Python

Theory

Ved Jani

1. Introduction to Python

Q. Introduction to Python and its Features (simple, high-level, interpreted language).

Ans:-

Python is a popular programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python is used in many areas like web development, data science, artificial intelligence, automation, and more.

Key Features of Python

1. **Simple and Easy to Learn**
    - Python has a clean and easy-to-understand syntax.
    - It looks like plain English, so beginners can learn it quickly.
      Example:
      print("Hello, World!")
2. **High-Level Language**
    - Python abstracts many complex details from the user (like memory management).
    - You can focus on *what* to do instead of *how* to do it at a low level.
3. **Interpreted Language**
    - Python code is executed line-by-line.
    - No need to compile code before running it.
    - Makes debugging easier.
4. **Portability**
    - Python is platform-independent.
    - A program written on Windows can run on Linux or Mac without changes.
5. **Object-Oriented**
    - Supports OOP (Object-Oriented Programming) concepts like classes and objects.

6. **Rich Library Support**
    - Python has a large standard library and thousands of external libraries.
    - Useful for tasks like math, web scraping, machine learning, and database access.
7. **Open Source**
    - It’s free to use and its source code is available to all.

Q. History and evolution of Python.

Ans:-

History and Evolution of Python

Python was created by Guido van Rossum, a Dutch programmer, in the late 1980s. He started working on it in December 1989 at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. His goal was to make a language that was easy to read and simple to use.

Year	**Version**	**Key Features / Events**
1989	-	Python was first conceived by Guido van Rossum.
1991	Python 0.9.0	First public release. Included functions, exception handling, and data types.
1994	Python 1.0	Full version with modules, functions, and standard library.
2000	Python 2.0	Introduced list comprehensions, garbage collection, Unicode support.
2008	Python 3.0	Major redesign to fix inconsistencies. Not backward compatible with Python 2.x.
2020	-	Python 2 reached end of life. Support officially ended.
Present	Python 3.x series	Active version with improved features like type hinting, async support, etc.

Q. Advantages of using Python over other programming languages.

Ans:-

1. **Easy to Learn and Use**
   - Python's syntax is close to English, making it beginner-friendly.
   - Reduces the time required to write and understand code.

     Example:

     print("Hello, Python!")
2. **Interpreted Language**
   - No need to compile. You can run the code directly.
   - Easier for debugging and testing.
3. **Extensive Standard Library**
   - Comes with built-in modules for tasks like math, file handling, networking, etc.
   - You don't have to write everything from scratch.
4. **Large Community Support**
   - Python has millions of users worldwide.
   - You can easily get help, find code samples, and ask questions on forums like StackOverflow.
5. **Cross-platform**
   - Write code once and run it anywhere (Windows, Mac, Linux).
6. **Supports Multiple Programming Paradigms**
   - Works with Object-Oriented, Procedural, and Functional programming styles.
7. **Wide Range of Applications**
   - Used in web development (Django, Flask), AI/ML (TensorFlow, PyTorch), data science (Pandas, NumPy), automation, game development, and more.
8. **Free and Open Source**
   - No cost to download or use.
   - Source code is available to everyone.
9. **Great for Rapid Development**
   - Less code, faster development.
   - Helps startups and companies build products quicker.

Q. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).
Ans:-

1. Visit the official Python website (python.org) and download the latest Python installer for your operating system (Windows/macOS/Linux).

2. Run the installer and make sure to check the "Add Python to PATH" option before clicking Install. This allows Python to be accessible from the command line.

3. After installation, verify it by opening the command prompt (or terminal) and typing python --version to see the installed Python version.

4. Download and install Visual Studio Code (VS Code) from its official website (code.visualstudio.com). It is a lightweight, free code editor.

5. Open VS Code and install the Python extension by Microsoft from the Extensions marketplace (search for "Python" and install it).

6. Open a new folder or workspace in VS Code where you want to create Python projects.

7. To run Python code, open a new file with .py extension, write your Python script, and save it.
8. Use the terminal inside VS Code (View -> Terminal) to run Python scripts by typing python filename.py.

9. You can also use the Run and Debug feature in VS Code to execute and debug Python programs easily.

10. Customize your Python interpreter in VS Code by clicking on the interpreter selection button at the bottom-left and selecting the installed Python path if there are multiple versions

Q. Writing and executing your first Python program.

Ans:-

Step 1: Open VS Code

- Click the VS Code icon on your desktop or start menu.

Step 2: Open a Folder (Workspace)

1. Click File → Open Folder...
2. Select a folder (or create one like C:\PythonProjects)
3. Click Select Folder

Step 3: Create a New Python File

1. Click the New File icon on the left Explorer panel
2. Name it: first.py (make sure it ends with .py)

Step 4: Write Code

Inside first.py, type this:

```
print("Hello Bhai")
```

Step 5: Run the Program

Option 1: Use Run Button

- Click the Run (button at the top-right of the editor)

Option 2: Use Right-Click

- Right-click anywhere in the editor
- Select "Run Python File in Terminal"

Step 6: See the Output

The output will appear in the terminal at the bottom:

```
Hello Bhai
```

2. Programming Style

Q. Understanding Python's PEP 8 guidelines.

Ans:-

- **Key PEP 8 Guidelines**
    - Use 4 spaces for indentation (never use tabs).
    - Keep your lines short—no more than 79 characters per line.
    - Name your variables, functions, and classes clearly. Classes use CapitalizedWords, variables and functions use lowercase_with_underscores.
    - Add extra blank lines to separate major code sections. Use 2 blank lines before a function/class, and single blank lines sparingly inside functions for clarity.
    - Compare to None using is or is not, not ==.
    - Put spaces around operators like + or =, but not right inside brackets (like [1, 2, 3] not [1, 2, 3]).
    - Write comments with good explanations, starting comments with a # and a space.
- **Why PEP 8 Is Important**
    - Following PEP 8 makes your code look professional, helps others understand it, and prevents mistakes when working in teams.
    - Keep your code consistent with spacing, naming, and indentation throughout the file.

Q. Indentation, comments, and naming conventions in Python

Ans:-

Indentation

- Python uses indentation (spaces at the start of a line) to define code blocks.
- Standard rule: 4 spaces for each indentation level.
- Without proper indentation, Python will give an error.

Comments

- Comments are used to explain code and make it easier to understand.
- Single-line comment starts with #.
- Multi-line explanations are written using """ triple quotes """.

Naming Conventions

- Variables and functions use snake_case (example: total_marks, get_name()).
- Class names use PascalCase (example: StudentInfo).
- Constants are written in UPPER_CASE (example: PI = 3.14).

Q. Writing readable and maintainable code.

Ans:-

Key Practices for Writing Readable and Maintainable Code

- Use meaningful variable and function names to describe their purpose.
- Write short functions that each do one thing well.
- Add comments only where logic is complex or non-obvious.
- Stick to consistent formatting and indentation throughout the code.
- Avoid magic numbers; use constants with descriptive names instead.
- Refactor repeated code into reusable functions or modules.
- Keep related code together—organize by functionality.
- Follow naming conventions and style guides specific to the language.
- Make sure error handling is clear and predictable.
- Write tests where possible to catch bugs and make future changes safer.

3. Core Python Concepts

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

Ans:-

Here is a simple explanation of common data types in programming:

- **Integers (int):** Whole numbers without decimals. Example: 5, -10.
- **Floats (float):** Numbers with decimal parts. Example: 3.14, -0.001.
- **Strings (str):** Ordered sequences of characters, used for text. Example: "hello".
- **Lists:** Ordered collections of items allowing duplicates. Items can be different types. Example: [1, "apple", 3.5].
- **Tuples:** Like lists but immutable (cannot be changed). Example: (2, "banana", 4.5).
- **Dictionaries (dict**): Collections of key-value pairs, unordered. Keys are unique. Example: {"name": "Ali", "age": 25}.
- **Sets:** Unordered collections of unique items, no duplicates allowed. Example: {1, 2, 3}.

Q. Python variables and memory allocation

Ans:-

Python variables and memory allocation:

- Variables in Python are names pointing to objects stored in memory, not the data itself.
- When you assign a value, Python creates an object in memory and the variable points to its address.
- Memory is managed in a private heap reserved for Python objects, and developers don't control allocation directly.
- Updating a variable creates a new object in memory; the variable is redirected to this new object.
- Python uses reference counting and garbage collection to free memory when objects are no longer used.
- Local variables (inside functions) are stored on the stack while objects themselves live in the heap.
- Python optimizes small objects with a specialized allocator called pymalloc for efficient memory use.
- Dynamic structures like lists grow in memory as needed by allocating new blocks and copying data over.
- Memory allocation in Python is mostly automatic, allowing programmers to focus on code logic rather than manual memory control.

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

Ans:-

Python operators explained:

- **Arithmetic operators:** Perform math operations.
    - **Examples:** + (add), - (subtract), * (multiply), / (divide), % (modulus), ** (power), // (floor division).
- **Comparison operators:** Compare values, return True or False.
    - **Examples:** == (equal), != (not equal), > (greater than), < (less than), >= (greater or equal), <= (less or equal).
- **Logical operators:** Combine boolean conditions.
    - **Examples:** and (both true), or (either true), not (negate condition).
- **Bitwise operators**: Work on binary digits of integers.
    - **Examples**: & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift) .

These operators are essential tools for performing calculations, making decisions, and manipulating data in Python code efficiently and clearly.

4. Conditional Statements

Q. Introduction to conditionalstatements: if, else, elif.

Ans:-

Introduction to Python conditional statements:

- **if statement:** Executes code block if a condition is True.
  - **Syntax:**

```
if condition:
  # code to run if condition is true
```

- **else statement:** Executes code block if the if condition is False.
  - **Syntax:**

```
if condition:
  # code if true
else:
  # code if false
```

- **elif (else if) statement:** Checks another condition if the previous if/elif was False.
  - **Allows multiple conditions to be tested in sequence.**
  - **Syntax:**

```
if condition1:
  # code if condition1 true
elif condition2:
  # code if condition2 true
else:
  # code if none true
```

- **Only one block among if, elif, else runs per execution.**
- **Useful for decision-making in programs to control flow based on conditions.**

Example:

```
x = 10
if x > 20:
   print("x is bigger than 20")
elif x == 20:
   print("x is 20")
else:
   print("x is less than 20")
```

Output: "x is less than 20"

Q.Nested if-else conditions.

Ans:-

Nested if-else conditions in Python:

- Nested if-else means having an if or else block inside another if or else block.
- It lets you check multiple layers of conditions step-by-step for more detailed decisions.
- **Syntax example:**

```
if condition1:
  if condition2:
    # runs if both condition1 and condition2 are true
  else:
    # runs if condition1 true but condition2 false
else:
  # runs if condition1 is false
```

- Useful when you want to perform additional checks only after a certain condition passes.
- **Example:**

```
num = 10
if num > 0:
  if num % 2 == 0:
    print("Positive and even")
  else:
    print("Positive but odd")
else:
  print("Not positive")
```

- Outputs "Positive and even" because 10 is greater than 0 and even.
- Keep proper indentation for readability and to avoid errors.
  This structure helps write clear programs with complex decision logic.

5.Conditional Statements

Q. Introduction to for and while loops.

Ans:-

Introduction to for and while loops in Python:

- **for loop:** Runs a block of code a fixed number of times, iterating over items in a sequence (like list, string, or range).
  - **Syntax:**
    ```
    for item in sequence:
      # code block
    ```
  - **Example:**
    ```
    for i in range(5):
      print(i)
    ```
  - **Prints 0 to 4, one per line.**

- **while loop**: Repeats a code block as long as a given condition is True.
  - **Syntax:**
    ```
    while condition:
      # code block
    ```
  - **Example:**
    ```
    count = 1
    while count <= 5:
      print(count)
      count += 1
    ```
  - **Prints 1 to 5, then stops.**
- Both loops help automate repetitive tasks in programs.
- Use for loops when the number of iterations is known or finite.
- Use while loops when you want to repeat until a condition changes dynamically.

This understanding is key for efficient and clear programming in Python.

Q. How loops work in Python.

Ans:-

How loops work in Python:

- A loop lets you run the same code repeatedly without rewriting it.

- **For loop:**
    - Iterates over each item in a sequence (like list, string, range).
    - Executes the indented code block for each item.
    - Moves to the next item until the sequence ends.

- **While loop:**
    - Repeatedly runs the code block as long as a condition is True.
    - Checks the condition before each iteration.
    - Stops when the condition becomes False.

- Both loops can use break to exit early or continue to skip to the next iteration.
- After the loop ends, execution continues with the code following the loop.
- Proper indentation is essential to define loop body.
- Loops help automate repetitive tasks and handle dynamic conditions efficiently in Python programs.

Understanding this is key to writing clean, effective Python code that repeats actions as needed.

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

Ans:-

Using loops with collections (like lists, tuples, etc.) in Python:

- **for loop is the most common way to iterate over each element in a list or tuple.**

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
   print(fruit)
```

*This prints each fruit, one per line.

- **You can also use for loops to access items by their index with range() and len().**

```
for i in range(len(fruits)):
   print(fruits[i])
```

*Prints each fruit by index.

- **while loops can also iterate using an index variable.**

```
i = 0
while i < len(fruits):
   print(fruits[i])
   i += 1
```

*Loops until all items are printed.

- **for loop can unpack tuples directly:**

```
pairs = [(1, 'a'), (2, 'b')]
```

for number, letter in pairs:

```
  print(number, letter)
```

Prints both items from each tuple easily.

- **List comprehensions offer a compact way to create new lists by looping:**

```
upper_fruits = [x.upper() for x in fruits]
print(upper_fruits)
```

Makes a new list with uppercase names.

- **These techniques work for any sequence type: lists, tuples, even strings.This keeps code short, readable, and efficient when working with collections in Python.**

Ved Jani

6. Generators and Iterators

Q. Understanding how generators work in Python.

Ans:-

Understanding how generators work in Python:

- Generators are special functions that produce values one at a time using the yield keyword instead of returning all values at once.
- When called, a generator function returns a generator object but does not execute its code immediately.
- Each time the generator's next() method is called (such as by a loop), it runs until it hits a yield, returns that value, and pauses its state.
- The generator resumes from where it paused on the next next() call, saving memory because values are generated on-the-fly, not stored all at once.
- This is useful for large data sequences or infinite streams where storing all values is inefficient.
- **Example:**

```
def count_up_to(n):
   count = 1
   while count <= n:
      yield count
      count += 1

for num in count_up_to(3):
   print(num)
```

Output: 1 2 3, generating values sequentially without storing the whole list.

- Generators can be created with generator expressions too, a concise syntax similar to list comprehensions but using parentheses.
- They help write memory-efficient, clean, and lazy-evaluated code for sequences and data streams.

Q. Difference between yield and return.

Ans:-

Difference between yield and return in Python:

- **return:**
  - Ends the function immediately and sends a value back to the caller.
  - Function cannot resume after return; it runs once per call.
  - Use return when you want to send a single complete result.
  - Code after return in a function does not run.
- **yield:**
  - Pauses the function, sends a value back, but saves state to resume later.
  - Makes the function a generator, producing a sequence of values over time.
  - Use yield when you want to produce multiple values lazily, one at a time.
  - Code after yield can run when function resumes on next call.
- yield is memory efficient for large or infinite sequence generation; return sends all results at once, which can be costly.
- **Example difference:**

```
def gen_nums():
  for i in range(3):
    yield i  # produces 0, then pauses; continues on next call

def ret_nums():
  nums = []
  for i in range(3):
    nums.append(i)
  return nums  # returns full list at once
```

- yield supports multiple "returns" by pausing and resuming; return only finishes once per call.

Q. Understanding iterators and creating custom iterators.

Ans:-

Understanding iterators and creating custom iterators in Python:

- **Iterator:** An object that lets you traverse (access one item at a time) through a collection like lists, tuples, etc.
- **Iterators implement two key methods:**
  - **__iter__()** returns the iterator object itself**.**
  - **__next__()** returns the next item; raises StopIteration when no more items.
- Using iter() on an iterable creates an iterator; next() fetches items one by one.
- **Custom iterator creation in a class involves:**
  - **Defining __iter__()** to return self or a new iterator.
  - **Defining __next__()** to control what item comes next or stop iteration.
- **Example custom iterator generating numbers 1 to 5:**

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    if self.a <= 5:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

my_iter = iter(MyNumbers())
for num in my_iter:
  print(num)
```

- This prints numbers 1 to 5, showing how you can control iteration behavior.

- Custom iterators are useful for sequences requiring special iteration logic, infinite sequences, or lazy evaluation.

This understanding helps with writing flexible, memory-efficient code managing data traversal in Python.

7. Functions and Methods

Q. Defining and calling functions in Python.

Ans:-

Defining and calling functions in Python:

- Define a function using the def keyword, followed by the function name and parentheses with optional parameters, ending with a colon.
- Inside the function, indent the code block that runs when the function is called.
- Optionally, use return to send back a value from the function.
- **Example:**

```
def greet(name):
   print(f"Hello, {name}!")
```

- **Call the function by writing its name with arguments in parentheses:**

```
greet("Alice")  # Output: Hello, Alice!
```

- Functions help avoid repeating code by wrapping reusable logic in one place.
- **Functions can have:**
    - No parameters.
    - Positional or keyword parameters.
    - Default values for parameters.
    - Variable-length argument lists.
- **Example with return:**

```
def add(a, b):
   return a + b
result = add(3, 5)
print(result)                    # Output: 8
```

- Clear function names and simple parameters improve code readability and maintainability.

Q. Function arguments (positional, keyword, default)

Ans:-

Python function arguments:

- **Positional arguments:**
    - Passed in order, matching function parameters.
    - **Example:** def add(a, b): return a + b
    - **Call as:** add(3, 5).
- **Keyword arguments:**
    - Named explicitly with parameter=value.
    - Can be used in any order**.**
    - **Example: add(b=4, a=2).**
- **Default arguments:**
    - Have preset values if no argument is given.
    - **Example:** def greet(name, message="Hello"): print(message, name)
    - **Call as:** greet("Alice") (uses default). Or greet("Alice", "Hi") (overrides default).
- **Arbitrary arguments:**
    - ***args:** Accepts any number of positional arguments as a tuple.
        - **Example:**

```
def show_all(*args):
   for arg in args:
      print(arg)
```

    - ****kwargs:** Accepts any number of keyword arguments as a dictionary.
        - **Example:**

```
def show_info(**kwargs):
   for key, value in kwargs.items():
      print(f"{key}: {value}")
```

- These argument types make functions flexible and reusable, improving code clarity and flexibility.

Q. Scope of variables in Python.

Ans:-

Understanding scope of variables in Python:

- Scope defines where a variable can be accessed or used in the code.
- **Python follows the LEGB rule to find variable names:**
  - **Local (L):** Inside the current function or block.
  - **Enclosing (E):** Inside any outer (enclosing) functions for nested functions.
  - **Global (G):** At the top level of the module or script.
  - **Built-in (B):** Python's built-in names like print, len.
- Variables defined inside a function have local scope and are not accessible outside it.
- Variables defined outside functions are global and accessible throughout the module.
- global keyword lets you modify a global variable inside a function.
- nonlocal keyword lets you modify a variable in the enclosing outer function scope.
- Blocks like if, for, while do not create new scopes in Python.
- **Example:**

```
x = 10  # global variable
def func():
   x = 5  # local variable, different from global
   print(x)
func()  # prints 5
print(x)  # prints 10
```

- Understanding scope avoids errors like NameError and unexpected variable shadowing.

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

Ans:-

Built-in methods for strings, lists, etc. in Python:

String built-in methods:

- **upper():** Converts string to uppercase.
- **lower():** Converts string to lowercase.
- **strip():** Removes leading/trailing whitespace.
- **split():** Splits string into list based on separator.
- **join():** Joins list of strings into one string with separator.
- **replace(old, new):** Replaces occurrences of old substring with new.
- **find(sub):** Returns index of substring or -1 if not found.
- **startswith(prefix):** Checks if string starts with prefix.
- **endswith(suffix):** Checks if string ends with suffix.
- **capitalize():** Capitalizes first character.
- **title():** Capitalizes first letter of each word.
- **count(sub):** Counts occurrences of substring.

List built-in methods:

- **append(item):** Adds item to end.
- **extend(iterable):** Adds all items from iterable.
- **insert(index, item):** Inserts item at index.
- **remove(item):** Removes first matching item.
- **pop(index):** Removes and returns item at index (default last).
- **index(item):** Returns index of first matching item.
- **count(item):** Counts occurrences of item.
- **sort():** Sorts list in place.
- **reverse():** Reverses list order.
- **clear():** Removes all items.

These built-in methods help efficiently manipulate and query strings and lists without extra coding.

8. Control Statements (Break, Continue, Pass)

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

Ans:-

Understanding break, continue, and pass in Python loops:

- **break:**
    - Exits the loop immediately when executed.
    - Useful to stop looping early if a condition is met.
    - Only exits the innermost loop where it's used.
    - **Example:**

```
for i in range(5):
  if i == 3:
    break
  print(i)
```

Output: 0 1 2 (loop stops at 3)

- **continue:**
    - Skips the rest of the current iteration and moves to the next iteration.
    - Useful to ignore certain items but keep looping.
    - **Example:**

```
for i in range(5):
  if i == 3:
    continue
  print(i)
```

Output: 0 1 2 4 (skips 3)

- **pass:**
    - Does nothing, acts as a placeholder for code to be written later.
    - Useful in loops, functions, conditionals where code is syntactically required but not ready.

    - **Example:**

```
for i in range(5):
   if i == 3:
      pass  # placeholder, does nothing
   print(i)
```

Output: 0 1 2 3 4 (all printed)

These statements give precise control to manage loop execution flow and structure code cleanly.

9. String Manipulation

Q. Understanding how to access and manipulate strings.

Ans:-

Understanding how to access and manipulate strings in Python:

- **Strings are sequences of characters, supporting indexing and slicing:**
  - Access single character: s[0] gets first character.
  - Slice substring: s[1:4] gets characters from index 1 to 3.
- **Common string manipulation methods:**
  - **upper():** Convert all characters to uppercase.
  - **lower():** Convert all characters to lowercase.
  - **strip():** Remove whitespace from start and end.
  - **replace(old, new):** Replace occurrences of substring.
  - **split(separator):** Split string into list by separator.
  - **join(iterable):** Join items from iterable into a string.
  - **find(sub):** Find substring position or -1 if not found.
  - **startswith(prefix), endswith(suffix):** Check start or end of string.
  - **capitalize():** Capitalize first letter, rest lower.
  - **title():** Capitalize first letter of each word.
- Strings are immutable; methods return new strings, original stays unchanged.
- You can concatenate strings with + and repeat with *.
- **Example:**

```
s = " hello world "
print(s.strip().upper())  # "HELLO WORLD"
print(s.replace("world", "Python"))  # " hello Python "
print(s.split())  # ["hello", "world"]
```

- These operations make string handling flexible and powerful in Python.

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

Ans:-

Basic operations with strings in Python:

- **Concatenation:**
    - Use + operator to join strings.
    - **Example:** "Hello " + "World" → "Hello World".
    - Use " ".join(list_of_strings) to join many strings efficiently with separator.
- **Repetition:**
    - Use * operator to repeat strings.
    - **Example:** "Hi! " * 3 → "Hi! Hi! Hi! ".
- **Common string methods:**
    - **upper():** Converts string to uppercase. "abc".upper() → "ABC".
    - **lower():** Converts string to lowercase. "ABC".lower() → "abc".
    - **strip():** Removes whitespace from start and end. " hello ".strip() → "hello".
    - **replace(old, new):** Replaces substring. "2020".replace("0", "1") → "2121".
    - **split(sep**): Splits string into a list by separator. "a,b,c".split(",") → ["a","b","c"**].**
- Using these operations, strings can be combined, repeated, and transformed easily for text processing tasks.

Q. String slicing.

Ans:-

String slicing in Python explained simply for assignments:

- **Slicing extracts a substring from a string using the syntax:**

$$s[start:end:step]$$

- **Parameters:**
    - **start:** index to begin (inclusive, default 0)
    - **end:** index to stop (exclusive, default end of string)
    - **step:** step interval between indices (default 1)
- **Examples:**
    - **s =** "Hello, World!"
    - **s[0:5]** → "Hello" (characters from index 0 to 4)
    - **s[7:]** → "World!" (from index 7 to end)
    - **s[:5]** → "Hello" (from start to index 4)
    - **s[::2]** → "Hlo ol!" (every 2nd character)
    - **s[-6:-1]** → "World" (negative indices count from the end)
    - **s[::-1]** → "!dlroW ,olleH" (reverses the string)
- The end index is exclusive, so the character at end is not included.
- Negative step slices backwards.
- Slicing is a powerful way to quickly access substrings or reverse strings without loops.

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

Q. How functional programming works in Python

Ans:-

Functional programming in Python:

- **Functional programming (FP) is a paradigm focusing on writing programs using pure functions that:**
  - Always produce the same output for the same inputs.
  - Do not cause side effects (no modifying external state).
- **Key FP concepts supported in Python:**
  - First-class functions: Functions can be assigned to variables, passed as arguments, and returned from other functions.
  - Higher-order functions: Functions that take functions as arguments or return them.
  - Immutability: Data is not changed but copied or transformed.
  - Recursion: Replacing loops by recursive function calls.
- **Python provides functional tools like:**
  - lambda to create anonymous functions.
  - map() to apply a function to all items in an iterable.
  - filter() to select items that satisfy a function.
  - reduce() (from functools) to combine items using a function.
- **Example of map():**

```
numbers = [1, 2, 3]
doubled = list(map(lambda x: x * 2, numbers))  # [2, 4, 6]
```

- Functional style leads to concise, modular, and predictable code.
- Python is multi-paradigm and lets you mix functional programming with other styles.

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

Ans:-

Using map(), reduce(), and filter() functions for processing data in Python:

- **map():**
  - Applies a function to each item in an iterable (like list) and returns a map object.
  - **Example:**
    ```
    numbers = [1, 2, 3]
    doubled = list(map(lambda x: x * 2, numbers))  # [2, 4, 6]
    ```
- **filter():**
  - Filters items in an iterable based on a function that returns True or False.
  - Only items returning True are included.
  - **Example:**
    ```
    numbers = [1, 2, 3, 4]
    evens = list(filter(lambda x: x % 2 == 0, numbers))  # [2, 4]
    ```
- **reduce():**
  - Applies a function cumulatively to reduce iterable to a single value.
  - Needs to be imported from functools module.
  - **Example:**
    ```
    from functools import reduce
    numbers = [1, 2, 3, 4]
    product = reduce(lambda x, y: x * y, numbers)  # 24
    ```
- These functions enable concise, functional-style data processing without explicit loops.
- They can be combined to, for example, filter data, transform it, then aggregate results.
- Benefits: more expressive, cleaner code, suitable for functional programming style.

Q. Introduction to closures and decorators.

Ans:-

Introduction to closures and decorators in Python:

- **Closures:**
  - A closure is a nested function that remembers variables from its enclosing (outer) function even after the outer function has finished executing.
  - It "closes" over the environment, capturing local variables so the inner function can use them later.
  - **Example:**

```
def outer(x):
   def inner(y):
      return x + y  # x is remembered from outer
   return inner

add5 = outer(5)
print(add5(3))  # Outputs 8
```

  - Closures enable data encapsulation and function factories.
- **Decorators:**
  - Decorators are functions that modify or enhance other functions without changing their code directly.
  - They take a function as input, wrap it with additional behavior, and return a new function.
  - Often implemented using closures.
  - Syntax uses @decorator_name above the target function.

- **Example:**

```
def decorator(func):
  def wrapper():
    print("Before function runs")
    func()
    print("After function runs")
  return wrapper

@decorator
def say_hello():
  print("Hello!")

say_hello()
```

Output:

```
Before function runs
Hello!
After function runs
```

- Both closures and decorators help write modular, reusable, and clean code.
- Closures focus on preserving state; decorators focus on extending behavior.

This concise introduction covers basics needed for assignments on these advanced Python concepts.