Module 13 : Python Fundamentals

1.Introduction To Python

1.Introduction to Python and its Features

Ans :

Python is a high-level, general-purpose programming language created by Guido van Rossum and first released in 1991. Python is famous for its friendly syntax, code readability, and versatility, which means you can use it for many kinds of problems such as web development, data analysis, automation, software development, and more.

Why Learn Python?

- Beginner-friendly: Python has simple, clean syntax similar to English, which makes it easy for beginners to read and write code.
- Popular and in-demand: It's widely used in tech, data science, AI, and finance. Learning Python can open doors to many career paths.
- Versatile: Python can be used for small scripts or complex applications like websites, games, or machine learning projects. You can write code in different styles: procedural, object-oriented, or functional.

Key Features of Python :

1. Easy to Learn and Read

Python's syntax is straightforward and emphasizes readability. Compared to languages like C++ or Java, Python uses fewer symbols and organizes its code blocks using indentation (spaces), not curly braces. This makes it visually cleaner and reduces common code errors.

2. Dynamically Typed

You don't have to declare variable types. Python figures out variable types when you run the program. For example:

```
python
```

```
x = 10  # This is an integer
y = "hello"  # This is a string
```

This flexibility allows you to write code quickly, but also means you have to be careful to avoid type-related mistakes.

3. Interpreted Language

Python runs each line of code one by one, instead of first compiling everything. If your code has an error, Python stops at that line and shows you the problem. This helps with debugging and experimentation.

4. Multi-Paradigm

Python doesn't force you into one style:

- You can write quick, simple scripts (procedural)
- Build programs using classes and objects (object-oriented)
- Or use functions as first-class objects (functional programming)

5. Extensive Standard Library and Ecosystem

Python comes with a large collection of modules (the "standard library")—you can use them for math, file operations, web communication, etc. Plus, there are thousands of third-party libraries for scientific computing (NumPy, pandas), machine learning (scikit-learn, TensorFlow), web development (Django, Flask), and much more.

6. Cross-Platform and Open Source

You can run Python on Windows, Mac, Linux, or even on smaller devices like Raspberry Pi. It's free to use, share, and modify—making it popular among hobbyists and professionals alike.

7. Community Support

Python has a huge, active community. You'll find plenty of tutorials, forums, and resources to help you learn or solve problems you run into.

2.History Of Python

Ans:

Python was created by Guido van Rossum in December 1989 at the Centrum Wiskunde & Informatica (CWI) in the Netherlands, initially as a hobby project to keep him occupied during the holidays. The language was heavily inspired by the ABC programming language, aiming to improve on ABC’s limitations while keeping its strengths, especially simple syntax and readability. The name "Python" was chosen after the British comedy show "Monty Python’s Flying Circus," reflecting the language’s philosophy of making programming fun and approachable.

The first public release, Python 0.9.0, was launched in February 1991. Even this version included features that are core to Python today—like exception handling, functions, the module system, classes with inheritance, and key data types such as lists and dictionaries. Discussion around Python development quickly grew, with an early user forum (comp.lang.python) established in 1994.

Major Milestones in Python's Evolution

- **Python 1.0 (1994)**: Introduced features like lambda functions, map, filter, and reduce, as well as error-handling mechanisms and basic support for object-oriented programming.
- **Python 2.0 (2000)**: Marked a shift to a more community-driven development process. Added major enhancements such as list comprehensions, a garbage collector, reference counting for better memory management, Unicode support, and many other features. Python 2 became the industry standard for many years, remaining in use until its support ended in 2020.
- **Python 3.0 (2008)**: A major overhaul that was intentionally not backward compatible, meant to fix inconsistencies and modernize the language. Added better Unicode support, new syntax, and cleaner integer division rules. This version also included tools to help migrate old Python 2 code to Python 3. Over time, Python 3 became the default version, and almost all new projects now use it.

After Guido van Rossum stepped down as Python's "Benevolent Dictator for Life" in 2018, the leadership transitioned to a community-elected steering council, ensuring continued evolution of the language with open, transparent development.

Current State

As of August 2025, Python remains one of the world's most popular programming languages, reaching version 3.13.6 with ongoing security and bug-fix updates. Its straightforward syntax, open-source model, and robust community support ensure that Python continues to evolve and maintain its relevance in a fast-changing software landscape.

Python's history is a testament to its enduring appeal—rooted in simplicity, flexibility, and the idea that code should be both powerful and accessible to all programmers.

Python Version Milestones :

Version	Release Year	Major Features and Milestones
0.9.0	1991	First public release. Introduced classes, inheritance, exception handling, functions, modules, and core data types like lists and dictionaries.
1.0	1994	Added lambda/functional programming features (lambda, map, filter, reduce), exception handling improvements, and started the standard module system.
1.5	1997	Expanded standard library, initial Unicode support, and improved usability for internationalization.
2.0	2000	Introduced list comprehensions, full Unicode support, garbage collection, augmented assignment, and major ecosystem/library growth.
2.7	2010	Last major Python 2 version, added set literals, ordered dictionaries, improved string formatting, and many Python 3 forward-compatibility features.
3.0	2008	Major redesign (not backward compatible). Introduced print as a function, new integer division, improved Unicode (all strings are Unicode), removed deprecated features.

3.4	2014	Introduced asyncio (asynchronous programming), pathlib (filesystem path library), and enumeration support.
3.5	2015	Added type hints (function signatures), async/await syntax for coroutines, and further improvements to async programming.
3.12	2023	Enhanced performance, improved error messages, expanded f-string formatting, and type parameter syntax.
3.13/3.14	2024-2025	Added experimental Just-In-Time (JIT) compiler, free threaded mode, deferred annotation evaluation, template strings, and better error messages.

These milestones reflect Python’s evolution from a beginner-friendly scripting language to a leading choice for web development, automation, data science, and more, with each version adding new capabilities and modern features while maintaining code readability and simplicity.

3.Advantages of using Python over other programming languages.

Ans : Python offers several important advantages over other programming languages, making it a top choice for developers in many fields. Its popularity continues to grow due to its ease of use, flexibility, and powerful features.

Readability and Simplicity

Python code is designed to be easy to read and write, with a clear, human-friendly syntax. Programs are typically shorter and more understandable compared to other languages like C++ or Java, allowing faster development and easier collaboration among teams.

Versatility and Multiparadigm Support

Python supports multiple programming styles, including procedural, object-oriented, and functional programming. This allows developers to pick the approach best suited to the problem at hand. Python is used for web development, automation, data analysis, artificial intelligence, software testing, and education.

Extensive Libraries and Ecosystem

Python's vast standard library and thousands of third-party packages simplify tasks ranging from web development (Django, Flask) and data science (NumPy, pandas) to machine learning (TensorFlow, scikit-learn) and automation (Selenium).

Platform Independence

Python is cross-platform and works on Windows, Mac, Linux, and even mobile devices without code modification, which means projects can be easily deployed across different systems.

Community and Support

With a huge global community, Python offers great documentation, hundreds of forums and tutorials, and rapid help for troubleshooting. Its open-source nature encourages continued growth, improvement, and user contributions.

Integration and Extensibility

Python can easily integrate with languages like C, C++, and Java, and with various technologies and APIs. This makes it pertinent for projects that combine legacy code with modern features.

Rapid Development and Prototyping

Python's simplicity and dynamism allow developers to go from idea to working prototype quickly, facilitating testing, experimentation, and iteration in both academic and industrial settings.

In summary, Python's major advantages—readability, flexibility, rich ecosystem, platform independence, and vibrant community—make it an excellent choice for beginners and professionals tackling almost any modern programming challenge.

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

Ans:

To get started with Python, you need to install the Python interpreter and set up a development environment. Common tools for writing and running Python code include Anaconda (especially for data science), PyCharm, and Visual Studio Code (VS Code).

Installing Python

1. Official Website Download
    - Go to the Python official website (python.org).
    - Download the latest stable version for your operating system (Windows, Mac, or Linux).
    - Run the installer and ensure the option “Add Python to PATH” is checked before clicking “Install Now”.
    - After installation, open your terminal or command prompt, type `python --version` to verify the installation.
2. Anaconda Distribution
    - Anaconda is recommended for data science and scientific programming as it comes with Python and many useful libraries.
    - Visit the Anaconda website (anaconda.com) and download the installer for your OS.
    - Install Anaconda, which will also set up its package manager, Conda.
    - After installation, use the “Anaconda Navigator” GUI or launch the "Anaconda Prompt" for command-line control.
    - Create new environments with `conda create -n myenv python=3.12` and activate them using `conda activate myenv`.

Setting Up Your Development Environment :

1. PyCharm

- Download Community (free) or Professional (paid) version from jetbrains.com/pycharm.
- Install and launch PyCharm.
- Create a new Python project or open an existing one.
- Set the Python interpreter in Settings/Preferences > Project > Python Interpreter.
- PyCharm offers powerful code completion, debugging, and project management.

2. Visual Studio Code (VS Code)

- Download from code.visualstudio.com.
- Install and launch VS Code.
- Go to Extensions and search for "Python", then install the official Python extension.
- Open your project folder or create a new file with `.py` extension.
- Set your Python interpreter by clicking on the interpreter path at the bottom left.
- VS Code provides smart code completion, integrated terminal, linting, and debugging tools.

3. Anaconda Navigator (for Data Science)

- Open “Anaconda Navigator” after Anaconda installation.
- Use it to launch graphical tools like Jupyter Notebook or Spyder.
- Create and manage virtual environments, install libraries, and run code interactively.

- Always use virtual environments (via `venv`, `conda`, or PyCharm’s environment manager) to keep project dependencies organized and avoid conflicts.
- Try starting with VS Code if undecided, as it’s lightweight, extensible, and beginner-friendly.
- For data science or machine learning, Anaconda and Jupyter Notebooks are excellent starting points.

5. Writing and executing your first Python program.

Ans:

Writing Your First Python Program :

1. Open Your Development Environment
    - This can be a simple text editor like Notepad or a code editor like Visual Studio Code or PyCharm.
    - Alternatively, you can use an interactive environment like Jupyter Notebook (with Anaconda).
2. Write the Code

- Type the classic introductory program:

```
python

print("Hello, World!")
```

- This code uses the `print` function to display the text "Hello, World!" on the screen.

3. Save the File
    - Save the file with a `.py` extension, for example `hello.py`.
    - Make sure you save it in a folder where you can easily locate it.

Executing Your Python Program:

Using Command Line or Terminal:

1. Open your terminal (Command Prompt on Windows, Terminal on Mac/Linux).
2. Navigate to the directory where your file is saved using the `cd` command.
3. Run the program by typing:

```
python hello.py
```

4. You should see the output:

```
Hello, World!
```

Using an IDE or Code Editor:

- In PyCharm or VS Code, open your file and look for a “Run” button or use the terminal integrated into the editor to run the command above.
- In Jupyter Notebook, type `print("Hello, World!")` in a cell and run the cell to see the output directly.

2. Programming Style

1.Understanding Python’s PEP 8 guidelines.

Ans:

PEP 8 is the official style guide for Python code, created to promote readability, consistency, and maintainability of Python programs. It provides detailed guidelines for writing clean, well-structured Python code, which is widely adopted across the Python community.

Key Principles of PEP 8

- Indentation: Use 4 spaces per indentation level; no tabs.
- Line Length: Limit lines to 79 characters to improve readability and allow multiple files to be open side-by-side.
- Blank Lines: Use blank lines to separate functions, classes, and blocks of code.
- Imports: Imports should be on separate lines and grouped (standard libraries, third-party, local).
- Whitespace: Avoid unnecessary whitespace; for example:
    - No spaces inside parentheses, brackets, or braces.
    - No space before commas, colons, or semicolons.
    - Choose to break lines before binary operators to enhance readability.
- Naming Conventions: Use descriptive, lowercase names with underscores for variables and functions; CamelCase for classes.
- Comments and Docstrings: Use comments and docstrings to explain code functionality clearly.
- Consistent Style: Follow the standards throughout your code for consistency.

Practice and Tools

Following PEP 8 improves collaboration and code quality, especially in teams. Automated tools like `flake8`, `black`, and IDE integrations can help enforce PEP 8 compliance.

PEP 8 is essential for writing Python code that is clean, readable, and maintainable, making it a cornerstone of professional Python development.

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

❖ Indentation in Python

Indentation is fundamental in Python and is used to define the structure and flow of the code. Unlike many other languages that use braces `{}` to mark blocks of code, Python uses indentation levels:

- Use 4 spaces per indentation level, as recommended by PEP 8.
- Never mix tabs and spaces in the same file; Python 3 will raise an error if they are mixed.
- Indentation groups statements together, such as the body of functions, loops, and conditionals.
- Proper indentation is crucial because Python uses it to determine which statements belong to which blocks.

Example:

```
python

    x = 10
    if x > 5:
        print("x is greater than 5")  # Indented by 4 spaces, inside
the if block
```

Comments in Python

Comments are used to explain and clarify code. They are ignored by the interpreter during execution.

- Single-line comments start with `#` and should be used to explain complex or unclear sections.
- Avoid obvious comments that state what the code already clearly does.
- Docstrings are multi-line string literals enclosed in triple quotes `"""` and are used to document modules, functions, classes, and methods.
- Good commenting improves code readability and maintainability.

Example of a single-line comment:

```
python

    # This function adds two numbers
    def add(a, b):
        return a + b
```

Example of a docstring:

```
    def add(a, b):
        """
        Return the sum of a and b.
        """
        return a + b
```

❖ Naming Conventions in Python

Naming conventions help make code more readable and consistent.

- Use snake_case (all lowercase with words separated by underscores) for variable names, function names, and methods.
    - Example: `total_sum`, `calculate_area()`

- Use CamelCase (capitalize first letter of each word, no underscores) for class names.
    - Example: `MyClass`, `DataAnalyzer`

- Constants should be in all uppercase letters with underscores separating words.
    - Example: `MAX_SIZE = 100`

- Avoid using names that are too short or cryptic; names should be descriptive but concise.

3.Writing readable and maintainable code.

Ans:

Writing readable and maintainable Python code is essential for collaboration, debugging, and long-term project success. It ensures that others (and your future self) can understand and modify the code easily without introducing errors.

Key Principles for Readability and Maintainability

1. Follow PEP 8 Style Guide
    - Use consistent indentation (4 spaces).
    - Limit line length to 79 characters.
    - Use meaningful variable, function, and class names following naming conventions.
    - Add comments and docstrings where necessary to explain complex logic or the purpose of code sections.

2. Write Clear and Simple Code
    - Prefer readability over cleverness.
    - Avoid deeply nested code and long functions; break them into smaller, modular functions.
    - Use descriptive names that convey intent clearly.
    - Use whitespace effectively to separate code blocks and improve visual clarity.

3. Document Your Code
    - Use docstrings for modules, classes, and functions to describe their purpose, inputs, outputs, and side effects.
    - Keep comments up-to-date and focused on explaining “why” rather than “what” (which should be apparent from the code).

4. Use Consistent Formatting
    - Stick to one style for quotes, spacing, and parentheses.
    - Use automated tools like `black` or `flake8` to enforce style consistency.

5. Write Modular and Reusable Code
    - Create functions or classes that do one thing well.
    - Avoid duplicating code—reuse functions and classes to centralize logic.
    - Separate concerns by organizing code into modules and packages logically.

6. Handle Errors Gracefully
    - Use exceptions and error handling to make your program robust.
    - Provide informative error messages to ease debugging.

7. Test Your Code
    - Write unit tests to verify functionality.
    - Use testing frameworks like `unittest` or `pytest`.
    - Testing improves confidence and prevents regressions.

---

3. Core Python Concepts

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

- **Python Data Types :**

---

1. Integers (`int`)

- Integers represent whole numbers without decimals, such as `-3`, `0`, `42`, or `123456`.
- They can be positive, negative, or zero, and Python supports arbitrarily large integers limited by available memory.
- Integers are immutable, meaning their value cannot be changed once created.

Example:

```
x = 10
print(type(x))  # Output: <class 'int'>
```

- Integers are primarily used for counting, indexing, and arithmetic operations involving whole numbers.

---

2. Floating-Point Numbers (`float`)

- Floats represent real numbers with fractional parts, like `3.14`, `-0.001`, or `2.0`.
- They use double-precision (64-bit) IEEE 754 format internally.
- Floats are also immutable.

Example:

```
python

pi = 3.14159
temperature = -5.2
print(type(pi))  # Output: <class 'float'>
```

- Floats are used where fractions or decimal precision are needed, such as scientific calculations or measurements.

3. Strings (`str`)

- Strings are sequences of Unicode characters used to represent text.
- They are immutable and support operations like concatenation, slicing, and formatting.

Example:

```
name = "Python"
print(type(name))  # Output: <class 'str'>
print(name[0])     # Output: 'P'
```

- Strings can store any textual data such as words, sentences, or longer documents.

---

4. Lists (`list`)

- Lists are ordered, mutable collections of items.
- They can hold mixed data types but often contain items of the same type.
- Lists can be changed by adding, removing, or replacing elements.

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("date")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']
```

- Lists are great for collections that may need modification, such as queues, stacks, or grouped data.

5. Tuples (`tuple`)

- Tuples are ordered, immutable sequences.
- Once created, their content cannot be changed.
- Useful for fixed collections of heterogeneous data.

Example:

```
point = (10, 20)
print(type(point))  # Output: <class 'tuple'>
```

- Tuples are often used for function returns, fixed groups of related items, or dictionary keys.

6. Dictionaries (`dict`)

- Dictionaries store key-value pairs, where keys must be unique and immutable.
- They provide fast lookup, insertion, and deletion by keys.
- Useful for representing structured data or mappings.

Example:

```
python

person = {"name": "Alice", "age": 30}
print(person["name"])  # Output: Alice
```

Dictionaries are essential in most Python programs for associative arrays, JSON-like data, and configurations.

---

7. Sets (`set`)

- Sets are unordered collections of unique elements.
- Useful to eliminate duplicates and perform mathematical set operations like union, intersection.

Example:

```
unique_numbers = {1, 2, 3, 3, 4}
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

- Sets provide efficient membership testing and operations on groups of unique items.

2.Python Variable And Memory Allocation

Ans:

In Python, variables are fundamental building blocks used to store and manipulate data. Unlike many programming languages, Python does not require explicit declaration of variables with specific data types. Instead, variables are created automatically at the moment you assign a value to them. This makes Python simple and flexible for beginners and advanced users alike.

- ❖ **How Variables Are Created in Python**

- No explicit declaration: You don't declare a variable with a keyword like `var` or `int`. Instead, you simply assign a value to a name.
- Assignment operator: The equals sign `=` is used to assign values to variables. For example:

```
age = 25
name = "Alice"
```

- These lines create variables `age` and `name` and assign corresponding values.
- Automatic type determination: Python infers the data type based on the value assigned. For example:

```
x = 10          # integer
x = "Hello"     # now x is a string
```

- The variable `x` dynamically changes its type based on the assigned value.

Important Characteristics of Python Variables

- Dynamic typing: Variables can hold different types during the execution of the program, and their types can change over time.
- Case-sensitive: Variable names are case-sensitive, so `age`, `Age`, and `AGE` are considered different variables.
- Good naming practice: Use meaningful names following the snake_case convention (e.g., `user_age`, `total_score`) for clarity and maintainability.

- No reserved keywords: Variable names should not be Python keywords like `for`, `if`, `while`, etc.

Variable Naming Rules :

- Start with a letter or an underscore `_`.
- Can include letters, digits, and underscores.
- Cannot start with a digit.
- Should be descriptive to clarify purpose.

Examples:

```
valid_variable = 42
_user_name = "John"
x1 = 5
```

- Changing Variable Types and Reassignment
- Variables in Python are mutable in terms of reassignment:

```
python

x = 10        # x is an integer
x = “Ten"     # now x is a string
```

This flexibility allows you to reuse variable names for different data types, which contributes to rapid development.

❖ Memory Allocation Mechanism in Python:

1. Heap and Stack Memory
    - Python allocates objects and data structures on a private heap, which is a region of memory reserved for dynamic allocation.
    - Variables in Python are references (or pointers) to objects stored on the heap. These references are kept in the stack memory.

2. Dynamic Allocation
    - When you create an object (like an integer, list, or dictionary), Python dynamically allocates memory for that object on the heap.
    - The amount of memory allocated depends on the nature and size of the object.
    - Python is dynamically typed, so you don't declare variable sizes upfront; memory is allocated at runtime.
3. Memory Optimizations
    - Python uses object interning for some immutable objects (e.g., small integers from -5 to 256, some strings). This means these objects are stored only once in memory and reused, saving space and improving performance.
4. Memory Managers and Allocators
    - Python's memory manager includes layers of allocators:
        - A general-purpose allocator interacts with the operating system’s memory manager.
        - A raw memory allocator manages memory blocks from the operating system.
        - An object allocator (called `pymalloc`) manages small objects up to 512 bytes efficiently by preallocating memory blocks in pools and arenas.
    - Larger objects request memory directly from the raw allocator.

5. Reference Counting
    - Python maintains a reference count for each object, representing how many references point to it.
    - When the reference count drops to zero (no active references), Python deallocates the object's memory automatically.

6. Garbage Collection
    - Besides reference counting, Python uses a cyclic garbage collector to clean up objects involved in reference cycles that reference counting alone can't reclaim.

Example: Interning and Reference Counting:

python

```
a = 10
b = 10
print(id(a) == id(b))  # True, both refer to same memory location
```

Python

```
c = [1, 2, 3]
d = c
print(id(c) == id(d))  # True, d points to the same list object
```

3.Python Operators :

1. Arithmetic Operators:

These operators perform mathematical calculations on numeric values.

Operator	Name	Description	Example
+	Addition	Adds two operands	5 + 2 = 7
-	Subtraction	Subtracts second operand from first	4 - 2 = 2
*	Multiplication	Multiplies two operands	2 * 3 = 6
/	Division	Divides first operand by second (float)	4 / 2 = 2.0
//	Floor Division	Divides and floors the result	7 // 2 = 3
%	Modulus	Returns remainder of division	7 % 2 = 1
**	Exponentiation	Raises first operand to the power of second	3 ** 2 = 9

Example:

```python
a, b = 7, 2
print('Sum:', a + b)           # 9
print('Subtraction:', a - b)   # 5
print('Multiplication:', a * b) # 14
print('Division:', a / b)       # 3.5
print('Floor Division:', a // b)# 3
print('Modulo:', a % b)         # 1
print('Power:', a ** b)         # 49
```

---

2. Comparison Operators

Used to compare two values, the outcome is a boolean (`True` or `False`).

Operator	Name	Description	Example
==	Equal to	True if operands are equal	5 == 5 → True
!=	Not equal to	True if operands are not equal	5 != 2 → True
>	Greater than	True if left operand is greater	7 > 4 → True
<	Less than	True if left operand is smaller	3 < 6 → True

>=	Greater than or equal	True if left operand ≥ right	5 >= 5 → True
<=	Less than or equal	True if left operand ≤ right	2 <= 4 → True

Example:

python

```
x, y = 5, 10
print(x == y)  # False
print(x < y)   # True
print(x != y)  # True
```

---

3. Logical Operators

Combine conditional statements, producing boolean results.

Operator	Name	Description	Example
and	Logical AND	True if both operands are True	True and False → False
or	Logical OR	True if at least one operand is True	True or False → True
not	Logical NOT	Inverts the boolean value	not True → False

Example:

```python
x, y = True, False
print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

---

4. Bitwise Operators

Operate on integers at the binary level, manipulating individual bits.

Operator	Name	Description	Example
&	AND	Bits set in both operands	5 & 3 = 1 (0101 & 0011)
\	OR	OR	Bits set in either operand
^	XOR	Bits set in one operand only	5 ^ 3 = 6 (0101 ^ 0011)
~	NOT	Inverts all bits (one's complement)	~5 = -6

Example:

```python
x = 5    # binary: 0101
y = 3    # binary: 0011
print(x & y)  # 1  (0001)
print(x | y)  # 7  (0111)
print(x ^ y)  # 6  (0110)
print(~x)     # -6 (inverts bits)
print(x << 1) # 10 (1010)
print(x >> 1) # 2  (0010)
```

With these operators, you can perform a wide range of computations and logical expressions in Python, from simple math to complex bit manipulation.

4.Conditional Statements

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

Ans:

Conditional statements let your program make decisions and execute different code based on certain conditions. Python provides three key statements for this flow control:

- if: Executes code only if a condition is true.
- elif: (short for "else if") Checks another condition if the previous `if` or `elif` was false.
- else: Executes code if all the preceding conditions were false.

The If Statement

The simplest decision maker. You give Python a condition, and if it evaluates to `True`, Python runs the indented code block beneath it.

```
python

if condition:
    # this code runs if condition is True
```

Example:

```
python

x = 10
if x > 5:
    print("x is greater than 5")
```

- Since `x` is 10, which is greater than 5, the message prints.

The if-else Statement

Sometimes you want to do one thing if the condition is true, *and* another thing if it is false. Use `else` to specify what happens when the `if` condition fails.

```
if condition:

    # runs if condition is True
else:
    # runs if condition is False
```

Example:

```python
number = 3
if number % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

- Because 3 is not even, the else block prints "Odd number."

• The `if-elif-else` Chain

When there are multiple conditions to test, `elif` lets you check them sequentially until one is true.

```python
if condition1:
    # runs if condition1 is True
elif condition2:
    # else if condition2 is True
elif condition3:
    # else if condition3 is True
else:
    # runs if none of the above conditions are True
```

Example:

```python
letter = 'A'
if letter == 'B':
    print("Letter is B")
elif letter == 'C':
    print("Letter is C")
elif letter == 'A':
    print("Letter is A")
else:
    print("Letter isn't A, B, or C")
```

- O/p : This prints "Letter is A" because the third condition matches.

How It Works

- Python evaluates the `if` condition first.
- If `True`, it runs that block and skips the rest.
- If `False`, it tests the next `elif`...
- If no conditions match, the `else` block runs (if present).
- Only one block runs in an `if-elif-else` chain.

2.Nested If -Else Conditions

Ans：

Nested if-else Conditions in Python

Nested if-else statements are conditional statements written inside another if or else block. They allow for more complex decision-making by testing additional conditions only when an outer condition is satisfied.

- **What is a Nested if-else?**

A nested if-else means placing one if or if-else statement inside another if or else block. This hierarchical structure lets you check multiple levels of conditions sequentially.

- **Syntax of Nested if-else**

python

```
if condition1:
    # Executes if condition1 is True
    if condition2:
        # Executes if both condition1 and condition2 are True
    else:
        # Executes if condition1 is True but condition2 is False
else:
    # Executes if condition1 is False
```

Example

Python

```
num = 10

    if num > 0:
        if num % 2 == 0:
            print("The number is positive and even.")
        else:
            print("The number is positive but odd.")
    else:
        print("The number is not positive.")
```

Output: The number is positive and even.

Explanation

- First, the code checks if `num` is greater than 0.
- If true, it then checks if `num` is even using the modulus `%` operator.
- Depending on the result, it prints whether the number is even or odd.
- If the first condition is false, it will print that the number is not positive.

❖ **Nested if-else with elif**

You can combine nested if-else with `elif` for more refined control:

```
    age = 25

    if age >= 18:
        if age >= 65:
            print("Senior citizen")
        else:
            print("Adult")
    elif age >= 13:
        print("Teenager")
    else:
        print("Child")
```

- This helps check complex conditions step-by-step in a logical manner.

5. Looping (For, While)

1.Introduction to `for` and `while` loops.

Ans:

For Loop:

A for loop in Python is used to iterate over a sequence (such as a list, tuple, string, dictionary, or range) and execute a block of code repeatedly for each item in that sequence. It simplifies the process of looping without requiring manual management of loop counters or indices.

Syntax:

python

```
for variable in sequence:
    # code block to execute
```

- `variable` takes the value of each item in the sequence one by one.
- The indented code block under the `for` statement runs once for each item.

Example 1: Looping through a list

python

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

Output:

```
apple
banana
cherry
```

Example 2: Looping through a string

python

```
for letter in "banana":
    print(letter)
```

Output:

```
b
a
```

```
n
a
n
a
```

Example 3: Using `range()` with for loop

The `range()` function generates a sequence of numbers, which the for loop can iterate over.

python

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

Explanation:

- The for loop automatically handles the iteration.
- Useful for processing items in collections or repeating a code block a fixed number of times.
- Helps write concise and readable looping logic.

Python's for loop is a powerful, readable tool for iterating over sequences without extra complexity like manual counters or index tracking.

5.1.2 While Loop:

The while loop is a control flow statement that repeatedly executes a block of code as long as a given condition is true. Unlike a for loop, which typically runs a fixed number of times, a while loop is ideal when you want to continue looping until some dynamic condition changes.

Syntax:

python

```
while condition:
    # code to execute repeatedly
```

- The loop *checks* the condition before each iteration.
- If the condition is True, it executes the indented block.
- Once the condition becomes False, the loop terminates.

Example 1: Basic While Loop

python

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output:

```
0
1
2
3
4
```

Explanation:

- The loop runs while `count` is less than 5.
- Each iteration, it prints the current value and then increments `count`.
- When `count` reaches 5, the condition is False and the loop stops.

Example 2: Using a Break Statement

You can use `break` inside a while loop to exit early based on a condition:

python

```
n = 0
while True:  # Infinite loop
    if n == 3:
        break  # Exit loop when n == 3
    print(n)
    n += 1
```

Output:

```
0
1
2
```

Key Points About While Loops

- If the condition is initially False, the code block inside the while loop will not execute at all.
- Be careful to update variables involved in the condition inside the loop to avoid infinite loops.
- You can use `continue` to skip the rest of the loop body and proceed with the next iteration.
- Use `else` with while loops to run a block of code after the loop finishes normally (without `break`).

Use Case of While Loop

- When you don't know in advance how many times the loop will run.
- Waiting for user input or some external event.
- Running loops until a dynamic condition changes.

5.2 How Loop Works In Python

Ans:

Loops in Python work by repeatedly executing a block of code based on a condition or sequence of elements. There are two main types: for loops and while loops.

How loops work conceptually:

1. Initialization: The loop starts with an initial condition or sequence.
2. Condition Check:
    - For a `while` loop, Python evaluates a condition before each iteration. If it is `True`, the loop executes; if `False`, the loop terminates.
    - For a `for` loop, Python iterates over each item in a sequence or iterable.
3. Execution: The block of code inside the loop runs.
4. Update: In `while` loops, variables involved in the condition typically change within the loop body to eventually end the loop.
5. Repeat or Exit: Steps 2-4 continue until the condition fails (`while`) or the sequence is exhausted (`for`).

For Loop Example Flow:

python

```
for i in range(3):   # range produces 0, 1, 2
    print(i)         # prints 0, then 1, then 2
```

- Python gets the first value (`0`), runs the code.
- Then next value (`1`), runs code again.
- Finally `2`.
- When there are no more values, loop ends.

While Loop Example Flow:

python

```
count = 0
while count < 3:
    print(count)
    count += 1
```

- Python checks if `count < 3` (initially 0 < 3 → True),
- Executes the block (prints `0`),
- Updates `count` to 1,
- Checks again (1 < 3 → True),
- Continues until condition is False (3 < 3 → False),
- Loop exits.

❖ Break and Continue

In Python, break and continue are loop control statements that modify the normal flow of loops like `for` and `while`.

Break Statement:

- The `break` statement immediately terminates the loop when executed.
- Control moves to the code right after the loop.
- It is useful to exit a loop early when a condition is met.

Example:

python

```
for i in range(5):
    if i == 3:
        break  # Exit loop when i equals 3
    print(i)
```

Output:

```
0
1
2
```

Explanation: The loop stops as soon as `i` is 3, so values 3 and 4 are not printed.

❖ Continue Statement:

- The `continue` statement skips the rest of the current iteration.
- Control jumps back to the start of the loop for the next iteration.
- It's used to bypass certain iterations without exiting the entire loop.

Example:

python

```
for i in range(5):
    if i == 3:
        continue  # Skip the iteration when i is 3
    print(i)
```

Output:

```
0
1
2
4
```

Summary of Differences

Statement	Effect	Usage
break	Exits the entire loop	Exit early when condition met
continue	Skips current iteration	Skip unwanted iterations

Use Cases

- Use break to stop searching a list once a target is found.
- Use continue to skip invalid data or irregular cases during loop processing.

These control statements make loops flexible and efficient by enabling early exit or selective iteration.

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

Ans:

Using loops with collections like lists, tuples, strings, dictionaries, and sets is a common and powerful way to process data in Python. Here's a detailed overview of how loops work with these collections:

1. Iterating Over Lists

Lists are ordered, mutable collections. You can loop through the entire list easily with a `for` loop.

`python`

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

This prints each fruit in order.

You can also loop by index using `range` and `len`:

python

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

2. Iterating Over Tuples

Tuples are ordered, immutable collections. Iterating over tuples is similar to lists:

python

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)
```

You can also iterate using indices, or while loops, similarly to lists.

3. Iterating Over Strings

Strings are sequences of characters. A `for` loop can iterate over each character:

python

```
for letter in "Python":
    print(letter)
```

4. Iterating Over Dictionaries

Dictionaries store key-value pairs. Loops can iterate over keys, values, or both:

python

```
person = {"name": "Alice", "age": 30}
for key in person:
    print(key, person[key])
```

Or iterate keys and values directly:

python

```
for key, value in person.items():
    print(f"{key}: {value}")
```

5. Iterating Over Sets

- Sets are unordered collections of unique elements. You can loop over them as well:

python

```
unique_numbers = {1, 2, 3}
for num in unique_numbers:
    print(num)
```

Additional Techniques

- Using **`enumerate()`** for index and item in one:

python

```
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

- Using **`zip()`** to iterate multiple sequences in parallel:

```
names = ["Alice", "Bob"]
ages = [25, 30]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

- **List Comprehensions as compact iteration**:

python

```
squares = [x**2 for x in range(5)]
```

- Loops in Python simplify traversing collections like lists, tuples, strings, dictionaries, and sets. They allow sequential access, indexed iteration, parallel iteration, and more, adapting to the data and task you have. Knowing how to iterate effectively over these collections is essential for efficient Python programming.

6. Generators and Iterators

1.Understanding how generators work in Python.

Ans:

Python generators are special kinds of iterators that allow you to generate values lazily, meaning values are produced one at a time and only as needed, rather than storing an entire sequence in memory. They are created using functions with the `yield` keyword or with generator expressions.

How Generators Work

- A generator function looks like a normal function but contains one or more `yield` statements.
- When called, the generator function returns a generator object without executing the function body immediately.
- Each call to `next()` on the generator resumes execution from where it last left off until it hits another `yield`, which produces a value and pauses again.
- This allows efficient iteration over large sequences or streams without the overhead of creating a full list in memory.

Creating Generators

You can create a generator function like this:

`python`

```
def my_generator(n):
    count = 0
    while count < n:
        yield count
        count += 1
```

Calling `my_generator(3)` returns a generator object, which yields `0`, then `1`, then `2` on successive iterations.

Using Generators

Generators can be used directly in loops:

`python`

```
for value in my_generator(3):
    print(value,end=" ")
```

Output: 0 1 2

Or manually advanced using `next()`:

python

```
g = my_generator(3)
print(next(g))  # 0
print(next(g))  # 1
print(next(g))  # 2
```

❖ **Generator Expressions**

Python also supports generator expressions, a concise syntax similar to list comprehensions but with parentheses:

python

```
squares = (x * x for x in range(5))
for square in squares:
    print(square)
```

This prints squares of numbers 0 to 4.

➢ **Benefits of Generators**

- Memory efficiency: Values generated on-the-fly; large sequences don't consume lots of memory.
- Represent infinite streams: Example, infinite sequence of numbers.
- Pipeline data processing: Combine multiple generators to process data stepwise.

Generators provide a powerful, memory-efficient way to produce sequences and streams of data, leveraging `yield` to pause and resume function execution. They are a cornerstone for Pythonic iteration, especially for large or complex data pipelines.

2. Difference between `yield` and `return`.

Ans:

The key difference between `yield` and `return` in Python lies in how they handle function execution and the values they produce:

1. `return`

- Terminates the function and sends a value back to the caller immediately.
- The function exits after the `return` statement executes.
- Returns a single value (or a tuple if multiple values are separated by commas).
- Subsequent code in the function after `return` does not execute.
- Used in regular functions to output results once.

Example:

```
def square(x):
    return x * x

print(square(4))  # Output: 16
```

2. `yield`

- Used in generator functions to pause execution and produce a value to the caller.
- The function's state (local variables, instruction pointer) is saved; it can resume where it left off when called again.
- Can produce multiple intermediate values one at a time, rather than all at once.
- Does not terminate the function immediately.
- Efficient for generating large sequences or streams of data without loading everything into memory.

Example:

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

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

Output:

```
1
2
3
```

Summary Comparison

Aspect	`return`	`yield`
Function Behavior	Exits the function completely	Pauses function, to be resumed later
Values Produced	Single value or tuple	Multiple values over time
Function Type	Regular function	Generator function
Memory Usage	Returns entire result at once	Produces items one at a time
Code Execution After Statement	Does not execute	Resumes on next call

When to Use

- Use `return` when you want to send a final result and end the function.
- Use `yield` when you want to generate a sequence of values lazily, especially when dealing with large or infinite data streams.

3. Understanding iterators and creating custom iterators.

Ans:

An iterator is an object in Python that allows you to traverse through all the elements in a collection (like lists, tuples, or dictionaries) one at a time. It implements a specific protocol involving two methods:

- `__iter__()` — returns the iterator object itself.
- `__next__()` — returns the next item in the collection, and raises a `StopIteration` exception when there are no more items.

Iterables vs Iterators

- An iterable is any Python object capable of returning an iterator, typically by implementing `__iter__()`.
- Common iterables include lists, tuples, strings, dictionaries, and sets.
- You get an iterator from an iterable by calling the built-in `iter()` function.

How Iterators Work

When you call `next()` on an iterator, it returns the next item and remembers its current position. On reaching the end, it raises a `StopIteration` exception to signal the end of iteration.

Using Built-in Iterators

`python`

```
my_list = [4, 7, 0]
iterator = iter(my_list)
print(next(iterator))  # 4
print(next(iterator))  # 7
print(next(iterator))  # 0
```

After calling `next()` three times, further calls raise `StopIteration`.

Creating Custom Iterators

You can define your own iterator by creating a class that implements `__iter__()` and `__next__()`.

Steps:

- `__iter__(self)`: Initialize iterator state and return the iterator object (usually `self`).
- `__next__(self)`: Return the next item; raise `StopIteration` when done.

Example: Iterator for Even Numbers

python

```
class EvenNumbers:
    def __iter__(self):
        self.n = 2  # Start from first even number
        return self

    def __next__(self):
        current = self.n
        self.n += 2
        return current

# Usage
even_iter = iter(EvenNumbers())
print(next(even_iter))  # 2
print(next(even_iter))  # 4
print(next(even_iter))  # 6
```

This iterator yields even numbers by maintaining internal state between `next()` calls.

- Iterators allow lazy traversal of sequences, yielding one item at a time.
- Python's iterator protocol requires implementing `__iter__()` and `__next__()`.
- Custom iterators are useful for generating sequences on the fly without storing them in memory.

7.Function And Methods

1.Defining and calling functions in Python.

Ans:

Functions are reusable blocks of code designed to perform specific tasks. They help make programs modular, readable, and maintainable.

Defining a Function

- Use the `def` keyword followed by the function name and parentheses `()`.
- Inside the parentheses, optionally specify parameters (inputs).
- End the function header line with a colon `:`.
- Indent the function body (usually 4 spaces) beneath the header.

Basic Syntax:

```
python

def function_name(parameters):
    # code block
    # perform actions
    return value  # optional
```

Simple Example:

```
def greet():
    print("Hello from a function")
```

This defines a function named `greet` which prints a message.

Calling a Function

- To execute a function, use its name followed by parentheses.
- If the function expects arguments, pass them inside the parentheses.
- Parentheses are required even if the function accepts no arguments.

Example:

```
python

greet()  # Prints: Hello from a function
```

❖ **Functions with Parameters and Arguments:**

- Parameters are variables listed in the function definition.
- Arguments are actual values passed when calling the function.
- You can define multiple parameters, separated by commas.

Example with Parameters:

```
python

def say_hello(name):
    print(name + " Refsnes")

say_hello("Emil")  # Output: Emil Refsnes
say_hello("Tobias")  # Output: Tobias Refsnes
```

- **Returning Values**

➔ Functions can return values using the `return` statement.

```
python

def add(a, b):
    return a + b

result = add(3, 4)
print(result)  # Outputs: 7
```

- Define functions with `def function_name(parameters):`.
- Use indentation for the function body.
- Call functions with parentheses and pass arguments if needed.
- Optionally use `return` to send a value back.

2.Functions Arguments :

Ans:

In Python, function arguments are the values you pass to a function when you call it. They allow you to customize the behavior of functions and make them more flexible. There are several types of arguments in Python:

1. Positional Arguments:

- The most common type.
- Values are assigned to parameters based on their position in the function call.
- The order must match the order in the function definition.

Example:

python

```
def greet(name, age):
    print(f"{name} is {age} years old")

greet("Alice", 30)
```

2. Keyword Arguments (Named Arguments)

- Values are assigned to parameters by explicitly specifying parameter names during the function call.
- The order of arguments does not matter.

Example:

python

```
def greet(name, age):
    print(f"{name} is {age} years old")

greet(age=30, name="Alice")
```

3. Default Arguments

- Parameters can have default values specified in the function definition.
- If the caller omits these arguments, default values are used.
- Default arguments must follow positional arguments.

Example:

python

```
def greet(name, message="Hello"):
    print(f"{message}, {name}")
greet("Bob")  # Uses default message
greet("Bob", "Hi")  # Overrides default message
```

4. Arbitrary Arguments (`*args`)

- Use `*args` to pass a variable number of non-keyword arguments.
- Inside the function, `args` is a tuple containing all excess positional arguments.

Example:

python

```
def add_numbers(*args):
    return sum(args)
add_numbers(1, 2, 3, 4)  # Sum of all arguments
```

5. Arbitrary Keyword Arguments (`**kwargs`)

- Use `**kwargs` to pass a variable number of keyword arguments.
- Inside the function, `kwargs` is a dictionary containing all excess named arguments.

Example:

```
def describe_person(**kwargs):

    for key, value in kwargs.items():

        print(f"{key}: {value}")

describe_person(name="Alice", age=30, city="NY")
```

Notes:

- Positional: order matters.
- Keyword: specify values with names.
- Default: optional arguments with preset values.
- `*args`: variable number of non-keyword arguments.
- `**kwargs`: variable number of keyword arguments.

3.Scope of variables in Python.

Ans:

The scope of a variable refers to the region in the program where that variable is accessible and can be used. Python variable scope determines the visibility and lifetime of variables.

❖ **Main Types of Variable Scope**

1. Local Scope
    - Variables created inside a function are local to that function.
    - Accessible only inside that function.
    - They cease to exist once the function finishes executing.

Example:

```
def my_func():
    x = 10  # x is local to my_func
    print(x)

my_func()    # Outputs: 10
print(x)     # Raises error: x is not defined
```

2. **Enclosing (Nonlocal) Scope**
    - Applies to nested or inner functions.
    - The inner function can access variables from the enclosing (outer) function.
    - Use `nonlocal` keyword inside inner functions to modify enclosing function variables.

Example:

```
def outer_func():
    x = 5
    def inner_func():
        nonlocal x
        x = 10
        print(x)  # 10
    inner_func()
    print(x)  # 10

outer_func()
```

3. **Global Scope**
    - Variables defined at the top level of a file or declared `global` inside functions.
    - Accessible throughout the module or script.

Example:

```
 python
y = 20  # global variable

def func():
    print(y)  # accesses global y

func()     # 20
print(y)   # 20
```

4. **Built-in Scope**
    - Contains names preassigned in Python, like `len`, `print`.
    - Accessible everywhere unless shadowed by local or global names.

LEGB Rule for Scope Resolution

Python resolves variable names by searching scopes in this order:

- Local — current function/local block.
- Enclosing — any and all enclosing functions (in nested functions).
- Global — module-level variables.
- Built-in — names in the built-in namespace.

Notes:

- Local variables exist inside functions only.
- Nonlocal/enclosing variables belong to outer functions in nested functions.
- Global variables exist module-wide.
- Built-in variables are predefined by Python.

4.Built In Methods For Strings , List , etc…

Ans:

1.String Methods

➢ **S**trings are immutable sequences of characters.

- `str.lower()`: Returns the string in lowercase.
- `str.upper()`: Returns the string in uppercase.
- `str.strip()`: Removes leading and trailing whitespace.
- `str.split(sep)`: Splits the string into a list by separator.
- `str.join(iterable)`: Joins elements of iterable into a string separated by the given string.
- `str.replace(old, new)`: Replaces occurrences of old substring with new substring.
- `str.find(sub)`: Returns the lowest index where substring is found or -1.
- `str.startswith(prefix)`, `str.endswith(suffix)`: Checks if string starts or ends with specified substring.

2.List Methods

➢ Lists are mutable sequences.

- `list.append(x)`: Adds element `x` to the end.
- `list.extend(iterable)`: Extends list by appending elements from the iterable.
- `list.insert(i, x)`: Inserts `x` at index `i`.
- `list.remove(x)`: Removes first occurrence of `x`.
- `list.pop([i])`: Removes and returns item at index `i` (default last).

- `list.index(x)`: Returns index of the first occurrence of `x`.
- `list.count(x)`: Returns number of occurrences of `x`.
- `list.sort()`: Sorts the list in place.
- `list.reverse()`: Reverses the list in place.

3.Tuple Methods

➢ Tuples are immutable sequences.

- `tuple.count(x)`: Returns number of times `x` appears.
- `tuple.index(x)`: Returns the first index of `x` in the tuple.

(Note: Tuples have few methods due to immutability.)

4.Dictionary Methods

➢ Dictionaries store key-value pairs.

- `dict.keys()`: Returns a view of keys.
- `dict.values()`: Returns a view of values.
- `dict.items()`: Returns a view of (key, value) pairs.
- `dict.get(key, default)`: Returns value for `key`, or `default` if not found.
- `dict.pop(key)`: Removes key and returns its value.
- `dict.popitem()`: Removes and returns an arbitrary (key, value) pair.
- `dict.update(other_dict)`: Updates dictionary with key-value pairs from another dict.
- `dict.clear()`: Removes all items.

5.Set Methods

Sets are unordered collections of unique elements.

- `set.add(elem)`: Adds element to the set.
- `set.remove(elem)`: Removes element; raises KeyError if not present.
- `set.discard(elem)`: Removes element if present, else does nothing.
- `set.pop()`: Removes and returns an arbitrary element.
- `set.clear()`: Removes all elements.
- `set.union(other)`: Returns union of sets.
- `set.intersection(other)`: Returns intersection.
- `set.difference(other)`: Returns difference.
- `set.issubset(other)`: Checks if set is subset.
- `set.issuperset(other)`: Checks if set is superset.

● Summary Table

Data Type	**Common Methods**
String	lower, upper, strip, split, join, replace, find
List	append, extend, insert, remove, pop, index, count, sort, reverse
Tuple	count, index
Dictionary	keys, values, items, get, pop, popitem, update, clear
Set	add, remove, discard, pop, clear, union, intersection, difference
Integer	bit_length, to_bytes, from_bytes

8.Control Statements (Break , Continue And Pass)

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

Ans: Python provides three important loop control statements that allow you to control the behavior of loops (`for` or `while`) beyond the default sequential execution:

---

1. `break` Statement

- Immediately exits the current loop, skipping any remaining iterations.
- Control passes to the first statement after the loop.
- Useful when you want to stop looping early based on a condition.

Example:

python

```
for i in range(5):
    if i == 3:
        break  # Exit loop when i is 3
    print(i)
```

Output:

```
0
1
2
```

---

2. `continue` Statement

- Skips the rest of the current loop iteration and jumps to the next iteration.
- Control goes back to the loop condition check immediately.
- Useful to bypass certain values or conditions inside the loop.

Example:

python

```
for i in range(5):
    if i == 3:
        continue  # Skip printing when i is 3
    print(i)
```

Output:

```
0
1
2
4
```

---

3. **`pass`** Statement

- Does nothing; it's a placeholder statement.
- Used when syntax requires a statement but no action is needed yet.
- Commonly used as a stub for unfinished code blocks like functions, loops, or conditionals.

Example:

python

```
for i in range(5):
    if i == 3:
        pass  # Placeholder, no action taken
    print(i)
```

Output:

```
0
1
2
3
4
```

Summary of Differences

Statement	Effect	Typical Use Case
`break`	Exits the loop immediately	Stop looping once a condition is satisfied
`continue`	Skips current iteration and moves to next	Skip processing for specific cases
`pass`	Does nothing (placeholder)	Placeholder for code to be added later

9. String Manipulation

1.Understanding how to access and manipulate strings.

Ans:

Accessing Characters and Substrings

- Use indexing with square brackets `[]` to access individual characters: `string[index]`.
  - Indexing starts at 0 for the first character.
  - Negative indices count from the end, e.g., `string[-1]` is the last character.
- Use slicing to get substrings: `string[start:end]` extracts characters from position `start` up to but not including `end`.

Example:

python

```
word = "Hello World"
print(word[0])       # H
print(word[-1])      # d
print(word[0:5])     # Hello
print(word[6:])      # World
```

Common String Manipulations

- Concatenation: Combine strings using `+`.
- Repetition: Repeat strings using `*`.
- Searching: Find substring position with `.find()`, returns `-1` if not found.
- Replacing: Replace parts of strings with `.replace(old, new)`.
- Case Conversion: Change character cases with `.lower()`, `.upper()`, `.title()`, `.capitalize()`.
- Trimming: Remove extra spaces with `.strip()`, `.lstrip()`, `.rstrip()`.

Example:

python

```
text = " Hello Python! "
print(text.strip())          # "Hello Python!"
print(text.replace("Python", "World"))  # " Hello World! "
print(text.upper())          # " HELLO PYTHON! "
```

- **Immutability**

Strings are immutable in Python, meaning methods like `.replace()` or `.upper()` return new strings; they do not change the original.

Notes:

- Use indexing and slicing to access parts of strings.
- Use built-in methods to transform and analyze strings.
- Remember strings are immutable; methods create new modified strings.

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

❖ Accessing Characters and Slices

```
python
text = "Hello Python"

# Access individual characters
print(text[0])    # Output: H
print(text[-1])   # Output: n (last character)

# Slicing substrings
print(text[0:5])  # Output: Hello (characters at index 0 through 4)
print(text[6:])   # Output: Python (from index 6 to end)
print(text[:5])   # Output: Hello (from start to index 4)
print(text[-6:-1]) # Output: Pytho (from index -6 to -2)
```

❖ String Concatenation and Repetition

python

```python
str1 = "Hello"
str2 = "World"

# Concatenate
combined = str1 + " " + str2
print(combined)   # Output: Hello World

# Repeat
repeat = str1 * 3
print(repeat)     # Output: HelloHelloHello
```

❖ Searching and Replacing

python

```python
sentence = "I love Python programming"

# Find position of substring

pos = sentence.find("Python")
print(pos)  # Output: 7

# Replace substring

new_sentence = sentence.replace("Python", "Java")
print(new_sentence)  # Output: I love Java programming
```

❖ Case Conversion and Trimming

python

```
raw = "  Hello World!  "

# Trim whitespace
print(raw.strip())     # Output: Hello World!

# Change case
print(raw.upper())     # Output:   HELLO WORLD!
print(raw.lower())     # Output:   hello world!

# Capitalize first letter only
print(raw.capitalize()) # Output:   hello world!
```

---

3. String slicing.

Ans:

String slicing is a way to extract a substring from a string by specifying a start index, an end index, and optionally a step value.

Syntax

python

```
substring = string[start:end:step]
```

- start: The beginning index of the slice (inclusive). Defaults to 0 if omitted.
- end: The ending index of the slice (exclusive). Defaults to the length of the string if omitted.
- step: The interval between characters to include in the slice. Defaults to 1.

Key Points

- Indexing starts at 0.
- Negative indices count from the end (`-1` is the last character).
- The character at the end index is not included in the result.
- Omitting parameters uses default values.
- Step can be negative to slice in reverse order.

Examples

python

```
s = "Hello, World!"

print(s[0:5])     # Output: Hello (characters from index 0 to 4)
print(s[:5])      # Output: Hello (start defaults to 0)
print(s[7:])      # Output: World! (to the end)
print(s[-6:-1])   # Output: World (negative indexing)
print(s[::2])     # Output: Hlo ol! (every 2nd character)
print(s[::-1])    # Output: !dlroW ,olleH (reversed string)
print(s[7:12:1])  # Output: World (step of 1 is default)
```

❖ Examples

Example 1: Slicing with only start index

python

```
text = "Python Programming"
print(text[7:])   # Output: Programming
```

- Starts from index 7 to the end of the string.

Example 2: Slicing with only end index

python

```
text = "Python Programming"
print(text[:6])   # Output: Python
```

- Extracts from the beginning up to index 6 (exclusive).

Example 3: Slicing with step

python

```
text = "Python Programming"
print(text[::3])  # Output: Ph rgm
```

- Takes every 3rd character from the string.

Example 4: Slicing backwards

python

```
text = "Python"
print(text[::-1])  # Output: nohtyP
```

- Reverses the string.

Example 5: Extracting a single character

python

```
text = "Python"
print(text[1])  # Output: y
```

- Access character at index 1.

Example 6: Using negative step and indexes

```
python

    text = "Python Programming"
print(text[10:5:-1])  # Output: mmarg
```

- Slices the string backward from index 10 down to (but not including) 6.