Python Notes

Table of Contents

2.5 Taking User Input (`input()` function)

2.6 Printing Output (`print()` function, f-strings, format method)

2.7 Comments in Python (Single-line and Multi-line)

---

3. Operators and Expressions

3.1 Arithmetic Operators (`+`, `-`, `*`, `/`, `%`, `//`, `**`)

3.2 Comparison Operators (`==`, `!=`, `>`, `<`, `>=`, `<=`)

3.3 Logical Operators (`and`, `or`, `not`)

3.4 Bitwise Operators (`&`, `|`, `^`, `~`, `<<`, `>>`)

3.5 Assignment Operators (`=`, `+=`, `-=`, `*=`, etc.)

3.6 Identity Operators (`is`, `is not`)

3.7 Membership Operators (`in`, `not in`)

3.8 Operator Precedence and Associativity

---

4. Control Flow Statements

4.1 Conditional Statements

- `if` Statement
- `if-else` Statement
- `if-elif-else` Statement

4.2 Looping in Python

- `for` Loop
- `while` Loop

4.3 Nested Loops

4.4 Loop Control Statements

- `break` Statement
- `continue` Statement
- `pass` Statement

---

5. Data Structures in Python

5.1 Lists

- Creating Lists
- Accessing Elements (Indexing & Slicing)
- Adding and Removing Elements (`append()`, `extend()`, `insert()`, `remove()`, `pop()`)
- Sorting and Reversing Lists

5.2 Tuples

- Tuple Characteristics (Immutable)
- Creating and Accessing Tuples
- Tuple Methods

5.3 Sets

- Set Properties (Unordered, Unique Elements)
- Creating and Manipulating Sets
- Set Operations (`union()`, `intersection()`, `difference()`)

5.4 Dictionaries

- Key-Value Pair Structure
- Adding, Accessing, and Deleting Elements
- Dictionary Methods (`keys()`, `values()`, `items()`)

5.5 Strings

- String Operations (`upper()`, `lower()`, `strip()`, `split()`, `join()`)
- String Formatting (`f-strings`, `format()`, `%` Operator)

---

6. Functions

6.1 Defining and Calling Functions

6.2 Function Arguments

- Positional Arguments
- Default Arguments
- Keyword Arguments
- Arbitrary Arguments (`*args`, `**kwargs`)

6.3 Returning Values from Functions

6.4 Scope and Lifetime of Variables

6.5 Recursive Functions

6.6 Anonymous (Lambda) Functions

---

7. Modules

7.1 Understanding Modules in Python

7.2 Importing Modules (`import`, `from ... import`, `as`)

7.3 Creating and Using Custom Modules

7.4 Standard Library Modules (`math`, `random`, `datetime`, `os`, etc.)

7.5 The `sys` Module and Command-line Arguments

7.6 Using `pip` to Install External Modules

---

8. File Handling

8.1 Opening and Closing Files (`open()`, `close()`)

8.2 Reading and Writing Files (`read()`, `write()`, `writelines()`)

8.3 File Modes (`r`, `w`, `a`, `rb`, `wb`)

8.4 Working with CSV Files (`csv` module)

8.5 Handling JSON Data (`json` module)

---

9. Exception Handling

9.1 Understanding Errors in Python

9.2 Using `try` and `except` Blocks

9.3 Handling Multiple Exceptions

9.4 The `finally` Block

9.5 Raising Custom Exceptions (`raise` keyword)

---

10. Object-Oriented Programming (OOP)

10.1 Understanding Classes and Objects

10.2 Defining and Using a Class (`__init__` Constructor)

10.3 Instance and Class Variables

10.4 Inheritance and Types of Inheritance

10.5 Method Overriding and Polymorphism

10.6 Encapsulation and Abstraction

10.7 Magic Methods (`__str__`, `__len__`, etc.)

---

11. Python Libraries Overview

11.1 Data Analysis

- NumPy (Arrays, Mathematical Operations)
- Pandas (DataFrames, Series, Data Manipulation)

11.2 Data Visualization

- Matplotlib (Basic and Advanced Plotting)
- Seaborn (Statistical Data Visualization)

11.3 Machine Learning

- Scikit-learn (Regression, Classification, Clustering)

11.4 Web Scraping

- Requests (Fetching Web Pages)

- BeautifulSoup (Parsing HTML)

11.5 Automation

- Selenium (Automating Browsers)

11.6 Working with Databases

- SQLite (`sqlite3` module)
- MySQL (`MySQL Connector`)

---

12. Working with APIs

12.1 Understanding APIs and RESTful Services

12.2 Making HTTP Requests (`requests` module)

12.3 Parsing JSON Data (`json` module)

12.4 Working with Public APIs (OpenWeather, GitHub API)

12.5 Creating a Simple API using Flask

---

13. Web Development with Python

13.1 Flask Framework

- Setting Up Flask
- Creating Routes and Views
- Handling Forms

Notes begins here :)

▼ Introduction to Python

1. Introduction to Python

1.1 What is Python?

Python is a **high-level, interpreted, and general-purpose programming language** known for its **simplicity, readability, and versatility**. It was created by **Guido van Rossum** and first released in **1991**. Python supports multiple programming paradigms, including **procedural, object-oriented, and functional programming**.

🔷 **Key Characteristics of Python:**

- **Interpreted:** Python does not need compilation; the interpreter executes code line by line.
- **Dynamically Typed:** No need to declare variable types explicitly.
- **Cross-platform:** Runs on Windows, macOS, and Linux without modification.
- **Extensible:** Can integrate with C, C++, Java, and more.

1.2 History and Evolution of Python

Version	Release Year	Key Features
Python 1.0	1991	Initial release by Guido van Rossum
Python 2.0	2000	List comprehensions, garbage collection with reference counting
Python 3.0	2008	Unicode support, print function, new integer division
Python 3.6+	2016+	f-strings, type hints, async/await
Python 3.10+	2021+	Pattern matching, structural pattern matching

🔷 **Python 2 vs. Python 3:**

- **Python 2** is outdated and no longer maintained (official support ended in 2020).
- **Python 3** is the current and actively developed version.

1.3 Features and Advantages of Python

🔷 **Key Features:**

✔️ **Easy to Read and Write:** Uses indentation instead of braces `{}`.

✔️ **Extensive Standard Library:** Includes built-in modules for OS, networking, math, etc.

✔️ **Automatic Memory Management:** Uses garbage collection.

✔️ **Interpreted Language:** No need for compilation like C/C++.

✔️ **High-level Language:** Allows focusing on logic rather than low-level details.

🔷 **Advantages of Python:**

✅ **Beginner-friendly:** Simple syntax, easy to learn.

✅ **Large Community Support:** Active developers and open-source contributions.

✅ **Versatile:** Used in web development, AI, data science, automation, and more.

1.4 Real-World Applications of Python

🔷 **1. Web Development:** Flask, Django (Used by Instagram, YouTube)

🔷 **2. Data Science & Machine Learning:** NumPy, Pandas, Scikit-learn (Used by Google, Netflix)

🔷 **3. Artificial Intelligence:** TensorFlow, PyTorch (Used by OpenAI, DeepMind)

🔷 **4. Automation & Scripting:** Selenium, BeautifulSoup (Used for bots, web scraping)

🔷 **5. Cybersecurity:** Used in penetration testing (e.g., Kali Linux tools)

🔷 **6. Game Development:** Pygame (Used for 2D game prototyping)

🔷 **7. Internet of Things (IoT):** MicroPython (Used for Raspberry Pi projects)

📌 **Example Use Case:** NASA uses Python for scientific computing.

1.5 Installing Python and Setting Up the Environment

🔷 **1. Install Python:**

- **Windows:** Download from python.org.
- **macOS:** Pre-installed (upgrade using `brew install python3`).
- **Linux:** Install via `sudo apt install python3` (Debian/Ubuntu) or `sudo dnf install python3` (Fedora).

🔷 **2. Verify Installation:**

Run:

```
python --version
```

or

```
python3 --version
```

🔷 **3. Install a Code Editor (IDE):**

- **Beginner-friendly:** IDLE (built-in), Thonny
- **Popular Choices:** VS Code, PyCharm, Jupyter Notebook

🔷 **4. Install Dependencies (PIP):**

Python's package manager **pip** is used to install external libraries:

```
pip install numpy
```

1.6 Running Python Scripts (Interactive Mode vs. Script Mode)

🔷 **1. Interactive Mode (REPL - Read-Eval-Print Loop)**

- Run Python directly in the terminal using:

```
python
```

- Example:

```
>>> print("Hello, Python!")
Hello, Python!
```

- Best for quick tests and debugging.

🔷 **2. Script Mode (Running `.py` files)**

- Write Python code in a `.py` file and execute:

```
python my_script.py
```

- Example (`hello.py`):
  Run it:

```
print("Welcome to Python!")
```

```
python hello.py
```

1.7 Understanding Python's REPL (Read-Eval-Print Loop)

📌 **What is REPL?**

REPL stands for **Read-Eval-Print Loop**, a command-line environment that allows **interactive execution** of Python commands.

🔷 **How REPL Works:**

1. **Read**: Takes user input (`>>>`)
2. **Evaluate**: Executes the statement
3. **Print**: Displays output

4. **Loop**: Repeats the process

🔷 **Example:**

```
>>> a = 10
>>> b = 20
>>> a + b
30
```

🔷 **Useful REPL Commands:**

- `help()` → Opens interactive help
- `exit()` → Exits Python REPL
- `_` (underscore) → Stores last evaluated expression

1.8 Writing and Executing a Simple Python Program

📌 **Let's write our first Python program!**

🔷 **Step 1: Open a text editor (VS Code, PyCharm, Notepad++)**

🔷 **Step 2: Write the following code in a file named `first_program.py`:**

```
# This is a simple Python program
print("Hello, World!")
```

🔷 **Step 3: Run the program in the terminal:**

```
python first_program.py
```

🔷 **Expected Output:**

```
Hello, World!
```

▼ Python Basics

2. Python Basics

2.1 Python Syntax and Indentation Rules

Python syntax is **clean and easy to read**. Unlike other languages that use curly braces `{}` to define blocks of code, Python **relies on indentation**.

🔷 **Key Syntax Rules:**

✔️ **Statements end at a new line** (no semicolons required)

✔️ **Indentation defines blocks of code** (instead of `{}`)

✔️ **Case-sensitive language** (`Var` and `var` are different)

✔️ **Colons (`:`) indicate code blocks** (e.g., functions, loops)

🔷 **Example:**

✅ Correct:

```
def greet():
    print("Hello, Python!")  # Indentation is required
greet()
```

❌ Incorrect (Indentation Error):

```
def greet():
print("Hello, Python!")  # Missing indentation
```

🔷 **Best Practices:**

- Use **4 spaces** for indentation (PEP 8 standard).
- Avoid mixing spaces and tabs.

2.2 Variables and Constants

🔷 **Variables:**

- Variables store data in memory.

- No need to declare types explicitly (dynamic typing).

Example:

```
name = "Python"
age = 30
pi = 3.14
print(name, age, pi)
```

🔷 **Rules for Naming Variables:**

✔️ Must start with a **letter (A-Z, a-z) or underscore (_)**
✔️ Can contain
letters, numbers, and underscores

✔️ Cannot be a Python **keyword** (e.g., `class`, `if`, `for`)

🔷 **Constants:**

- Python does not have built-in constants, but we use **UPPERCASE** naming convention.

```
PI = 3.14159  # A constant by convention
MAX_USERS = 100
```

2.3 Data Types in Python

Python provides several **built-in data types**:

Numeric Types: `int`, `float`, `complex`

```
num1 = 10      # int
num2 = 3.14    # float
num3 = 3 + 4j  # complex
print(type(num1), type(num2), type(num3))
```

Text Type: `str`

```
text = "Hello, Python!"
print(type(text))
```

Sequence Types: `list`, `tuple`, `range`

✔ List (Mutable, Ordered)

```
my_list = [1, 2, 3, "Python"]
my_list.append(4)  # Adding element
print(my_list)
```

✔ Tuple (Immutable, Ordered)

```
my_tuple = (1, 2, 3, "Python")
print(my_tuple[0])  # Accessing elements
```

✔ Range (Sequence of Numbers)

```
r = range(1, 10, 2)  # Start, Stop, Step
print(list(r))  # Convert range to list
```

Set Types: `set`, `frozenset`

✔ Set (Unordered, Unique Elements)

```
my_set = {1, 2, 3, 3, 4}
print(my_set)  # Duplicates are removed
```

✔ Frozenset (Immutable Set)

```
f_set = frozenset([1, 2, 3, 4])
print(f_set)
```

Mapping Type: `dict` (Key-Value Pairs)

```
my_dict = {"name": "Python", "year": 1991}
print(my_dict["name"])
```

2.4 Type Casting and Type Checking

🔷 **Type Checking:**

```
num = 10
print(type(num))  # <class 'int'>
```

🔷 **Type Casting (Converting Data Types):**

✔️ **int()** – Convert to integer

✔️ **float()** – Convert to float

✔️ **str()** – Convert to string

✔️ **list()** – Convert to list

```
a = "100"
b = int(a)  # Convert string to integer
c = float(b)  # Convert integer to float
print(b, c)
```

2.5 Taking User Input (`input()` function)

Python allows **user input** using the `input()` function.

```
name = input("Enter your name: ")
print("Hello,", name)
```

🔷 **By default, `input()` returns a string.**

For numeric input, convert it using `int()` or `float()`.

```
age = int(input("Enter your age: "))
print("Next year, you will be", age + 1)
```

2.6 Printing Output (`print()` function, f-strings, format method)

🔷 **Using `print()` function:**

```
print("Hello, Python!")
print(10, 20, 30)  # Multiple values
print("Python", "is", "fun", sep="-")  # Custom separator
```

🔷 **Using `format()` method:**

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

🔷 **Using f-strings (Recommended, Python 3.6+):**

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

🔷 **Printing with Special Characters:**

✔️ `\\n` – New line

✔️ `\\t` – Tab space

```
print("Hello\\nWorld")  # New line
print("Name:\\tAlice")  # Tab space
```

2.7 Comments in Python (Single-line and Multi-line)

🔷 **Single-line comments (`#`)**

```
# This is a comment
print("Hello, World!")  # This is an inline comment
```

🔷 **Multi-line comments (`""" """` or `''' '''`)**

```
"""
This is a multi-line comment.
It can span multiple lines.
"""
print("Python Basics")
```

🔷 **Use Comments for:**

✔️ Code documentation

✔️ Temporarily disabling code

✔️ Explaining complex logic

▼ Operators and Expressions

3. Operators and Expressions

Operators are special symbols in Python that **perform operations** on variables and values. Expressions are combinations of **variables, operators, and values** that produce a result.

3.1 Arithmetic Operators (`+` , `-` , `*` , `/` , `%` , `//` , `*`)

Arithmetic operators perform **mathematical calculations**.

Operator	Description	Example	Output
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`10 - 4`	`6`
`*`	Multiplication	`6 * 2`	`12`

`/`	Division (floating-point)	`7 / 2`	`3.5`
`%`	Modulus (remainder)	`7 % 2`	`1`
`//`	Floor division (quotient without decimal)	`7 // 2`	`3`
`**`	Exponentiation (power)	`2 ** 3`	`8`

🔷 **Examples:**

```
a, b = 10, 3
print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.3333
print(a % b)   # 1
print(a // b)  # 3
print(a ** b)  # 1000 (10^3)
```

3.2 Comparison Operators (`==` , `!=` , `>` , `<` , `>=` , `<=`)

Comparison operators are used to **compare values**. They return a **Boolean** (`True` **or** `False`).

Operator	Description	Example	Output
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`5 < 3`	`False`
`>=`	Greater than or equal to	`5 >= 5`	`True`
`<=`	Less than or equal to	`5 <= 3`	`False`

🔷 **Examples:**

```
x, y = 10, 20
print(x == y)   # False
print(x != y)   # True
print(x > y)    # False
print(x < y)    # True
print(x >= y)   # False
print(x <= y)   # True
```

3.3 Logical Operators (`and` , `or` , `not`)

Logical operators are used to **combine multiple conditions**.

Operator	Description	Example	Output
`and`	Returns `True` if **both** conditions are `True`	`(5 > 3) and (10 > 5)`	`True`
`or`	Returns `True` if **at least one** condition is `True`	`(5 > 10) or (10 > 5)`	`True`
`not`	Reverses the Boolean result	`not(5 > 3)`	`False`

🔷 **Examples:**

```
a, b = True, False
print(a and b)  # False
print(a or b)   # True
print(not a)    # False
```

3.4 Bitwise Operators (`&` , `|` , `^` , `~` , `<<` , `>>`)

Bitwise operators perform **operations on binary representations** of numbers.

Operator	Description	Example (Binary)	Output
`&`	AND	`5 & 3` → `0101 & 0011`	`0001` (1)
`` ` ``	`` ` ``	OR	`` `5 ``
`^`	XOR	`5 ^ 3` → `0101 ^ 0011`	`0110` (6)

`~`	NOT (Inverts bits)	`~5`	`-6`
`<<`	Left Shift	`5 << 1` (0101 → 1010)	`10`
`>>`	Right Shift	`5 >> 1` (0101 → 0010)	`2`

🔷 **Examples:**

```
x, y = 5, 3
print(x & y)  # 1 (AND)
print(x | y)  # 7 (OR)
print(x ^ y)  # 6 (XOR)
print(~x)     # -6 (NOT)
print(x << 1) # 10 (Left shift)
print(x >> 1) # 2 (Right shift)
```

3.5 Assignment Operators (`=` , `+=` , `=` , `=` , etc.)

Assignment operators are used to **assign values** to variables.

Operator	Example	Equivalent To
`=`	`x = 5`	`x = 5`
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 3`	`x = x * 3`
`/=`	`x /= 3`	`x = x / 3`
`%=`	`x %= 3`	`x = x % 3`
`//=`	`x //= 3`	`x = x // 3`
`**=`	`x **= 3`	`x = x ** 3`

🔷 **Examples:**

```
x = 10
x += 5   # x = x + 5 → 15
```

```
x *= 2   # x = x * 2 → 30
print(x)
```

3.6 Identity Operators (`is` , `is not`)

Identity operators check if **two variables refer to the same object** in memory.

Operator	Example	Output
`is`	`x is y`	`True` if x and y are same object
`is not`	`x is not y`	`True` if x and y are different objects

🔷 **Examples:**

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)   # True (same object)
print(a is c)   # False (different objects)
print(a == c)   # True (same values)
```

3.7 Membership Operators (`in` , `not in`)

Membership operators check whether a **value is in a sequence** (like a list, tuple, or string).

Operator	Example	Output
`in`	`"a" in "apple"`	`True`
`not in`	`"x" not in "apple"`	`True`

🔷 **Examples:**

```
numbers = [1, 2, 3, 4]
print(2 in numbers)   # True
print(5 not in numbers)  # True
```

3.8 Operator Precedence and Associativity

Operator precedence determines **the order in which operations are performed**.

🔷 **Precedence Order (Highest to Lowest):**

1. `()` Parentheses
2. `*` Exponentiation
3. `+x`, `x`, `~x` Unary operators
4. `*`, `/`, `//`, `%` Multiplication, division, modulus
5. `+`, `-` Addition and subtraction
6. `<<`, `>>` Bitwise shifts
7. `&` Bitwise AND
8. `^` Bitwise XOR
9. `|` Bitwise OR
10. `==`, `!=`, `>`, `<`, `>=`, `<=` Comparisons
11. `not` Logical NOT
12. `and` Logical AND
13. `or` Logical OR
14. `=` Assignment

🔷 **Examples:**

```
print(2 + 3 * 4)   # 14 (* has higher precedence than +)
print((2 + 3) * 4) # 20 (Parentheses change precedence)
```

▼ Control Flow Statements

4. Control Flow Statements in Python

Control flow statements **determine the execution order** of statements in a program. They allow decision-making and looping, enabling programs to respond dynamically to different conditions.

4.1 Conditional Statements

Conditional statements allow **decision-making** based on conditions. They evaluate **Boolean expressions** and execute different code blocks accordingly.

🔷 `if` Statement

- The `if` statement executes a block of code **only if** a condition is `True`.
- If the condition is `False`, the code inside the `if` block is skipped.

Syntax:

```
if condition:
    # Code to execute when the condition is True
```

Example:

```
age = 20
if age >= 18:
    print("You are eligible to vote.")
```

Output:

```
You are eligible to vote.
```

🔷 `if-else` Statement

- The `else` block executes **when the** `if` **condition is False**.

Syntax:

```
if condition:
    # Code when condition is True
```

```
else:
    # Code when condition is False
```

Example:

```
num = 5
if num % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

Output:

```
Odd number
```

🔷 `if-elif-else` Statement

- The `elif` block checks **multiple conditions**.
- It runs the first block that evaluates to `True`. If none are `True`, `else` executes.

Syntax:

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

Example:

```
marks = 85

if marks >= 90:
    print("Grade: A")
```

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

Output:

```
Grade: B
```

4.2 Looping in Python

Loops **repeat a block of code** multiple times.

🔷 `for` Loop

- Used to iterate over **sequences** (lists, tuples, strings, ranges, etc.).
- The loop **automatically stops** when it reaches the end of the sequence.

Syntax:

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

Example:

```
for num in range(1, 6):
    print(num)
```

Output:

```
1
2
3
```

```
4
5
```

Looping through a list:

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

Output:

```
apple
banana
cherry
```

🔷 `while` Loop

- The `while` loop runs as long as a **condition remains** `True`.
- It is useful when the number of iterations is **not known in advance**.

Syntax:

```
while condition:
    # Code to execute while condition is True
```

Example:

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

Output:

```
1
2
3
4
5
```

⚠ Be careful of infinite loops!

```
while True:
    print("This will run forever!")  # Avoid this unless intentional
```

4.3 Nested Loops

- A loop inside another loop.
- The inner loop **executes completely** for each iteration of the outer loop.

Example:

```
for i in range(1, 3):  # Outer loop
    for j in range(1, 4):  # Inner loop
        print(f"i={i}, j={j}")
```

Output:

```
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
i=2, j=3
```

4.4 Loop Control Statements

Loop control statements modify the execution of loops.

🔷 `break` Statement

- Exits the loop **immediately** when encountered.

Example:

```
for num in range(1, 6):
    if num == 4:
        break  # Exits the loop when num is 4
    print(num)
```

Output:

```
1
2
3
```

🔷 `continue` Statement

- **Skips** the rest of the loop's code for the current iteration **but continues with the next iteration**.

Example:

```
for num in range(1, 6):
    if num == 3:
        continue  # Skips the print statement for num == 3
    print(num)
```

Output:

```
1
2
4
5
```

🔷 `pass` Statement

- A placeholder that **does nothing** but allows the program to run.

Example:

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

Output:

```
0
1
2
3
4
```

▼ Data Structures in Python

5. Data Structures in Python

Python provides **built-in data structures** that allow efficient storage and manipulation of data. The primary data structures include:

- **Lists** (Ordered, Mutable)
- **Tuples** (Ordered, Immutable)
- **Sets** (Unordered, Unique elements)
- **Dictionaries** (Key-Value pairs, Mutable)
- **Strings** (Immutable sequence of characters)

5.1 Lists

Definition

A **list** is an **ordered**, **mutable** (changeable) collection of items. Lists allow **duplicate elements** and can store different data types in a single list.

Creating Lists

```
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed_list = ["Python", 3.14, True, 42]
nested_list = [[1, 2], [3, 4]]
```

Accessing Elements

```
numbers = [10, 20, 30, 40, 50]
print(numbers[0])     # First element: 10
print(numbers[-1])    # Last element: 50
print(numbers[1:4])   # Slicing: [20, 30, 40]
```

List Methods

Method	Description	Example
`append(x)`	Adds `x` to the end	`lst.append(6)`
`extend(iterable)`	Adds multiple elements	`lst.extend([7, 8, 9])`
`insert(i, x)`	Inserts `x` at index `i`	`lst.insert(2, 15)`
`remove(x)`	Removes first occurrence of `x`	`lst.remove(3)`
`pop(i)`	Removes and returns element at index `i` (last if not provided)	`lst.pop(1)`
`clear()`	Removes all elements	`lst.clear()`
`index(x)`	Returns index of `x`	`lst.index(10)`
`count(x)`	Counts occurrences of `x`	`lst.count(5)`
`sort()`	Sorts the list	`lst.sort()`
`reverse()`	Reverses elements in-place	`lst.reverse()`
`copy()`	Returns a copy of the list	`new_list = lst.copy()`

5.2 Tuples

Definition

A **tuple** is an **ordered** and **immutable** collection. Once a tuple is created, its elements cannot be modified.

Creating Tuples

```
empty_tuple = ()
single_element_tuple = (42,)  # Comma required
numbers = (10, 20, 30, 40)
```

Accessing Elements

```
numbers = (10, 20, 30, 40, 50)
print(numbers[0])
print(numbers[-1])
print(numbers[1:3])
```

Tuple Methods

Method	Description	Example
`count(x)`	Counts occurrences of `x`	`tpl.count(2)`
`index(x)`	Returns index of `x`	`tpl.index(10)`
`len()`	Returns the length	`len(tpl)`
`max()`	Returns max value	`max(tpl)`
`min()`	Returns min value	`min(tpl)`
`sum()`	Returns sum of elements	`sum(tpl)`
`sorted()`	Returns sorted list	`sorted(tpl)`
`tuple()`	Converts iterable to tuple	`tuple(lst)`

5.3 Sets

Definition

A **set** is an **unordered** collection of **unique** elements.

Creating Sets

```
empty_set = set()
my_set = {1, 2, 3, 4, 5}
```

Set Methods

Method	Description	Example
`add(x)`	Adds `x` to the set	`set.add(6)`
`remove(x)`	Removes `x` (error if not found)	`set.remove(2)`
`discard(x)`	Removes `x` (no error if not found)	`set.discard(3)`
`pop()`	Removes a random element	`set.pop()`
`clear()`	Removes all elements	`set.clear()`
`union(B)`	Combines two sets	`A.union(B)`
`intersection(B)`	Common elements	`A.intersection(B)`
`difference(B)`	Elements in A, not B	`A.difference(B)`
`symmetric_difference(B)`	Elements in either A or B but not both	`A.symmetric_difference(B)`
`isdisjoint(B)`	Returns `True` if A and B have no common elements	`A.isdisjoint(B)`

5.4 Dictionaries

Definition

A **dictionary** is a **key-value** data structure that allows fast lookups.

Creating Dictionaries

```
student = {"name": "Alice", "age": 21}
```

Dictionary Methods

Method	Description	Example
`keys()`	Returns all keys	`dict.keys()`
`values()`	Returns all values	`dict.values()`
`items()`	Returns key-value pairs	`dict.items()`
`get(x)`	Returns value of `x` (None if not found)	`dict.get("name")`
`pop(x)`	Removes key `x` and returns its value	`dict.pop("age")`
`update(B)`	Merges dictionary B into A	`dict.update(new_data)`
`setdefault(x, y)`	Returns value of `x` or sets `y` if not found	`dict.setdefault("city", "NY")`
`copy()`	Returns a copy of the dictionary	`new_dict = dict.copy()`
`fromkeys(iterable, value)`	Creates a new dictionary with keys from iterable and default values	`dict.fromkeys(["name", "age"], None)`

5.5 Strings

Definition

A **string** is an **immutable sequence** of characters.

String Methods

Method	Description	Example
`upper()`	Converts to uppercase	`"hello".upper()`
`lower()`	Converts to lowercase	`"HELLO".lower()`
`strip()`	Removes spaces	`" hello ".strip()`
`split()`	Splits into a list	`"a,b,c".split(",")`
`join()`	Joins list into a string	`",".join(["a", "b", "c"])`
`replace(old, new)`	Replaces substrings	`"hello".replace("h", "H")`
`find(x)`	Returns index of first occurrence	`"hello".find("l")`
`count(x)`	Counts occurrences of `x`	`"hello".count("l")`

`startswith(x)`	Checks if string starts with `x`	`"hello".startswith("h")`
`endswith(x)`	Checks if string ends with `x`	`"hello".endswith("o")`

▼ Functions

6. Functions in Python

A **function** is a block of reusable code designed to perform a specific task. Functions improve **code reusability**, **modularity**, and **readability**.

6.1 Defining and Calling Functions

Defining a Function

A function is defined using the `def` keyword.

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

# Calling the function
greet()
```

Function with Parameters

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

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

6.2 Function Arguments

Python functions can accept **different types of arguments**.

1. Positional Arguments

- The most common type of argument.
- Arguments are assigned based on their position.

```
def full_name(first, last):
    print(f"Full Name: {first} {last}")

full_name("Nihar", "Dodagatta")
# Output: Full Name: Nihar Dodagatta
```

2. Default Arguments

- Used when a function has a default value for a parameter.
- If no value is provided, it uses the default.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()        # Output: Hello, Guest!
greet("Nihar")  # Output: Hello, Nihar!
```

3. Keyword Arguments

- Arguments are passed using parameter names.
- Order does not matter.

```
def student_details(name, age, course):
    print(f"Name: {name}, Age: {age}, Course: {course}")

student_details(age=22, name="Harsha", course="CS")
# Output: Name: Harsha, Age: 22, Course: CS
```

4. Arbitrary Arguments (`args` , `*kwargs`)

`args` (Multiple Positional Arguments)

- Allows passing **multiple positional arguments** as a tuple.

```
def sum_numbers(*numbers):
    total = sum(numbers)
    print(f"Sum: {total}")

sum_numbers(1, 2, 3, 4, 5)
# Output: Sum: 15
```

`*kwargs` (Multiple Keyword Arguments)

- Allows passing **multiple keyword arguments** as a dictionary.

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

display_info(name="Vasanta", age=24, city="Chennai")
# Output:
# name: Vasanta
# age: 24
# city: Chennai
```

6.3 Returning Values from Functions

A function can return values using the `return` statement.

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

result = square(6)
print(result)  # Output: 36
```

Returning Multiple Values

A function can return multiple values as a tuple.

```
def calculate(a, b):
    return a + b, a - b, a * b, a / b

add, sub, mul, div = calculate(10, 5)
print(add, sub, mul, div)
# Output: 15 5 50 2.0
```

6.4 Scope and Lifetime of Variables

- **Local Scope**: Variables defined inside a function exist only within that function.
- **Global Scope**: Variables defined outside all functions can be accessed anywhere.

```
x = 10  # Global variable

def my_function():
    x = 5  # Local variable
    print("Inside function:", x)

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

# Output:
# Inside function: 5
# Outside function: 10
```

Using `global` Keyword

- Modify a global variable inside a function.

```
x = 10

def modify():
    global x
    x = 20

modify()
print(x)  # Output: 20
```

6.5 Recursive Functions

A **recursive function** calls itself to solve a problem.

Factorial Using Recursion

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
# Output: 120
```

Fibonacci Series Using Recursion

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
print(fibonacci(6))
# Output: 8
```

6.6 Anonymous (Lambda) Functions

A **lambda function** is a small, anonymous function that can take any number of arguments but has only **one expression**.

Syntax

```
lambda arguments: expression
```

Lambda Function Example

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Lambda Function with Multiple Arguments

```
add = lambda a, b: a + b
print(add(3, 4))  # Output: 7
```

Using `lambda` with `map()`, `filter()`, `reduce()`

1. `map()` Function

Applies a function to all elements in an iterable.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

2. `filter()` Function

Filters elements based on a condition.

```
numbers = [10, 15, 21, 30, 45]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
# Output: [15, 21, 45]
```

3. `reduce()` Function (Requires `functools`)

Performs cumulative operations.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_all = reduce(lambda a, b: a + b, numbers)
print(sum_all)
# Output: 15
```

▼ Modules

7. Modules in Python

A **module** in Python is a file containing **Python code** (functions, classes, and variables) that can be imported and reused in other programs. Modules help in organizing code into separate files, improving readability and maintainability.

A **module file** has a `.py` extension.

7.1 Understanding Modules in Python

Python provides two types of modules:

1. **Built-in Modules**: Pre-installed modules like `math`, `random`, `os`, etc.
2. **User-defined Modules**: Custom modules created by programmers.

Why Use Modules?

- **Code Reusability**: Avoids rewriting the same code multiple times.

- **Organized Structure**: Breaks large programs into smaller, manageable files.
- **Namespace Management**: Helps prevent variable and function name conflicts.

7.2 Importing Modules

Python provides different ways to **import** and use modules.

1. Using `import`

The `import` statement is used to import a module.

```
import math

print(math.sqrt(25))  # Output: 5.0
print(math.pi)        # Output: 3.141592653589793
```

2. Using `from ... import`

This allows importing specific functions or variables from a module.

```
from math import sqrt, pi

print(sqrt(36))  # Output: 6.0
print(pi)        # Output: 3.141592653589793
```

3. Using `as` (Alias)

You can give a module or function an alias using `as`.

```
import math as m

print(m.factorial(5))  # Output: 120
```

4. Importing Everything Using ``

This imports all functions and variables from a module.

```
from math import *

print(sin(0))  # Output: 0.0
print(pow(2, 3))  # Output: 8.0
```

⚠️ **Warning:** Avoid using `*` as it can cause **namespace conflicts**.

7.3 Creating and Using Custom Modules

You can create your own Python module.

Step 1: Create a Module (`mymodule.py`)

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

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

Step 2: Import and Use the Module

Save `mymodule.py` in the same directory as your script.

```
import mymodule

print(mymodule.greet("Nihar"))  # Output: Hello, Nihar!
print(mymodule.add(5, 3))       # Output: 8
```

7.4 Standard Library Modules

Python provides a rich set of built-in modules.

1. `math` Module (Mathematical Functions)

```
import math

print(math.sqrt(16))       # Output: 4.0
print(math.factorial(5))   # Output: 120
print(math.log10(100))     # Output: 2.0
print(math.sin(math.pi/2)) # Output: 1.0
```

2. `random` Module (Random Number Generation)

```
import random

print(random.randint(1, 10))  # Random integer between 1 and 10
print(random.choice(["Python", "Java", "C++"]))  # Random choice from li
st
print(random.uniform(1.5, 3.5))  # Random float between 1.5 and 3.5
```

3. `datetime` Module (Date and Time Handling)

```
import datetime

now = datetime.datetime.now()
print(now)  # Output: 2025-02-19 12:34:56.789123

print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Output: 2025-02-19 12:
34:56
```

4. `os` Module (Operating System Interactions)

```
import os

print(os.getcwd())  # Get current working directory
print(os.listdir())  # List files in the current directory
```

5. `time` Module (Time-Related Functions)

```
import time

print("Wait for 3 seconds...")
time.sleep(3)  # Pauses execution for 3 seconds
print("Done!")
```

6. `sys` Module (System-Specific Parameters and Functions)

```
import sys

print(sys.version)  # Prints Python version
print(sys.platform) # Prints OS platform
```

7. `json` Module (JSON Handling)

```
import json

data = {"name": "Nihar", "age": 25}
json_data = json.dumps(data)
print(json_data)  # Output: {"name": "Nihar", "age": 25}

parsed_data = json.loads(json_data)
print(parsed_data["name"])  # Output: Nihar
```

8. `csv` Module (Handling CSV Files)

```
import csv

with open("data.csv", mode="w", newline="") as file:
    writer = csv.writer(file)
```

```
    writer.writerow(["Name", "Age"])
    writer.writerow(["Nihar", 25])
```

9. `re` Module (Regular Expressions)

```
import re

pattern = r"\\d+"
text = "There are 15 cats and 3 dogs."

matches = re.findall(pattern, text)
print(matches)  # Output: ['15', '3']
```

10. `urllib` Module (Web Requests)

```
import urllib.request

response = urllib.request.urlopen("<http://www.google.com>")
print(response.status)  # Output: 200
```

7.5 The `sys` Module and Command-line Arguments

The `sys` module allows handling **command-line arguments**.

Example: `sys.argv`

Save the following code in `script.py`:

```
import sys

print("Arguments:", sys.argv)
```

Run the script in the terminal:

```
python script.py hello world
```

Output:

```
Arguments: ['script.py', 'hello', 'world']
```

- `sys.argv[0]` → Script name
- `sys.argv[1:]` → Passed arguments

7.6 Using `pip` to Install External Modules

Python has **third-party libraries** that can be installed using `pip`.

Checking Installed Modules

```
pip list
```

Installing a Module

```
pip install requests
```

Uninstalling a Module

```
pip uninstall requests
```

Using an Installed Module (`requests`)

```
import requests

response = requests.get("<https://api.github.com>")
print(response.json())  # Output: JSON response from GitHub API
```

▼ File Handling

8. File Handling in Python

Python provides built-in functions and modules to work with files. Using **file handling**, we can **read, write, and modify** files stored on disk.

8.1 Opening and Closing Files

Python provides the `open()` function to open files.

Syntax:

```
file = open("filename", "mode")
```

- `"filename"` → Name of the file (with extension).
- `"mode"` → Specifies how the file will be opened.

Opening a File

```
file = open("example.txt", "r")  # Opens file in read mode
```

Closing a File

Always **close a file** after performing operations to free system resources.

```
file.close()
```

Best Practice: Use `with` Statement

The `with` statement **automatically closes the file** after use.

```
with open("example.txt", "r") as file:
    data = file.read()
```

8.2 Reading and Writing Files

Python provides various methods to read and write data in files.

Reading a File (`read()` , `readline()` , `readlines()`)

```
# Example file: sample.txt
# Content:
# Hello, Python!
# Welcome to file handling.

with open("sample.txt", "r") as file:
    content = file.read()  # Reads entire file
    print(content)
```

Other Read Methods:

- `read(size)` : Reads a specific number of characters.
- `readline()` : Reads one line at a time.
- `readlines()` : Reads all lines and returns them as a **list**.

```
with open("sample.txt", "r") as file:
    print(file.readline())  # Reads first line
    print(file.readlines())  # Reads remaining lines as a list
```

Writing to a File (`write()` , `writelines()`)

Overwriting a File (`w` mode)

```
with open("sample.txt", "w") as file:
    file.write("New content added!\\n")
```

- **Warning:** This **erases** existing content.

Appending to a File (`a` mode)

```
with open("sample.txt", "a") as file:
    file.write("Appending a new line.\\n")
```

- **Appends** data **without** deleting existing content.

Writing Multiple Lines (`writelines()`)

```
lines = ["Line 1\\n", "Line 2\\n", "Line 3\\n"]
with open("sample.txt", "w") as file:
    file.writelines(lines)
```

8.3 File Modes

Python supports multiple file **modes** to control how a file is opened.

Mode	Description
`r`	Read (default) – Error if file doesn't exist
`w`	Write – Creates file if not exists, overwrites existing content
`a`	Append – Adds new data, doesn't delete existing content
`r+`	Read and Write – Raises error if file doesn't exist
`w+`	Read and Write – Creates file if not exists, overwrites existing content
`a+`	Read and Append – Creates file if not exists
`rb`	Read in Binary mode
`wb`	Write in Binary mode
`ab`	Append in Binary mode

Example: Reading and Writing in Binary Mode

```
with open("image.jpg", "rb") as file:
    binary_data = file.read()

with open("copy.jpg", "wb") as file:
    file.write(binary_data)
```

8.4 Working with CSV Files (`csv` Module)

CSV (Comma-Separated Values) files store tabular data in **plain text**.

1. Writing to a CSV File

```
import csv

with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
    writer.writerow(["Nihar", 25, "Hyderabad"])
    writer.writerow(["Harsha", 24, "Bangalore"])
```

2. Reading a CSV File

```
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

3. Writing and Reading CSV with Dictionary

```
import csv

# Writing Dictionary Data
with open("data.csv", "w", newline="") as file:
    fieldnames = ["Name", "Age", "City"]
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({"Name": "Nihar", "Age": 25, "City": "Hyderabad"})
    writer.writerow({"Name": "Harsha", "Age": 24, "City": "Bangalore"})

# Reading Dictionary Data
with open("data.csv", "r") as file:
```

```
    reader = csv.DictReader(file)
    for row in reader:
        print(row["Name"], row["Age"], row["City"])
```

8.5 Handling JSON Data (`json` Module)

JSON (JavaScript Object Notation) is used to store and exchange **structured data**.

1. Writing JSON Data

```
import json

data = {
    "name": "Nihar",
    "age": 25,
    "city": "Hyderabad"
}

with open("data.json", "w") as file:
    json.dump(data, file, indent=4)
```

2. Reading JSON Data

```
import json

with open("data.json", "r") as file:
    data = json.load(file)

print(data["name"])  # Output: Nihar
```

3. Converting Python Dictionary to JSON String

```
import json

data = {"name": "Nihar", "age": 25}
json_string = json.dumps(data, indent=4)
print(json_string)
```

4. Converting JSON String to Python Dictionary

```
import json

json_string = '{"name": "Nihar", "age": 25}'
data = json.loads(json_string)
print(data["name"])  # Output: Nihar
```

▼ Exception Handling

9. Exception Handling in Python

Exception handling allows a program to **gracefully handle errors** instead of crashing. Python provides mechanisms to **catch and handle errors** using the `try-except` block.

9.1 Understanding Errors in Python

Errors in Python can be broadly classified into:

1. **Syntax Errors** – Occur when the Python interpreter cannot understand the code.
2. **Exceptions (Runtime Errors)** – Occur during execution due to **invalid operations**.

Example: Syntax Error

```
# Missing colon in 'if' statement (SyntaxError)
if True
    print("Hello")
```

Example: Exception (Runtime Error)

```
# Division by zero error (ZeroDivisionError)
result = 10 / 0
```

Common Built-in Exceptions

Exception	Description
`ZeroDivisionError`	Raised when dividing by zero
`ValueError`	Raised when a function gets an incorrect argument
`TypeError`	Raised when an operation is performed on an invalid type
`IndexError`	Raised when trying to access an out-of-range index in a list
`KeyError`	Raised when a dictionary key is not found
`FileNotFoundError`	Raised when attempting to open a non-existent file

9.2 Using `try` and `except` Blocks

The `try` block allows Python to execute **error-prone** code, while the `except` block **handles the error** if one occurs.

Basic Example

```
try:
    result = 10 / 0  # Code that may raise an error
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Output:

```
Error: Cannot divide by zero!
```

9.3 Handling Multiple Exceptions

A program can encounter **different types of exceptions**, and we can handle them separately.

Example: Handling Multiple Exceptions

```
try:
    num = int(input("Enter a number: "))  # May raise ValueError
    result = 10 / num                     # May raise ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input! Please enter a valid integer.")
```

Catching Multiple Exceptions in One Except Block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError) as e:
    print("An error occurred:", e)
```

9.4 The `finally` Block

- The `finally` block **always executes**, regardless of whether an exception occurred.
- It is used to **release resources** (e.g., closing a file, closing a database connection).

Example: Using `finally`

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Closing file...")
    file.close()  # Ensuring the file is closed
```

9.5 Raising Custom Exceptions (`raise` keyword)

Python allows **manually raising exceptions** using the `raise` keyword.

Example: Raising an Exception

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("You must be at least 18 years old.")
```

Output (if age < 18):

```
Traceback (most recent call last):
  File "script.py", line 3, in <module>
ValueError: You must be at least 18 years old.
```

Defining Custom Exceptions

We can define our own exceptions by **creating a subclass of** `Exception`.

```
class NegativeNumberError(Exception):
    """Custom exception for negative numbers"""
    pass

num = int(input("Enter a number: "))
```

```
if num < 0:
    raise NegativeNumberError("Negative numbers are not allowed!")
```

▼ Object-Oriented Programming (OOP)

10. Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which contain **data** (attributes) and **behavior** (methods). Python is **highly object-oriented**, meaning **everything in Python is an object**, including numbers, strings, functions, and classes.

10.1 Understanding Classes and Objects

- **Class** → A blueprint for creating objects.
- **Object** → An instance of a class.

Example: Creating a Class and an Object

```
class Car:
    brand = "Toyota"

# Creating an object (instance) of the Car class
car1 = Car()
print(car1.brand)  # Output: Toyota
```

10.2 Defining and Using a Class (`__init__` Constructor)

- The `__init__` method is a **special method (constructor)** that initializes object attributes.

Example: Using `__init__`

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Instance variable
        self.model = model  # Instance variable

# Creating objects
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

print(car1.brand, car1.model)  # Output: Toyota Corolla
print(car2.brand, car2.model)  # Output: Honda Civic
```

10.3 Instance and Class Variables

Instance Variables

- Defined inside the `__init__` method.
- **Specific to each object**.

Class Variables

- Shared among all instances of the class.
- Defined outside `__init__`.

Example: Instance vs. Class Variables

```
class Employee:
    company = "TechCorp"  # Class variable

    def __init__(self, name, salary):
        self.name = name    # Instance variable
        self.salary = salary  # Instance variable

emp1 = Employee("Alice", 50000)
```

```
emp2 = Employee("Bob", 60000)

print(emp1.company)  # Output: TechCorp
print(emp2.company)  # Output: TechCorp

# Changing class variable
Employee.company = "NewTech"

print(emp1.company)  # Output: NewTech
print(emp2.company)  # Output: NewTech
```

10.4 Inheritance and Types of Inheritance

Inheritance allows a class to derive properties and methods from another class, promoting **code reusability**.

Types of Inheritance in Python

1. **Single Inheritance** → One class inherits from another.
2. **Multiple Inheritance** → A class inherits from multiple classes.
3. **Multilevel Inheritance** → A class inherits from a derived class.
4. **Hierarchical Inheritance** → Multiple classes inherit from a single parent.
5. **Hybrid Inheritance** → Combination of multiple inheritance types.

Example: Single Inheritance

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):  # Inheriting from Animal
    def speak(self):
        print("Dog barks")
```

```
dog = Dog()
dog.speak()  # Output: Dog barks
```

10.5 Method Overriding and Polymorphism

Method Overriding

- A subclass **redefines** a method from its parent class.

Example: Method Overriding

```
class Parent:
    def show(self):
        print("This is the parent class")

class Child(Parent):
    def show(self):
        print("This is the child class")

obj = Child()
obj.show()  # Output: This is the child class
```

Polymorphism

- Allows different objects to be treated as instances of the same class through **method overriding** or **duck typing**.

Example: Polymorphism

```
class Bird:
    def fly(self):
        print("Birds can fly")

class Penguin(Bird):
    def fly(self):
```

```
        print("Penguins cannot fly")

def flying_test(bird):
    bird.fly()

sparrow = Bird()
penguin = Penguin()

flying_test(sparrow)  # Output: Birds can fly
flying_test(penguin)  # Output: Penguins cannot fly
```

10.6 Encapsulation and Abstraction

Encapsulation

- **Hides** internal implementation using **private variables**.
- Private attributes are prefixed with `__` **(double underscore)**.

Example: Encapsulation

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

```
# Accessing private variable directly raises an error
# print(account.__balance)  # AttributeError
```

Abstraction

- **Hides complex details** from the user using **abstract classes**.

Example: Abstraction using `ABC` module

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

dog = Dog()
cat = Cat()

print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

10.7 Magic Methods (`__str__` , `__len__` , etc.)

Magic methods (also called **dunder methods**) in Python **start and end with double underscores** (`__`).

Common Magic Methods

Method	Description
`__init__(self, ...)`	Constructor (initialization method)
`__str__(self)`	Returns a string representation of an object
`__len__(self)`	Defines behavior for `len(obj)`
`__add__(self, other)`	Defines behavior for `obj1 + obj2`
`__eq__(self, other)`	Defines behavior for `obj1 == obj2`

Example: Using `__str__` and `__len__`

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"Book: {self.title}, Pages: {self.pages}"

    def __len__(self):
        return self.pages

book = Book("Python Programming", 450)
print(book)  # Output: Book: Python Programming, Pages: 450
print(len(book))  # Output: 450
```

▼ Python Libraries Overview

11. Python Libraries Overview

Python provides a vast collection of libraries that simplify complex tasks such as **data analysis, machine learning, web scraping, automation, and database management**. Below is a structured overview of some of the most commonly used libraries.

11.1 Data Analysis Libraries

NumPy (Numerical Python)

- Used for **numerical computing** with **multi-dimensional arrays**.
- Provides **mathematical functions**, **linear algebra**, and **random number generation**.

Example: Working with NumPy Arrays

```
import numpy as np

# Creating an array
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]

# Reshaping an array
arr2D = arr.reshape(1, 5)
print(arr2D)  # Output: [[1 2 3 4 5]]

# Performing mathematical operations
arr_squared = arr ** 2
print(arr_squared)  # Output: [1 4 9 16 25]
```

Pandas (Data Manipulation and Analysis)

- Provides **DataFrame and Series** structures to work with **structured data**.
- Supports **reading/writing CSV, Excel, SQL, and JSON files**.

Example: Creating a DataFrame

```
import pandas as pd

# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
```

```
df = pd.DataFrame(data)

# Display the first few rows
print(df.head())

# Reading a CSV file
df = pd.read_csv("data.csv")

# Selecting a column
print(df['Name'])
```

11.2 Data Visualization

Matplotlib (Basic and Advanced Plotting)

- Provides **2D and 3D plotting capabilities**.
- Supports **line charts, bar plots, histograms, scatter plots**.

Example: Line Plot

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

plt.plot(x, y, marker='o')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Plot Example")
plt.show()
```

Seaborn (Statistical Data Visualization)

- Built on **Matplotlib**, it provides **more aesthetically pleasing plots**.

- Supports **heatmaps, pair plots, violin plots, and more**.

Example: Creating a Heatmap

```
import seaborn as sns
import numpy as np

# Creating a random dataset
data = np.random.rand(5, 5)

# Creating a heatmap
sns.heatmap(data, annot=True, cmap="coolwarm")
plt.show()
```

11.3 Machine Learning

Scikit-learn (Machine Learning)

- A powerful library for **classification, regression, clustering, and model evaluation**.
- Supports **decision trees, random forests, SVMs, and neural networks**.

Example: Linear Regression

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Creating sample data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

# Training the model
model = LinearRegression()
model.fit(X, y)
```

```
# Predicting a new value
print(model.predict([[6]]))  # Output: [12.]
```

11.4 Web Scraping

Requests (Fetching Web Pages)

- Used to send **HTTP requests** to fetch web pages.

Example: Fetching a Web Page

```
import requests

response = requests.get("<https://www.example.com>")
print(response.text)  # Prints the HTML content of the page
```

BeautifulSoup (Parsing HTML & XML)

- Helps extract **data from HTML pages** efficiently.

Example: Extracting Titles from a Web Page

```
from bs4 import BeautifulSoup

html = "<html><head><title>Sample Page</title></head><body></body
></html>"
soup = BeautifulSoup(html, "html.parser")

print(soup.title.text)  # Output: Sample Page
```

11.5 Automation

Selenium (Automating Browsers)

- Automates **browser actions** such as clicking buttons, filling forms, and navigating pages.

Example: Automating Google Search

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

# Open Chrome browser
driver = webdriver.Chrome()

# Open Google
driver.get("<https://www.google.com>")

# Search for "Python"
search_box = driver.find_element("name", "q")
search_box.send_keys("Python")
search_box.send_keys(Keys.RETURN)

# Close the browser
driver.quit()
```

11.6 Working with Databases

SQLite (Lightweight Database)

- **Embedded database** used in mobile and small-scale applications.

Example: Creating a Database and Table

```
import sqlite3

conn = sqlite3.connect("example.db")  # Create or open a database
cursor = conn.cursor()
```

```
# Creating a table
cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMA
RY KEY, name TEXT)")

# Inserting data
cursor.execute("INSERT INTO users (name) VALUES ('Alice')")
conn.commit()

# Retrieving data
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())  # Output: [(1, 'Alice')]

conn.close()
```

MySQL (Relational Database Management System - RDBMS)

- Used for **scalable database solutions**.
- Requires the **MySQL connector** to interact with databases.

Example: Connecting to MySQL Database

```
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="testdb"
)

cursor = conn.cursor()

# Creating a table
cursor.execute("CREATE TABLE IF NOT EXISTS users (id INT AUTO_INCRE
MENT PRIMARY KEY, name VARCHAR(100))")
```

```
# Inserting data
cursor.execute("INSERT INTO users (name) VALUES ('Alice')")
conn.commit()

# Retrieving data
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())

conn.close()
```

▼ Working with APIs

12. Working with APIs in Python

APIs (Application Programming Interfaces) allow applications to communicate and exchange data. Python provides powerful tools for interacting with **RESTful APIs**, fetching data, and even creating your own APIs.

12.1 Understanding APIs and RESTful Services

- APIs allow different applications to communicate using predefined rules.
- **RESTful APIs** follow the **REST (Representational State Transfer)** architecture, using HTTP methods such as:
  - `GET` → Retrieve data
  - `POST` → Send data
  - `PUT` → Update data
  - `DELETE` → Remove data

Example of a REST API endpoint:

```
https://api.example.com/users/1
```

This may return data in **JSON format**, like:

```
{
   "id": 1,
   "name": "Alice",
   "email": "alice@example.com"
}
```

12.2 Making HTTP Requests (`requests` module)

Python's `requests` module allows us to send HTTP requests and interact with APIs.

Installing the requests module

```
pip install requests
```

Example: Making a GET Request

```
import requests

# Sending a GET request to an API
response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

# Checking if the request was successful
if response.status_code == 200:
   print(response.json())  # Prints the response in JSON format
else:
   print("Failed to retrieve data")
```

Example: Sending a POST Request

```
import requests

url = "https://jsonplaceholder.typicode.com/posts"
```

```
# Data to send
data = {
    "title": "New Post",
    "body": "This is a sample post",
    "userId": 1
}

# Sending POST request
response = requests.post(url, json=data)

print(response.json())  # Output: Created post details
```

12.3 Parsing JSON Data (`json` module)

APIs often return data in JSON format. Python's `json` module helps parse and manipulate this data.

Example: Parsing JSON Response

```
import json

json_data = '{"name": "Alice", "age": 25, "city": "New York"}'

# Convert JSON string to Python dictionary
data = json.loads(json_data)
print(data["name"])  # Output: Alice
```

Example: Converting Python Data to JSON

```
import json

python_dict = {"name": "Bob", "age": 30, "city": "Los Angeles"}

# Convert dictionary to JSON string
```

```
json_output = json.dumps(python_dict, indent=4)
print(json_output)
```

12.4 Working with Public APIs (OpenWeather, GitHub API, etc.)

Python can interact with **public APIs** like OpenWeatherMap, GitHub API, and more.

Example: Fetching Weather Data from OpenWeather API

```
import requests

api_key = "your_api_key_here"
city = "London"
url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid
={api_key}"

response = requests.get(url)
weather_data = response.json()

print(f"Weather in {city}: {weather_data['weather'][0]['description']}")
```

> 🔷 Note: You need to sign up at OpenWeather to get an API key.

Example: Fetching GitHub User Data

```
import requests

username = "octocat"
url = f"https://api.github.com/users/{username}"
```

```
response = requests.get(url)
data = response.json()

print(f"GitHub User: {data['login']}")
print(f"Public Repos: {data['public_repos']}")
```

12.5 Creating a Simple API using Flask

Python's **Flask** framework allows us to create our own APIs.

Installing Flask

```
pip install flask
```

Example: Creating a Simple API

```
from flask import Flask, jsonify

app = Flask(__name__)

# Sample data
users = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]

@app.route("/users", methods=["GET"])
def get_users():
    return jsonify(users)

@app.route("/users/<int:user_id>", methods=["GET"])
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if user:
        return jsonify(user)
```

```
    return jsonify({"error": "User not found"}), 404

if __name__ == "__main__":
    app.run(debug=True)
```

- Run this script and open http://127.0.0.1:5000/users in your browser to see the API response.

▼ Web Development with Python

13. Web Development with Python

Python is widely used for web development, with frameworks like **Flask** and **Django** making it easy to build web applications. This section focuses on **Flask**, a lightweight and flexible framework.

13.1 Flask Framework

Flask is a **micro web framework** that provides essential tools for building web applications without unnecessary complexity. It is ideal for small to medium-sized applications and APIs.

Installing Flask

To install Flask, use:

```
pip install flask
```

Basic Flask Application

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
```

```
    return "Hello, Flask!"

if __name__ == "__main__":
    app.run(debug=True)
```

- Save this as `app.py` and run `python app.py`.
- Open http://127.0.0.1:5000/ in your browser to see the output.

Creating Routes and Views

Routes define the URLs of your application and map them to specific functions.

Example: Creating Multiple Routes

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Welcome to the Home Page"

@app.route("/about")
def about():
    return "This is the About Page"

if __name__ == "__main__":
    app.run(debug=True)
```

- Visiting http://127.0.0.1:5000/about will show `"This is the About Page"`.

Dynamic Routes

Dynamic routes allow passing parameters in URLs.

```
@app.route("/user/<name>")
def user(name):
    return f"Hello, {name}!"
```

- http://127.0.0.1:5000/user/Nihar → `"Hello, Nihar!"`

Handling Forms in Flask

Flask makes handling user input from forms easy.

Installing Flask-WTF (For Forms)

```
pip install flask-wtf
```

Example: Simple Form Handling

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        name = request.form["name"]
        return f"Hello, {name}!"
    return '''
        <form method="post">
            Name: <input type="text" name="name">
            <input type="submit">
        </form>
    '''

if __name__ == "__main__":
    app.run(debug=True)
```

- Accepts user input and returns a personalized greeting.
