

❖ Variables

- A name which can store a value.
- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- E.g. number=20, age = 21
- Note: Variables do not need to be declared with any particular type.

❖ Memory Allocation

- **Dynamic Allocation**: Memory is allocated as needed during runtime.
- **Private Heap**: Python objects are stored in a private heap, inaccessible directly by programmers.
- **Memory Allocators**: Python uses built-in memory allocators, optimizing for small objects by using "free lists" to recycle memory blocks. Dynamic allocation is used for complex objects like lists and dictionaries.
- **Memory Pools**: Memory is organized into pools based on object size to minimize overhead and fragmentation.
- **Object-Specific Allocators**: Different object types (e.g., integers, strings) may have specific allocators for optimized memory management.
- **Reference Counting**: Python uses reference counting to track object usage. When an object's reference count drops to zero, it's deallocated.
- **Garbage Collection**: Python's garbage collector detects and reclaims memory occupied by unreachable objects, preventing memory leaks.

- **Memory Management Techniques**: Python uses techniques such as best fit, memory pools, and reference counting to manage memory effectively.
- **Immutability**: Immutable objects, like tuples, offer memory efficiency due to their fixed size.
- **Memory Allocation APIs**: Python provides specific APIs for allocating and deallocating memory, such as PyMem_Malloc(), PyMem_RawMalloc(), and malloc (), which are used for non-Python objects.
- **Memory Reusage**: Python keeps freed memory blocks within its process to reuse them later, instead of releasing them back to the operating system immediately.

3. Python operators: arithmetic, comparison, logical, bitwise.

❖ Arithmetic Operators:

- These operators perform mathematical calculations:
  - **+:** Addition
  - **-:** Subtraction
  - ***:** Multiplication
  - **/:** Division (returns a float)
  - **//:** Floor division (returns an integer)
  - **%:** Modulo (remainder of division)
  - ****:** Exponentiation

❖ Comparison Operators:

- These operators compare two values and return a Boolean (True or False):
  - = =: Equal to
  - ! =: Not equal to
  - >: Greater than
  - <: Less than
  - >=: Greater than or equal to
  - <=: Less than or equal to

❖ Logical Operators:

- These operators combine or modify Boolean expressions:
  - and: Returns true if both operands are true
  - or: Returns true if at least one operand is true
  - not: Returns true if operand is true and vice versa

❖ Bitwise Operators:

- These operators perform operations on the binary representation of integers:
  - &: Bitwise AND
  - |: Bitwise OR
  - ^: Bitwise XOR
  - ~: Bitwise NOT
  - <<: Left shift
  - >>: Right shift

Conditional Statements

1. Introduction to conditional statements: if, else, elif.

❖ if statement:

- It is similar to that of other languages.
- The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.
- Syntax: if condition:
  Statements

❖ if... else statement:

- It is similar to that of other languages.
- It is frequently the case that you want one thing to happen when a condition it true, and something else to happen when it is false.
- Syntax: if condition:
  Statements
  else:
  Statements

❖ if...elif...else statement:

- It is similar to that of other languages.
- The elif is short for else if. It allows us to check for multiple expressions.
- If the condition for if is False, it checks the condition of the next elif block and so on.

- If all the conditions are false, body of else is executed.
- Only one block among the several if...elif...else blocks is executed according to the condition.
- Syntax: if condition:
  Statements
  elif condition:
  Statements

2. Nested if-else conditions.

- There may be a situation when you want to check for another condition after a condition resolves to true.
- In such a situation, you can use the nested if construct.
- Syntax : if condition:
  Statements
  if condition:
  Statements
  else:
  Statements

Looping (For, While)

1. Introduction to for and while loops.

❖ For Loop:

- For loop has the ability to iterate over the items of anysequence, such as a list or a string.
- Syntax: for iterating_var in sequence:
  statements(s)
- If a sequence contains an expression list, it is evaluated first.
- Then, the first item in the sequence is assigned to the iterating variable iterating_var.
- Next, the statements block is executed.

- Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

➢ **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.
- The range () function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

❖ While Loop:

- A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value.
- Syntax: while expression:
  statement(s)

2. How loops work in Python.

- In Python, loops are used to repeatedly execute a block of code. There are two main types of loops: for loops and while loops

❖ For Loop

- For loops are used to iterate over a sequence (such as a list, tuple, string, or range) or other iterable objects. The loop executes once for each item in the sequence.

➢ **Key features of for loop:**

- Iteration: For loops iterate over a sequence (like a list, tuple, string, or range)
- Execution**:** The code block within the loop executes once for each item in the sequence.

❖ While Loop

- While loops are used to execute a block of code repeatedly as long as a condition is true.

➢ **Key features for while loop:**

- Condition-Based: While loops execute a block of code repeatedly as long as a specific condition is true.
- Execution: The loop continues until the condition becomes false.

3. Using loops with collections (lists, tuples, etc.).

❖ **Lists:**

- Lists are one of the most powerful data structures in python. Lists are sequenced data types.
- In Python, an empty list is created using list () function. They are just like the arrays declared in other languages.
- But the most powerful thing is that list need not be always homogeneous. A single list can contain strings, integers, as well as other objects.
- Lists can also be used for implementing stacks and queues.
- Lists are mutable, i.e., they can be altered once declared.
- The elements of list can be accessed using indexing and slicing operations.

❖ **Tuples:**

- A tuple is a sequence of immutable Python objects. Tuples are just like lists with the exception that tuples cannot be changed once declared. Tuples are usually faster than lists.

Generators and Iterators

1. Understanding how generators work in Python.

- Python generators are a way to create iterators. They are defined using functions, but instead of return, they use the yield keyword.
- When a generator function is called, it doesn't execute immediately. Instead, it returns a generator object that can be iterated over.

❖ **Here's how they work:**

➢ Lazy Evaluation: Generators produce values on demand, only when requested. This is memory-efficient, especially when dealing with large datasets.

- State Retention: They retain their internal state between calls. When next() is called on a generator, it resumes execution from where it left off, remembering the values of local variables.
- yield Keyword: The yield keyword pauses the function's execution and returns a value to the caller. The function's state is saved, allowing it to resume later.

❖ Benefits of using Generators:

- Memory Efficiency: They are ideal for handling large datasets or infinite sequences because they generate values on the fly without needing to store everything in memory.
- Simpler Code: They can make code more readable and concise when you need to iterate over a sequence of values.
- State Management: They automatically manage their state during iteration, simplifying code.

2. Difference between yield and return.

❖ return:

- Purpose: Terminates the function's execution and sends a single value back to the caller.
- Behaviour: Once a return statement is encountered, the function immediately stops executing, and any code after the return statement is not run.
- Output: Returns a single value or None if no value is specified.
- Function Type: Used in regular functions

❖ yield:

- Purpose: Creates a generator function that produces a sequence of values over time.
- Behaviour: When a yield statement is encountered, the function pauses its execution, saves its current state, and returns the yielded value to the caller. The function's execution can be resumed from where it left off the next time the generator is called.
- Output: Returns a generator object, which can be iterated over to retrieve the yielded values.
- Function Type: Used in generator functions.

3. Understanding iterators and creating custom iterators.

❖ What is an Iterator?

- An iterator is an object that allows you to iterate (loop) over a sequence of values. In Python, an object is called an *iterator* if it implements two methods:
  - __iter__ () → returns the iterator object itself.
  - __next__ () → returns the next item in the sequence. Raises Stop Iteration when there are no more items.

❖ Iterable vs Iterator

- Iterable: An object capable of returning its members one at a time (e.g., list, tuple, string).
- **Iterator**: An object that keeps state and knows how to get the next value.

❖ Creating a Custom Iterator

- Steps:
  - Define a class.
  - Implement __iter__() → returns self.
  - Implement __next__() → returns next value, raises Stop Iteration when done.

Functions and Methods

1. Defining and calling functions in Python.

❖ Defining a Function

- In Python, a function is defined using the def keyword, followed by the function name, parentheses (), and a colon:
- Any parameters the function takes are placed inside the parentheses.
- The code block that makes up the function is indented below the def statement.

❖ Calling a Function

- To call a function, use the function name followed by parentheses (). If the function takes parameters, pass the arguments inside the parentheses.

❖ Key Points

- def keyword: Starts the function definition.
- Function name: Follows the def keyword and should be descriptive.
- Parameters: Variables inside the parentheses that receive values when the function is called.
- Arguments: Actual values passed to the function when it is called.
- Indentation: Code inside the function must be indented.
- return statement: Used to send a value back from the function to the caller. If no return statement is included, the function will return None.
- Function call: Executes the code inside the function.

2. Function arguments (positional, keyword, default).

- Python functions can accept different types of arguments, providing flexibility in how they are called.

❖ Positional Arguments:

- These are the most common type of arguments.
- They are passed to the function based on their order or position in the function call.
- The function definition specifies the order in which the arguments are to be received.
- When calling the function, the values must be provided in the same order as defined.

❖ Keyword Arguments:

- Keyword arguments are passed with the parameter name during the function call, using the syntax parameter_name=value.
- This allows you to pass arguments in any order, as long as you specify the parameter names.
- They enhance readability, especially when a function takes many arguments.
- Keyword arguments must follow positional arguments in the function call.

❖ Default Arguments:

- Default arguments are parameters in a function that have a default value specified in the function definition.
- If the caller doesn't provide a value for these arguments, the default value is used.
- Default arguments are defined using the = operator during function definition.
- Default arguments must follow non-default arguments in the parameter list.

3. Scope of variables in Python.

- In Python, the scope of a variable refers to the region of the code where the variable is accessible.
- It determines the visibility and lifetime of a variable within a program.
- Understanding variable scope is essential for writing structured and maintainable code.

❖ Python follows the LEGB rule to determine the scope of a variable, which stands for:

- Local: Variables defined inside a function are local to that function and are only accessible within that function.
- Enclosing: If a function is nested inside another function, the inner function can access variables defined in the outer function's scope, which is called the enclosing scope.
- Global: Variables defined outside any function or class are global and can be accessed from anywhere in the program.
- Built-in: This scope contains pre-defined functions, exceptions, and objects that are available throughout the program without needing to be defined or imported.

➢ **When a variable is referenced, Python searches for it in the following order: local, enclosing, global, and built-in scope**

❖ The lifetime of a variable depends on its scope:

- Local variables exist only during the function's execution.
- Global variables exist throughout the program's execution.
- Enclosing variables exist as long as the outer function is running.

➢ Benefits of limiting scope

- Limiting a variable's scope to only what's necessary and restricting global variable use makes a program easier to debug, maintain, and update.

- It helps avoid variable naming conflicts and makes the program more organized.

4. Built-in methods for strings, lists, etc.

❖ Methods for strings:

- len (): Returns the length of a string.
- upper (): Converts all characters to uppercase.
- lower (): Converts all characters to lowercase.
- strip (): Removes leading and trailing whitespace.
- replace (old, new): Replaces all occurrences of a substring with another.
- split (): Splits a string into a list of substrings based on a delimiter.
- capitalize (): Converts the first character to uppercase and the rest to lowercase.
- count(sub): Returns the number of times a substring appears.
- casefold(): Converts a string to lowercase for case-insensitive comparisons.
- isalnum(): Checks if all characters are alphanumeric.
- startswith(prefix): Checks if a string starts with a prefix.
- endswith(suffix): Checks if a string ends with a suffix.
- isdecimal(): Checks if all characters are decimal characters.
- isdigit(): Checks if all characters are digits.

❖ Methods of Lists:

- append(item): Adds an item to the end of the list.
- extend(iterable): Appends all elements from an iterable (like another list) to the end of the list.
- insert (index, item): Inserts an item at a specified index.
- remove(item): Removes the first occurrence of an item from the list.
- pop(index): Removes and returns the item at the specified index. If no index is provided, it removes and returns the last item.
- clear (): Removes all items from the list.
- index(item): Returns the index of the first occurrence of an item.
- count(item): Returns the number of times an item appears in the list.
- sort (): Sorts the list in ascending order by default.
- reverse (): Reverses the order of the elements in the list.
- copy (): Returns a shallow copy of the list.

Control Statements (Break, Continue, Pass)

1. Understanding the role of break, continue, and pass in Python loops.

❖ Break Statement:

- It brings control out of the loop and transfers execution to the statement immediately following the loop.
- Syntax: break

❖ Continue Statement:

- It continues with the next iteration of the loop.
- Syntax: continue

❖ Pass Statement:

- The pass statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.
- Syntax: pass

String Manipulation

1. Understanding how to access and manipulate strings.

❖ Accessing Characters:

- Python strings are sequences of characters, and you can access individual characters using indexing, starting from 0 for the first character.
- Negative indexing allows access from the end of the string, with -1 representing the last character.
- Slicing allows you to extract a range of characters using the syntax string[start:end]. The start index is inclusive, and the end index is exclusive.

❖ String Manipulation Methods:

- Concatenation: Use the + operator to combine strings.
- . join (): Joins the elements of an iterable (like a list or tuple) into a single string, using the string it's called on as a separator.
- . replace (): Replaces occurrences of a substring with another substring.
- . lower (): Converts all characters to lowercase.
- . upper (): Converts all characters to uppercase.
- . strip (): Removes whitespace from the beginning and end of a string.
- . split (): Splits a string into a list of substrings based on a delimiter.
- . find(): Returns the index of the first occurrence of a substring.
- . count (): Counts the occurrences of a substring.
- .startswith() and. endswith (): Checks if a string starts or ends with a specified substring.
- . format (): Formats strings by inserting values into placeholders.
- f-strings: Provides a concise way to embed expressions inside string literals.

2. Basic operations: concatenation, repetition, string methods (upper (), lower (), etc.).

❖ Concatenation:

- String concatenation is the process of combining two or more strings into a single string.
- The + operator is used for string concatenation.

❖ Repetition:

- String repetition involves repeating a string a specified number of times.
- The * operator is used for string repetition.

❖ String Methods:

- Python provides various built-in string methods for manipulating strings.
- Some commonly used methods include:
- upper (): Converts the string to uppercase.
- lower (): Converts the string to lowercase.
- strip (): Removes leading and trailing whitespace.
- replace (old, new): Replaces all occurrences of the old substring with the new substring.
- find(substring): Returns the index of the first occurrence of the substring. Returns -1 if the substring is not found.

- count(substring): Returns the number of occurrences of the substring.
- startswith(prefix): Checks if the string starts with the given prefix.
- endswith(suffix): Checks if the string ends with the given suffix.
- split(delimiter): Splits the string into a list of substrings based on the delimiter.

3. String slicing.

- String slicing in Python allows you to extract a portion of a string by specifying a start and end index.

  ➢ **The syntax for string slicing is string[start:end:step].**
  - start: The index where the slice begins (inclusive). If omitted, it defaults to 0.
  - end: The index where the slice ends (exclusive). If omitted, it defaults to the end of the string.
  - step: The increment between indices. If omitted, it defaults to 1.

❖ Negative Indexing:

- Negative indices can also be used to slice strings, where -1 refers to the last character, -2 refers to the second last character, and so on.

❖ Step Value:

- The step parameter allows you to extract characters at specific intervals.

❖ Note:

- The end index is exclusive, meaning the character at that index is not included in the slice.
- If the start index is greater than or equal to the end index, an empty string is returned.
- String slicing creates a new string object; it does not modify the original string.

Advanced Python (map (), reduce (), filter (), Closures and Decorators)

1. How functional programming works in Python.

- Functional programming in Python revolves around treating functions as first-class citizens, emphasizing immutability and avoiding side effects. It leverages higher-order functions and techniques like lambda expressions, map, filter, and reduce to transform and process data. Python offers a variety of features and libraries, like functools, itertools, and operator, that support a functional style of programming.

➢ Functions as First-Class Citizens:

- In Python, functions can be passed as arguments to other functions, returned as values from functions, and assigned to variables, just like any other data type.
- This allows for flexible function composition and higher-order functions (functions that take other functions as arguments or return them).

➢ Immutability:

- Functional programming encourages minimizing or avoiding state changes (mutation) in data. Instead of modifying existing data, new data is created during transformations.
- This immutability simplifies debugging, testing, and reasoning about code because the state of data is predictable.

➢ Avoiding Side Effects:

- Side effects (modifying global variables, performing I/O operations outside the function) are generally avoided in functional programming.
- Functions should ideally be pure, meaning they only depend on their inputs and produce outputs without altering the program's state or interacting with the outside world.

➢ Key Functional Programming Constructs:

- Lambda Functions: Anonymous functions defined using the lambda keyword, often used for simple, concise operations.
- map ():Applies a function to each item in an iterable and returns an iterator of the results.
- filter ():Creates an iterator from elements of an iterable for which a function returns True.
- reduce ():(from functools) Applies a function cumulatively to the items of an iterable, reducing it to a single value.

➢ Functional Programming in Python: How to Approach It:

- While Python is not a purely functional language, it provides features to support functional programming.
- You can adopt functional style for specific parts of your code, especially for data transformation pipelines.
- It's about choosing the right paradigm for a particular problem and leveraging the strengths of functional programming in Python.

2. Using map (), reduce (), and filter () functions for processing data.

- map (), filter (), and reduce () are powerful built-in functions in Python, often used to process data efficiently. They are considered functional programming tools, allowing for concise and expressive code.

❖ Map ():

- The map () function applies a given function to each item of an iterable (like a list, tuple, etc.) and returns a new iterable with the results.

❖ Filter ():

- The filter () function filters elements of an iterable based on a condition defined by a function. It returns an iterator containing the elements for which the function returns True.

❖ Reduce ():

- The reduce () function in Python is a tool for applying a function cumulatively to the items of an iterable, reducing the iterable to a single value. It is part of the functools module and requires importing before use.

3. Introduction to closures and decorators.

- Closures and decorators are powerful features in Python that enhance code flexibility and reusability.

❖ Closures:

- A closure is a nested function that remembers the values from its enclosing scope, even after the outer function has finished executing.
- This means the inner function retains access to variables from the outer function's scope.

- Closures are created when:
  - An inner function is defined within an outer function.
  - The inner function references variables from the outer function's scope.
  - The outer function returns the inner function.

❖ Decorators:

- Decorators are a way to modify the behaviour of a function or method without altering its original code.
- They use closures to wrap the original function.
- A decorator is essentially a function that takes another function as an argument, modifies it, and returns the modified function.
- Decorators are applied using the @ symbol followed by the decorator's name above the function definition.

➢ Key Differences:

- Closures: are about creating functions that remember their environment.
- Decorators: are a specific use case of closures, used to modify the behaviour of functions or methods.

❖ Use Cases:

➢ Closures:

- Creating function factories.
- Implementing callback mechanisms.
- Maintaining state between function calls.

➢ Decorators:

- Adding logging, timing, or authentication to functions.
- Validating input.
- Implementing caching.
- Enforcing access control.