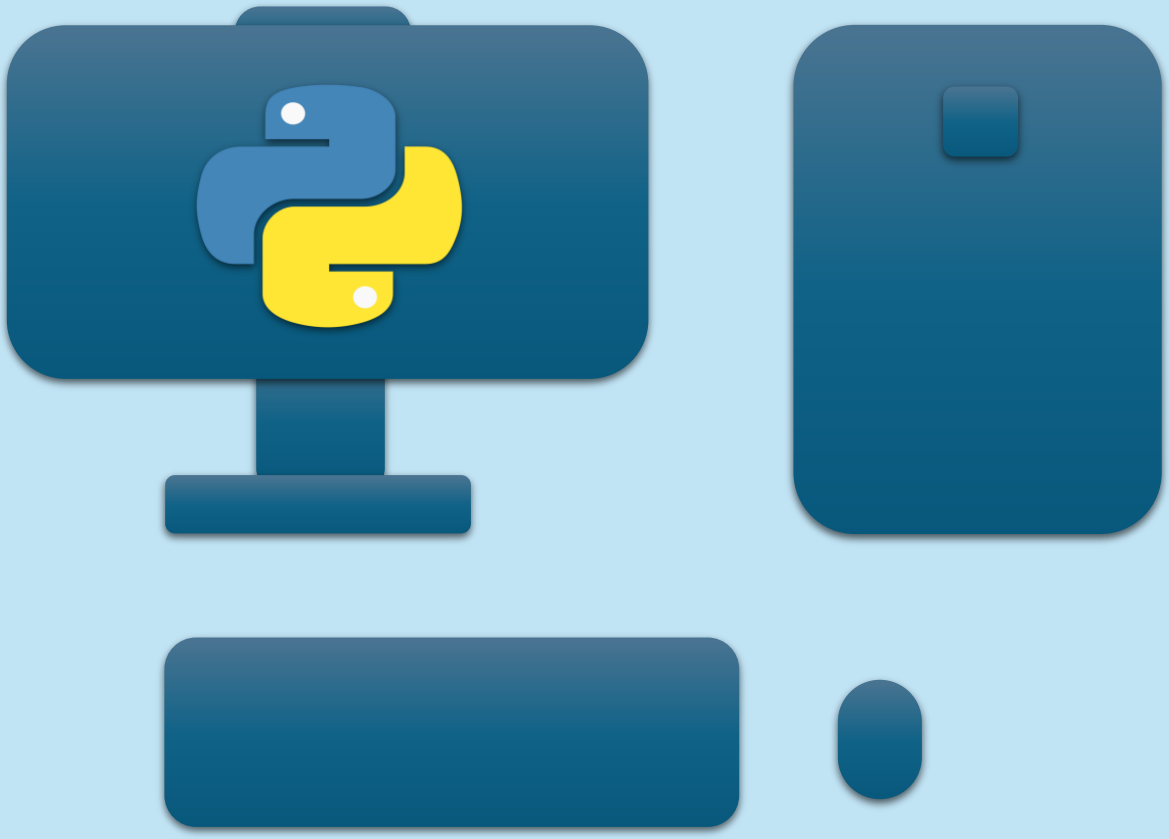
PYTHON HANDBOOK

BY ANIMESH SINGHA & AI



Preface

Welcome to this Python journey!

This eBook is designed for students, beginners, and enthusiasts who wants to learn **Python programming** step-by-step – from the very basics to advanced concepts, and finally into real-world projects.

Python is more than just a programming language. It's the **language of problem** solving, powering websites, apps, data analysis, artificial intelligence, automation, and even games. Whether you dream of becoming a developer, an ethical hacker, or a data scientist, Python is the foundation that will carry you forward.

In this book, you will find:

- **Clear explanation** of every concept
- **Simple examples** that make learning fun
- **Cheatsheets & tips** for quick revision
- **Mini projects** to apply your knowledge
- **Advanced topics** to prepare you for real-world applications

The goal is not just to teach you how to code, but to help you **think like a programmer**. By the end, you'll be confident in writing your own programs, solving problems, and creating amazing things with Python.

This is not just a book – it's a **hands-on guide** to turning your curiosity into skill. So, grab your keyboard, open your Python editor, and let's start this exciting journey together.

Edition-1 *(2025-2027)*

CONTENTS

Chapter 1: Python Basics

1.1 What is Python?

Python is a high-level, interpreted programming language known for being:

Easy to learn (uses simple English-like syntax)

Powerful (used in AI, hacking, games, apps, web, data science, automation)

Free & Open-source

1.2 Installing Python

1. Download from www.python.org

2. Install an IDE:

- VS Code (recommended)
- PyCharm
- Or use online compilers like Replit/Jupyter

Check installation by typing in CMD/Terminal:

python --version

1.3 Your First Python Program

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

Output:

Hello, World!

1.4 Variables & Data Types

Variables store values. In Python, you don’t need to declare the type.

```
# Examples
name = "Animesh"          # string
age = 14                  # integer
height = 5.6              # float
is_student = True         # boolean
```

1.5 Input & Output

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

1.6 Basic Operations

```
a = 10
b = 3
print("Addition:", a + b)          # 13
print("Subtraction:", a - b)       # 7
print("Multiplication:", a * b)    # 30
print("Division:", a / b)          # 3.33
print("Floor Division:", a // b)   # 3
print("Modulus:", a % b)           # 1
print("Power:", a ** b)            # 1000
```

1.7 Comments

Comments help explain code.

```
# This is a single-line comment
"""
This is a
multi-line comment
"""
```

1.8 Exercises 📝

1. Write a program to print your name, age, and class.

2. Take two numbers from the user and print their sum, difference, product, and division.

3. Write a program that asks the user's name and greets them with:

Hello <name>, nice to meet you!

Chapter 2: Control Flow

Control flow lets your program decide what to do or repeat actions. It's like giving your code a brain 🧠.

2.1 if, else, elif Statements

Syntax:

```
if condition:
   # code if condition is True
elif another_condition:
   # code if another condition is True
else:
   # code if all conditions are False
```

Example:

```
age = 18
if age < 13:
   print("You are a child.")
elif age < 20:
   print("You are a teenager.")
else:
   print("You are an adult.")
```

Output:
You are a teenager.

2.2 Comparison & Logical Operators

- == (equal)
- != (not equal)
- >, < , >= , <=
- and , or , not

Example:

```
marks = 75
if marks >= 90:
   print("Grade A")
elif marks >= 60 and marks < 90:
   print("Grade B")
else:
   print("Grade C")
```

2.3 Loops

For Loop

Used when you know how many times to repeat.

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

Output:
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4

While Loop

Used when you don’t know how many times, but need to repeat until a condition is false.

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

Output:
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5

2.4 break & continue

break → stops the loop immediately.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Output:

0 1 2 3 4

continue → skips the current iteration and goes to the next.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output:

0 1 3 4

2.5 Nested Loops

Loops inside loops.

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

Output:

0 0
0 1
1 0
1 1
2 0
2 1

2.6 Exercises 📝

1. Write a program that checks if a number is positive, negative, or zero

2. Print all numbers from 1 to 50, but skip multiples of 5 using continue.

3. Write a program that asks for a number and prints its multiplication table using a loop.

4. Make a simple password checker:

    - Ask user for a password
    - If password = "python123", print "Access Granted"
    - Else, print "Wrong Password"

Chapter 3: Functions & Modules

Functions help you organize, reuse, and simplify your code. Modules let you use extra features without writing everything yourself.

3.1 What is a Function?

A function is a block of code that runs only when you call it.

Defining & Calling a Function

```
def greet():
    print("Hello, welcome to Python!")
greet()  # calling the function
```

Output:

Hello, welcome to Python!

3.2 Parameters & Return Values

Function with Parameters:

```
def greet(name):
    print("Hello", name)
greet("Animesh")
```

Output:
Hello Animesh

Function with Return:

```
def add(a, b):
    return a + b
result = add(5, 3)
print("Sum =", result)
```

Output:

Sum = 8

3.3 Default Parameters

If you don’t give a value, Python uses the default.

```
def greet(name="User"):
    print("Hello", name)
greet()                 # Hello User
greet("Ani")            # Hello Ani
```

3.4 Keyword & Positional Arguments

```
def student(name, age):
    print("Name:", name, "Age:", age)
student("Rana", 15)                     # Positional
student(age=16, name="Parijat")         # Keyword
```

3.5 Modules in Python

A module is a file with Python code (functions, classes, variables) you can reuse.

Importing a Module

```
import math
print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.141592653589793
```

Import Specific Function

```
from math import sqrt
print(sqrt(25))   # 5.0
```

3.6 Creating Your Own Module

1. Create a file mymath.py:

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

2. Use it in another file:

```
import mymath
print(mymath.add(10, 5))
print(mymath.sub(10, 5))
```

3.7 Mini Project: Calculator 🧮

```
def add(a, b): return a + b
def sub(a, b): return a - b
def mul(a, b): return a * b
def div(a, b): return a / b

print("Simple Calculator")
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print("1. Add\n2. Subtract\n3. Multiply\n4. Divide")
choice = int(input("Enter choice: "))

if choice == 1:
    print("Result:", add(a, b))
elif choice == 2:
    print("Result:", sub(a, b))
elif choice == 3:
    print("Result:", mul(a, b))
elif choice == 4:
    print("Result:", div(a, b))
else:
    print("Invalid choice")
```

3.8 Exercises 📝

1. Write a function that takes a number and returns whether it’s even or odd.

2. Create a function factorial(n) that returns the factorial of a number.

3. Make a module geometry.py with functions:

    - area_circle(r)
    - area_square(s)
    - Import and use it in another file

Chapter 4: Data Structures

Data structures are ways to store and organize data in Python. The main ones are:

- List → Ordered, changeable, allows duplicates
- Tuple → Ordered, unchangeable, allows duplicates
- Set → Unordered, no duplicates
- Dictionary → Key-value pairs

4.1 Lists 📝

A list is like a container that can store multiple items.

```
fruits = ["apple", "banana", "mango"]
print(fruits)        # ['apple', 'banana', 'mango']
print(fruits[0])     # apple
```

Adding & Removing Items

```
fruits.append("orange")    # add at end
fruits.insert(1, "grape")  # add at index
fruits.remove("banana")    # remove by value
fruits.pop(0)              # remove by index
print(fruits)
```

Slicing

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])  # [20, 30, 40]
print(numbers[:3])   # [10, 20, 30]
print(numbers[-1])   # 50
```

4.2 Tuples 🔒

Tuples are like lists but cannot be changed.

```
colors = ("red", "green", "blue")
print(colors[0])  # red
# colors[0] = "yellow" ❌ Error (immutable)
```

4.3 Sets 🎲

Sets are unordered, no duplicates allowed.

```
nums = {1, 2, 3, 3, 4}
print(nums)   # {1, 2, 3, 4} → duplicates removed
```

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))       # {1, 2, 3, 4, 5}
print(a.intersection(b)) # {3}
print(a.difference(b))   # {1, 2}
```

4.4 Dictionaries 📖

Dictionaries store data in key-value pairs.

```
student = {
   "name": "Animesh",
   "age": 14,
   "class": 9
}
print(student["name"])    # Animesh
print(student.get("age"))  # 14
```

Adding & Removing Items

```
student["school"] = "DPS"
student.pop("class")
print(student)
```

4.5 Dictionary Methods

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

Output:

```
name : Animesh
age : 14
school : DJPS
```

4.6 List & Dictionary Comprehensions

A short way to create lists/dictionaries.

```
# List comprehension
squares = [x**2 for x in range(5)]
print(squares)   # [0, 1, 4, 9, 16]

# Dictionary comprehension
nums = {x: x**2 for x in range(5)}
print(nums)   # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

4.7 Mini Project: Contact Book 📒

```
contacts = {}

while True:
    print("\n1. Add Contact\n2. View Contacts\n3. Search\n4. Exit")
    choice = int(input("Enter choice: "))

    if choice == 1:
        name = input("Enter name: ")
        number = input("Enter number: ")
        contacts[name] = number
        print("Contact saved!")
    elif choice == 2:
        for name, number in contacts.items():
            print(name, ":", number)
    elif choice == 3:
        name = input("Enter name to search: ")
        if name in contacts:
            print("Number:", contacts[name])
        else:
            print("Not found!")
    elif choice == 4:
        break
    else:
        print("Invalid choice!")
```

4.8 Exercises 📝

1. Create a list of 5 numbers and print their sum.

2. Store 5 student names in a tuple and print each one.

3. Make a set of numbers 1–10 and another set of even numbers. Find their intersection.

4. Create a dictionary of 3 friends with their ages. Print only the names of friends above 15.

Chapter 5: File Handling

File handling lets your program store data permanently in files instead of just memory. You can read, write, update, and delete files.

5.1 Opening & Closing Files

Syntax:

```
file = open("filename.txt", "mode")
# modes: "r" = read, "w" = write, "a" = append, "x" = create
file.close()
```

Example:

```
file = open("test.txt", "w")
file.write("Hello, this is my first file!")
file.close()
```

5.2 Reading Files

```
file = open("test.txt", "r")
content = file.read()
print(content)
file.close()
```

Read Line by Line

```
file = open("test.txt", "r")
for line in file:
    print(line.strip())
file.close()
```

5.3 Using with (Best Practice ✅)

with automatically closes the file.

```
with open("test.txt", "r") as f:
    print(f.read())
```

5.4 Writing & Appending

 # Writing (overwrites old content)

```
with open("test.txt", "w") as f:
    f.write("This will replace old content.\n")
```

 # Appending (adds to file)

```
with open("test.txt", "a") as f:
    f.write("This is new content.\n")
```

5.5 Working with CSV Files 📊

CSV (Comma Separated Values) is used for storing tabular data.

```
import csv

# Writing CSV
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Animesh", 14])
    writer.writerow(["Parijat", 15])

# Reading CSV
with open("students.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age']
['Animesh', '14']
['Parijat', '15']
```

5.6 Working with JSON Files 🌐

JSON is used for storing structured data (common in APIs & databases).

```
import json

data = {
  "name": "Animesh",
  "age": 14,
  "skills": ["Python", "Hacking", "Gaming"]
}
# Writing JSON
with open("data.json", "w") as f:
  json.dump(data, f, indent=4)
# Reading JSON
with open("data.json", "r") as f:
  content = json.load(f)
  print(content)
```

5.7 Mini Project: To-Do App ✅

```
import json

# Load existing tasks
try:
  with open("todo.json", "r") as f:
    tasks = json.load(f)
except:
  tasks = []

while True:
  print("\n1. Add Task\n2. View Tasks\n3. Exit")
  choice = int(input("Enter choice: "))

  if choice == 1:
    task = input("Enter task: ")
    tasks.append(task)
    with open("todo.json", "w") as f:
      json.dump(tasks, f, indent=4)
```

```
        print("Task added!")
    elif choice == 2:
        print("\nYour Tasks:")
        for i, t in enumerate(tasks, 1):
            print(i, "-", t)
    elif choice == 3:
        break
    else:
        print("Invalid choice!")
```

5.8 Exercises 📝

1. Write a program to create a file and write your name, age, and class.

2. Write a program that reads a text file and counts how many lines are in it.

3. Create a CSV file marks.csv with 3 students' marks. Read the file and print the average marks.

4. Create a JSON file that stores your favorite games and then read & display them.

Chapter 6: OOP in Python

OOP is a way of writing programs using objects (real-world things) and classes (blueprints).
It makes code more organized, reusable, and scalable.

6.1 Classes & Objects

Defining a Class

```
class Student:
  def __init__(self, name, age):  # constructor
    self.name = name
    self.age = age
  def introduce(self):  # method
    print("Hi, I am", self.name, "and I am", self.age, "years old.")

# Creating Objects
s1 = Student("Animesh", 14)
s2 = Student("Parijat", 15)

s1.introduce()
s2.introduce()
```

Output:
Hi, I am Animesh and I am 14 years old.
Hi, I am Parijat and I am 15 years old.

6.2 Attributes & Methods

- Attributes = variables inside a class (like name, age)
- Methods = functions inside a class

```
class Car:
  def __init__(self, brand, speed):
    self.brand = brand
    self.speed = speed

  def drive(self):
    print(self.brand, "is driving at", self.speed, "km/h")

car1 = Car("BMW", 120)
car1.drive()
```

6.3 Inheritance

A class can inherit features from another class.

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

class Dog(Animal):  # Dog inherits Animal
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

d = Dog()
c = Cat()
d.speak()
c.speak()
```

Output:
Dog barks
Cat meows

6.4 Encapsulation (Data Hiding)

Use private variables with _ or __.

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

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

    def get_balance(self):
        return self.__balance
```

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

6.5 Polymorphism (Many Forms)

Different classes can have the same method name.

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

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly")

b = Bird()
p = Penguin()
b.fly()
p.fly()
```

6.6 Class vs Instance Variables

```
class Student:
    school = "DPS"  # class variable (same for all)

    def __init__(self, name):
        self.name = name  # instance variable (different for each)

s1 = Student("Animesh")
s2 = Student("Parijat")

print(s1.name, "-", s1.school)
print(s2.name, "-", s2.school)
```

6.7 Mini Project: Student Management System 🎓

```
class Student:
    def __init__(self, name, roll, marks):
        self.name = name
        self.roll = roll
        self.marks = marks

    def display(self):
        print(f"Name: {self.name}, Roll: {self.roll}, Marks: {self.marks}")

students = []

while True:
    print("\n1. Add Student\n2. View Students\n3. Exit")
    choice = int(input("Enter choice: "))

    if choice == 1:
        name = input("Enter name: ")
        roll = input("Enter roll: ")
        marks = int(input("Enter marks: "))
        students.append(Student(name, roll, marks))
        print("Student added!")
    elif choice == 2:
        for s in students:
            s.display()
    elif choice == 3:
        break
    else:
        print("Invalid choice!")
```

6.8 Exercises 📝

1. Create a class Rectangle with methods to calculate area and perimeter.

2. Create a class Employee with attributes name, salary. Add a method to check if salary is above 50,000.

3. Implement a Library System with classes Book and Library. Allow adding books and viewing them.

Chapter 7: Exception Handling & Debugging

When programs crash because of errors, it's called an exception.
Exception handling makes sure your program doesn't stop suddenly. Instead, it gives a safe message.

7.1 Errors vs Exceptions

- Errors → Problems in code (e.g., syntax error, missing colon).
- Exceptions → Runtime problems (e.g., dividing by zero, missing file).

Example of an exception:

```
print(10 / 0)   # ZeroDivisionError
```

7.2 try and except

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print("Result =", a / b)
except:
    print("Something went wrong!")
```

7.3 Handling Specific Exceptions

```
try:
    x = int("abc")
except ValueError:
    print("Invalid input! Please enter numbers only.")
```

7.4 finally Block

finally always runs — useful for closing files or cleanup.

```
try:
    f = open("test.txt", "r")
    print(f.read())
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing file...")
```

7.5 else with try

else runs if no exception occurs.

```
try:
   num = int(input("Enter a number: "))
except ValueError:
   print("Invalid number!")
else:
   print("You entered:", num)
```

7.6 Raising Exceptions Manually 🚨

```
age = int(input("Enter your age: "))
if age < 0:
   raise ValueError("Age cannot be negative!")
```

7.7 Debugging Tips 🛠️

1. Use print() to check variable values.

2. Use Python's built-in debugger:

```
import pdb
pdb.set_trace()
```

(Lets you pause code and check values step by step.)

3. Write clean, modular code with functions.

4. Test small parts of the program before combining.

7.8 Mini Project: Safe Calculator 🧮

```
while True:
   try:
      a = int(input("Enter first number: "))
      b = int(input("Enter second number: "))
      print("Division =", a / b)
   except ValueError:
      print("Error: Please enter numbers only!")
   except ZeroDivisionError:
      print("Error: Cannot divide by zero!")
```

```
else:
    print("Operation successful!")
    break
finally:
    print("End of attempt.\n")
```

7.9 Exercises 📝

1. Write a program that asks for a number and prints its square. Handle the case when the user enters a string.

2. Make a program that opens a file. If the file doesn’t exist, show "File Missing!".

3. Create a function that takes an age. Raise an exception if age is below 0 or above 120.

4. Extend the calculator from Chapter 3 to handle invalid input gracefully.

Chapter 6.1: Functions in Python

6.1 What is a Function?

A function is a block of reusable code that performs a specific task.
Instead of writing the same code again and again, you can put it inside a function and call it whenever needed.

6.2 Defining a Function

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

Calling the function:

```
greet()
```

Output:

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

6.3 Function with Parameters

You can pass data (called arguments) into a function.

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

Calling it:

```
greet("Animesh")
greet("Parijat")
```

Output:

```
Hello Animesh !
Hello Parijat !
```

6.4 Function with Return Value

Functions can return values using the return keyword.

```
def add(a, b):
    return a + b
result = add(5, 7)
print("Sum =", result)
```

Output:
Sum = 12

6.5 Default Arguments

If a value is not given, Python uses the default value.

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

Output:
Hello Animesh
Hello Guest

6.6 Keyword Arguments

You can specify which parameter you are passing.

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

intro(age=14, name="Animesh")
```

Output:
Name: Animesh
Age: 14

6.7 Variable Number of Arguments

```
*args → Multiple arguments
def add_all(*numbers):
    return sum(numbers)
print(add_all(1, 2, 3, 4, 5))   # 15

**kwargs → Multiple keyword arguments
def info(**details):
    for key, value in details.items():
        print(key, ":", value)
info(name="Animesh", age=14, country="India")
```

6.8 Lambda Functions (Anonymous Functions)

A lambda function is a small one-line function.

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

6.9 Exercises 📝

1. Write a function that takes two numbers and returns their product.

2. Write a function is_even(n) that returns True if the number is even, otherwise False.

3. Write a function that takes a name and age, and prints:

Hello <name>, you are <age> years old.

4. Write a lambda function that returns the cube of a number.

Chapter 7: Strings in Python

7.1 What is a String?

A string is a sequence of characters enclosed in single quotes, double quotes, or triple quotes.

```
str1 = 'Hello'
str2 = "World"
str3 = '''This is
a multi-line
string.'''
```

7.2 Accessing Characters

Strings work like arrays (indexed from 0).

```
text = "Python"
print(text[0])   # P
print(text[2])   # t
print(text[-1])  # n (last character)
```

7.3 String Slicing

You can extract parts of a string.

```
word = "Programming"
print(word[0:6])   # Progra
print(word[3:])    # gramming
print(word[:5])    # Progr
print(word[-4:])   # ming
```

7.4 String Operations

```
a = "Hello"
b = "World"

# Concatenation
print(a + " " + b)          # Hello World

# Repetition
print(a * 3)                # HelloHelloHello

# Length
print(len(a))               # 5
```

7.5 String Methods

Python has many built-in functions for strings:

```
text = "  python programming  "
print(text.upper())      # PYTHON PROGRAMMING
print(text.lower())      # python programming
print(text.title())      # Python Programming
print(text.strip())      # removes extra spaces
print(text.replace("python", "java"))  # java programming
print(text.find("pro"))   # 8 (index)
print(text.split())      # ['python', 'programming']
```

7.6 Checking String Content

```
name = "Animesh14"
print(name.isalpha())   # False (because of numbers)
print(name.isdigit())   # False
print(name.isalnum())   # True (letters + numbers)
print("hello".startswith("he"))  # True
print("hello".endswith("lo"))    # True
```

7.7 String Formatting

Using f-strings (recommended)

```
name = "Animesh"
age = 14
print(f"My name is {name} and I am {age} years old.")
```

Using .format()

```
print("My name is {} and I am {} years old.".format("Animesh", 14))
```

7.8 Escape Characters

```
print("Hello\nWorld")   # New line
print("Hello\tWorld")   # Tab space
print("He said \"Python is fun!\"")  # Quotes
```

7.9 Exercises 📝

1. Write a program to input your name and print:

Hello <name>, welcome to Python.

2. Take a string and print it in reverse.

3. Count the number of vowels in a string.

4. Check if a string is a palindrome (same forward and backward, e.g., "madam").

5. Write a program that asks for first name and last name, then prints full name in title case.

Chapter 8: Lists in Python

8.1 What is a List?

A list is a collection of ordered, changeable (mutable) items. Lists can hold different data types.

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = ["Animesh", 14, True, 5.6]
```

8.2 Accessing List Elements

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])   # apple
print(fruits[1])   # banana
print(fruits[-1])  # cherry (last item)
```

8.3 List Slicing

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])   # [20, 30, 40]
print(numbers[:3])    # [10, 20, 30]
print(numbers[2:])    # [30, 40, 50]
print(numbers[-2:])   # [40, 50]
```

8.4 Modifying Lists

Lists are mutable (can be changed).

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits)   # ['apple', 'mango', 'cherry']
```

8.5 Adding Elements

```
fruits = ["apple", "banana"]
fruits.append("cherry")   # add at end
fruits.insert(1, "mango") # add at position
print(fruits)
```

Output:
['apple', 'mango', 'banana', 'cherry']

8.6 Removing Elements

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

fruits.remove("banana")   # remove by value
print(fruits)
fruits.pop(1)          # remove by index
print(fruits)
del fruits[0]          # delete item
print(fruits)
fruits.clear()         # empty the list
print(fruits)
```

8.7 List Functions & Methods

```
numbers = [4, 1, 9, 3, 7]
print(len(numbers))     # 5
print(max(numbers))     # 9
print(min(numbers))     # 1
numbers.sort()        # sort ascending
print(numbers)        # [1, 3, 4, 7, 9]
numbers.reverse()      # reverse list
print(numbers)        # [9, 7, 4, 3, 1]
```

8.8 Looping through a List

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

8.9 List Comprehension

A short way to create lists.

```
squares = [x**2 for x in range(1, 6)]
print(squares)   # [1, 4, 9, 16, 25]
```

8.10 Nested Lists

Lists inside lists.

```
matrix = [[1, 2], [3, 4], [5, 6]]
print(matrix[0][1])   # 2
```

8.11 Exercises 📝

1. Create a list of 5 subjects and print them using a loop.
2. Write a program to find the largest number in a list.
3. Write a program to remove all even numbers from a list.
4. Create a 2D list (matrix) and print its elements.
5. Use list comprehension to generate a list of all odd numbers from 1–20.

Chapter 9: Tuples in Python

9.1 What is a Tuple?

- A tuple is like a list, but it is immutable (cannot be changed after creation).
- Written with round brackets ( )
- Faster than lists
- Used for data that should not change

```
fruits = ("apple", "banana", "cherry")
numbers = (1, 2, 3, 4, 5)
mixed = ("Animesh", 14, True, 5.6)
```

9.2 Accessing Tuple Elements

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])   # apple
print(fruits[-1])  # cherry
```

9.3 Tuple Slicing

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4])   # (20, 30, 40)
print(numbers[:3])    # (10, 20, 30)
print(numbers[-2:])   # (40, 50)
```

9.4 Tuple Immutability

Unlike lists, tuples cannot be modified.

```
fruits = ("apple", "banana", "cherry")
# fruits[1] = "mango" ❌ Error: Tuples are immutable
```

9.5 Tuple with One Item

Be careful — single item tuples need a comma.

```
t1 = ("apple",)  # ✅ tuple
t2 = ("apple")   # ❌ just a string
```

9.6 Tuple Functions

```
numbers = (4, 1, 9, 3, 7)
print(len(numbers))   # 5
print(max(numbers))   # 9
print(min(numbers))   # 1
print(sum(numbers))   # 24
```

9.7 Looping through Tuples

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

9.8 Tuple Packing and Unpacking

```
person = ("Animesh", 14, "India")
name, age, country = person
print(name)    # Animesh
print(age)     # 14
print(country)  # India
```

9.9 Nested Tuples

```
nested = (("a", 1), ("b", 2), ("c", 3))
print(nested[1][0])   # b
```

9.10 Why Use Tuples Instead of Lists?

✔ Faster than lists

✔ Use less memory

✔ Good for fixed data (like coordinates, database records, etc.)

9.11 Exercises 📝

1. Create a tuple with your name, age, and school name.
2. Write a program to find the maximum and minimum value in a tuple of numbers.
3. Unpack a tuple (10, 20, 30) into three variables and print them.
4. Create a nested tuple and access its inner elements.
5. Convert a list into a tuple using tuple() function.

Chapter 10: Sets in Python

10.1 What is a Set?

A set is a collection of unique and unordered elements.

- Written with curly braces { }
- No duplicate values allowed
- Order is not guaranteed

```
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}
mixed = {"Animesh", 14, True, 5.6}
```

10.2 Characteristics of Sets

✔ No duplicate elements
✔ Unordered (indexing doesn't work)
✔ Mutable (you can add/remove elements)
✔ Good for mathematical operations

```
nums = {1, 2, 2, 3, 4, 4, 5}
print(nums)   # {1, 2, 3, 4, 5}
```

10.3 Creating Sets

```
s1 = {1, 2, 3}
s2 = set([4, 5, 6])  # using set() constructor
empty = set()       # ✅ empty set
```

⚠ {} creates an empty dictionary, not a set.

10.4 Adding & Removing Elements

```
fruits = {"apple", "banana"}
fruits.add("cherry")
print(fruits)   # {'apple', 'banana', 'cherry'}
fruits.remove("banana")
print(fruits)   # {'apple', 'cherry'}
fruits.discard("mango")  # no error if not present
print(fruits)
fruits.clear()   # remove all
print(fruits)    # set()
```

10.5 Set Operations

Sets are great for math-like operations.

```
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}
```

10.6 Checking Membership

```
fruits = {"apple", "banana", "cherry"}
print("apple" in fruits)   # True
print("mango" not in fruits)  # True
```

10.7 Looping through Sets

```
for item in {"a", "b", "c"}:
    print(item)
```

10.8 Frozen Sets

A frozenset is an immutable set (cannot be changed).

```
fs = frozenset([1, 2, 3])
print(fs)
# fs.add(4) ❌ Error: cannot add to frozenset
```

10.9 Exercises 📝

1. Create a set of your 5 favorite movies.

2. Write a program to find common subjects between two students using intersection.

3. Remove duplicates from a list using a set.

4. Create two sets of numbers and find their union, intersection, and difference.

5. Convert a set into a list and sort it.

Chapter 11: Dictionaries in Python

11.1 What is a Dictionary?

- A dictionary is a collection of key-value pairs.
- Keys are unique
- Values can be anything (string, number, list, etc.)
- Written with curly braces {}

```
student = {
   "name": "Animesh",
   "age": 14,
   "school": "Darjeeling Public School"
}
```

11.2 Accessing Dictionary Items

```
print(student["name"])   # Animesh
print(student.get("age"))  # 14
```

11.3 Adding & Updating Items

```
student["grade"] = "Class 9"   # add new key-value
student["age"] = 15          # update value
print(student)
```

11.4 Removing Items

```
student.pop("school")    # remove by key
print(student)
student.popitem()       # remove last inserted
print(student)
del student["age"]      # delete key
print(student)
student.clear()         # remove all
print(student)
```

11.5 Looping through Dictionary

```
person = {"name": "Animesh", "age": 14, "country": "India"}
for key in person:
   print(key, ":", person[key])

# OR
for key, value in person.items():
   print(key, "->", value)
```

11.6 Dictionary Methods

```
student = {"name": "Animesh", "age": 14}

print(student.keys())    # dict_keys(['name', 'age'])
print(student.values())  # dict_values(['Animesh', 14])
print(student.items())   # dict_items([('name', 'Animesh'), ('age', 14)])
```

11.7 Nested Dictionaries

```
students = {
   "s1": {"name": "Animesh", "age": 14},
   "s2": {"name": "Parijat", "age": 15}
}
print(students["s1"]["name"])  # Animesh
```

11.8 Dictionary Comprehension

```
squares = {x: x**2 for x in range(1, 6)}
print(squares)   # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

11.9 When to Use Dictionaries?

- ✔ When you need fast lookups by key
- ✔ To store related information (like a database row)
- ✔ For structured data like JSON

11.10 Exercises 📝

1. Create a dictionary of 3 friends with their name as key and age as value.

2. Write a program to find the student with the highest marks from a dictionary.

3. Make a nested dictionary of 3 countries with name, capital, and population.

4. Write a program to count the frequency of each character in a string using a dictionary.

5. Convert two lists into a dictionary (keys = list1, values = list2).

Chapter 12: Functions Advanced

12.1 Function Scope

In Python, variables have scope → the region where they can be accessed.

```
x = 10  # global variable

def func():
    y = 5  # local variable
    print("Inside function:", x, y)

func()
print("Outside function:", x)
# print(y) ❌ Error: y is not accessible here
```

12.2 Global Keyword

You can modify a global variable inside a function using global.

```
count = 0
def increase():
    global count
    count += 1
increase()
print(count)  # 1
```

12.3 Recursion

A function calling itself is called recursion.

Example: Factorial

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))   # 120
```

Example: Fibonacci

```
def fibonacci(n):
    if n <= 1:
        return n
```

```
    return fibonacci(n-1) + fibonacci(n-2)
for i in range(6):
    print(fibonacci(i), end=" ")
```

Output:

0 1 1 2 3 5

12.4 Modules in Python

A module is a Python file containing functions and variables.
You can import built-in or custom modules.

Importing Built-in Module

```
import math
print(math.sqrt(16))   # 4.0
print(math.pi)       # 3.14159...
```

Importing Specific Functions

```
from math import sqrt, pi
print(sqrt(25))
print(pi)
```

Renaming a Module

```
import math as m
print(m.factorial(5))   # 120
```

12.5 Creating Your Own Module

Create a file mymath.py:

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

Use it in another file:

```
import mymath
print(mymath.add(5, 3))
```

12.6 The __name__ == "__main__" Trick

When a Python file runs directly, __name__ is "__main__".

```
def greet():
    print("Hello from module!")
if __name__ == "__main__":
    greet()
```

12.7 Built-in Useful Modules

- **random** → random numbers
- **datetime** → dates & times
- **os** → file & system operations
- **sys** → system-specific functions
- **json** → JSON parsing

Example:

```
import random
print(random.randint(1, 10))  # random number 1–10
```

12.8 Exercises 

1. Write a recursive function to calculate the sum of first n natural numbers.

2. Create a module calculator.py with add, subtract, multiply, divide functions. Import and use it.

3. Write a program to generate 5 random numbers between 1 and 50.

4. Use datetime module to print today's date and current time.

5. Write a recursive function to reverse a string.

Chapter 13: File Handling in Python

13.1 Why File Handling?

File handling allows Python programs to:

- = Store data permanently
- = Read/write data from text files (.txt), CSV files, etc.
- = Handle logs, configuration, and user data

13.2 Opening and Closing Files

Python provides the built-in open() function.

```
file = open("demo.txt", "w")  # open in write mode
file.write("Hello, Python!")
file.close()
```

Modes for opening files:

- "r" → read (default)
- "w" → write (overwrites)
- "a" → append
- "b" → binary mode (images, etc.)
- "x" → create new file

13.3 Reading Files

```
file = open("demo.txt", "r")
content = file.read()
print(content)
file.close()
```

Read line by line:

```
file = open("demo.txt", "r")
for line in file:
    print(line.strip())
file.close()
```

13.4 Writing and Appending

```
# Write (overwrites file)
file = open("demo.txt", "w")
file.write("First Line\n")
file.close()

# Append (adds to file)
file = open("demo.txt", "a")
file.write("Second Line\n")
file.close()
```

13.5 Using with Statement (Best Practice)

Automatically closes the file after use.

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

13.6 Working with CSV Files

```
import csv

# Writing CSV
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Animesh", 14])
    writer.writerow(["Parijat", 15])

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

13.7 Handling JSON Files

```
import json

# Writing JSON
data = {"name": "Animesh", "age": 14}
with open("data.json", "w") as file:
    json.dump(data, file)
# Reading JSON
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

13.8 File Methods

```
file = open("demo.txt", "r")
print(file.read(5))   # read first 5 chars
print(file.readline()) # read first line
print(file.readlines()) # read all lines as list
file.close()
```

13.9 Exception Handling in Files

```
try:
    with open("not_exist.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
```

13.10 Exercises 

1. Write a program to create a text file and write your name, class, and age.

2. Write a program to read a text file and count the number of lines.

3. Store a dictionary (student info) into a JSON file and read it back.

4. Create a CSV file with 5 students and their marks, then print all records.

5. Write a program to reverse the contents of a text file.

Chapter 14: Object-Oriented Programming (OOP)

14.1 What is OOP?

OOP is a programming paradigm based on objects that contain data (attributes) and functions (methods).

Key concepts:

- Class → Blueprint/template
- Object → Instance of a class
- Method → Function inside a class
- Attribute → Variable inside a class

14.2 Creating a Class and Object

```
class Student:
    def __init__(self, name, age):  # Constructor
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating objects
s1 = Student("Animesh", 14)
s2 = Student("Parijat", 15)

s1.display()
s2.display()
```

14.3 The __init__ Constructor

Runs automatically when an object is created.

Used to initialize object data.

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

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

```
car1 = Car("Tesla", "Model S")
car1.info()
```

14.4 Instance vs Class Variables

```
class Dog:
    species = "Mammal"  # Class variable (common for all)

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

dog1 = Dog("Tommy")
dog2 = Dog("Bruno")

print(dog1.species, dog1.name)
print(dog2.species, dog2.name)
```

14.5 Inheritance (Reusing Classes)

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        print("Animal makes sound")

# Child class inherits
class Dog(Animal):
    def speak(self):
        print("Woof!")
dog = Dog("Bruno")
dog.speak()
```

14.6 Multiple Inheritance

```
class A:
    def methodA(self):
        print("Method A")
class B:
    def methodB(self):
        print("Method B")

class C(A, B):
    pass
```

```
obj = C()
obj.methodA()
obj.methodB()
```

14.7 Polymorphism (Same name, different behavior)

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

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly")

b1 = Bird()
b2 = Penguin()

b1.fly()
b2.fly()
```

14.8 Encapsulation (Hiding Data)

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

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

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())
```

14.9 Abstraction (Hiding Implementation)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2

c = Circle(5)
print(c.area())
```

14.10 Exercises 

1. Create a Student class with attributes (name, roll, marks) and a method to display them.

2. Create a Calculator class with methods for add, subtract, multiply, divide.

3. Implement inheritance: Class Person → Teacher & Student.

4. Create a BankAccount class where deposit and withdraw methods modify balance.

5. Write a program to demonstrate polymorphism using Cat and Dog classes.

Chapter 15: Modules and Packages

15.1 What is a Module?

A module is just a Python file (.py) that contains code (functions, classes, variables).
Helps in code reusability and organization.

Example: math is a built-in module.

```
import math
print(math.sqrt(16))   # 4.0
print(math.factorial(5))  # 120
```

15.2 Importing Modules

Different ways to import:

```
import math
print(math.pi)

from math import sqrt, pi
print(sqrt(25), pi)

import math as m
print(m.sin(90))

from math import *
print(cos(0))
```

15.3 Creating Your Own Module

👉 **Create a file mymodule.py**

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

👉 **Use it in another file:**

```
import mymodule
print(mymodule.greet("Animesh"))
print(mymodule.add(10, 20))
```

15.4 Built-in Modules

Some useful Python modules:

- = math → math functions
- = random → random numbers
- = datetime → date & time
- = os → operating system
- = sys → system-specific functions

Example:

```
import random
print(random.randint(1, 10))   # random number
```

15.5 Packages in Python

A package is a collection of modules in a folder with __init__.py file.

Used for organizing large projects.

Example:

```
mypackage/
  __init__.py
  module1.py
  module2.py
```

Using package:

```
from mypackage import module1, module2
```

15.6 The __name__ == "__main__" Trick

```
# mymodule.py
def greet():
  print("Hello from module!")

if __name__ == "__main__":
  print("Running directly")
else:
  print("Imported as module")
```

15.7 Installing External Packages (pip)

pip install requests

Example:

```
import requests
response = requests.get("https://api.github.com")
print(response.json())
```

15.8 Exercises 📝

1. Write a module calculator.py with add, subtract, multiply, divide functions. Import it and use in another file.

2. Use the random module to generate a 6-digit OTP.

3. Create a package school with modules student.py and teacher.py. Import them in a main program.

4. Write a program using datetime module to print today's date in DD-MM-YYYY format.

5. Install requests and fetch data from https://jsonplaceholder.typicode.com/todos/1.

Chapter 16: Exception Handling

16.1 What is an Exception?

An exception is an error that occurs during program execution.

Example:

- Dividing by zero → ZeroDivisionError
- Using undefined variable → NameError
- Wrong data type → TypeError

Without handling, exceptions crash the program.

16.2 Try and Except

```
try:
   x = int("abc")   # error
except ValueError:
   print("Invalid input! Please enter numbers only.")
```

16.3 Multiple Exceptions

```
try:
   a = 10 / 0
except ZeroDivisionError:
   print("You cannot divide by zero!")
except ValueError:
   print("Invalid value!")
```

16.4 Using else and finally

```
try:
   num = int(input("Enter a number: "))
except ValueError:
   print("That's not a number!")
else:
   print(f"You entered {num}")
finally:
   print("Execution complete.")
```

else → runs if no exception
finally → always runs (used for cleanup, closing files, etc.)

16.5 Handling Multiple Errors in One Line

```
try:
    a = int("abc")
except (ValueError, TypeError):
    print("Something went wrong!")
```

16.6 Raising Exceptions

```
def withdraw(amount):
    if amount < 0:
        raise ValueError("Amount cannot be negative")
    else:
        print(f"Withdrew {amount} successfully!")
withdraw(-100)
```

16.7 Custom Exceptions

```
class TooYoungError(Exception):
    pass

age = 12
try:
    if age < 18:
        raise TooYoungError("You must be 18+ to register.")
except TooYoungError as e:
    print(e)
```

16.8 Real-Life Example

```
try:
    with open("data.txt", "r") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("File not found, please check the filename!")
```

16.9 Exercises 📝

1. Write a program that handles division by zero using try-except.

2. Create a program that asks for age and raises a custom exception if the user is under 18.

3. Handle file reading errors (file not found).

4. Write a calculator that handles invalid input using try-except.

5. Demonstrate try-except-else-finally in a program that reads a number.

Chapter 17: File System and OS Module

17.1 What is the OS Module?

The os module allows Python programs to interact with the operating system.
Useful for file/folder operations, paths, and environment info.

```
import os
```

17.2 Getting Current Working Directory

```
import os
cwd = os.getcwd()
print("Current Directory:", cwd)
```

17.3 Changing Directory

```
os.chdir("C:/Users")
print("Directory changed to:", os.getcwd())
```

17.4 Listing Files and Folders

```
files = os.listdir()
print(files)  # list of files and folders in current directory
```

17.5 Creating and Removing Directories

```
# Create folder
os.mkdir("new_folder")
os.makedirs("parent/child")  # create nested folders

# Remove folder
os.rmdir("new_folder")
os.removedirs("parent/child")  # removes nested folders
```

17.6 File Operations with OS

```
# Rename file
os.rename("old.txt", "new.txt")

# Remove file
os.remove("new.txt")
```

17.7 Checking File or Directory

```
print(os.path.exists("demo.txt"))     # True or False
print(os.path.isfile("demo.txt"))     # True or False
print(os.path.isdir("new_folder"))    # True or False
```

17.8 Path Operations

```
print(os.path.join("folder", "file.txt"))  # folder/file.txt
print(os.path.basename("folder/file.txt")) # file.txt
print(os.path.dirname("folder/file.txt"))  # folder
print(os.path.split("folder/file.txt"))    # ('folder', 'file.txt')
```

17.9 Environment Variables

```
print(os.environ)         # all env variables
print(os.environ.get("HOME"))  # get specific variable
```

17.10 Walking Through Directory

```
for root, dirs, files in os.walk("."):
    print("Root:", root)
    print("Directories:", dirs)
    print("Files:", files)
```

17.11 Exercises 📝

1. Write a program to list all files in your current directory.

2. Create a folder named test_folder, then delete it.

3. Write a program to rename a file safely.

4. Use os.walk to print all files in a folder and its subfolders.

5. Write a program to check if a file exists before reading it.

Chapter 18: Regular Expressions (Regex)

18.1 What is Regex?

Regex is a sequence of characters used to match patterns in text.
Useful for searching, validation, and text manipulation.
Python provides the re module for regex.

```
import re
```

18.2 Basic Functions

```
text = "My number is 9876543210"

# Search for a pattern
match = re.search(r"\d+", text)  # \d+ = one or more digits
if match:
    print("Found:", match.group())
```

18.3 Match vs Search

```
text = "Hello World"
print(re.match(r"Hello", text))  # matches at start
print(re.match(r"World", text))  # None
print(re.search(r"World", text)) # matches anywhere
```

18.4 Find All Matches

```
text = "Call 123 or 456 or 789"
numbers = re.findall(r"\d+", text)
print(numbers)  # ['123', '456', '789']
```

18.5 Split Text Using Regex

```
text = "apple,banana;cherry orange"
fruits = re.split(r"[ ,;]", text)
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

18.6 Replace Text (Substitute)

```
text = "My number is 9876543210"
new_text = re.sub(r"\d", "X", text)
print(new_text)  # My number is XXXXXXXXXX
```

18.7 Common Regex Patterns

Pattern	Meaning
\d	Digit (0-9)
\D	Non-digit
\w	Alphanumeric (a-z, A-Z, 0-9, _)
\W	Non-alphanumeric
\s	Whitespace (space, tab)
\S	Non-whitespace
.	Any character except newline
^	Start of string
$	End of string
+	One or more
*	Zero or more
?	Zero or one
{n}	Exactly n times
[abc]	a or b or c
[^abc]	Not a, b, or c

18.8 Validating an Email

```
email = "animesh@example.com"
pattern = r"^[a-zA-Z0-9._]+@[a-zA-Z]+\.[a-zA-Z]{2,3}$"
if re.match(pattern, email):
   print("Valid Email")
else:
   print("Invalid Email")
```

18.9 Validating a Phone Number

```
phone = "9876543210"
pattern = r"^[6-9]\d{9}$"  # starts with 6-9 and has 10 digits
if re.match(pattern, phone):
    print("Valid Phone Number")
else:
    print("Invalid Phone Number")
```

18.10 Exercises 

1. Extract all numbers from the text: "Call 123 or 456 or 789".

2. Validate emails like "user@domain.com".

3. Replace all digits in a string with #.

4. Split a text by spaces, commas, or semicolons.

5. Write a regex to validate Indian mobile numbers.

Chapter 19: Decorators and Generators

19.1 What is a Decorator?

A decorator is a function that modifies another function without changing its code.
Used for logging, authentication, timing, etc.

19.2 Basic Decorator

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

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

# Decorating manually
say_hello = decorator(say_hello)
say_hello()
```

Output:

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

19.3 Using @ Symbol

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

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

19.4 Decorator with Arguments

```
def decorator(func):
    def wrapper(name):
        print(f"Hello, {name}!")
        func(name)
    return wrapper

@decorator
def greet(name):
    print("Welcome!")
greet("Animesh")
```

19.5 What is a Generator?

- A generator is a function that yields values one by one instead of returning all at once.
- Uses the yield keyword.
- Saves memory for large data.

19.6 Basic Generator

```
def my_generator():
    yield 1
    yield 2
    yield 3
gen = my_generator()
for value in gen:
    print(value)
```

Output:

```
1
2
3
```

19.7 Fibonacci Generator

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
for num in fibonacci(6):
    print(num)
```

Output:

0 1 1 2 3 5

19.8 Generator Expressions

```
squares = (x**2 for x in range(5))
for num in squares:
    print(num)
```

19.9 Advantages of Generators

Memory efficient (yields one value at a time)

Faster for large sequences

Can be iterated only once

19.10 Exercises 📝

1. Write a decorator to print execution time of a function.

2. Create a decorator to capitalize the output string of a function.

3. Write a generator to yield even numbers up to n.

4. Create a Fibonacci generator that yields numbers less than 100.

5. Convert a list comprehension into a generator expression.

Chapter 20: Python Advanced Topics

This chapter covers Context Managers, Lambda Functions, Map, Filter, and Reduce.

20.1 Context Managers (with Statement)

Used to automatically manage resources, like files.

Ensures proper cleanup even if exceptions occur.

```
# Without context manager
file = open("demo.txt", "w")
file.write("Hello Python!")
file.close()

# With context manager
with open("demo.txt", "w") as file:
    file.write("Hello Python!")
# file is automatically closed
```

20.2 Lambda Functions (Anonymous Functions)

Anonymous one-line functions.

Syntax: lambda arguments: expression

```
# Normal function
def add(a, b):
    return a + b
# Lambda function
add = lambda a, b: a + b
print(add(5, 3))  # 8
```

20.3 Map Function

Applies a function to each item of an iterable.

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)  # [1, 4, 9, 16]
```

20.4 Filter Function

Filters items in an iterable based on a condition.

```
nums = [1, 2, 3, 4, 5, 6]
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)  # [2, 4, 6]
```

20.5 Reduce Function

Reduces an iterable to a single value using a function.

Requires functools module.

```
from functools import reduce

nums = [1, 2, 3, 4]
sum_all = reduce(lambda a, b: a + b, nums)
print(sum_all)  # 10
```

20.6 Combining Map, Filter, Reduce

```
from functools import reduce

nums = [1, 2, 3, 4, 5, 6]
# Filter even numbers
even = filter(lambda x: x % 2 == 0, nums)
# Square them
squared = map(lambda x: x**2, even)
# Sum all squares
result = reduce(lambda a, b: a + b, squared)
print(result)  # 56  (4+16+36)
```

20.7 Other Advanced Features

List/Dictionary/Set Comprehensions → concise loops

Enumerate → index + value in loop

```
names = ["Animesh", "Parijat"]
for i, name in enumerate(names, start=1):
    print(i, name)
```

Zip → combine iterables

```
a = [1, 2, 3]
b = ["a", "b", "c"]
for x, y in zip(a, b):
    print(x, y)
```

20.8 Exercises 📝

1. Write a lambda function to cube a number.

2. Use map to double all numbers in a list.

3. Use filter to keep numbers greater than 10.

4. Use reduce to find the product of all numbers in a list.

5. Write a program that uses with to read a file safely.

6. Combine map, filter, reduce to calculate the sum of squares of even numbers from a list.

I HOPE THIS HANDBOOK HELP YOU A LOT

DO PRACTICE MORE SO THAT YOU CAN CODE PYTHON WITHOUT HELP

BY ANIMESH SINGHA

3 YEARS Experience of more than seven programming languages

Visit: https://vortexuser123.github.io/infosite_animesh/ Know more

Email: singhaanimesh509@gmail.com

Phone: +91 9544342332 [SMS only]

AND Artificial Intelligence

[ChatGPT 5, Perplexity pro, Blackbox AI, HacknovaAI, and CybrBuddy]