Module – 13 Python Fundamental

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

- Python is a dynamic, high-level, interpreted , general purpose programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python has become one of the most popular programming languages due to its versatility and ease of learning.
  - **Feature of Python** :
    - Easy to Code
    - Easy to Read
    - Large community support
    - Object-Oriented language
    - Portable language
    - Integrated language
    - GUI programming support
    - Large standard library
    - Allocating memory dynamically
- **History and evolution of Python** :

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.
- The following programming languages influence Python:
  - ABC language.
  - Modula-3
- There is a fact behind choosing the name Python. **Guido van Rossum** was reading the script of a popular BBC comedy series "**Monty Python's Flying Circus**". It was late on-air 1970s.
- Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the **"Monty Python's Flying Circus"** for their newly created programming language.
- Python is also versatile and widely used in every technical field, such as Machine Learning, Artificial Intelligence, Web Development, Mobile Application, Desktop Application, Scientific CalculatComments areion, etc.

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

- **Easy to learn and use:** Python’s simple syntax makes it beginner-friendly
- **Extensive libraries and frameworks**: Offers rich libraries for web development, data science, AI, etc.
- **Versatile:** Supports multiple programming paradigms and applications
- **Cross-platform:** Works seamlessly across Windows, macOS, and Linux
- **Great for rapid prototyping**: Quick syntax allows for fast idea implementation

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

1.Installing the Python:

1.Download Python

- **Go to python official website**
- **Download the latest python version file.**

2.Install Python:

- Run the installer.
- Check "Add Python to PATH" before clicking "Install Now".
- Verify installation by running

```
python –version

        or
python3 --version
```

1. **Installing and Setting Up Anaconda (Optional):**

Anaconda is a Python distribution that includes Jupyter Notebook and pre-installed libraries.

- **Installation**

- Download Anaconda from its official website.
- Run the installer and follow the prompts.
- Open Anaconda Navigator or use the Anaconda Prompt.

- **Create a Virtual Environment in Anaconda**

```
conda create --name myenv python=3.11
conda activate myenv
```

- **Using Jupyter Notebook (Optional)**
  conda install jupyter
- jupyter notebook

2. Setting up an IDE :

We can use **PyCharm** or **VS Code** for development.

- **PyCharm Setup**

1. Download from JetBrains PyCharm.
2. Install and open PyCharm.
3. Go to **File** > **Settings** > **Project Interpreter** and select your Python/Anaconda environment.

- **VS Code Setup**

1. Download from VS Code.
2. Install the **Python Extension** from the Extensions Marketplace.
3. Select the correct Python interpreter (Ctrl + Shift + P → "Python: Select Interpreter").

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

```
Print(“Hello!”)
```

O/P :

```
Hello!
```

2. Programming Style

- **Understanding Python’s PEP 8 guidelines :**
  i. Use 4-space indentation and no tabs.
  ii. Use docstrings.
  iii. Wrap lines so that they don’t exceed 79 characters .
  iv. Use of regular and updated comments are valuable to both the coders and users**.**
  v. Use of trailing commas.
  vi. Use Python’s default UTF-8 or ASCII encodings and not any fancy encodings.
  vii. Use spaces around operators and after commas, but not directly inside bracketing constructs:
  viii. Naming Conventions.
  ix. Characters that should not be used for identifiers.
  x. Don’t use non-ASCII characters in identifiers.

xi. Name your classes and functions consistently :

- **Indentation, comments, and naming conventions in Python**.
- **Indentation:**
    - ➢ Whitespace is used for indentation in Python**.** Python indentation is mandatory.A block is a combination of all these statements.Block can be regarded as the grouping of statements for a specific purpose.
    - ➢ Python is its use of indentation to highlight the blocks of code. All statements with the same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right.

Example:

The lines print(‘Logging on to geeksforgeeks…’) and print(‘retype the URL.’) are two separate code blocks. The two blocks of code in our example if-statement are both indented four spaces. The final print(‘All set!’) is not indented, so it does not belong to the else-block.

```
# Python indentation

site = 'gfg'

if site == 'gfg':
        print('Logging on to geeksforgeeks...')
else:
        print('retype the URL.')
print('All set !')
```

- **Comments in Python:**
    - ➢ Python comments start with the hash symbol # and continue to the end of the line.
    - ➢ useful information that the developers provide to make the reader understand the source code. It explains the logic or a part of it used in the code.
    - ➢ These are often cited as useful programming convention that does not take part in the output of the program but improves the readability of the whole program.

➢ **Types of comments in Python :**

1. **Single-line comment in Python**

➢ Python single-line comment starts with a hash symbol (#) with no white spaces and lasts till the end of the line. If the comment exceeds one line then put a hashtag on the next line and continue the comment.

- Python’s single-line comments are proved useful for supplying short explanations for variables, function declarations, and expressions.
- See the following code snippet demonstrating single line comment:

```
# This is a comment

# Print “GeeksforGeeks” to console

print("GeeksforGeeks")
```

output:

```
GeeksforGeeks
```

2. **Multiline comment in Python**

- Use a hash **(#)** for each extra line to create a multiline comment.
- Python multi-line comments by using multiline strings. It is a piece of text enclosed in a delimiter (“””) on each end of the comment. Again there should be no white space between delimiter (“””).
- Ex :1

```
# This is a comment
# This is second comment
# Print “GeeksforGeeks” to console
print("GeeksforGeeks")
```

Ex : 2

```
"""
This would be a multiline comment in Python that spans several lines
and describes geeksforgeeks. A Computer Science portal for geeks.
"""
print("GeeksForGeeks")
```

3. **Naming Convention:**

In Python, naming conventions help make code readable and maintainable.

a. **Variables and Functions :**

Lowercase with underscores (snake_case)

```
user_name = "Alice"
total_amount = 100
```

b. **Constants (Values that shouldn’t change) :**
All uppercase with underscores

```
PI = 3.14159
MAX_LIMIT = 100
```

c. **Class Names :**
PascalCase (UpperCamelCase)

```
class UserProfile:
    pass
```

d. **Private Variables and Methods (By Convention) :**
Prefix with an underscore

```
class Example:
    def __init__(self):
        self._private_var = 10  # Private (by convention)

    def _private_method(self):
        pass
```

e. **Dunder (Double Underscore) Methods :**
Special methods (also called magic methods) start and end with double underscores (__).

```
class Sample:
    def __init__(self):  # Constructor
        pass

    def __str__(self):  # String representation
        return "Sample Object"
```

f. **Naming Modules and Packages**
Lowercase with underscores (for readability)

```
# Module name
import my_module

# Package name
import mypackage
```

- **Writing readable and maintainable code.**
  Readable and maintainable code makes collaboration easier and helps prevent bugs. Follow these best practices to improve your Python code quality.

  a. **Follow PEP 8 (Python Style Guide)** : PEP 8 is the official Python style guide. Some key points:

     - Use Meaningful Variable & Function Names

```
max_retries = 10
def calculate_total_price(price, quantity):
    return price * quantity
```

  b. **Keep Code Indentation and Spacing Clean** : Python enforces indentation, but follow PEP 8 for better readability. Use 4 spaces per indentation level. Avoid tabs or mixed indentation. Avoid cluttered expressions

     - Use 4 spaces per indentation level

```
def greet():
    print("Hello, World!")
```

     - Use spaces around operators and after commas

```
total = price * quantity + tax
```

  c. **Keep Functions Small and Focused :** A function should do **one thing only**.

     - **Bad (Too much in one function)**

```
def process_data(data):
            cleaned = [d.strip().lower() for d in data]
            sorted_data = sorted(cleaned)
            return sorted_data
```

     - **Good (Split into smaller functions)**

```
def clean_data(data):
           return [d.strip().lower() for d in data]

def sort_data(data):
           return sorted(data)

cleaned = clean_data(data)
sorted_data = sort_data(cleaned)
```

  d. **Use Comments Wisely** : Explain why something is done, not what (the code itself is the "what"). Avoid redundant comments.

e. **Use Docstrings for Functions and Classes :**

```
def add_numbers(a, b):
    """
    Adds two numbers and returns the result.

    Args:
        a (int): First number.
        b (int): Second number.

    Returns:
        int: Sum of a and b.
    """
    return a + b
```

f. **Use List Comprehensions for Clean Code :** Avoid long and unnecessary loop

```
squared_numbers = [x**2 for x in range(10)]
```

g. **Use F-Strings for String Formatting** :

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

h. **Handle Exceptions Properly** : Avoid catching all exception blindly.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

3. Core Python Concepts

- **Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.**
  - **Integer** : Whole numbers, positive or negative, without decimals.

    EX :
    ```
    age = 25
    temperature = -10
    print(type(age))

     # Output:
     <class 'int'>
    ```

  - **Floating-Point Numbers (float)** : Numbers that contain decimals.

    EX :
    ```
    price = 99.99
    ```

```
pi = 3.14159
print(type(price))

 # Output:
 <class 'float'>
```

- **String** : A sequence of characters enclosed in single or double quotes.

  ```
  EX :
  name = "Alice"
  message = 'Hello, World!'
  print(type(name))

  # Output:
   <class 'str'>
  ```

- **List** : A collection that **is** ordered, mutable (changeable), and allows duplicates**.**

  ```
  EX :
  fruits = ["apple", "banana", "cherry"]
  numbers = [1, 2, 3, 4, 5]
  print(type(fruits))

  # Output: <class 'list'>
  ```

- **Tuple** : A collection that is ordered, immutable (unchangeable), and allows duplicates**.**

  ```
  EX :
  coordinates = (10.5, 20.3)
  colors = ("red", "green", "blue")
  print(type(colors))

   # Output:
  <class 'tuple'>
  ```

- **Dictionaries** : A collection that is unordered, mutable, and stores key-value pairs**.**

  ```
  EX :
  student = {
     "name": "Alice",
     "age": 22,
     "grade": "A"
  }
  print(type(student))

  # Output:
  <class 'dict'>
  ```

- **Sets** : A collection that is unordered, mutable, and does NOT allow duplicates**.**

  EX :
  ```
  unique_numbers = {1, 2, 3, 4, 4, 5}
  print(unique_numbers)  # Output: {1, 2, 3, 4, 5}
  print(type(unique_numbers))

   # Output:
  <class 'set'>
  ```

• **Python variables and memory allocation**.

- In Python, variables are used to store data. Unlike some languages, Python dynamically allocates memory based on the type of data stored.
- A variable is a name that refers to a value stored in memory.

  EX :
  ```
  x = 10
  name = "Alice"
  pi = 3.14
  ```

- Python uses a dynamic memory management system and handles memory allocation automatically.
  a. Reference Counting : Each object in Python has **a** reference count—a count of how many variables refer to it.

     EX :
     ```
     a = [1, 2, 3]  # List is created in memory
     b = a          # Both 'a' and 'b' reference the same object

     print(id(a))  # Memory address of 'a'
     print(id(b))  # Memory address of 'b' (Same as 'a')
     ```

  b. **Garbage Collection (GC)** : Python has an automatic garbage collector that removes unused objects when their reference count reaches zero.

     EX :
     ```
     x = 100
     x = None  # The previous value (100) is garbage collected if not use elsewhere
     ```

  c. **Memory Optimization with Small Integers and Strings** : Python optimizes memory usage for small integers (-5 to 256) and commonly used strings.

     EX : (integer caching)
     ```
     a = 100
     b = 100
     print(a is b)  # Output: True (Both point to the same memory location)
     ```
     EX : (string interning)
     ```
     str1 = "hello"
     ```

```
str2 = "hello"
print(str1 is str2)  # Output: True (Both refer to the same memory)
```

However, this does not work for dynamically created strings:

d. **Mutable vs Immutable Objects :**

- **Immutable objects** (cannot be changed): int, float, str, tuple
  EX :
  ```
  x = 10
  y = x
  y = 20  # A new memory location is assigned to y
  print(x)

  # Output:
  10 (x is unchanged)
  ```

- **Mutable objects** (can be changed): list, dict, set
  EX :
  ```
  list1 = [1, 2, 3]
  list2 = list1
  list2.append(4)

  print(list1)

  # Output: [1, 2, 3, 4] (Both lists are modified)
  ```

e. **Understanding id() and sys.getrefcount():**

- id(object): Returns the memory address of the object.
- sys.getrefcount(object): Returns the reference count.

  EX :
  ```
  import sys

  a = [1, 2, 3]
  print(id(a))  # Memory address
  print(sys.getrefcount(a))  # Number of references to 'a'
  ```

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

Operators are symbols that perform operations on variables and values. Python has several types of operators:

1. **Arithmetic Operators :** Used for mathematical operations.

Operator	**Name**	**Example (a=10 , b=3)**	**Result**
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30

/	Division	a / b	3.333
//	Floor Division	a // b	3 (remove decimal)
%	Modulus	a % b	1 (reminder)
**	Exponentiation	a ** b	1000

2. **Comparison (Relational) Operators** : Used to compare values. Returns True or False.

Operator	**Name**	**Example (a=10 , b=3)**	**Result**
= =	Equal to	a == b	False
!=	Not equal to	a != b	True
>	Greater than	a > b	True
<	Less than	a < b	False
>=	Greater than or equal to	a >= b	True
<=	Less than or equal to	a <= b	False

3. **Logical Operators :** Used to combine conditional statements.

Operator	**Meaning**	**Example (x=True, y=False)**	**Result**
And	True if both conditions are True	x and y	False
Or	True if at least one condition is True	x or y	True
Not	Reverses the condition	not x	False

4. **Bitwise Operators :** Perform operations on binary numbers (bit-level operations).

Operator	**Meaning**	**Example (a = 5 (0101 in binary), b = 3 (0011 in binary))**	**Result (Binary)**
&	AND	a & b	0001
`	OR	A ` b	0111
^	XOR	a ^ b	0110
~	NOT	~a	-0110
<<	Left Shift	a << 1	1010
>>	Right Shift	a >> 1	0010

4. Conditional Statements

- **Introduction to conditional statements: if, else, elif.**

  Conditional statements in Python allow the program to make decisions based on conditions. The basic structure includes if, elif, and else statements.

  - **The if Statement** : The if statement checks a condition and executes a block of code only if the condition evaluates to True.

    Syntax :
    ```
    if condition:
       # Code to execute if condition is True
    ```

    EX :
    ```
    age = 18
    if age >= 18:
       print("You are eligible to vote.")
    ```

    O/P :
    ```
    You are eligible to vote.
    ```

  - **The else Statement :** The else statement runs a block of code if the condition in the if statement is False**.**

    Syntax :
    ```
    if condition:
       # Code if condition is True
    else:
       # Code if condition is False
    ```

    EX :
    ```
    age = 16

    if age >= 18:
       print("You are eligible to vote.")
    else:
       print("You are not eligible to vote.")
    ```

    O/P :
    ```
    You are not eligible to vote.
    ```

  - **The elif (else if) Statement :** The elif statement is used when we have multiple conditions to check.

Syntax:
```
if condition1:
   # Code if condition1 is True
elif condition2:
   # Code if condition2 is True
else:
   # Code if none of the above conditions are True
```

EX :
```
marks = 75

if marks >= 90:
   print("Grade: A")
elif marks >= 75:
   print("Grade: B")
elif marks >= 50:
   print("Grade: C")
else:
   print("Grade: F")
```

O/P :
```
Grade: B
```

- **Nested if-else conditions** :
  A nested if-else statement means an if or else statement inside another if or else. This is useful when you need to check multiple conditions in a hierarchical manner.

  Syntax :
```
if condition1:
   if condition2:
      # Code to execute if both condition1 and condition2 are True
   else:
      # Code to execute if condition1 is True but condition2 is False
else:
   # Code to execute if condition1 is False
```

  EX : Checking eligibility for voting and senior citizen benefits

```
age = 70



if age >= 18:
    print("You are eligible to vote.")

    if age >= 65:

        print("You are also eligible for senior citizen benefits.")

    else:

        print("You are not yet a senior citizen.")

else:

    print("You are not eligible to vote.")
```

O/P :

```
You are eligible to vote.
You are also eligible for senior citizen benefits.
```

5. Looping (For, While)

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

Loops in Python allow us to execute a block of code multiple times. The two main types of loops in Python are:

- **for loop** : Used when we know the number of iterations. The for loop is used for iterating over a sequence (like a list, tuple, string, or range).

Syntax :

```
for variable in sequence:
    # Code to execute
```

EX : Looping through a range of numbers

```
for i in range(5):  # Loops from 0 to 4
    print("Iteration:", i)
```

O/P :

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

EX : Looping through a list

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

O/P :
```
apple
banana
cherry
```

- **while loop** : Used when we repeat a block of code until a condition becomes False.

  Syantax **:**
  ```
  while condition:
      # Code to execute
  ```

  EX :
  ```
  i = 1
  while i <= 5:
              print(i)
               i += 1 # Increment i
  ```

  O/P :

  ```
  1
  2
  3
  4
  5
  ```

- **How loops work in Python.**
  Answer is same as above**.**

- **Using loops with collections (lists, tuples, etc.).**
  In Python, loops are commonly used to iterate over collections such as lists, tuples, sets, and dictionaries. The for loop is the best way to traverse these collections.

  1. Iterating Over a List : Lists are ordered and mutable collections.
  EX : Using for Loop with a List
  ```
  fruits = ["apple", "banana", "cherry"]
  for fruit in fruits:
      print(fruit)
  ```

  O/P :
  ```
  apple
  banana
  cherry
  ```

EX : Using while Loop with a List

```
numbers = [10, 20, 30, 40]
i = 0

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

O/P :

```
10
20
30
40
```

2. Iterating Over a Tuple : Tuples are ordered but immutable collections.
   EX : Using for Loop with a Tuple

```
colors = ("red", "green", "blue")

for color in colors:
    print(color)
```

O/P :

```
red
green
blue
```

3. Iterating Over a Set : Sets are unordered and do not allow duplicates**.**
   EX **:** Using for Loop with a Set

```
unique_numbers = {1, 2, 3, 4}

for num in unique_numbers:
    print(num)
```

O/P **:** Output (order may vary due to set being unordered):

```
1
2
3
4
```

4. Iterating Over a Dictionary : Dictionaries store data as key-value pairs**.**
   EX : Iterating Over Keys

```
student = {"name": "John", "age": 20, "grade": "A"}
```

```
for key in student:
    print(key, ":", student[key])
```

O/P :

```
name : John
age : 20
grade : A
```

EX : Iterating Over Values

```
student = {"name": "John", "age": 20, "grade": "A"}

for value in student.values():
    print(value)
```

O/P :

```
John
20
A
```

EX : Iterating Over Key-Value Pairs

```
student = {"name": "John", "age": 20, "grade": "A"}

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

O/P :

```
name -> John
age -> 20
grade -> A
```

6. Generators and Iterators

- **Understanding how generators work in Python.**
    - Generators in Python are a memory-efficient way to create iterators.
    - Generators are special functions that yield values one at a time instead of returning them all at once.
    - They use the yield keyword instead of return.
    - The function's execution pauses when it encounters `yield` and resumes from where it left off when called again.

Ex :

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

- yield pauses the function and remembers its state.
- Calling next(gen) resumes execution until the next yield.

Ex : Generating Fibonacci series

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b  # Move to the next numbers

fib_gen = fibonacci()

for _ in range(10):
    print(next(fib_gen), end=" ")
```

O/P :

```
0 1 1 2 3 5 8 13 21 34 55
```

- **Difference between yield and return.**

Feature	**Yield**	**Return**
Behavior	Pauses function execution	Exits function completely
State memory	Remember previous stat	No memory(restart each time)
Output	Produces a sequence of values	Return a single value
Use case	Large data,infinite loop,streaming	One-time computations

- **Understanding iterators and creating custom iterators**.
  - An iterator in Python is an object that allows you to traverse through a sequence (like a list, tuple, or string) one element at a time.

- Uses **__iter__()** to return itself.
- Uses **__next__()** to return the next value.
- Raises **StopIteration** when there are no more items.
  Ex :
```
nums = [1, 2, 3]
iterator = iter(nums)  # Get an iterator

print(next(iterator))  # Output: 1
print(next(iterator))  # Output: 2
print(next(iterator))  # Output: 3
# print(next(iterator))  # Raises StopIteration
```
note :

- `iter()` gets an iterator from an iterable.
- `next()` fetches the next value.
- When no values are left, it raises `StopIteration`.

- **Creating a custom Iterator :**
  You can create a **custom iterator** by defining a class with:
  - `__iter__()` → Returns the iterator object.
  - `__next__()` → Returns the next item.

  Ex :
```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self  # The iterator itself

    def __next__(self):
        if self.current > self.end:
            raise StopIteration  # Stop when reaching the end
        value = self.current
        self.current += 1  # Move to the next value
        return value

# Using the custom iterator
counter = Counter(1, 5)

for num in counter:
    print(num)  # Output: 1 2 3 4 5
```

7. Functions and Methods

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

A function in Python is a block of reusable code that performs a specific task. Functions help in writing modular, reusable, and organized code.

- **Defining a Function :**
  Functions in Python are defined using the def keyword.

  Syntax :
  ```
  def  function_name(parameters):
      """Optional docstring describing the function"""
      # Function body (code)
      return value  # (Optional) Returns a value
  ```

- **Calling a Function :**
- To execute a function, you **call it** by its name and pass required arguments (if any).

  Example: Simple Function
  ```
  def greet():
      print("Hello, welcome to Python!")

  # Calling the function
  greet()
  ```

  O/P :
  ```
  Hello, welcome to Python!
  ```

- **Function arguments (positional, keyword, default).**
  Functions in Python can accept arguments in different ways:

1. **Positional Arguments** (default method)

   Positional arguments(Ordered Argument) are passed in the same order as they are unction. **Order matters** in positional arguments!

   Ex :
   ```
   def greet(name, age):
       print(f"Hello {name}, you are {age} years old.")

   # Calling function with positional arguments
   greet("Alice", 25)  # Correct order
   greet(25, "Alice")  # Incorrect order (wrong output)
   ```

   O/P :
   ```
   Hello Alice, you are 25 years old.
   Hello 25, you are Alice years old.  # Incorrect order changes meaning
   ```

2. **Keyword Arguments** (named Argument) :
   You can specify arguments by their **parameter name**, allowing flexibility in order.

Order doesn't matter when using keyword arguments.
Makes the function call more readable.

Ex :
```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

# Calling function with keyword arguments
greet(age=30, name="Bob")  # Order does not matter
```

O/P :
```
Hello Bob, you are 30 years old.
```

3. **Default Arguments** (predefined values) :
   If an argument is not provided, a default value is used.

   Ex :
```
def greet(name="Guest", age=18):
    print(f"Hello {name}, you are {age} years old.")

# Calling function without arguments
greet()  # Uses default values

# Calling function with one argument
greet("Charlie")  # Uses default age

# Calling function with all arguments
greet("Diana", 22)
```

   O/P :
```
Hello Guest, you are 18 years old.
Hello Charlie, you are 18 years old.
Hello Diana, you are 22 years old.
```

4. **Variable-Length Arguments** (`*args` and `**kwargs)` :
   Sometimes, we don't know how many arguments will be passed.

   1)*args (Multiple Positional Arguments):
   Allows passing any number of positional arguments as a tuple.

   Ex :
```
def  add_numbers(*args):
    total = sum(args)
    print("Sum:", total)
```

```
add_numbers(2, 3)       # Sum: 5
add_numbers(1, 2, 3, 4) # Sum: 10
```

# *args collects all extra positional arguments into a tuple.

2) ****kwargs (Multiple Keyword Arguments**) :

Allows passing any number of keyword arguments as a dictionary.

Ex :

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

show_details(name="Eve", age=28, city="New York")
```

O/P :

```
name: Eve
age: 28
city: New York
```

# **kwargs collects all extra keyword arguments into a dictionary.

- **Scope of variables in Python.**

  Variable scope determines where in the program a variable can be accessed or Python, variables can have different scopes depending on where they are declared.

  There are **four types** of variable scopes in Python:

1. **Local Scope** (inside a function) : A variable declared inside a function is local and can only be accessed within that function.

   Ex :

   ```
   def my_function():
      x = 10  # Local variable
      print("Inside function:", x)

   my_function()

   # print(x)  # Error: x is not defined outside the function
   ```

   O/P :

   ```
   Inside function: 10
   ```

   # The variable **x** only exists inside my_function() and cannot be used outside.

2. **Enclosing Scope** (nested function) : A variable inside an outer function is accessible to an inner (nested) function using the nonlocal keyword.

Ex :

```
def outer_function():
    a = 5  # Enclosing variable

    def  inner_function():
        nonlocal a  # Refers to the variable in the outer function
        a += 1
        print("Inside inner function:", a)

    inner_function()
    print("Inside outer function:", a)

outer_function()
```

O/P:

```
Inside inner function: 6
Inside outer function: 6
```

# Without nonlocal, the inner function cannot modify a from the outer function

3. **Global Scope** (outside a function & Accessible Everywhere) : A variable declared outside any function is global and can be accessed anywhere except inside functions (unless explicitly declared global).

   Ex :

```
x = 100  # Global variable

def my_function():
    print("Inside function:", x)  # Accessing global variable

my_function()
print("Outside function:", x)  # Global variable accessible everywhere
```

O/P :

```
Inside function: 100
Outside function: 100
```

- To modify a global variable inside a function, use the global keyword.

  Ex :

```
y = 50  # Global variable

def change_global():
    global y  # Declaring y as global
    y += 10
    print("Inside function:", y)
```

```
change_global()
print("Outside function:", y)  # Global variable is modified

O/P :
Inside function: 60
Outside function: 60
```

- **Built-in methods for strings, lists, etc**.
- **String** :

  Python string methods *is a collection of in-built Python functions that operates on strings.* Python string is a sequence of Unicode characters that is enclosed in quotation marks.

  The below Python functions are used to change the case of the strings. Let's look at some Python string methods with examples:
- **lower( ):** Converts all uppercase characters in a string into lowercase
- **upper( ):** Converts all lowercase characters in a string into uppercase
- **title( ):** Convert string to title case
- **swapcase( ):** Swap the cases of all characters in a string
- **capitalize( ):** Convert the first character of a string to uppercase
- **join(iterable)**: Concatenates elements of an iterable with a specified separator.
- **startswith(prefix)**: Checks if the string starts with the specified prefix.
- **endswith(suffix)**: Checks if the string ends with the specified suffix.
- **strip( ):** Removes any leading and trailing characters (space is the default).

- **List :** Python list methods are built-in functions that allow us to perform various operations on lists, such as adding, removing, or modifying element.

  Let's look at different list methods in Python:
- **append( ):** Adds an element to the end of the list.
- **copy( ):** Returns a shallow copy of the list.
- **clear( ):** Removes all elements from the list.
- **count( ):** Returns the number of times a specified element appears in the list.
- **extend( ):** Adds elements from another list to the end of the current list.
- **index( ):** Returns the index of the first occurrence of a specified element.
- **insert( ):** Inserts an element at a specified position.
- **pop( ):** Removes and returns the element at the specified position (or the last element if no index is specified).
- **remove( )**: Removes the first occurrence of a specified element.
- **reverse( ):** Reverses the order of the elements in the list.
- **sort( ):** Sorts the list in ascending order (by default).

8. Control Statements (Break, Continue, Pass)

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

  Loop control statements in Python are special statements that help control the execution of loops (for or while).

  They let you modify the default behavior of the loop, such as stopping it early, skipping an iteration, or doing nothing temporarily.

  Python supports the following control statements:

- **Break statement** : The break statement in Python is used to exit or “break” out of a loop (either a for or while loop) prematurely, before the loop has iterated through all its items or reached its condition. When the break statement is executed, the program immediately exits the loop, and the control moves to the next line of code after the loop.

  Ex :
  ```
  # Using For Loop
  for i in range(5):
      if i == 3:
          break  # Exit the loop when i is 3
      print(i)

  # Using While Loop
  i = 0
  while i < 5:
      if i == 3:
          break  # Exit the loop when i is 3
      print(i)
      i += 1
  ```

  O/P :
  ```
  0

  1

  2

  0

  1

  2
  ```

- **Continue statement** : Python Continue statement is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.

  Ex :
  ```
  for i in range(5):
      if i == 3:
          continue  # Skip the rest of the code for i = 3
      print(i)
  ```

  O/P :
  ```
  0
  1
  2
  4
  ```

- **Pass statement** : Pass statement in Python is a null operation or a placeholder. It is used when a statement is syntactically required but we don’t want to execute any code. It does nothing but allows us to maintain the structure of our program.

  Ex :
  ```
  for i in range(5):
      if i == 3:
          pass  # Placeholder for future code
      print(i)
  ```

  O/P :
  ```
  0
  1
  2
  3
  4
  ```

9. String Manipulation

- **Understanding how to access and manipulate strings.**
  - A string is a sequence of characters. Python treats anything inside quotes as a string. This includes letters, numbers, and symbols. Strings can be created using either single (‘) or double (“) quotes.
  - Python has no character data type so single character is a string of length 1.
  - Strings in Python are sequences of characters, so we can access individual characters using **indexing.** Strings are indexed starting from **0** and **-1** from end**.** This allows us to retrieve specific characters from the string.

    Ex :
    ```
    s = "GfG"
    print(s[1])      # access 2nd char
    s1 = s + s[0]    # update
    print(s1)        # print
    ```

    O/P :
    ```
    f
    GfGG
    ```

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

  In python, concatenations and repetitions are supported by sequence data types both mutable(list) and immutable(tuple, strings).

- **Concatenation**: Concatenation is done by + operator. Concatenation is supported by sequence data types(string, list, tuple). Concatenation is done between the same data types only.

  Syntax:

  a+b

  String Concatenation: The string is an immutable sequence data type. Concatenating immutable sequence data types always results in a new object.

  Ex :

```
s1 = “Python”
s2 = “Welcome!”
s3 = “Hello”
s4 = s2 +s1 +s3
print(s4)          #O/P : WelcomePythonHello
```

- **Repetition :** Sequences datatypes (both mutable and immutable) support a **repetition** operator * .The repetition operator * will make multiple copies of that particular object and combines them together. When * is used with an integer it performs multiplication but with list, tuple or strings it performs a repetition

  Syntax:

  a*b

  Ex :

```
s1 = “Python”
print(s1*3)      #O/P : PythonPythonPython
```

- **upper( ) and lower( )**: upper( ) method converts all characters to uppercase. lower() method converts all characters to lowercase.

  Ex :

```
s = "Hello World"

print(s.upper())   # output: HELLO WORLD

print(s.lower())   # output: hello world
```

- **String slicing.**
  - Slicing is a way to extract portion of a string by specifying the **start** and **end** indexes. The syntax for slicing is **string[start:end]**, where **start** starting index and **end** is stopping index (excluded).

    Ex :
    ```
    s = "GeeksforGeeks"

    # Retrieves characters from index 1 to 3: 'eek'
    print(s[1:4])

    # Retrieves characters from beginning to index 2: 'Gee'
    print(s[:3])

    # Retrieves characters from index 3 to the end: 'ksforGeeks'
    print(s[3:])

    # Reverse a string
    print(s[::-1])
    ```

    O/P :
    ```
    eek
    Gee
    ksforGeeks
    skeeGrofskeeG
    ```

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

- **How functional programming works in Python.**
  Functional programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions. It avoids changing state and mutable data, making code cleaner, more predictable, and easier to debug**.**
  Key concepts of Functional Programming in python
  - **First-Class Functions** (Functions are treated like variables)
  - **Pure Functions** (No side effects, same input → same output)
  - **Higher-Order Functions** (Functions that take or return functions)
  - **Immutability** (Avoid modifying data)
  - **Recursion** (Loops using function calls)
  - **Functional Tools** (`map()`, `filter()`, `reduce()`, `lambda`)

  1. **First-Class Functions :**
     Functions can be assigned to variables, passed as arguments, and returned from other functions**.**
     Ex :
     ```
     def greet(name):
         return f"Hello, {name}!"
     ```

```
# Assign function to a variable
greeting = greet
print(greeting("Alice"))  # Output: Hello, Alice!
```

2. **Pure Functions :**

   A function is pure if :

   1.It always returns the same output for the same input.

   2.It has no side effects (no modifying global variables, files,etc.).

   Ex :

```
def pure_function(x, y):
    return x + y

print(pure_function(3, 4))  # Output: 7
```

3. **Higher-Order Functions :**

   Functions that take functions as argument**s** or return functions**.**

   Ex : Passing a function as an argument

```
def apply_function(func, value):
    return func(value)

def square(x):
    return x ** 2

print(apply_function(square, 5))  # Output: 25
```

   Ex : Returning a function

```
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

double = multiplier(2)
print(double(5))  # Output: 10
```

4. **Immutability :**

   In FP, **we** avoid modifying existing data and instead return new data**.**

   Ex :

```
my_list = [1, 2, 3]
new_list = my_list + [4]  # Creates a new list instead of modifying the original
print(new_list)  # Output: [1, 2, 3, 4]
```

5. **Recursion :**
   Functional programming often uses recursion instead of loops**.**
   Ex :
   ```
   def factorial(n):
      if n == 0:
          return 1
      return n * factorial(n - 1)
   print(factorial(5))                  #O/P : 120

   print(factorial(5))  # Output: 120
   ```
6. Functional Tools :
   Map( ), reduce( ),filter( ) :
   - map(function, iterable): Applies a function to all elements in an iterable.
   - filter(function, iterable): Filters elements based on a function.
   - reduce(function, iterable): Applies a function cumulatively (from functools).
     Ex :
     ```
     from functools import reduce
     numbers = [1, 2, 3, 4, 5]

     squared = list(map(lambda x: x * x, numbers))
     evens = list(filter(lambda x: x % 2 == 0, numbers))
     total = reduce(lambda x, y: x + y, numbers)

     print(squared)  # Output: [1, 4, 9, 16, 25]
     print(evens)    # Output: [2, 4]
     print(total)    # Output: 15
     ```
   lamda func:
   A compact way to create small, anonymous functions.
   Ex :
   ```
   double = lambda x: x * 2
   print(double(5))  # Output: 1
   ```

- **Using map(), reduce(), and filter() functions for processing data.**
  - **map ( ) :** Apply function to all element
  - The map() function applies a given function to all elements in an iterable (e.g., list, tuple).
  - Ex :
    ```
    # Convert temperatures from Celsius to Fahrenheit
    temps_celsius = [0, 10, 20, 30, 40]

    # Formula: (C * 9/5) + 32
    ```

```
temps_fahrenheit = list(map(lambda c: (c * 9/5) + 32, temps_celsius))

print(temps_fahrenheit)
# Output: [32.0, 50.0, 68.0, 86.0, 104.0]
```

- **filter ( ) :** filter the element based on condition
- The filter() function removes elements that do not satisfy a given condition.
- Ex :

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

evens = list(filter(lambda x: x % 2 == 0, numbers))

print(evens)
# Output: [2, 4, 6, 8, 10]
```

- **reduce ( ) :** cumulatively apply a function
- The reduce() function applies a function to a sequence cumulatively, reducing it to a single value. It is available in the functools module.
- Ex : find the product oaf all the number in given list
- from functools import reduce

```
numbers = [1, 2, 3, 4, 5]

product = reduce(lambda x, y: x * y, numbers)

print(product)
# Output: 120 (1 × 2 × 3 × 4 × 5)
```

- **Introduction to closures and decorators**
    - Closures and decorators are powerful concepts in Python that allow for elegant function manipulation and code reusability.
    - **Closures ( ):**
      A closure is a function that remembers the environment in which it was created, even if that environment is no longer available.
      How Closure work ?
    - A nested function can capture variables from its enclosing scope.
    - Even after the outer function has finished execution, the inner function remembers the captured variables.
    - Ex :

```
def outer_function(message):
    def inner_function():
        print(f"Message: {message}")  # Inner function remembers
`message`
    return inner_function  # Returning the inner function
```

```
# Create a closure
closure_func = outer_function("Hello, Python!")

# Call the closure
closure_func()
# Output: Message: Hello, Python!
```

- **Decorators funct :**
- A decorator is a function that modifies the behavior of another function without changing its code**.**
  It wraps another function, adding extra functionality before or after its execution.
  Ex :

```
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed before", original_function.__name__)
        original_function()
        print("Wrapper executed after", original_function.__name__)
    return wrapper_function

@decorator_function  # Using the decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

O/P :

```
Wrapper executed before say_hello
Hello, World!
Wrapper executed after say_hello
```