**Module:5**

**Python - Fundamentals Of Python Language**

**1.Introduction to Python Theory:**

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

* Python is a widely used, general-purpose, high-level programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991.

**Key Features of Python:**

* **Simple and Easy to Learn:**
  + Python's syntax is designed to be clear and intuitive, resembling natural language more than other programming languages.
  + It emphasizes readability through significant indentation, making code easier to understand and maintain.
  + The relatively small number of keywords contributes to its ease of learning.
* **High-Level Language:**
  + Python abstracts away many of the complexities of low-level programming, such as memory management and hardware interaction.
  + This allows developers to focus on solving problems and expressing logic rather than dealing with intricate system details.
* **Interpreted Language:**
  + Unlike compiled languages (like C++ or Java) that require a separate compilation step before execution, Python code is executed directly by an interpreter.
  + The interpreter reads and executes the code line by line at runtime, which facilitates rapid development and easier debugging.
  + This interpreted nature makes Python a dynamically typed language, meaning variable types are inferred at runtime, and explicit type declarations are not require**d.**

**2. History and evolution of Python.**

* Python was conceived in the late 1980s by Guido van Rossum, a Dutch programmer, and first released in 1991 as a hobby project. Named after the British comedy troupe Monty Python's Flying Circus, the language's design prioritized code readability and simplicity. Its evolution has seen major versions 1.0 (1994), 2.0 (2000), and 3.0 (2008), with the latter being a significant, non-backward-compatible overhaul.
* **Origins & Early Development**

Late 1980s:

Guido van Rossum began developing Python at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.

* **Inspiration:**

Frustrated with existing languages, he aimed to create a successor to the ABC language that was both powerful and accessible, with good readability and modularity.

* **First Release (1991):**

The first public implementation, Python 0.9.0, was released, introducing core features like exception handling, classes with inheritance, and modular functionality.

**3. Advantages of using Python over other programming languages.**

* Python offers several advantages over other programming languages, contributing to its widespread adoption and popularity:
* **Simplicity and Readability:**

Python's syntax is designed for readability and ease of use, often requiring fewer lines of code to achieve the same functionality compared to languages like Java or C++. This makes it easier to learn, write, and maintain code.

* **Versatility and Wide Application:**

Python is a general-purpose language with applications across various domains, including web development (Django, Flask), data science and machine learning (NumPy, Pandas, scikit-learn, TensorFlow), automation, scientific computing, and more.

* **Extensive Libraries and Frameworks:**

Python boasts a rich ecosystem of pre-built libraries and frameworks that streamline development and provide ready-to-use solutions for complex tasks, significantly accelerating development time.

* **Cross-Platform Compatibility:**

Python code can be executed on various operating systems (Windows, macOS, Linux) without significant modifications, enhancing its portability.

* **Strong Community Support:**

Python has a large and active global community that contributes to its development, provides extensive documentation, and offers support to users, making it easier to find solutions to problems.

* **Rapid Development and Prototyping:**

The concise syntax and extensive libraries enable faster development cycles and efficient prototyping of applications, allowing for quick iteration and deployment.

* **Integration Capabilities:**

Python can be easily integrated with other programming languages and systems, facilitating the creation of complex, multi-language applications.

* **Dynamic Typing and Automatic Memory Management:**

Python handles memory management automatically through garbage collection and uses dynamic typing, reducing the burden on developers to manage memory manually or declare variable types explicitly.

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

* Setting up a Python development environment involves installing Python and choosing an Integrated Development Environment (IDE) or code editor like Anaconda, PyCharm, or VS Code.
* **1. Installing Python:**

Download Python:

Obtain the appropriate installer for your operating system (Windows, macOS, Linux) from the official Python website.

* **Run the Installer:**

Execute the downloaded installer. Crucially, ensure you select the option to "Add Python to PATH" during installation (on Windows) or follow the platform-specific instructions to ensure Python is accessible from your command line.

* **Verify Installation:**

Open a terminal or command prompt and type python --version or py --version to confirm Python is installed and accessible.

* **2. Setting up the Development Environment:**

**A) Anaconda (Recommended for Data Science and Machine Learning):**

* **Download and Install Anaconda:**

Download the Anaconda installer from the Anaconda website and follow the installation instructions. It is recommended to add Anaconda to your system's PATH during installation.

* **Use Anaconda Navigator or Conda:**

Anaconda Navigator provides a graphical interface to manage environments and launch applications like Jupyter Notebook/Lab. Alternatively, use the conda command in your terminal to create and manage environments and install packages:

* Code

conda create -n myenv python=3.9  
 conda activate myenv  
 conda install pandas numpy matplotlib

**B) PyCharm (Full-featured IDE):**

* **Download and Install PyCharm:**

Download the Community (free) or Professional edition of PyCharm from the JetBrains website and install it.

* **Configure Interpreter:**

When creating a new project or in an existing project, navigate to File > Settings/Preferences > Project: [Your Project Name] > Python Interpreter. Here, you can select an existing Python interpreter (including those from Anaconda environments) or create a new virtual environment.

* **Install Packages:**

Use PyCharm's built-in package manager in the Python Interpreter settings to install necessary libraries.

**C) VS Code (Lightweight and Versatile Code Editor):**

* **Download and Install VS Code:**

Download and install Visual Studio Code from the official website.

* **Install Python Extension:**

Open VS Code, go to the Extensions view (Ctrl+Shift+X), search for "Python" by Microsoft, and install it.

* **Select Interpreter:**

Open a Python file or a workspace, open the Command Palette (Ctrl+Shift+P), and search for "Python: Select Interpreter." Choose the desired Python interpreter, which can be a global installation, a virtual environment created with venv, or an Anaconda environment.

* **Create Virtual Environment (Optional but Recommended):**

Use the "Python: Create Environment" command in the Command Palette to create a project-specific virtual environment (venv or conda).

**5. Writing and executing your first Python program.**

* Writing and executing a first Python program involves a few key steps:
* **1. Install Python:**

Ensure Python is installed on the system. If not, download the appropriate installer for the operating system (Windows, macOS, or Linux) from the official Python website (python.org) and follow the installation instructions.

* **2. Choose a Text Editor or IDE:**

Select a text editor (e.g., Notepad, TextEdit) or an Integrated Development Environment (IDE) like PyCharm, Visual Studio Code, or IDLE (which often comes bundled with Python) to write the code. IDEs offer features like syntax highlighting, autocompletion, and debugging, which can enhance the coding experience.

* **3. Create a Python File:**

Open the chosen text editor or IDE and create a new file. Save this file with a .py extension (e.g., hello.py).

* **4. Write the Code:**

In the newly created .py file, write the Python code. A common first program is "Hello, World!":

Python

print("Hello, World!")

This line uses the print() function to display the string "Hello, World!" to the console.

* **5. Save the File:**

Save the changes made to the .py file.

* **6. Execute the Program:**

Using a Terminal/Command Prompt:

Open a terminal or command prompt.

Navigate to the directory where the .py file is saved using the cd command (e.g., cd C:\PythonPrograms).

Execute the program by typing python followed by the filename (e.g., python hello.py). The output "Hello, World!" should appear in the terminal.

**2. Programming Style**

**1.** **Understanding Python’s PEP 8 guidelines.**

* PEP 8, or Python Enhancement Proposal 8, is the official style guide for writing Python code. Its primary goal is to enhance readability and consistency across the Python community. Adhering to PEP 8 guidelines helps developers write clean, maintainable code that is easier for others to understand and collaborate on.
* **Here are some key aspects of PEP 8:**
* **Indentation:**

Use 4 spaces per indentation level. Avoid using tabs for indentation.

* **Line Length:**

Limit all lines to a maximum of 79 characters for code and 72 characters for docstrings and comments.

* **Blank Lines:**

Use two blank lines to separate top-level function and class definitions. Use one blank line to separate methods within a class and logical sections within functions.

* **Imports:**

Place all imports at the top of the file, after any module comments or docstrings.

Group imports in the following order: standard library imports, third-party library imports, local application/library specific imports.

Use one import statement per module, and avoid wildcard imports (e.g., from module import \*).

* **Naming Conventions:**

Variables and Functions: Use lowercase letters with words separated by underscores (e.g., my\_variable, calculate\_total()).

* **Classes:** Use capitalized words without underscores (CamelCase) (e.g., MyClass, CustomerOrder).
* **Constants:** Use all uppercase letters with words separated by underscores (e.g., MAX\_SPEED, PI\_VALUE).
* **Whitespace:**

Use single spaces around operators (e.g., x = y + z).

No spaces immediately inside parentheses, brackets, or braces.

One space after a comma or colon.

* **Comments:**

Use inline comments sparingly, explaining non-obvious code.

Separate inline comments from the statement by two or more spaces.

Start comments with # and a single space.

* **Boolean Comparisons:**

Avoid explicitly comparing boolean values to True or False (e.g., instead of if x == True:, use if x:).

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

* Python relies on specific conventions for indentation, comments, and naming to ensure code readability and maintainability, largely guided by PEP 8, the official style guide for Python code.
* **Indentation:**
* Python uses indentation to define code blocks, unlike other languages that might use curly braces.
* The standard indentation is four spaces per level. Tabs should not be mixed with spaces.
* Consistent indentation within a file and project is crucial to avoid IndentationError and ensure correct program execution.
* **Comments:**
* Comments are used to explain code and improve understanding for human readers; the Python interpreter ignores them.
* **Single-line comments**: begin with a # followed by a single space and are used for brief explanations.
* **Block comments**: are indented to the same level as the code they describe, with each line starting with # and a single space. They explain more complex logic or design decisions.
* **Inline comments**: appear on the same line as a code statement, separated by at least two spaces, and should be used sparingly for concise explanations.
* **Docstrings**: (documentation strings) are multi-line strings enclosed in triple quotes ("""Docstring content""") used to document modules, classes, and functions, explaining their purpose, arguments, and return values.
* Naming Conventions (PEP 8):
* **Variables and Functions:** Use lowercase\_with\_underscores (snake\_case).
* **Classes:** Use CamelCase (CapWords).
* **Constants:** Use ALL\_CAPS\_WITH\_UNDERSCORES.
* **Modules:** Use lowercase\_with\_underscores.
* **Package Names:** Use lowercase\_with\_underscores.
* **Private Identifiers:** Start with a single underscore (e.g., \_private\_variable).
* **Special/Magic Methods:** Start and end with double underscores (e.g., \_\_init\_\_, \_\_str\_\_).
* Adhering to these conventions significantly enhances code clarity, making it easier for developers to read, understand, and collaborate on Python projects.

**3. Writing readable and maintainable code.**

* Writing readable and maintainable code is essential for creating high-quality software that is easier to debug, extend, and collaborate on. A clean codebase reduces the long-term effort and cost of maintenance, decreases the risk of introducing new bugs, and makes the project more resilient to change.

Core principles for readable and maintainable code

* **Make code "self-documenting" with meaningful names**. Choose clear, descriptive names for variables, functions, and classes that convey their purpose and intent. Avoid single-letter variables or cryptic abbreviations, and use constants with clear names instead of "magic numbers".
* **Bad:** int d;
* **Good:** int daysSinceLastUpdate;
* **Adhere to the "Don't Repeat Yourself" (DRY) principle**. Avoid duplicating logic by creating reusable functions, classes, or modules. When you need to update replicated code, fixing it in a single location saves time and reduces the risk of errors.
* **Follow the Single Responsibility Principle (SRP)**. Each function, class, or module should have one, and only one, reason to change. By breaking down complex tasks into smaller, focused units, your code becomes easier to understand, test, and maintain.
* **Keep functions and methods small**. Concise functions that perform a single task are easier to reason about and maximize code reuse. If a function is too long or complex, refactor it into smaller, more manageable pieces.
* **Prioritize simplicity and avoid complexity**. Strive for straightforward solutions over "clever" or overly complex ones. Future developers—including yourself—will appreciate code that is simple to understand, even if it is less elegant.

**3. Core Python Concepts**

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

* Data types are classifications that specify the kind of values a variable can hold and the operations that can be performed on them. In Python, common built-in data types include:
* **1. Numeric Types:**

Integers (int): Represent whole numbers, both positive and negative, without decimal points. There is no practical limit to their size in Python.

Python

my\_integer = 100

Floats (float): Represent real numbers with decimal points. They are stored as floating-point numbers and offer a certain level of precision (typically up to 15 decimal places).

Python

my\_float = 3.14159

* **2. Sequence Types:**

Strings (str): Represent sequences of characters, used for text. They are immutable, meaning their content cannot be changed after creation.

Python

my\_string = "Hello, World!"

Lists (list): Represent ordered, mutable collections of items. Items can be of different data types and can be added, removed, or modified.

Python

my\_list = [1, "apple", 3.0]

Tuples (tuple): Represent ordered, immutable collections of items. Similar to lists, but their content cannot be changed after creation.

Python

my\_tuple = (1, "banana", 4.5)

* **3. Mapping Type:**

Dictionaries (dict): Represent unordered collections of key-value pairs. Each key must be unique and immutable, while values can be of any data type.

Python

my\_dictionary = {"name": "Alice", "age": 30, "city": "New York"}

* **4. Set Type:**

Sets (set): Represent unordered collections of unique, immutable items. They do not allow duplicate elements.

Python

my\_set = {1, 2, 3, 3, 4} # will store as {1, 2, 3, 4}

**2.Python variables and memory allocation.**

* In Python, variables are not containers that directly store values, but rather references or labels that point to objects in memory. This is a fundamental concept in understanding Python's memory allocation.
* **How Variables and Memory Interact:**
* **Object Creation:**
* When you assign a value to a variable (e.g., x = 10), Python creates an object in memory (in this case, an integer object with the value 10).
* **Variable as Reference:**

The variable x then becomes a reference to this object's memory location.

* **Dynamic Typing:**

Python is dynamically typed, meaning you don't declare a variable's type. A variable can reference objects of different types throughout its lifetime (e.g., x = 10, then x = "hello"). When x is reassigned, it simply points to a new object, and the old object's reference count is decremented.

* **Reference Counting and Garbage Collection:**

Python uses a mechanism called reference counting to manage memory. Each object in memory has a count of how many variables (or other objects) are referencing it. When this count drops to zero, the object is no longer accessible and becomes eligible for garbage collection, where Python's memory manager reclaims the memory.

* **Memory Optimization (Integer Caching):**

For certain immutable objects, like small integers (typically in the range of -5 to 256), Python pre-allocates and caches these objects at startup. If you assign a variable to one of these cached values, it will point to the existing object in memory, rather than creating a new one. This is a memory optimization.

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

* Python operators are special symbols that perform operations on variables and values. The primary categories include:
* **1.Arithmetic Operators:**

These perform mathematical calculations.

* + (Addition): Adds two operands.
* - (Subtraction): Subtracts the second operand from the first.
* \* (Multiplication): Multiplies two operands.
* / (Division): Divides the first operand by the second, resulting in a float.
* // (Floor Division): Divides and rounds down to the nearest whole number.
* % (Modulo): Returns the remainder of the division.
* \*\* (Exponentiation): Raises the first operand to the power of the second.
* **2. Comparison Operators:**

These compare two values and return a Boolean result (True or False).

* == (Equal to): Checks if two operands are equal.
* != (Not equal to): Checks if two operands are not equal.
* > (Greater than): Checks if the first operand is greater than the second.
* < (Less than): Checks if the first operand is less than the second.
* >= (Greater than or equal to): Checks if the first operand is greater than or equal to the second.
* <= (Less than or equal to): Checks if the first operand is less than or equal to the second.
* **3. Logical Operators:**

These combine conditional statements and return a Boolean result.

* and: Returns True if both conditions are True.
* or: Returns True if at least one condition is True.
* not: Reverses the Boolean state of the operand.
* **4. Bitwise Operators:**

These perform operations on the individual bits of integers. The integers are treated as binary numbers.

* & (Bitwise AND): Sets each bit to 1 if both corresponding bits are 1.
* | (Bitwise OR): Sets each bit to 1 if at least one of the corresponding bits is 1.
* ^ (Bitwise XOR): Sets each bit to 1 if only one of the corresponding bits is 1.
* ~ (Bitwise NOT): Inverts all the bits.
* << (Left Shift): Shifts the bits to the left by a specified number of positions.
* >> (Right Shift): Shifts the bits to the right by a specified number of positions

**4. Conditional Statements**

**1.Introduction to conditionalstatements: if, else, elif.**

* Conditional statements, specifically if, else, and elif, are fundamental programming constructs that enable a program to make decisions and execute different blocks of code based on whether certain conditions are true or false.
* **1. if statement:**

The if statement is the most basic conditional statement. It executes a block of code only if a specified condition evaluates to True.

* **Python**

age = 20  
if age >= 18:  
 print("You are an adult.")

* **2. else statement:**

The else statement is used in conjunction with an if statement. It provides an alternative block of code to be executed if the if condition evaluates to False.

* **Python**

temperature = 25  
if temperature > 30:  
 print("It's a hot day.")  
else:  
 print("It's not too hot.")

* **3. elif statement:**

The elif (short for "else if") statement allows for checking multiple conditions sequentially. If the if condition is False, the program proceeds to check the elif condition. If that is also False, it can continue to check further elif conditions. If none of the if or elif conditions are True, the else block (if present) will be executed.

* **Python**

score = 85  
if score >= 90:  
 print("Grade A")  
elif score >= 80:  
 print("Grade B")  
elif score >= 70:  
 print("Grade C")

else:  
 print("Grade D or F")

**2. Nested if-else conditions.**

* Nested if-else conditions refer to the practice of placing one or more if-else statements inside another if or else block. This creates a hierarchical structure for decision-making, allowing a program to evaluate multiple conditions in a specific order.
* **Concept:**
* An outer if statement checks an initial condition.
* If the outer condition is true, the code block within that if statement is executed, which can include another if-else statement (the "nested" part).
* This nested if-else then evaluates a secondary condition, and its corresponding code block is executed based on the outcome of that condition.
* If the outer condition is false, the else block of the outer statement is executed, which can also contain further nested if-else statements.

**5. Looping (For, While)**

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

* **for Loops**
* Purpose: Use a for loop when you know the exact number of times you need to repeat a block of code, such as processing each item in a list or iterating over a specific range of numbers.
* How it works: It iterates through a sequence, processing each item one by one. Many for loop structures include an initialization, a condition, and an increment/decrement step, allowing for concise control over the loop.
* **Example (Conceptual):**

Code:

for item in list\_of\_items:

* while Loops
* **Purpose:** Use a while loop when you need to repeat a block of code an unknown number of times, continuing as long as a specific condition remains true.
* **How it works:** It checks a condition before each iteration. If the condition is true, the code block executes, and the condition is checked again. The loop terminates when the condition becomes false.
* **Example (Conceptual):**

Code:

while condition\_is\_true:

**2.How loops work in Python.**

* Loops in Python provide a mechanism to repeatedly execute a block of code. Python primarily offers two types of loops: for loops and while loops.
* **1. For Loops:**

**Purpose:** for loops are used to iterate over a sequence (like a list, tuple, string, or range) or any other iterable object. They are ideal when the number of iterations is known or determined by the length of the iterable.

* **Mechanism:**
* The for loop initializes an iteration variable.
* In each iteration, this variable is assigned the next item from the iterable.
* The indented block of code within the loop's body is executed with the current value of the iteration variable.
* This process repeats until all items in the iterable have been processed.
* **Example:**

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

* **2. While Loops:**

**Purpose:** while loops are used to repeatedly execute a block of code as long as a specified condition remains true. They are suitable when the number of iterations is not known in advance and depends on a dynamic condition.

* **Mechanism**:
* The while loop evaluates a test condition at the beginning of each iteration.
* If the condition is True, the indented block of code within the loop's body is executed.
* After the execution of the loop body, the condition is re-evaluated.
* This process continues until the test condition becomes False, at which point the loop terminates.
* **Example:**

count = 0  
 while count < 5

print(count)  
 count += 1

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

* Loops, particularly for loops, are commonly used to iterate over elements within collections like lists, tuples, dictionaries, and sets in Python.
* **Iterating over Lists and Tuples:**
* The most straightforward way to iterate through items in a list or tuple is directly using a for loop:
* **Python**

my\_list = ["apple", "banana", "cherry"]

for item in my\_list:

print(item)

my\_tuple = ("red", "green", "blue")

for color in my\_tuple:

print(color)

* **Iterating with Index (using range() or enumerate()):**

If access to the index of each element is required, range() or enumerate() can be used:

* **Python**

my\_list = ["apple", "banana", "cherry"]

for i in range(len(my\_list)):

print(f"Element at index {i}: {my\_list[i]}")

my\_tuple = ("red", "green", "blue")

for index, color in enumerate(my\_tuple):

print(f"Color at index {index}: {color}")

* **Iterating over Dictionaries:**

Dictionaries store data in key-value pairs. Iteration can target keys, values, or both:

* **Python**

my\_dict = {"name": "Alice", "age": 30, "city": "New York"}

for key in my\_dict:

print(f"Key: {key}")

for value in my\_dict.values():

print(f"Value: {value}")

for key, value in my\_dict.items():

print(f"Key: {key}, Value: {value}")

* **Iterating over Sets:**

Sets are unordered collections of unique elements. Iteration directly accesses each element:

* **Python**

my\_set = {"apple", "banana", "cherry"}

for fruit in my\_set:

print(fruit)

* **Using while loops with collections:**

While less common for simple iteration, while loops can be used, often requiring manual index management:

* **Python**

my\_list = ["apple", "banana", "cherry"]

i = 0

while i < len(my\_list):

print(my\_list[i])

i += 1

**6. Generators and Iterators**

**1.** **Understanding how generators work in Python.**

* Python generators are a powerful and memory-efficient way to create iterators. They allow you to generate a sequence of values on demand, one at a time, instead of constructing the entire sequence in memory upfront. This "lazy evaluation" is their key advantage, especially when dealing with large datasets or infinite sequences.
* **How Generators Work:**
* **Yield Keyword:**

The defining characteristic of a generator function is the presence of the yield keyword. Unlike return, which terminates a function and returns a value, yield pauses the function's execution, returns a value to the caller, and preserves the function's state.

* **Generator Object:**

When you call a generator function, it doesn't immediately execute the code within it. Instead, it returns a special "generator object." This object is an iterator, meaning it can be iterated over using a for loop or advanced explicitly using the next() function.

* **Resuming Execution:**

Each time next() is called on the generator object (or implicitly by a for loop), the generator function resumes execution from where it last yielded a value. It continues until it encounters another yield statement, at which point it pauses again and returns the new value.

* **StopIteration:**

When the generator function runs out of yield statements and completes its execution, a StopIteration exception is raised, signaling that there are no more values to yield. This exception is handled automatically by for loops.

**2. Difference between yield and return.**

* Yield is the periodic income from an investment (like dividends or interest) as a percentage of its price, whereas return is the total profit or loss of an investment over time, including income and any changes in the investment's market value (capital gains). Yield provides insight into regular income potential, while return offers a holistic view of an investment's overall performance.

Yield

* **What it is**: The income an investment generates over a period, such as interest from a bond or dividends from a stock.
* **How it's expressed**: As a percentage relative to the investment's price or face value.
* **What it includes**: Primarily periodic payments like dividends and interest.
* **Focus**: Income-generating potential, often used by investors seeking regular income.

Return

* **What it is**:

The total gain or loss from an investment over a specific period.

* **How it's expressed**:

As a dollar value or percentage, reflecting the overall change in the investment's value.

* **What it includes**:

Both the income received (yield) and any capital appreciation or depreciation (changes in market value).

* **Focus**:

Overall profitability of an investment, providing a complete picture of its performance.

**3. Understanding iterators and creating custom iterators.**

* **Understanding Iterators:**

An iterator in Python is an object that allows traversal through a sequence of data, one element at a time. It implements the iterator protocol, which consists of two methods:

* **\_\_iter\_\_():** This method should return the iterator object itself.
* **\_\_next\_\_():** This method should return the next item from the sequence. When there are no more items to return, it must raise a StopIteration exception.
* An iterable is an object that can be iterated over. It defines a \_\_iter\_\_() method that returns an iterator. Examples of built-in iterables include lists, tuples, strings, and dictionaries. Every iterator is also an iterable, but not every iterable is an iterator.
* **Creating Custom Iterators:**

To create a custom iterator in Python, you define a class that implements the \_\_iter\_\_() and \_\_next\_\_() methods:

* **Python:**

class MyCustomIterator:  
 def \_\_init\_\_(self, start, end):  
 self.current = start  
 self.end = end  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 if self.current < self.end:  
 value = self.current  
 self.current += 1  
 return value  
 else:  
 raise StopIteration  
  
my\_iterator = MyCustomIterator(1, 5)  
  
for num in my\_iterator:  
 print(num)  
another\_iterator = MyCustomIterator(10, 13)  
print(next(another\_iterator))  
print(next(another\_iterator))  
print(next(another\_iterator))

**7. Functions and Methods**

**1.** **Defining and calling functions in Python**

* Defining and calling functions in Python involves two distinct but related steps:
* **1. Defining a Function:**
* To define a function in Python, use the def keyword, followed by the function name, a pair of parentheses, and a colon. The code block that constitutes the function's body must be indented.
* **Python**

def function\_name(parameter1, parameter2):  
 print(f"Hello, {parameter1} and {parameter2}!")  
 return "Function executed successfully"

* def keyword: Signals the start of a function definition.
* function\_name: A descriptive name for the function, adhering to Python's naming conventions (lowercase, words separated by underscores).
* (): Parentheses enclose optional parameters.
* parameter1, parameter2: These are placeholders for values that will be passed into the function when it's called.
* :: A colon marks the end of the function header.
* Indented code block: This is the function's body, containing the instructions to be executed.
* return statement (optional): Used to send a value back to the part of the code that called the function.
* **2. Calling a Function:**
* To execute the code within a defined function, you "call" it. This involves writing the function's name followed by parentheses, providing any necessary arguments for the parameters.
* Python  
  result = function\_name("Alice", "Bob")  
  print(result)
* function\_name(...): You call the function by its name.
* "Alice", "Bob": These are the arguments passed to the function, corresponding to parameter1 and parameter2 in the definition.
* result = ...: If the function has a return statement, the returned value can be assigned to a variable.
* **Example:**

Python

* def calculate\_sum(a, b):  
   """This function calculates the sum of two numbers."""  
   total = a + b  
   return total  
  sum\_result = calculate\_sum(5, 3)  
  print(f"The sum is: {sum\_result}")  
    
  another\_sum = calculate\_sum(10, 20)  
  print(f"Another sum is: {another\_sum}")

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

* Function arguments in programming, particularly in languages like Python, can be categorized based on how they are passed and handled:
* **1. Positional Arguments:**

These arguments are assigned to parameters in a function based on their order or position during the function call.

The first value passed corresponds to the first parameter in the function definition, the second value to the second parameter, and so on.

The order of positional arguments is crucial; changing the order can lead to incorrect assignments and unexpected results.

The number of positional arguments passed must match the number of parameters defined in the function, unless default values are provided.

* **Python**

def greet(name, message):  
 print(f"Hello, {name}! {message}")  
  
greet("Alice", "How are you?") # "Alice" is assigned to 'name', "How are you?" to 'message'

* **2. Keyword Arguments:**

These arguments are passed to a function using the parameter's name explicitly during the function call (e.g., parameter\_name=value).

The order of keyword arguments does not matter, as the explicit naming ensures correct assignment.

Keyword arguments enhance code readability by clearly indicating which value corresponds to which parameter.

* **Python**

def create\_user(username, email):  
 print(f"User created: {username}, Email: {email}")  
  
create\_user(email="test@example.com", username="john\_doe") # Order doesn't matter

* **3. Default Arguments:**

These are parameters in a function definition that have a predefined value assigned to them.

If an argument for that parameter is not provided during the function call, the default value is used.

If an argument is provided during the function call, it overrides the default value.

Default arguments must be defined after any non-default (positional) arguments in the function signature.

* **Python**

def send\_email(to\_address, subject="No Subject", body=""):  
 print(f"Sending email to {to\_address}")  
 print(f"Subject: {subject}")  
 print(f"Body: {body}")  
  
send\_email("recipient@example.com") *# Uses default subject and body*  
send\_email("another@example.com", subject="Meeting Reminder")

**3. Scope of variables in Python.**

* The scope of a variable in Python determines where in your code that variable is visible and accessible. Python follows the LEGB rule for resolving names, which stands for Local, Enclosing, Global, and Built-in scopes.
* **1. Local Scope:**

Variables defined inside a function have local scope.

They are accessible only within that specific function and cease to exist once the function finishes execution.

* **Python**

def my\_function():  
 local\_var = "I am local"  
 print(local\_var)  
  
my\_function()

* **2. Enclosing (Nonlocal) Scope:**

This applies to nested functions. Variables defined in an outer (enclosing) function can be accessed by an inner (nested) function.

The nonlocal keyword is used within a nested function to modify a variable in the enclosing scope, rather than creating a new local variable.

* **Python**

def outer\_function():  
 enclosing\_var = "I am in the enclosing scope"  
  
 def inner\_function():  
 nonlocal enclosing\_var  
 enclosing\_var = "I was modified by the inner function"  
 print(enclosing\_var)  
  
 inner\_function()  
 print(enclosing\_var)

* **3. Global Scope:**

Variables defined outside of any function, at the top level of a script, have global scope.

They are accessible from anywhere in the program, including inside functions.

To modify a global variable within a function, the global keyword must be used. Otherwise, a new local variable with the same name will be created.

* **Python**

global\_var = "I am global"  
  
def another\_function():  
 print(global\_var) *# Accessing global\_var*  
 global global\_var  
 global\_var = "I was modified globally"  
  
another\_function()  
print(global\_var)

* **4. Built-in Scope:**

This is the broadest scope and contains Python's pre-defined functions, exceptions, and objects (e.g., print(), len(), True, False).

These are available throughout the program without needing explicit definition or import.

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

* Python offers a rich set of built-in methods for manipulating common data types like strings and lists. These methods provide efficient ways to perform various operations without needing to write custom functions.
* **String Methods:**

String methods are called on string objects and typically return new strings, leaving the original string unchanged.

* **Python**  
  my\_string = " Hello, World! "  
    
  print(my\_string.strip()) *# Removes leading/trailing whitespace: "Hello, World!"*  
  print(my\_string.lower()) *# Converts to lowercase: " hello, world! "*  
  print(my\_string.upper()) *# Converts to uppercase: " HELLO, WORLD! "*  
  print(my\_string.replace("World", "Python")) *# Replaces substring: " Hello, Python! "*  
  print(my\_string.split(",")) *# Splits into a list by delimiter: [' Hello', ' World! ']*  
  print("Python".join(["Learn", "is", "fun"])) *# Joins elements with the string: "LearnPythonisfun"*  
  print("Hello".startswith("He")) *# Checks if string starts with prefix: True*  
  print("World!".endswith("!")) *# Checks if string ends with suffix: True*  
  print("abc".isalpha()) *# Checks if all characters are alphabetic: True*  
  print("123".isdigit()) # Checks if all characters are digits: True
* **List Methods:**

List methods modify the list in-place or return information about the list.

* **Python**my\_list = [1, 2, 3, 4]  
    
  my\_list.append(5) *# Adds an element to the end: [1, 2, 3, 4, 5]*  
  my\_list.insert(1, 0) *# Inserts an element at a specific index: [1, 0, 2, 3, 4, 5]*  
  my\_list.extend([6, 7]) *# Adds elements from another iterable: [1, 0, 2, 3, 4, 5, 6, 7]*  
  my\_list.remove(0) *# Removes the first occurrence of a value: [1, 2, 3, 4, 5, 6, 7]*  
  popped\_element = my\_list.pop(0) *# Removes and returns element at index: popped\_element = 1, my\_list = [2, 3, 4, 5, 6, 7]*  
  print(my\_list.index(3)) *# Returns the index of the first occurrence of a value: 1*  
  print(my\_list.count(4)) *# Returns the number of occurrences of a value: 1*  
  my\_list.sort() *# Sorts the list in-place: [2, 3, 4, 5, 6, 7]*  
  my\_list.reverse() # Reverses the order of elements in-place: [7, 6, 5, 4, 3, 2]

**8. Control Statements (Break, Continue, Pass)**

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

* In Python, break, continue, and pass are control flow statements used within loops to alter their execution.
* **break Statement**

The break statement immediately terminates the current loop (either for or while) and transfers control to the statement immediately following the loop. It is used when a specific condition is met and further iterations of the loop are no longer necessary.

* **Python**
* for i in range(10):

if i == 5:

break # Exit the loop when i is 5

print(i)

# Output: 0, 1, 2, 3, 4

* **continue Statement**

The continue statement skips the rest of the current iteration of the loop and proceeds to the next iteration. It is used when a specific condition is met, and you want to bypass the remaining code within the current iteration without exiting the entire loop.

* **Python**
* for i in range(5):

if i == 2:

continue

print(i)

# Output: 0, 1, 3, 4

* **pass Statement**

The pass statement is a null operation; it does nothing. It is a placeholder used when a statement is syntactically required but you do not want any code to execute at that point. This is often used when defining empty functions, classes, or as a temporary placeholder within conditional blocks or loops during development.

* **Python**
* def my\_function():

pass

* for i in range(3):

if i == 1:

pass # Do nothing when i is 1

else:

print(i)

# Output: 0, 2

**9. String Manipulation**

**1.** **Understanding how to access and manipulate strings.**

* **Accessing Strings**
* **Indexing:**

You can get a single character from a string using its position, known as its index.

Forward Indexing: The first character is at index 0, the second at index 1, and so on.

Backward (Negative) Indexing: The last character is at index -1, the second-to-last at -2, etc.

* **Slicing:**

To get a part of a string (a substring), you use a range of indices. For example, string[start:end] returns characters from the start index up to (but not including) the end index.

* **Length:**

Most languages have a function or property to find the number of characters in a string.

Common String Manipulation Operations

* **Concatenation:**

Joining two or more strings together using an operator like +.

* **Transformation:**

Case Conversion: Changing a string to uppercase or lowercase.

Replacement: Substituting a specific substring with another.

* **Splitting and Joining:**

Split: Breaking a string into a list of smaller strings based on a delimiter (e.g., a comma or space).

Join: Combining a list of strings into a single string using a specified separator.

* **Whitespace Handling:**

Removing leading, trailing, or all whitespace from a string using methods like strip().

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

* String operations in programming languages like Python typically include concatenation, repetition, and various string methods for manipulation.
* **1. Concatenation:**

Concatenation involves joining two or more strings together to form a new, single string. This is commonly achieved using the + operator.

* **Python**

string1 = "Hello"  
string2 = " World"  
concatenated\_string = string1 + string2  
print(concatenated\_string)

* **2. Repetition:**

Repetition involves creating multiple copies of a string. The \* operator is used for this, where a string is multiplied by an integer to repeat it that many times.

* **Python**

original\_string = "Repeat "  
repeated\_string = original\_string \* 3  
print(repeated\_string)

* **3. String Methods:**

String methods are built-in functions that can be called on string objects to perform various transformations and analyses. Some common examples include:

.upper(): Converts all characters in the string to uppercase.

* **Python**

text = "hello world"  
 uppercase\_text = text.upper()  
 print(uppercase\_text)

.lower(): Converts all characters in the string to lowercase.

* **Python**

text = "HELLO WORLD"  
 lowercase\_text = text.lower()  
 print(lowercase\_text)

.strip(): Removes leading and trailing whitespace from the string.

* **Python**

text = " spaced out "  
 stripped\_text = text.strip()  
 print(stripped\_text)

.replace(old, new): Replaces all occurrences of a specified substring (old) with another specified substring (new).

* **Python**

sentence = "I like apples."  
 new\_sentence = sentence.replace("apples", "oranges")  
 print(new\_sentence)

.split(delimiter): Splits the string into a list of substrings based on a specified delimiter.

* **Python**

data = "apple,banana,cherry"  
 fruits = data.split(",")  
 print(fruits)

**3.String slicing.**

* String slicing is a technique used in programming to extract a portion, or "substring," from a larger string. This operation creates a new string containing the selected characters, leaving the original string unchanged.
* **Key Concepts of String Slicing:**

**Syntax:**

String slicing typically uses a bracket notation with a colon to specify the start and end indices, and optionally, a step. The general format is string[start:end:step].

* **Indices:**

Start Index (inclusive): The index where the slice begins. If omitted, it defaults to the beginning of the string (index 0).

* End Index (exclusive): The index where the slice ends. The character at this index is not included in the resulting substring. If omitted, it defaults to the end of the string.
* Negative Indices: Negative indices can be used to count from the end of the string. For example, -1 refers to the last character, -2 to the second to last, and so on.
* **Step (optional):**

The step value determines the increment between characters in the slice. If omitted, it defaults to 1 (meaning every character is included). A step of -1 can be used to reverse a string.

* **Examples (using Python):**

**Python**

my\_string = "Hello, World!"  
substring1 = my\_string[0:5]   
substring2 = my\_string[7:12]   
substring3 = my\_string[:5]

substring4 = my\_string[7:]

substring5 = my\_string[-6:-1]

substring6 = my\_string[-1]

substring7 = my\_string[::2]   
substring8 = my\_string[::-1]

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

**1.** **How functional programming works in Python.**

* Functional programming in Python emphasizes treating computation as the evaluation of mathematical functions and avoiding changing state and mutable data. While Python is not a purely functional language, it offers features and constructs that allow programmers to adopt a functional style.
* **Key Concepts and How they Apply in Python:**

**Pure Functions:**

Concept: Functions that, given the same input, always return the same output and have no side effects (i.e., they don't modify global variables, perform I/O, or change the state of objects outside their scope).

* Python Application: Define functions that only depend on their arguments and return a value without altering anything else.
* **Immutability:**

Concept: Data cannot be changed after it's created. Instead of modifying existing data, new data structures are created with the desired changes.

Python Application: Utilize immutable data types like tuples and strings. When working with mutable types like lists, create new lists for transformations instead of modifying the original in-place.

* **First-Class Functions:**

Concept:Functions can be treated like any other variable – assigned to variables, passed as arguments to other functions, and returned as values from other functions.

Python Application:This is a core feature of Python. You can pass functions to map, filter, and reduce, or define functions within other functions.

* **Higher-Order Functions:**

Concept: Functions that take other functions as arguments or return functions as results.

* Python Application: Built-in functions like map(), filter(), and functools.reduce() are examples of higher-order functions. You can also define your own.
* **Recursion:**

Concept: A function calling itself to solve a problem, often used in functional programming to avoid explicit loops.

Python Application: Python supports recursion, although it has a default recursion limit. For deeply recursive problems, iterative solutions or tail-call optimization (though not natively supported in Python) might be considered.

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

* The map(), filter(), and reduce() functions are powerful tools in functional programming for processing data, particularly with collections or arrays. They promote cleaner, more concise code by abstracting common data manipulation patterns.
* **1. map()**

Purpose: Transforms each element in a collection by applying a given function, creating a new collection of the same size with the transformed elements.

Operation: Iterates over each element, applies a function to it, and collects the results into a new array.

* **Example (Python):**

Python

numbers = [1, 2, 3, 4]  
 squared\_numbers = list(map(lambda x: x \* x, numbers))  
 # squared\_numbers will be [1, 4, 9, 16]

* **2. filter()**

Purpose: Creates a new collection containing only the elements from the original collection that satisfy a given condition (i.e., for which a provided function returns True).

* Operation: Iterates over each element, applies a predicate function to it, and includes the element in the new collection only if the predicate returns True.
* **Example (Python):**

Python

numbers = [1, 2, 3, 4, 5, 6]  
 even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))  
 # even\_numbers will be [2, 4, 6]

* **3. reduce()**

Purpose: Applies a function to the elements of a collection cumulatively, reducing the collection to a single value.

Operation: Takes a function that accepts two arguments (an accumulator and the current element) and an optional initial value for the accumulator. It applies the function sequentially to the elements, updating the accumulator with each step.

* **Example (Python):**
* Python

from functools import reduce  
 numbers = [1, 2, 3, 4]  
 sum\_of\_numbers = reduce(lambda acc, x: acc + x, numbers)  
 # sum\_of\_numbers will be 10

**3.Introduction to closures and decorators.**

* **Closures:**
* **What they are:**

A closure is a function that retains a link to its enclosing scope's variables (called "free variables").

* **How they work:**

When an inner function is defined within an outer function and then returned from the outer function, it forms a closure.

* **Key characteristic:**

The inner function can access, and even modify (with keywords like nonlocal in Python), variables that were defined in the outer function, even after the outer function has completed its execution.

* **Example:**

A function factory that creates specialized functions is a common use case for closures.

* **Decorators:**
* **What they are:**

Decorators are functions that take another function as an argument and return a modified or extended version of that function.

* **How they work:**

They act as "syntactic sugar" in Python, wrapping a function call to add functionality before or after its original execution.

* **Benefits:**

Decorators provide a clean and readable way to add features like logging, access control, or performance monitoring to functions without altering their core logic.

* **Use of Closures:**

The inner function within a decorator is often a closure, capturing and using the original function and any added logic.