Module-6
Python Fundamentals(theory)

1. Introduction to Python

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

Python is a **high-level, interpreted, and general-purpose programming language**.
It is famous because it is **easy to read, simple to write, and very powerful**.
Python is used in many areas like **web development, data science, artificial intelligence, machine learning, and automation**.

Features of Python:

- **Easy to learn**: Looks like English, so beginners understand it quickly.
- **Interpreted**: Runs line by line, so errors are easier to find.
- **Dynamically typed**: You don't need to declare the type of variable (e.g., int, float).
- **Object-Oriented**: Supports classes and objects.
- **Large standard library**: Many built-in functions and modules.
- **Cross-platform**: Works on Windows, macOS, and Linux.
- **Open source**: Free to use and modify.
- **Extensible**: Can connect with other languages like C, C++, and Java.

2. History and evolution of Python.

- Python was created by **Guido van Rossum** and first released in **1991**.

- It became popular because it is **simple, flexible, and powerful**.

- Over time, new versions improved Python:

  - **Python 2** (older, now outdated).
  - **Python 3** (current and widely used).

3. Advantages of using Python over other programming languages.

- **Simple and easy to learn** (syntax is close to English).
- **Works on all platforms** (Windows, Linux, macOS).

- **Rich libraries and frameworks**:
  - Django, Flask (Web development)
  - Pandas, NumPy, TensorFlow (Data Science & AI)
  - Tkinter, PyQt (GUI development)
  - Selenium (Automation)
- **Large community support** – easy to find help.
- **Integrates well** with other languages and databases.
- **Used in many fields** – web, AI, data analysis, automation, games.
- **Fast development** – great for creating prototypes quickly.

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

You can install Python in different ways:

- **Download Python** from the official website (python.org).
- **Anaconda** – useful for data science and machine learning projects.
- **PyCharm** – a powerful IDE (Integrated Development Environment).
- **VS Code** – a lightweight and popular code editor.

After installing, you can open the Python shell or use an IDE to start coding.

5. Writing and executing your first Python program.

To write your first program:

1. Open Python (IDLE, PyCharm, or VS Code).
2. Type this code:

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

3. Run the program.
4. You will see the output:

```
Hello, World!
```

This is the first step in learning Python programming.

2. Programming Style

6. Understanding Python’s PEP 8 guidelines.

What is PEP 8?

- PEP stands for **Python Enhancement Proposal**.
- **PEP 8** is the official **style guide** for writing Python code.
- It gives rules on how to format Python programs so that code is **clean, consistent, and easy to read**.

Key PEP 8 Guidelines:

1. **Indentation**
    - Use **4 spaces** for indentation.
    - Don’t mix tabs and spaces.
2. **Maximum Line Length**
    - Keep each line up to **79 characters**.
    - For comments/docstrings, max **72 characters**.
3. **Blank Lines**
    - Leave blank lines to separate functions, classes, and sections of code.
4. **Naming Conventions**
    - **Variables & functions**: lowercase_with_underscores
      Example: `student_name`, `get_value()`
    - **Classes**: CamelCase (first letter capitalized, no underscores)
      Example: `StudentRecord`, `BankAccount`
    - **Constants**: ALL_UPPERCASE
      Example: `PI = 3.14`, `MAX_LIMIT = 100`
5. **Imports**
    - Each import should be on a new line.

```
   import os
import sys
```

6. **Spaces Around Operators**
    - Add spaces around operators and after commas.
    - Example:
    - `result = a + b`
    - `numbers = [1, 2, 3]`
7. **Comments**
    - Write comments to explain code.
    - Use `#` for single-line comments.
    - Use triple quotes (`""" ... """`) for docstrings in functions and classes.
8. **Docstrings**
    - Every function/class/module should have a short description.

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

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

Indentation

- In Python, indentation (spaces at the start of a line) is **very important**.
- It tells Python which block of code belongs together.
- Example:
- `if True:`
- `    print("This is inside the block")   # 4 spaces`
- `print("This is outside the block")`
- Rule: Use **4 spaces** for each indentation level (not tabs).

Comments

- Comments are notes you add in the code to explain what it does.
- They are ignored by Python, but they make code easier to understand.
- Types of comments:
    - **Single-line**:
    - `# This is a single line comment`
    - **Multi-line**:
    - `"""`
    - `This is a`
    - `multi-line comment`
    - `"""`

Naming Conventions (PEP 8 guidelines)

- **Variables & functions** → lowercase with underscores
  Example: `student_name`, `calculate_sum()`
- **Classes** → Capitalized words (CamelCase)
  Example: `StudentRecord`, `BankAccount`
- **Constants** → All uppercase
  Example: `PI = 3.14`, `MAX_LIMIT = 100`
- Names should be **meaningful**, not random like `x1`, `y2`.

8. Writing readable and maintainable code.

Readable and maintainable code means **anyone (including you later) can understand the code easily**.
Some practices are:

1. **Use proper indentation** (4 spaces).
2. **Add comments** to explain complex parts.
3. **Use meaningful names** for variables, functions, and classes.
    - Bad: `a = 20`

   - Good: `age = 20`

4. **Follow PEP 8 style guide** (Python’s official style rules).
5. **Keep lines short** (max 79 characters).
6. **Break big code into small functions**.
7. **Avoid duplicate code** → reuse functions or classes.
8. **Write docstrings** (multi-line comments at the start of functions/classes) to describe their purpose.

Example:

```
def calculate_area(radius):
    """
    This function calculates the area of a circle
    given its radius.
    """
    pi = 3.14
    return pi * radius * radius
```

3. Core Python Concepts

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

Python has different data types to store different kinds of values.

1. **Integer (`int`)**
   - Whole numbers (positive, negative, or zero).
   - Example: `10`, `-5`, `0`
2. **Float (`float`)**
   - Numbers with decimal points.
   - Example: `3.14`, `-0.5`, `2.0`
3. **String (`str`)**
   - A sequence of characters enclosed in single (`'`), double (`"`), or triple quotes.
   - Example: `"Hello"`, `'Python'`
4. **List (`list`)**
   - An **ordered, changeable (mutable)** collection.
   - Can store different types of data.
   - Example: `[1, 2, "apple", 3.5]`
5. **Tuple (`tuple`)**
   - An **ordered, unchangeable (immutable)** collection.
   - Example: `(10, 20, "banana")`
6. **Dictionary (`dict`)**
   - Stores **key-value pairs**.
   - Example: `{"name": "Alice", "age": 22}`
7. **Set (`set`)**

- An **unordered collection of unique items** (no duplicates).
- Example: `{1, 2, 3, 3, 4}` → `{1, 2, 3, 4}`

10.Python variables and memory allocation.

Variables

- A variable is a name used to store a value.
- Example:
- `age = 20`
- `name = "John"`

Rules for Variables

- Must start with a **letter or underscore** (_).
- Can contain letters, digits, and underscores.
- Case-sensitive → `Age` and `age` are different.
- Cannot use reserved keywords (`if`, `class`, etc.).

Memory Allocation

- Python manages memory **automatically**.
- **Heap memory** → Objects (like lists, strings, etc.) are stored here.
- **Stack memory** → Function calls and local variables.
- Python uses **reference counting** and **garbage collection** to free memory when variables are no longer needed.

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

Operators are special symbols used to perform operations on variables/values.

1. **Arithmetic Operators**
   - `+` (Addition) → `10 + 5 = 15`
   - `-` (Subtraction) → `10 - 5 = 5`
   - `*` (Multiplication) → `10 * 5 = 50`
   - `/` (Division) → `10 / 5 = 2.0`
   - `%` (Modulus – remainder) → `10 % 3 = 1`
   - `**` (Exponent) → `2 ** 3 = 8`
   - `//` (Floor division) → `10 // 3 = 3`
2. **Comparison (Relational) Operators**
   - `==` Equal to
   - `!=` Not equal to
   - `>` Greater than

    - `<` Less than
    - `>=` Greater than or equal to
    - `<=` Less than or equal to
3. **Logical Operators**
    - `and` → True if both are True
    - `or` → True if at least one is True
    - `not` → Reverses True/False
4. **Bitwise Operators** (work on binary numbers)
    - `&` (AND)
    - `|` (OR)
    - `^` (XOR – exclusive OR)
    - `~` (NOT – flips bits)
    - `<<` (Left shift)
    - `>>` (Right shift)

4. Conditional Statements

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

Conditional statements help you **make decisions** in a program.
They check conditions and execute code based on whether the condition is **True** or **False**.

if statement

- Executes a block of code only if the condition is True.

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

---

if...else statement

- Runs one block if the condition is True, otherwise runs another block.

```
marks = 40
if marks >= 35:
    print("You passed the exam")
else:
    print("You failed the exam")
```

---

if...elif...else statement

- Used when you have **multiple conditions**.

```
number = 0
```

```
if number > 0:
    print("Positive number")
elif number < 0:
    print("Negative number")
else:
    print("Zero")
```

Key point:

- `if` → checks first condition.
- `elif` → checks additional conditions.
- `else` → runs if no condition is True.

13.Nested `if-else` conditions.

- A **nested if** means an `if` statement **inside another if/else**.
- Used when you need to check multiple levels of conditions.

Example:

```
age = 20
citizen = "yes"

if age >= 18:
    if citizen == "yes":
        print("You are eligible to vote in this country")
    else:
        print("You must be a citizen to vote")
else:
    print("You are not old enough to vote")
```

5. Looping (For, While)

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

Loops are used when you want to **repeat a block of code** multiple times.

for loop

- Used to **iterate (go through)** a sequence like list, tuple, string, or range.
- Example:

```
for i in range(5):
    print("Number:", i)
```

□ This prints numbers from `0` to `4`.

while loop

- Repeats code **as long as the condition is True**.
- Example:

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

□ This prints numbers `1` to `5`.

15. How loops work in Python.

- A loop checks a **condition**.
- If condition is **True**, the code inside runs.
- After one cycle (called **iteration**), the condition is checked again.
- When condition becomes **False**, the loop stops.

□ Example of flow:

while loop example

```
Step 1 → Check condition
Step 2 → If True → run code
Step 3 → Go back to condition
Step 4 → Repeat until False
```

for loop example

```
Step 1 → Pick first item from collection/range
Step 2 → Run code
Step 3 → Pick next item
Step 4 → Repeat until all items are done
```

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

Python loops are very useful for **collections** like lists, tuples, sets, dictionaries.

Loop with List

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

Loop with Tuple

```
numbers = (10, 20, 30)
for n in numbers:
    print(n)
```

Loop with Set

```
colors = {"red", "green", "blue"}
for c in colors:
    print(c)
```

Loop with Dictionary

```
student = {"name": "Alice", "age": 22, "course": "BCA"}
for key, value in student.items():
    print(key, ":", value)
```

6. Generators and Iterators

17.Understanding how generators work in Python.

A **generator** is like a list, but it doesn’t store all the values in memory.

Instead, it produces values **one at a time**, only when you ask for them.

This makes generators very **memory efficient**, especially with large amounts of data.

You can make generators in two ways:

a. **Generator function** – uses the `yield` keyword.
b. **Generator expression** – looks like a list comprehension but with `()` instead of `[]`.

Example:

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

gen = my_generator()
print(next(gen))   # 1
print(next(gen))   # 2
print(next(gen))   # 3
```

18. Difference between `yield` and `return`.

`return`: Ends the function completely and sends back a single value.

`yield`: Pauses the function, saves its state, and gives back a value.

a. When called again, it continues from where it left off.
b. This is why `yield` is used in generators.

Example:

```
def normal_func():
    return 10  # ends here

def generator_func():
    yield 1
    yield 2  # can continue later
```

So, **`return`** = **one-time output**, **`yield`** = **step-by-step output**.

19.Understanding iterators and creating custom iterators.

An **iterator** is an object you can loop through, like lists or tuples.

It must have two special methods:

a. `__iter__()` → returns the iterator object.
b. `__next__()` → returns the next value, and raises **StopIteration** when no items are left.

Example of a custom iterator:

```
class CountUpTo:
    def __init__(self, max):
        self.max = max
        self.num = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.num <= self.max:
            val = self.num
            self.num += 1
            return val
        else:
            raise StopIteration

counter = CountUpTo(3)
for num in counter:
    print(num)  # 1, 2, 3
```

7. Functions and Methods

20. Defining and calling functions in Python.

- A **function** is a block of code that runs only when called.
- It helps **reuse code** and makes programs easier to understand.

Defining a function:

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

Calling a function:

```
greet()   # Output: Hello, welcome!
```

Functions can also take **parameters** and return values:

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

print(add(5, 3))  # Output: 8
```

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

Functions in Python can accept arguments in different ways:

1. **Positional arguments** – Order matters.

```
def student(name, age):
    print(name, age)

student("Alice", 20)   # Alice 20
```

2. **Keyword arguments** – You specify the parameter name.

```
student(age=20, name="Alice")   # Alice 20
```

3. **Default arguments** – If not provided, a default value is used.

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

greet()           # Hello Guest
greet("Darshana") # Hello Darshana
```

22.Scope of variables in Python.

- **Local variable** → declared inside a function, accessible only there.
- **Global variable** → declared outside all functions, accessible everywhere.

Example:

```
x = 10  # Global

def my_func():
    y = 5  # Local
    print("Inside function:", x, y)

my_func()
print("Outside function:", x)  # but y not accessible here
```

If you want to change a global variable inside a function, use `global` keyword:

```
def update():
    global x
    x = 20
```

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

•

□ **String methods** (work on text):

- `upper()` → `"hello".upper()` → `"HELLO"`
- `lower()` → `"HELLO".lower()` → `"hello"`
- `replace("a", "b")` → `"banana".replace("a", "o")` → `"bonono"`
- `split()` → `"a b c".split()` → `['a', 'b', 'c']`
- `join()` → `"-".join(["a", "b"])` → `"a-b"`

□ **List methods** (work on lists):

- `append(x)` → adds item → `[1,2].append(3)` → `[1,2,3]`

- `remove(x)` → removes first match → `[1,2,3].remove(2)` → `[1,3]`
- `pop(i)` → removes by index → `[1,2,3].pop(1)` → `[1,3]`
- `sort()` → sorts list → `[3,1,2].sort()` → `[1,2,3]`
- `reverse()` → reverses order → `[1,2,3].reverse()` → `[3,2,1]`

8. Control Statements (Break, Continue, Pass)

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

`break`

- Used to **stop a loop completely**, even if the condition is still true.
- After `break`, the loop ends, and the program moves to the next statement outside the loop.

Example:

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

# Output: 0, 1, 2
```

`continue`

- Used to **skip the current iteration** and go to the next loop cycle.
- The loop does not stop, it just jumps to the next round.

Example:

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

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

`pass`

- Does **nothing**.
- It is a placeholder when you don’t want any action in the loop (or function/class).

Example:

```
for i in range(5):
    if i == 3:
        pass  # Do nothing here
    print(i)

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

9. String Manipulation

25. Understanding how to access and manipulate strings.

- A **string** is a sequence of characters inside quotes (`" "` or `' '`).
- You can access characters using **indexing**:
  - Index starts at `0` for the first character.
  - Negative index `-1` means last character.

Example:

```
s = "Python"
print(s[0])   # P (first character)
print(s[-1])  # n (last character)
```

- Strings can be manipulated using methods like **replace, join, split, etc.**

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

1. **Concatenation (joining strings)** → use `+`

```
a = "Hello"
b = "World"
print(a + " " + b)  # Hello World
```

2. **Repetition** → use `*`

```
print("Hi! " * 3)   # Hi! Hi! Hi!
```

3. **Common string methods:**

- `upper()` → `"python".upper()` → `"PYTHON"`
- `lower()` → `"PYTHON".lower()` → `"python"`
- `capitalize()` → `"hello".capitalize()` → `"Hello"`

- `title()` → `"hello world".title()` → `"Hello World"`
- `strip()` → `" hello ".strip()` → `"hello"`
- `replace("a", "o")` → `"banana".replace("a","o")` → `"bonono"`
- `find("a")` → `"banana".find("a")` → `1`

27.String slicing.

- Slicing means cutting a part of a string using **start:end:step**.
- Syntax:

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

Examples:

```
s = "Python"

print(s[0:4])   # Pyth   (from index 0 to 3)
print(s[:4])    # Pyth   (start is 0 by default)
print(s[2:])    # thon   (goes till end)
print(s[::2])   # Pto    (skip every 2nd char)
print(s[::-1])  # nohtyP (reverse string)
```

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

28.How functional programming works in Python.

- **Functional programming** is a style of coding where you write programs using **functions** instead of modifying data directly.
- It focuses on:
  - **Pure functions** → same input always gives the same output, no side effects.
  - **Immutability** → don't change original data, create new ones.
  - **Higher-order functions** → functions that take other functions as input or return them.

Example:

```
nums = [1, 2, 3, 4]

# Square each number (functional way)
squared = list(map(lambda x: x**2, nums))
print(squared)  # [1, 4, 9, 16]
```

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

These are built-in functions for processing data in a **functional style**:

1. **`map(function, iterable)`**

Applies a function to each item in an iterable.

```
nums = [1, 2, 3, 4]
result = list(map(lambda x: x*2, nums))
print(result)  # [2, 4, 6, 8]
```

2. **`filter(function, iterable)`**

Keeps only the items where the function returns `True`.

```
nums = [1, 2, 3, 4, 5]
result = list(filter(lambda x: x % 2 == 0, nums))
print(result)  # [2, 4]
```

3. **`reduce(function, iterable)`** (from `functools` module)

Repeatedly applies a function to reduce iterable to a single value.

```
from functools import reduce
nums = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, nums)
print(result)  # 10
```

30.Introduction to closures and decorators.

⍰ *Closures*

- A **closure** is a function that remembers variables from the scope where it was created, even after that scope is gone.

Example:

```
def outer(x):
    def inner(y):
        return x + y  # remembers x
    return inner

add5 = outer(5)
```

```
print(add5(10))  # 15
```

⍰ ***Decorators***

- A **decorator** is a special function that **adds extra features** to another function **without changing its code**.
- It's built using closures.

Example:

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

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

greet()
```

Output:

```
Before function call
Hello!
After function call
```