Module 6 : Python Fundamentals

Introduction to Python:

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

Ans:-

- Python is a simple, high-level, and interpreted programming language.
- It is easy to learn because its syntax is like English, which makes it beginner-friendly.
- Python is widely used in web development, data science, machine learning, automation, and more.

Features of Python:

1. Simple and Easy:

   Easy to read, write, and understand.

2. High-Level Language:

   We don't need to worry about complex machine details.

3. Interpreted Language:

   Code runs line by line, so errors are easy to find.

4. Portable:

   Works on different platforms (Windows, Linux, Mac).

5. Object-Oriented:

   Supports classes and objects.

6. Large Library Support:

   Has many built-in modules and external libraries.

2. History and evolution of Python.

Ans:-

- Python was created by Guido van Rossum in 1989 at the National Research Institute (CWI) in the Netherlands.
- It was first released in 1991 as Python 0.9.0, which already had features like functions and exception handling.
- python 1.0 came in 1994, adding new features like modules.
- Python 2.0 was released in 2000, introducing features like garbage collection and Unicode support.
- Python 3.0 was released in 2008 with major improvements but not backward compatible with Python 2.
- Today, Python is one of the most popular programming languages, widely used in web development, AI, machine learning, data science, automation, and more.

3. Advantages of using Python over other programming languages.

Ans:-

1. Easy to Learn and Use:

   Python has simple English-like syntax, so beginners can learn it quickly compared to other programming language.

2. Cross-Platform:

   Python programs can run on Windows, Mac, and Linux without changes.

3. Large Library Support:

   Comes with a huge standard library and many external libraries for AI, data science, web development, etc.

4. Interpreted Language:

   Runs code line by line, making debugging easier.

5. Object-Oriented and Procedural:

Supports multiple programming styles.

6. Community Support:

   Has a large and active community for help, tutorials, and tools.

7. Rapid Development:

   Less code compared to C, C++, or Java, which saves time in development.

8. Versatile:

   Used in many areas: web development, data science, machine learning, automation, IoT, etc.

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

Ans:-

Installing Python and Setting Up the Development Environment:

1. Install Python:

   - Go to the official website: https://www.python.org. o Download the latest version of Python (3.x).
   - During installation, tick "Add Python to PATH" and then click Install Now.

2. Check Installation:

   - Open Command Prompt / Terminal.
   - Type: python --version (or python3 --version) o If version shows, Python is installed successfully.

Development Environment Options:

1. Anaconda (Best for Data Science & Machine Learning):

- Download from https://www.anaconda.com. o It comes with Python, Jupyter Notebook, and many useful libraries pre-installed.
- Good choice for data analysis, AI, ML projects.

2. PyCharm (Best for Professional Python Projects):

   - Download from https://www.jetbrains.com/pycharm. o It is a powerful IDE with debugging, testing, and project management tools.
   - Good for big projects and professional development.

3. Visual Studio Code (VS Code) (Lightweight & Popular):

   - Download from https://code.visualstudio.com. o Install the Python extension for better coding support. o Lightweight, easy to use, and supports multiple languages.

5. Writing and executing your first Python program.

Ans:-

Writing and Executing Your First Python Program:

1. Open Python (IDE or Editor):

You can use:

- IDLE (comes with Python), □ VS Code,
- Anaconda.

2. Write Your First Program:

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

- print() is a built-in function in Python.
- It displays the text "Hello, World!" on the screen.

3. Save the File:

- Save the program with .py extension,

  for example: hello.py.

4. Execute the Program:

- Open Command Prompt / Terminal.
- Go to the folder where the file is saved.
- Type:

python hello.py

Output:

Hello, World!

Programming Style:

1. Understanding Python's PEP 8 guidelines.

Ans:-

1. Indentation:

   Always use 4 spaces.

2. Line Length:

   Max 79 characters.

3. Blank Lines:

   - 2 lines between functions/classes.
   - 1 line inside functions for clarity.

4. Imports:

   At top, order: standard → third-party → local.

5. Naming Rules:

    - Variables/Functions → snake_case
    - Classes → CamelCase
    - Constants → UPPER_CASE

6. Spaces:

    - a = b + c (right)
    - No extra space inside (), [], {}.

7. Comments & Docstrings:

    Write clear, explain purpose.

8. Strings:

    Use ' or " consistently.

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

Ans:-

Indentation, Comments, and Naming Conventions in Python:

1. Indentation:

- Defines code blocks in Python.
- Must use consistent spaces (usually 4).
- Example:

```
if True:
print("Hello")
```

2. Comments:

- Used to explain code (ignored by Python).
- Single-line: # comment
- Multi-line: ''' comment ''' or """ comment """

3. Naming Conventions:

- Variables & functions:

  lower_case_with_underscores

- Classes:

  CapitalizedWords

- Constants:

  UPPERCASE

- Use meaningful names.

3.Writing readable and maintainable code.

Ans:-

Writing Readable & Maintainable Code

1. Follow PEP 8 Guidelines:
   - Use proper indentation (4 spaces). o Keep line length ≤ 79 characters.
   - Add blank lines to separate code sections.
2. Use Meaningful Names:
   - Variables, functions, and classes should have clear names.
3. Add Comments & Docstrings:
   - Write short comments to explain tricky logic.
   - Use docstrings (""" ... """) for functions and classes.
4. Keep Functions Small:
   - Each function should do only one task. o Easier to test and reuse.

5. Consistent Naming Style:
    - Variables/functions → snake_case o Constants → UPPERCASE
6. Avoid Hardcoding:
    - Use variables or constants instead of fixed values.
7. Handle Errors Gracefully:
    - Use try-except for exceptions.
8. Keep Code DRY (Don't Repeat Yourself):
    - Reuse code with functions or loops.

Core Python Concepts:

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

Ans:-

Understanding Data Types in Python

1. Integers (int):

- Whole numbers (positive, negative, or zero).

2. Floats (float):

- Numbers with decimals.

3. Strings (str):

- Text inside quotes (' ' or " ").

4. Lists (list):

- Ordered, changeable collection.
- Allows duplicates.

5. Tuples (tuple): □ Ordered, unchangeable collection.

- Allows duplicates.

6. Dictionaries (dict):

- Key–Value pairs, unordered, changeable.
- Keys must be unique.

7. Sets (set):

- Unordered, unique values only (no duplicates).

3. Python variables and memory allocation.

Ans:-

Python Variables & Memory Allocation:

- Variable = name that store value in memory.
- Python is dynamically typed (no need to declare type).
- Variables point to objects, not store values directly.

Example:

```
a = 5
b = 5   # both point to same object in memory  Immutable (int, float,
```

str, tuple): new object created when changed.

- Mutable (list, dict, set): modified in the same memory.
- Garbage Collector:

  frees memory when no variable refers to the object.

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

Ans:-

1. Arithmetic Operators (Math operations):

Operator	Meaning	Example (a=10, b=3)	Output
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3.33
//	Floor Division	a // b	3
%	Modulus (remainder)	a % b	1
**	Exponent (power)	a ** b	1000

2. Comparison Operators (Return True/False):

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

3. Logical Operators (Combine conditions):

Operator	Meaning	Example (a=10, b=3)	Output
and	Both conditions true	(a > 5 and b < 5)	True
or	At least one true	(a > 5 or b > 5)	True
not	Reverses result	not(a > b)	False

4. Bitwise Operators (Work on binary numbers):

Operator	Meaning	Example (a=10 → 1010, b=3 → 0011)	Output
&	AND	a & b → 1010 & 0011	2
||	OR	OR	`a
^	XOR	a ^ b	9
~	NOT	~a	-11
<<	Left shift	a << 1	20
>>	Right shift	a >> 1	5

5. Membership Operators:

Operator	Meaning	Example	Output
in	Returns True if a value is present	'a' in 'apple'	True
not in	Returns True if a value is not present	'x' not in 'apple'	True

6. Identity Operators:

Operator	Meaning	Example	Output
is	Returns True if both variables refer to the same object	a = [1,2]; b = a; a is b	True
is not	Returns True if both variables do not refer to same object	a = [1,2]; b = [1,2]; a is not b	True

Conditional Statements:

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

Ans:-

- Conditional statements are used to make decisions in a program.
- They check a condition (True/False) and run code accordingly.

1. if statement:

- Runs a block of code only if the condition is true.
- Executes a block only when the condition is True.
- Skips the block if the condition is False.

Example:

```
age = 18

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

2. if...else statement:

- else gives an alternative when the condition is false.
- Used to check multiple conditions after an if.
- Runs only if the previous conditions are False.

Example:

```
age = 16

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

3. if...elif...else statement:

- elif means else if.
- Used when we have multiple conditions to check.

- Executes when all previous conditions are False.

Example:

```
marks = 75

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

2. Nested if-else conditions.

Ans:-

- A nested if-else means writing an if or else inside another if or else block.

- It allows hierarchical decision-making (step-by-step checks).

Key Points:

  - Useful for multi-level conditions.
  - Indentation is very important to show which block belongs where.
  - Too many nested blocks may make code hard to read, so use wisely.

Syntax:

```
if condition1:      #
Outer if block     if
condition2:         #
Inner    if    block
statement1    else:
#  Inner   else   block
statement2 else:
#   Outer   else   block
```

statement3 Example:

```
num = 15

if num > 0:      if num % 2 == 0:
print("Positive   Even   number")
else:              print("Positive  Odd
number") else:
print("Number is Negative")
```

Looping (For, While)

1.Introduction to for and while loops.

Ans:-

Loops are used to repeat a block of code multiple times until a condition is met.

1. for loop:

- Used when we know how many times we want to repeat.

- Works with sequences (list, string, range, etc.).

Syntax:

for variable in sequence:

code block Example:

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

2. while loop:

- Used when we don't know the exact number of repetitions.
- Runs as long as the condition is True.

Syntax:

```
while condition:
# code block
```

Example:

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

2. How loops work in Python.

Ans:-

1. Loop Start:

- Python begins at the loop statement (for or while).

2. Condition Check:

- For for loop:

  It takes the next item from a sequence (list, range, string, etc.).

- For while loop:

  It checks if the condition is True.

3.Code Execution:

- If the condition is True (or an item is available), the loop body runs.

4.Update Step:

- for loop:

  Automatically moves to the next item.

- while loop:

  You must manually update the variable, otherwise it may run forever.

5.Repeat:

- Steps 2–4 repeat until there are no more items (for) or the condition becomes False (while).

6.Loop End:

- When the loop finishes, Python moves to the next statement after the loop.

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

Ans:- 1. Loop with

List:

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

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

Output:

```
apple banana
cherry
```

2. Loop with Tuple:

```
numbers = (10, 20, 30)

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

Output:

```
10
20 30
```

3. Loop with Set (unordered, no duplicates):

```
colors = {"red", "green", "blue"}

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

Output (order may vary):

```
red green
blue
```

4. Loop with Dictionary:

- By keys:

```
student = {"name": "Amit", "age": 21, "city": "Delhi"}

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

Output:

```
name : Amit
age : 21
city : Delhi
```

- By values:

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

- By key & value together:

```
for key, value in student.items():
print(key, "=", value)s
```

Generators and Iterators

1.Understanding how generators work in Python.

Ans:-

A generator is a special type of function that yields values instead of returning them all at once.

- Uses the yield keyword (instead of return)
- Remembers its state between calls
- Produces a sequence of values lazily (one by one)

Example:

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

# Using the generator
gen = simple_generator()

print(next(gen))  # 1 print(next(gen))
# 2 print(next(gen))  # 3
# print(next(gen))  # Raises StopIteration (no more values)
```

- Every time next() is called, the function runs until it reaches yield, then pauses and returns the value.

Why Use Generators?

Feature	Normal Function / List	Generator
Memory usage	Stores entire data	Generates on demand
Execution	Returns all at once	Returns one at a time
Infinite sequences	Not practical	Possible

Performance for big data	Slower	Faster / efficient

2.Difference between yield and return.

Ans:-

Feature	return	yield
Usage	Used in normal functions	Used in generator functions
What it does	Ends the function and returns a value	Pauses the function and yields a value
Number of values	Returns once	Can yield multiple times
Function behavior	Returns a single value or object	Creates a generator object (iterator)
Memory	Returns everything at once (may use more memory)	Generates values one by one (memoryefficient)
State	Function ends after return	Function remembers its state after yield
Iteration	You get one result	You can loop through many yielded values

3.Understanding iterators and creating custom iterators.

Ans:-

- An iterator is an object that allows you to loop through a sequence one element at a time using next().

It must define two methods:

- __iter__():

  returns the iterator object itself.

- __next__():

returns the next value or raises StopIteration when finished.

Example:

```
class EvenNumbers:
max_num = 10
current = 0

def __iter__(self):
return self

def __next__(self): if self.current
<= self.max_num:
num = self.current
self.current += 2
return num else:
raise StopIteration

for n in EvenNumbers():
print(n)
```

Functions and Methods Theory:

1.Defining and calling functions in Python.

Ans:-

- A function is a block of reusable code that performs a specific task.

- Instead of writing the same code again and again, we put it inside a function and call it whenever needed.

Defining a Function:

In Python, a function is defined using the def keyword:

```
def function_name(parameters):
# block of code
return result
```

- def → keyword to define a func on
- function_name → the name you give to the func on
- parameters (optional) → inputs the func on can take
- return (optional) → sends a value back a er execu on Example 1: Function without parameters

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

Calling the function:

```
greet()
```

Example 2: Function with parameters

```
def add_numbers(a, b):
result = a + b     return
result    Calling    the
function:

print(add_numbers(5, 3))
```

Example 3: Function with default parameter

```
def  greet_user(name="Guest"):
print("Hello,",   name)   Calling
the function:

greet_user("Hensi")
greet_user()
```

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

Ans:-

1. Positional Arguments:

- Values are passed in the same order as parameters are defined.
- Order matters here.

```
def student_info(name, age):
print("Name:", name)
print("Age:", age)

# Calling with positional arguments: student_info("Hensi",
22)
```

2. Keyword Arguments:

- You specify the parameter name while calling.
- Order doesn't matter, because Python matches by name.

```
def student_info(name, age):
print("Name:", name)
print("Age:", age)

# Calling with keyword arguments: student_info(age=22,
name="Hensi")
```

3. Default Arguments:

- If a value is not passed, Python uses the default value.
- If a value is passed, it overrides the default.

```
def                    greet(name="Guest"):
print("Hello,", name)

# Calling with and without argument:
greet("Hensi") greet()
```

3. Scope of variables in Python.

Ans:-

 Scope means the area of a program where a variable is accessible.

In Python, we have two main scopes:

1. Local Scope:

- A variable declared inside a function is local.
- It can be accessed only within that function.

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

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

Output: Inside

function: 10

2. Global Scope:

- A variable declared outside all functions is global.
- It can be accessed anywhere in the program (inside and outside functions).

```
x = 50  # global variable

def my_function():
print("Inside function:", x)

my_function() print("Outside
function:", x)
```

Output:

Inside function: 50
Outside function: 50

3. Modifying Global Variable inside a Function:

 If you want to change a global variable inside a function, use the global keyword.

```
x = 100

def           my_function():
global x
x = 200   # modifying global variable     print("Inside
function:", x)

my_function() print("Outside
function:", x)
```

Output:

```
Inside function: 200
Outside function: 200
```

4. LEGB Rule (Python Scope Resolution Order):

When Python looks for a variable, it follows the LEGB rule:

1. L → Local (inside current function)
2. E → Enclosing (inside nested/outer functions)
3. G → Global (defined at the top level of the program)
4. B → Built-in (Python's built-in names like len, print)

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

Ans:-

1. String Methods:

- Strings are sequences of characters. Python gives many built-in methods:

```
text = " Hello Python "
```

 Case Conversion

- text.upper() → " HELLO PYTHON "
- text.lower() → " hello python "
- text.title() → " Hello Python "

 Remove Spaces

- text.strip() → "Hello Python" (removes extra spaces at start & end)Search & Replace

- text.find("Python") → 7 (index of first occurrence)
- text.replace("Python", "World") → " Hello World "

 Check Content

- "hello".isalpha() → True (all letters)
- "123".isdigit() → True (all digits)
- "hello123".isalnum() → True (letters + numbers)

 Split & Join

```
text = "Python is fun"
print(text.split())   # ['Python', 'is', 'fun']
```

2. List Methods:

- Lists store multiple values in one variable.

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

 Adding Elements

- fruits.append("mango") → adds at end
- fruits.insert(1, "orange") → insert at index 1

 Removing Elements

- fruits.remove("banana") → remove by value
- fruits.pop(0) → remove by index (returns removed item)
- fruits.clear() → empty the list

 Searching & Counting

- fruits.index("cherry") → gives index of "cherry"
- fruits.count("apple") → number of times "apple" occurs

 Sorting

```
numbers = [4, 2, 9, 1]
numbers.sort()        # [1, 2, 4, 9] (ascending) numbers.sort(reverse=True)
# [9, 4, 2, 1] (descending)
```

 Copy & Reverse

- fruits.copy() → makes a copy
- fruits.reverse() → reverses list

Control Statements (Break, Continue, Pass):

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

Ans:-

1. break: Use: stop the loop completely.

- After break, the loop ends, and control goes to the next statement outside the loop.

Example:

```
for i in range(1, 6):
if i == 3:        break
print(i)
```

2. continue: Use: Skip the current iteration and go to the next iteration of the loop.

- The loop continues but ignores the remaining code for that iteration.

Example:

```
for i in range(1, 6):
if i == 3:
continue print(i)
```

3. pass: Use: Do nothing

(placeholder).

- Often used when you want to keep the loop/body syntactically correct but don't want any code there yet.

Example:

```
for i in range(1, 6):
if i == 3:
pass # does nothing
print(i)
```

String Manipulation:

1. Understanding how to access and manipulate strings. Ans:-

- Strings are a sequence of characters.
- In python ,string are enclosed within single(') or double(") or triple("""") quotation marks.

Accessing Strings:

- You can access characters using indexing and slicing.

Example:

```
s = "Hello" print(s[0]) #
H (first char) print(s[-1]) #
o (last char) print(s[1:4]) #
ell (slice)
```

Manipulating Strings:

- Strings are immutable (cannot change directly).
- You create new strings using operations and methods.

Common operations:

```
s = "hello world"

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

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

Ans:-

Basic String Operations:

1. Concatenation (joining strings):

- + = join strings.

```
s1 = "Hello" s2
= "World"
print(s1 + " " + s2)   # Hello World 2.
```

Repetition (repeating strings):

- * = repeat strings.

```
s = "Hi "
print(s * 3)   # Hi Hi Hi
```

3. String Methods:

- upper() = converts a string into uppercase. □ lower() = converts a string into lowercase.
- title() = convert the first character of each word to uppercase.
- strip() = removes any white space from stand and end.
- replace(old,new) = replaces part of string.
- split() = splits the string into list.
- len() = returns the length of a string.

Example:

```
text = "  hello world  " print(text.upper())
#  HELLO  WORLD  print(text.lower())
# hello world print(text.title())     # Hello
World print(text.strip())     # hello world
print(text.replace("world", "Python"))  # hello Python
print(text.split())     # ['hello', 'world'] print(len(text))
#12
```

3. String slicing.

Ans:-

- Slicing in python is a feature that enables accessing parts of the sequence.
- String slicing allows you to get subset of characters from a string using specified range of indices.

Syntax: starting[start : end : step]

- Start : the index to start slicing.default value is 0.
- End : the index to stop slicing. deafault value is length of string.
- Step :how muchto increment the index after each character. deafault value is 1.

Example:

name = “MADHAV”

name[0:1] = name[:1] = ‘M’ #first char

name[0:2] = name[:2] = ‘MA’ #first 2 chars

name[2:5] = ‘DHA’ #third to fifth chars

name[5:] = name[-1:] = ‘V’ #last char

name[4:] = name[-2:] = ‘AV’ #last 2 chars

name[0:5:2] = name[0::2] = ‘MDA’ #every second chars

name[1:-1] = ‘ADHA’ #exclude first & last chars

name[:] = name[::] = ‘MADHAV’ #all chars

name[::-1] = ‘VAHDAM’ #reverse the string

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

1.How functional programming works in Python.

Ans:-

- Functional programming in Python = using functions to transform data without changing it, making your code cleaner, shorter, and easier to test.

Main Points:

1. Functions are first-class:

You can store them in variables, pass them to other functions, or return them.

2. No changing data:

   Instead of modifying values, you create new ones.

3. Use built-in tools:

   like map(), filter(), and reduce() to work with data in a clean way.

4. Focus on “what to do”, not “how to do it.”

Common Functional Tools:

- map(func, list):

  apply a function to each item.

- filter(func, list):

  keep only items that return True.

- reduce(func, list): combine items step by step (from functools).
- lambda:

  small anonymous functions.

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

- These functions make data processing short, clean, and functional.

  1.map():transform data.

  2.filter():select data.

  3.reduce(): combine data.

Example:

```
from functools import reduce

# Sample data
numbers = [1, 2, 3, 4, 5, 6]

# 1.map() → Square each number squared = list(map(lambda x: x * x, numbers))
print("Squared:", squared)   # [1, 4, 9, 16, 25, 36]

# 2.filter() → Keep only even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers)) print("Even numbers:", even_numbers)  # [2, 4, 6]

# 3.reduce() → Find the sum of all numbers total_sum = reduce(lambda a, b: a + b, numbers)  print("Sum:", total_sum)   # 21
```

Explanation:

- map() applies the function to every element.
- filter() keeps elements that return True from the function.
- reduce() combines elements step by step into a single value.

3. Introduction to closures and decorators.

Ans:-

1.Closures — Functions inside Functions:

- A closure is a function that remembers variables from the outer function even after the outer function is finished.

Example:

```
def outer():
message = "Hello Closure!"

def inner():
print(message)  # inner remembers 'message' from outer     return inner

# Create closure my_func = outer()
my_func()  # Output: Hello Closure!
```

Here, inner() remembers message even after outer() is done.

This is useful when you want to create functions with remembered values.

2.Decorators — Add Extra Features to Functions:

- A decorator is a function that takes another function, adds some extra behavior, and returns a new function — without changing the original function code.

Example:

```
def       decorator(func):
def wrapper():
print("Before the function runs")        func()
# run the original function         print("After
the function runs")
return wrapper

@decorator
```

```
def say_hello():
print("Hello!") say_hello()
```

Output

```
Before the function runs Hello!
After the function runs The @decorator
```

line is the shortcut for:

```
say_hello = decorator(say_hello)
```