📓 Python Notes – Part 1 (Basics, Memory, Data Types)

---

1. 💠 Introduction to Python

- **High-level Language** → Easy to read & write (close to human language).
- **Interpreted** → Executes line by line (no need of compilation).
- **Dynamically Typed** → No need to declare variable type, Python decides at runtime.
- **Cross-platform** → Works on Windows, Linux, macOS.
- **Object-Oriented + Functional** → Supports both paradigms.

👉 Created by **Guido van Rossum** in 1991.

---

2. 💠 Python Execution & Memory Model

How Python Code Runs

1. **Write Code (.py file)**
2. **Python Interpreter** converts code into **Bytecode** (.pyc).
3. **Python Virtual Machine (PVM)** executes bytecode.

👉 Example:

```
print("Hello Python")
```

- `print("Hello Python")` → Converted into bytecode → Executed by PVM.

Memory Management in Python

- **Automatic Memory Management** handled by **Python Memory Manager**.
- **Key components:**
  - **Stack Memory** → Stores function calls, local variables.
  - **Heap Memory** → Stores objects, dynamic memory.
  - **Garbage Collector** → Frees unused objects.

👉 Example of Garbage Collection:

```
import gc

x = [1, 2, 3]
```

```
y = x
del x
print(y)  # still accessible
del y
gc.collect()  # unused memory cleaned
```

- **Reference Counting** → Each object has a count of references. When count = 0 → destroyed.
- **Generational Garbage Collection** → Objects are divided into **young, old generations** → frequent short-lived objects cleared quickly.

---

3. 🔹 Variables in Python

- Variables are **references** (labels) to objects in memory.
- No need to declare type → decided at runtime.

```
x = 10        # integer
y = "Hello"   # string
z = [1, 2, 3] # list
```

👉 **id() function** → Returns memory address.

```
a = 10
b = 10
print(id(a), id(b))  # same address (integer caching for small numbers)
```

---

4. 🔹 Data Types in Python

Built-in Data Types

1. **Numeric Types** → `int`, `float`, `complex`

   ```
   a = 5         # int
   b = 3.14      # float
   c = 2 + 3j    # complex
   ```

2. **Sequence Types** → `list`, `tuple`, `range`

   ```
   lst = [1, 2, 3]        # list (mutable)
   tpl = (1, 2, 3)        # tuple (immutable)
   rng = range(5)         # 0 to 4
   ```

3. **Text Type** → `str`

   ```
   s = "Python"
   ```

4. **Set Types** → `set`, `frozenset`

```
s = {1, 2, 3}
fs = frozenset([1, 2, 3])  # immutable
```

5. **Mapping Type** → `dict`

```
d = {"name": "Prince", "age": 21}
```

6. **Boolean Type** → `bool`

```
is_valid = True
```

7. **Binary Types** → `bytes`, `bytearray`, `memoryview`

```
b = b"hello"          # immutable
ba = bytearray(b)     # mutable
mv = memoryview(b)    # memory efficient
```

Type Conversion

- **Implicit (Type Casting by Python)**

```
x = 10
y = 2.5
z = x + y   # 12.5 (int → float automatically)
```

- **Explicit (Manual Casting)**

```
a = int("10")      # 10
b = float("3.14") # 3.14
c = str(100)      # "100"
```

Important Notes about Python Data Types

- **Mutable vs Immutable**
  - Mutable → `list`, `dict`, `set`, `bytearray`
  - Immutable → `int`, `float`, `str`, `tuple`, `frozenset`, `bytes`
- **Everything in Python is an Object**
  Even numbers, functions, and classes are objects.

  👉 Example:

```
print(type(10))      # <class 'int'>
print(type("Hello")) # <class 'str'>
```

📓 Python Notes – Part 2 (Operators)

---

1. ❖ What are Operators?

Operators are **symbols** that perform operations on variables/values.
Example:

```
a = 10
b = 3
print(a + b)  # 13
```

---

2. ❖ Types of Operators in Python

(A) Arithmetic Operators

Used for basic math operations:

```
a = 10
b = 3

print(a + b)   # Addition → 13
print(a - b)   # Subtraction → 7
print(a * b)   # Multiplication → 30
print(a / b)   # Division (float) → 3.3333
print(a // b)  # Floor Division → 3
print(a % b)   # Modulus (remainder) → 1
print(a ** b)  # Exponentiation → 1000 (10^3)
```

(B) Relational / Comparison Operators

Used to compare values → returns **True/False**.

```
a = 5
b = 10

print(a == b)  # False
print(a != b)  # True
print(a > b)   # False
print(a < b)   # True
print(a >= b)  # False
print(a <= b)  # True
```

(C) Logical Operators

Used for **boolean logic**.

```
x = True
y = False

print(x and y) # False
print(x or y)  # True
print(not x)   # False
```

👉 Works with non-boolean values too:

```
print(0 and 5)   # 0  (because 0 is False)
print(5 and 0)   # 0
print(5 and 10)  # 10
print(0 or 5)    # 5
```

(D) Bitwise Operators

Work on binary representation of numbers.

```
a = 6   # (110 in binary)
b = 3   # (011 in binary)

print(a & b)   # AND → 2 (010)
print(a | b)   # OR  → 7 (111)
print(a ^ b)   # XOR → 5 (101)
print(~a)      # NOT → -7  (inverts bits)
print(a << 1)  # Left Shift  → 12 (1100)
print(a >> 1)  # Right Shift → 3 (011)
```

(E) Assignment Operators

Used to assign values.

```
x = 5
x += 3   # x = x + 3 → 8
x -= 2   # x = x - 2 → 6
x *= 2   # x = x * 2 → 12
x /= 3   # x = x / 3 → 4.0
x //= 2  # x = x // 2 → 2.0
x %= 2   # x = x % 2 → 0.0
x **= 3  # x = x ** 3 → 0.0
```

(F) Identity Operators

Used to compare **memory location** (whether two variables refer to the same object).

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

print(a is b)     # False (different objects in memory)
print(a is c)     # True  (same object reference)
print(a is not b) # True
```

(G) Membership Operators

Used to check membership in sequences (`list`, `tuple`, `str`, `set`, `dict`).

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

print(2 in lst)     # True
print(5 in lst)     # False
print(5 not in lst) # True
```

---

3. ❖ Operator Precedence (Priority Order)

When multiple operators are used, Python follows **precedence rules**.

Precedence Table (highest → lowest)

1. **Parentheses `()`**
2. **Exponent `**`**
3. **Unary +, - , ~**
4. **Multiplication `*`, Division `/`, Floor `//`, Modulus `%`**
5. **Addition `+`, Subtraction `-`**
6. **Bitwise Shift <<, >>**
7. **Bitwise AND `&`**
8. **Bitwise XOR `^`**
9. **Bitwise OR `|`**
10. **Comparison (==, <, >, etc.)**
11. **Logical NOT `not`**
12. **Logical AND `and`**
13. **Logical OR `or`**
14. **Assignment (=, +=, etc.)**

☞ Example:

```
expr = 2 + 3 * 4 ** 2
# 4 ** 2 = 16
# 3 * 16 = 48
# 2 + 48 = 50
print(expr)  # 50
```

---

4. ❖ Special Notes

- **Chained Comparison**:

```
x = 10
print(5 < x < 15)   # True (equivalent to 5 < x and x < 15)
```

  - **is vs ==**:
    - `==` → Compares **values**
    - `is` → Compares **identity (memory address)**
  - `a = 1000`
  - `b = 1000`
  - `print(a == b)  # True`

```
o  print(a is b)  # False (different memory for large integers)
```

📓 Python Notes – Part 3 (Control Flow)

---

1. 🔹 What is Control Flow?

Control flow decides **how code is executed**:

- **Decision Making** → `if`, `elif`, `else`
- **Loops** → `for`, `while`
- **Loop Control** → `break`, `continue`, `pass`, `else` with loop

---

2. 🔹 Conditional Statements

(A) `if` Statement

Executes block only if condition is **True**.

```
x = 10
if x > 5:
    print("x is greater than 5")
```

(B) `if-else`

Executes one block if condition true, otherwise another.

```
x = 3
if x % 2 == 0:
    print("Even")
else:
    print("Odd")
```

(C) `if-elif-else`

Used for multiple conditions.

```
marks = 85

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

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

(D) Nested if

`if` inside another `if`.

```
x = 15
if x > 10:
    if x < 20:
        print("x is between 10 and 20")
```

(E) Short-hand `if`

One-line `if`.

```
x = 5
if x > 0: print("Positive")
```

Shorthand with `else`:

```
x = 10
print("Even") if x % 2 == 0 else print("Odd")
```

---

3. � Loops in Python

(A) `for` Loop

Used to iterate over **sequence (list, tuple, str, range, dict, set)**.

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

Iterating over list:

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

(B) `while` Loop

Runs until condition becomes **False**.

```
x = 1
while x <= 5:
    print(x)
```

```
    x += 1
```

(C) Loop with `else`

Rare feature in Python → `else` runs if loop ends **normally** (not by `break`).

```
for i in range(5):
    print(i)
else:
    print("Loop finished")
x = 0
while x < 3:
    print(x)
    x += 1
else:
    print("While loop finished")
```

4. � Loop Control Statements

(A) `break`

Terminates the loop immediately.

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

Output → `0 1 2 3 4`

(B) `continue`

Skips current iteration and goes to next.

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

Output → `0 1 3 4`

(C) `pass`

Placeholder (does nothing).

```
for i in range(3):
```

```
    pass    # used when code is not written yet
```

---

5. ◈ Nested Loops

Loop inside loop.

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

---

6. ◈ Useful Functions with Loops

- `range(start, stop, step)`

  ```
  for i in range(2, 10, 2):
      print(i)
  # Output: 2 4 6 8
  ```

- `enumerate()` → get index with value

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

- `zip()` → iterate two lists together

  ```
  names = ["Prince", "Aman", "Ravi"]
  scores = [90, 85, 88]

  for name, score in zip(names, scores):
      print(name, score)
  ```

---

7. ◈ Special Notes for Interviews

1. `while True` → infinite loop until `break`.
2. `for-else` and `while-else` → unique to Python.
3. `break` skips the `else` block, but `continue` does not.
4. Python **does not have switch-case** (but can use `dict` for alternatives).

👉 Example of switch alternative:

```
def switch_example(choice):
    return {
        1: "One",
        2: "Two",
        3: "Three"
    }.get(choice, "Invalid")

print(switch_example(2))    # Two
```

✅ Now you have strong notes on **Control Flow**:

- `if`, `elif`, `else`, nested if, shorthand if
- `for` & `while` loops with `else`
- `break`, `continue`, `pass`
- Loop helpers (`range`, `enumerate`, `zip`)

📘 Python Notes – Part 4 (Functions + Collections: List, Tuple, Set, Dict)

---

1. 🔹 Functions in Python

What is a Function?

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

Syntax:

```
def function_name(parameters):
    """docstring (optional)"""
    # body
    return value
```

(A) Defining & Calling Functions

```
def greet(name):
    return f"Hello, {name}!"

print(greet("Prince"))  # Hello, Prince!
```

(B) Types of Function Arguments

1. **Positional Arguments**

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

   print(add(2, 3))  # 5
   ```

2. **Default Arguments**

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

print(greet())         # Hello, Guest
print(greet("Aman"))   # Hello, Aman
```

3. **Keyword Arguments**

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

student(age=20, name="Prince")
```

4. **Variable-length Arguments**

- `*args` → multiple positional arguments (tuple)
- `**kwargs` → multiple keyword arguments (dict)

```
def add(*args):
    return sum(args)

print(add(1, 2, 3, 4))  # 10

def details(**kwargs):
    print(kwargs)

details(name="Prince", age=21)
# {'name': 'Prince', 'age': 21}
```

(C) Return Statement

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

(D) Anonymous Functions (Lambda)

One-liner functions.

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

(E) Scope of Variables

- **Local** → inside function
- **Global** → outside function
- Use `global` keyword to modify global variable.

```
x = 10
def func():
    global x
    x = 20
```

```
    func()
    print(x)  # 20
```

(F) Recursion

Function calling itself.

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

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

---

2. ◈ Collections in Python (Data Structures)

(A) List

- **Ordered, Mutable, Allows duplicates**

  ```
  fruits = ["apple", "banana", "cherry"]
  print(fruits[0])       # apple
  fruits[1] = "mango"    # modification allowed
  print(fruits)          # ['apple', 'mango', 'cherry']
  ```

  List Methods:

  ```
  lst = [1, 2, 3]
  lst.append(4)        # [1, 2, 3, 4]
  lst.insert(1, 5)     # [1, 5, 2, 3, 4]
  lst.remove(2)        # remove value
  lst.pop()            # remove last element
  lst.sort()           # sort ascending
  lst.reverse()        # reverse list
  lst.extend([7, 8])   # add multiple
  ```

  List Comprehension:

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

(B) Tuple

- **Ordered, Immutable, Allows duplicates**

  ```
  tpl = (1, 2, 3, 2)
  print(tpl[0])      # 1
  ```

```
print(tpl.count(2))# 2
```

👉 Used when data should not be changed.
👉 Tuples are faster than lists.

(C) Set

- **Unordered, Mutable, No duplicates**

```
s = {1, 2, 3, 3}
print(s)  # {1, 2, 3}
```

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}
```

👉 Useful for mathematical operations and removing duplicates.

(D) Dictionary

- **Key-Value Pairs, Mutable, Keys unique**

```
student = {"name": "Prince", "age": 21}
print(student["name"])    # Prince
student["age"] = 22
print(student)
```

Dictionary Methods:

```
d = {"a": 1, "b": 2}
print(d.keys())       # dict_keys(['a','b'])
print(d.values())     # dict_values([1,2])
print(d.items())      # dict_items([('a',1),('b',2)])
print(d.get("a"))     # 1
d.update({"c": 3})
d.pop("b")            # removes key
```

👉 Nested Dictionary:

```
students = {
    1: {"name": "Prince", "age": 21},
    2: {"name": "Aman", "age": 20}
}
print(students[1]["name"])  # Prince
```

(E) Other Useful Types

1. **frozenset** → immutable set

   ```
   fs = frozenset([1, 2, 3])
   ```

2. **range** → sequence generator

   ```
   for i in range(3, 10, 2):
       print(i)  # 3, 5, 7, 9
   ```

3. **collections module** → `deque`, `defaultdict`, `Counter`, `OrderedDict`, `namedtuple`

---

3. 🔹 Choosing Between List, Tuple, Set, Dict

- **List** → ordered collection when duplicates allowed & frequent updates.
- **Tuple** → ordered & immutable (safe for constants, faster lookups).
- **Set** → unique elements, fast membership test, math operations.
- **Dict** → key-value mapping (like JSON).

✅ Now you have detailed notes on:

- Functions (arguments, return, lambda, recursion, scope)
- Collections: List, Tuple, Set, Dict (+ methods & use cases)

📘 Python Notes – Part 5 (Strings)

---

1. 🔹 What is a String?

- A **sequence of characters** enclosed in single `' '`, double `" "`, or triple quotes `''' '''` / `""" """`.
- Strings are **immutable** → once created, cannot be changed.

```
s1 = 'hello'
s2 = "world"
s3 = '''multi
line'''
```

---

2. ◈ Accessing Characters

- Indexing (0-based)
- Negative Index → from end

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

---

3. ◈ String Slicing

```
s = "Python"
print(s[0:4])   # Pyth
print(s[:4])    # Pyth
print(s[2:])    # thon
print(s[::-1])  # nohtyP (reverse)
```

☞ Syntax: `s[start:end:step]`

- `start` → index to begin (default 0)
- `end` → index to stop (exclusive)
- `step` → jump (default 1)

---

4. ◈ String Operations

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

print(a + b)        # Concatenation → HelloWorld
print(a * 3)        # Repetition → HelloHelloHello
print("H" in a)     # True
print("z" not in a)# True
```

---

5. ◈ String Methods

Case Conversion

```
s = "python"
print(s.upper())      # PYTHON
print(s.lower())      # python
print(s.title())      # Python
print(s.capitalize())# Python
print(s.swapcase())   # PYTHON → python, python → PYTHON
```

Search & Replace

```
s = "hello world"
print(s.find("world"))   # 6
print(s.index("world"))  # 6 (error if not found)
print(s.replace("world","Python")) # hello Python
```

Checking

```
s = "Python123"
print(s.isalpha())   # False
print(s.isdigit())   # False
print(s.isalnum())   # True
print(" ".isspace()) # True
```

Strip (remove spaces/characters)

```
s = "  hello  "
print(s.strip())   # hello
print(s.lstrip())  # "hello  "
print(s.rstrip())  # "  hello"
```

Splitting & Joining

```
s = "apple,banana,cherry"
lst = s.split(",")        # ['apple','banana','cherry']
print("-".join(lst))      # apple-banana-cherry
```

Alignment

```
s = "Python"
print(s.center(10, "*"))  # **Python**
print(s.ljust(10, "-"))   # Python----
print(s.rjust(10, "-"))   # ----Python
```

---

6. ◈ String Formatting

Old Style (% operator)

```
name = "Prince"
age = 21
print("My name is %s and I am %d years old" % (name, age))
```

`str.format()`

```
print("My name is {} and I am {}".format(name, age))
print("My name is {1} and I am {0}".format(age, name))
```

f-Strings (Python 3.6+)

```
print(f"My name is {name} and I am {age}")
print(f"Next year I will be {age+1}")
```

---

7. 🔹 Raw Strings

Prefix `r` → ignores escape sequences.

```
print(r"Line1\nLine2")  # prints \n literally
```

---

8. 🔹 Escape Characters

```
print("Hello\nWorld")   # new line
print("Hello\tWorld")   # tab
print("He said \"Hi\"") # quotes
```

---

9. 🔹 Unicode & Encoding

Python strings are **Unicode** by default.

```
s = "हिंदी"
print(s.encode("utf-8"))  # b'\xe0\xa4\xb9\xe0\xa4\xbf...'
```

---

10. 🔹 Advanced: Regular Expressions (regex)

Regex is used for **pattern matching** (via `re` module).

```
import re

s = "My phone is 9876543210"
pattern = r"\d{10}"   # match 10 digits
match = re.search(pattern, s)

if match:
    print("Found:", match.group())  # Found: 9876543210
```

Common Regex Patterns:

- `\d` → digit
- `\w` → word char (a-z, A-Z, 0-9, _)
- `\s` → whitespace
- `.` → any char
- `^` → start, `$` → end
- `+` → one or more, `*` → zero or more, `?` → optional

---

11. 🔹 String Interning (Optimization)

Python optimizes memory by reusing some strings.

```
a = "hello"
b = "hello"
print(a is b)   # True (same memory for literals)
```

✅ Now you have complete notes on **Strings**:

- Creation, slicing, operations
- Methods (search, replace, strip, split, join, align)
- Formatting (`%`, `.format()`, f-strings)
- Escape sequences & raw strings
- Regex basics (interview favorite)
- String immutability & interning

📘 Python Notes – Part 6 (Modules & Packages)

---

1. 🔹 What is a Module?

- A **module** is simply a Python file (`.py`) containing functions, classes, or variables.
- It allows **code reusability** and better organization.

👉 Example: `math.py`, `os.py`, or your own file.

---

2. 🔹 Importing a Module

Basic Import

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

Import Specific Function

```
from math import sqrt, pi
print(sqrt(25))  # 5.0
print(pi)        # 3.14159...
```

Import with Alias

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

Import All (not recommended in large projects)

```
from math import *
print(sin(90))   # works without math.
```

---

3. 🔹 Common Built-in Modules

`math`

```
import math
print(math.ceil(2.3))   # 3
print(math.floor(2.9))  # 2
print(math.gcd(12, 15)) # 3
```

`random`

```
import random
print(random.randint(1, 10))     # random int
print(random.choice([1, 2, 3]))  # random element
```

`datetime`

```
import datetime
now = datetime.datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

`os`

```
import os
print(os.getcwd())      # current directory
os.mkdir("test")        # create folder
```

`sys`

```
import sys
print(sys.version)      # Python version
print(sys.path)         # search paths
```

`statistics`

```
import statistics
data = [10, 20, 30, 40]
print(statistics.mean(data))   # 25
print(statistics.median(data)) # 25
```

---

4. 🔹 Creating a Custom Module

Step 1: Create `mymodule.py`

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

def greet(name):
    return f"Hello, {name}"
```

Step 2: Use it in another file

```
import mymodule

print(mymodule.add(5, 3))      # 8
print(mymodule.greet("Prince")) # Hello, Prince
```

---

5. ◈ What is a Package?

- A **package** is a **collection of modules** inside a folder.
- Must contain an `__init__.py` file (can be empty).

☞ Example Structure:

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

Using a Package:

```
from mypackage import module1
from mypackage.module2 import func
```

---

6. ◈ Installing External Packages with `pip`

Python has **PyPI (Python Package Index)** → huge collection of libraries.

- Install package:

  ```
  pip install requests
  ```

- Use package:

  ```
  import requests
  res = requests.get("https://api.github.com")
  print(res.status_code)  # 200
  ```

- Upgrade package:

  ```
  pip install --upgrade requests
  ```

- List installed packages:

```
pip list
```

---

7. 💠 Virtual Environments

Used to isolate project dependencies.

```
# Create virtual environment
python -m venv myenv

# Activate (Windows)
myenv\Scripts\activate

# Activate (Linux/Mac)
source myenv/bin/activate

# Deactivate
deactivate
```

---

8. 💠 Standard vs Third-Party vs Custom

- **Standard Modules** → come with Python (`os`, `math`, `sys`)
- **Third-Party Modules** → install using `pip` (`numpy`, `pandas`, `flask`)
- **Custom Modules** → your own `.py` files

---

9. 💠 Interview Notes

1. **Difference between module & package**
    - Module = single `.py` file
    - Package = folder with `__init__.py` containing multiple modules
2. **Why use virtual environment?**
    - To avoid dependency conflicts between projects.
3. **`__name__ == "__main__"` usage**
    - Used to check if a file is run directly or imported as a module.

```
# mymodule.py
def hello():
    print("Hello!")

if __name__ == "__main__":
    hello()
```

✅ Now you have strong notes on **Modules & Packages**:

- Importing modules (ways)
- Built-in modules (`math`, `os`, `sys`, `datetime`, etc.)
- Creating custom modules & packages

- `pip` and virtual environments
- `__name__ == "__main__"`

📓 Python Notes – Part 7 (File Handling)

---

1. 🔹 What is File Handling?

- File handling allows you to **create, read, write, update, and delete** files.
- Python provides built-in functions for file handling using the **`open()`** function.

---

2. 🔹 Opening a File

```
file = open("example.txt", "r")
```

Modes in `open()`

Mode	Meaning
`"r"`	Read (default), error if file doesn't exist
`"w"`	Write (overwrite, creates file if not exist)
`"a"`	Append (adds data at end, creates file if not exist)
`"x"`	Create (error if file exists)
`"b"`	Binary mode (e.g., images, pdfs)
`"t"`	Text mode (default)

👉 Example:

```
f = open("data.txt", "w")   # write mode
f.write("Hello, Prince!")
f.close()
```

---

3. 🔹 Reading from a File

```
f = open("data.txt", "r")
print(f.read())         # read entire file
print(f.readline())     # read one line
print(f.readlines())    # read all lines as list
f.close()
```

---

4. ❖ Writing to a File

```
f = open("data.txt", "w")
f.write("This will overwrite the file.\n")
f.write("Second line.\n")
f.close()
```

---

5. ❖ Appending to a File

```
f = open("data.txt", "a")
f.write("This will add at the end.\n")
f.close()
```

---

6. ❖ Using `with` Statement (Best Practice)

- Automatically closes the file.

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

---

7. ❖ File Pointer

- After reading, the file pointer moves forward.
- Use `f.seek(0)` to move back to the beginning.

```
f = open("data.txt", "r")
print(f.read(5))   # first 5 chars
f.seek(0)
print(f.read())    # full content
f.close()
```

---

8. ❖ Working with JSON

- JSON (JavaScript Object Notation) is used for APIs & configs.

```
import json

# Python dict → JSON
data = {"name": "Prince", "age": 21}
json_str = json.dumps(data)
print(json_str)  # '{"name": "Prince", "age": 21}'

# JSON → Python dict
parsed = json.loads(json_str)
print(parsed["name"])  # Prince
```

```
# Write JSON to file
with open("data.json", "w") as f:
    json.dump(data, f)

# Read JSON from file
with open("data.json", "r") as f:
    d = json.load(f)
    print(d)
```

---

9. 🔹 Working with CSV

- CSV (Comma Separated Values) → used in data handling.

```
import csv

# Writing CSV
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Prince", 21])
    writer.writerow(["Rahul", 22])

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

---

10. 🔹 Deleting & Checking Files

```
import os

# Delete file
if os.path.exists("data.txt"):
    os.remove("data.txt")
else:
    print("File does not exist.")

# Remove folder
os.rmdir("testfolder")
```

---

11. 🔹 File Handling Errors

Always handle exceptions:

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

---

12. 🔹 Interview Notes

1. **Difference between `w` and `a` mode?**
   - `w`: overwrite file, `a`: append at the end.
2. **Why use `with open()`?**
   - Auto closes file (context manager).
3. **Difference between `read()`, `readline()`, `readlines()`?**
   - `read()`: whole content
   - `readline()`: single line
   - `readlines()`: list of all lines
4. **What is `seek()` and `tell()`?**
   - `seek(pos)`: move file pointer
   - `tell()`: current position

✅ Now you know everything about **File Handling in Python**:

- Opening, reading, writing, appending
- `with` statement
- JSON & CSV handling
- File deletion & exception handling

📘 Python Notes – Part 8 (Object-Oriented Programming – OOP)

---

1. 🔹 What is OOP?

- **Object-Oriented Programming (OOP)** is a programming paradigm based on **objects** and **classes**.
- Helps in **code reusability, modularity, maintainability**.

👉 Key principles: **Encapsulation, Inheritance, Polymorphism, Abstraction**

---

2. 🔹 Class and Object

Defining a Class

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

    def show(self):
        print(f"Car: {self.brand}, Model: {self.model}")
```

Creating an Object

```
c1 = Car("Tesla", "Model X")
c1.show()   # Car: Tesla, Model: Model X
```

---

3. ◆ `__init__` Constructor

- Special method called automatically when object is created.
- Used for **initializing attributes**.

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

---

4. ◆ Instance vs Class Variables

```
class Dog:
    species = "Mammal"   # class variable

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

d1 = Dog("Tommy")
print(d1.species)  # Mammal
print(d1.name)     # Tommy
```

---

5. ◆ Methods in Class

- **Instance Methods** → Work with objects (`self`)
- **Class Methods** → Work with class (`@classmethod`, `cls`)
- **Static Methods** → Utility functions (`@staticmethod`)

```
class Math:
    def add(self, a, b):   # instance method
        return a + b

    @classmethod
    def info(cls):         # class method
        return "This is Math class."

    @staticmethod
```

```
    def pi():
        return 3.14

m = Math()
print(m.add(2, 3))      # 5
print(Math.info())      # class method
print(Math.pi())        # 3.14
```

---

6. ❖ Encapsulation

- Wrapping data & methods together.
- **Access Modifiers**:
  - Public (`self.name`) → default
  - Protected (`self._name`) → convention (use carefully)
  - Private (`self.__name`) → not directly accessible

```
class Person:
    def __init__(self, name, age):
        self._name = name       # protected
        self.__age = age        # private

    def get_age(self):
        return self.__age

p = Person("Prince", 21)
print(p._name)          # accessible but not recommended
print(p.get_age())      # use getter
```

---

7. ❖ Inheritance

- Allows a class to **inherit** methods & attributes from another class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

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

d = Dog()
d.speak()    # Animal speaks
d.bark()     # Dog barks
```

☞ Types of Inheritance:

- Single (one parent → one child)
- Multiple (child inherits from multiple parents)
- Multilevel (grandparent → parent → child)
- Hierarchical (one parent → multiple children)

8. ◈ Method Overriding

```
class Animal:
    def sound(self):
        print("Animal sound")

class Dog(Animal):
    def sound(self):   # overriding
        print("Bark")

d = Dog()
d.sound()   # Bark
```

9. ◈ Polymorphism

- **Poly = many, Morph = forms**
- Same function name, different behavior.

```
class Cat:
    def sound(self):
        print("Meow")

class Dog:
    def sound(self):
        print("Bark")

for animal in (Cat(), Dog()):
    animal.sound()
```

10. ◈ Abstraction

- Hiding implementation details, showing only essentials.
- Achieved using **Abstract Base Class (ABC)**.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r

c = Circle(5)
print(c.area())  # 78.5
```

11. ◈ Special (Magic/Dunder) Methods

- Methods starting & ending with __.

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

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

b = Book("Python Basics")
print(b)   # Book: Python Basics
```

👉 Common dunder methods:

- `__init__` → constructor
- `__str__` → string representation
- `__len__` → length of object
- `__add__` → overloading + operator

---

12. ◈ Operator Overloading

```
class Number:
    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        return self.val + other.val

n1 = Number(5)
n2 = Number(10)
print(n1 + n2)   # 15
```

---

13. ◈ Interview Notes

1. **Four pillars of OOP?**
    - Encapsulation, Inheritance, Polymorphism, Abstraction
2. **Difference between `@classmethod` and `@staticmethod`?**
    - `classmethod`: works with class (`cls`)
    - `staticmethod`: works independently
3. **Access specifiers in Python?**
    - Public, Protected (`_var`), Private (`__var`)
4. **Does Python support multiple inheritance?**
    - Yes (Method Resolution Order - MRO used).
5. **What is `super()` keyword?**
    - Used to call parent class methods/constructors.

✅ Now you know **OOP in Python**:

- Classes, objects, constructor
- Instance, class, static methods
- Encapsulation, inheritance, polymorphism, abstraction
- Dunder methods & operator overloading
- Interview-ready explanations

📘 Python Notes – Part 9 (Error & Exception Handling)

---

1. 🔹 What is an Exception?

- An **exception** is an error that occurs **during program execution** and interrupts the normal flow.
- Example: dividing by zero, accessing invalid index, opening a missing file.

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

---

2. 🔹 Errors vs Exceptions

- **Errors** → problems in the program itself (e.g., syntax error).
- **Exceptions** → runtime problems that can be handled.

```
# Error (syntax issue)
if True
    print("Hello")    # ❌ SyntaxError

# Exception (runtime issue)
print(10 / 0)         # ❌ ZeroDivisionError
```

---

3. 🔹 Handling Exceptions with `try-except`

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

👉 Multiple Except:

```
try:
    num = int("abc")
except ValueError:
    print("Invalid number")
except ZeroDivisionError:
    print("Divide by zero error")
```

👉 Catch Any Exception:

```
try:
    print(10 / 0)
except Exception as e:
    print("Error:", e)
```

---

4. 🔷 `else` and `finally` with Try

- `else`: runs if no exception occurs
- `finally`: always runs (cleanup, closing files, etc.)

```
try:
    num = 10 / 2
except ZeroDivisionError:
    print("Divide by zero")
else:
    print("Success:", num)
finally:
    print("Always executed")
```

---

5. 🔷 Raising Exceptions

You can raise exceptions manually using `raise`.

```
def withdraw(amount):
    if amount > 1000:
        raise ValueError("Insufficient balance")
    else:
        print("Withdrawal successful")

withdraw(1200)   # Raises ValueError
```

---

6. 🔷 Custom Exceptions

Create your own exception class by inheriting from `Exception`.

```
class MyError(Exception):
    pass

try:
    raise MyError("Something went wrong!")
except MyError as e:
```

```
        print("Caught:", e)
```

---

7. 🔹 Common Built-in Exceptions

- `ValueError` → wrong value type
- `TypeError` → wrong type
- `IndexError` → list index out of range
- `KeyError` → dict key missing
- `ZeroDivisionError` → divide by zero
- `FileNotFoundError` → missing file
- `ImportError` → module not found
- `AttributeError` → attribute not found

---

8. 🔹 Nested Try-Except

```
try:
    f = open("data.txt")
    try:
        content = f.read()
        print(content)
    except:
        print("Error while reading file")
    finally:
        f.close()
except FileNotFoundError:
    print("File not found")
```

---

9. 🔹 Best Practices

1. Catch specific exceptions instead of `Exception`.
2. Use `finally` for cleanup (closing files, DB connections).
3. Don't swallow exceptions silently (always log or print).
4. Use **custom exceptions** for application-specific errors.
5. Group exceptions if multiple are handled same way:

```
try:
    x = int("abc")
except (ValueError, TypeError) as e:
    print("Error:", e)
```

---

10. 🔹 Interview Notes

1. **Difference between error and exception?**
    - Error = syntax/logic issue; Exception = runtime issue.
2. **What is `finally` block used for?**

        - Executes always, even if error occurs (cleanup).
3. **Can Python have multiple `except`?**
    - Yes, to handle different exceptions separately.
4. **Why use custom exceptions?**
    - To make errors meaningful in large applications.
5. **What happens if exception not handled?**
    - Program crashes and Python prints traceback.

✅ Now you know **Error & Exception Handling in Python**:

- `try-except-else-finally`
- Raising & creating custom exceptions
- Common exceptions (`ValueError`, `KeyError`, etc.)
- Best practices & interview questions

📘 Python Notes – Part 10 (Iterators, Generators & Decorators)

---

1. 🔹 Iterators in Python

An **iterator** is an object that allows iteration (looping) over elements one at a time.

👉 Behind the scenes, Python uses `iter()` and `next()`.

Example

```
nums = [1, 2, 3]
it = iter(nums)    # get iterator
print(next(it))    # 1
print(next(it))    # 2
print(next(it))    # 3
# next(it) -> StopIteration
```

Custom Iterator

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self
```

```
    def __next__(self):
        if self.current <= self.end:
            val = self.current
            self.current += 1
            return val
        else:
            raise StopIteration

for num in Counter(1, 5):
    print(num)  # 1 2 3 4 5
```

---

2. 💠 Generators

A **generator** is a simpler way to create iterators using `yield` keyword.

- Saves memory (lazy evaluation → generates values one by one).
- Used for large datasets, streams, APIs.

Example

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

g = my_gen()
print(next(g))  # 1
print(next(g))  # 2
```

👉 Looping:

```
for val in my_gen():
    print(val)
```

---

3. 💠 Generator Expressions

Like list comprehensions but use `()` instead of `[]`.

```
gen = (x*x for x in range(5))
print(next(gen))  # 0
print(next(gen))  # 1
print(list(gen))  # [4, 9, 16]
```

---

4. 💠 Difference: Iterator vs Generator

Feature	Iterator	Generator
Implementation	Class with `__iter__`, `__next__`	Function with `yield`
Memory	Can use more memory	Memory efficient (lazy eval)

Feature	Iterator	Generator
Ease of use	More code	Simple to write

---

5. 🧩 Decorators

A **decorator** is a function that modifies the behavior of another function without changing its code.

👉 Uses **`@decorator_name`** above a function.

Basic Example

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

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

say_hello()
```

📌 Output:

```
Before function
Hello, Prince!
After function
```

---

6. 🧩 Decorators with Arguments

```
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("Hello", name)

greet("Prince")
```

---

7. 🧩 Function Decorators in Practice

- **Logging**
- **Authentication**
- **Timing Execution**

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} executed in {end-start:.5f} sec")
        return result
    return wrapper

@timer
def calc_square(n):
    return [i*i for i in range(n)]

calc_square(100000)
```

---

8. 🔹 Built-in Decorators

- `@staticmethod`
- `@classmethod`
- `@property`

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name    # getter

    @name.setter
    def name(self, value):
        self._name = value   # setter

p = Person("Prince")
print(p.name)    # Prince
p.name = "Rahul"
print(p.name)    # Rahul
```

---

9. 🔹 Interview Notes

1. **Difference between iterator and generator?**
   - Iterator → uses `__iter__` & `__next__`.
   - Generator → uses `yield`, memory efficient.
2. **Why use generators?**
   - For large data streams → saves memory.
3. **What is a decorator?**

   - A function that modifies another function's behavior.
4. **Can decorators take arguments?**
   - Yes, using nested functions.
5. **Real-life use cases of decorators?**
   - Logging, authentication, caching, execution time measurement.

✅ Now you know:

- Iterators (`iter`, `next`, custom)
- Generators (`yield`, generator expressions)
- Decorators (basic, with args, built-in, real-world uses)

📘 Python Notes – Part 11 (Multithreading, Multiprocessing & Async)

---

1. 🔹 Why Concurrency & Parallelism?

- **Concurrency** → Multiple tasks progress **together** (not necessarily at same time).
- **Parallelism** → Multiple tasks run **at the same time** (on multiple cores).

👉 Python provides:

- **Threading** → lightweight, good for I/O tasks
- **Multiprocessing** → real parallelism, good for CPU-heavy tasks
- **Asyncio** → async programming for scalable I/O

---

2. 🔹 Multithreading in Python

Python's `threading` module allows running multiple threads.

Example

```
import threading
import time

def print_numbers():
    for i in range(5):
        print("Number:", i)
        time.sleep(1)
```

```
def print_letters():
    for ch in "ABCDE":
        print("Letter:", ch)
        time.sleep(1)

t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

t1.start()
t2.start()

t1.join()
t2.join()
print("Done")
```

📌 Output interleaves numbers & letters.

🔹 GIL (Global Interpreter Lock)

- Python threads **don't run in parallel on multiple cores** due to GIL.
- Good for **I/O-bound tasks** (network requests, file read/write).
- Not efficient for **CPU-heavy tasks** (math, image processing).

---

3. 🔹 Multiprocessing in Python

The `multiprocessing` module bypasses GIL and uses multiple processes.

Example

```
import multiprocessing
import time

def square(n):
    print(f"Square of {n}: {n*n}")

if __name__ == "__main__":
    nums = [2, 3, 4, 5]
    processes = []
    for n in nums:
        p = multiprocessing.Process(target=square, args=(n,))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()
```

📌 Each process runs **truly in parallel**.

🔹 Multiprocessing Pool

```
from multiprocessing import Pool

def cube(n):
    return n**3

if __name__ == "__main__":
    with Pool(4) as p:
        result = p.map(cube, [1, 2, 3, 4, 5])
    print(result)  # [1, 8, 27, 64, 125]
```

---

4. 🔹 AsyncIO in Python

For **asynchronous programming** (non-blocking, scalable).
Best for **network requests, APIs, real-time applications**.

Example

```
import asyncio

async def task(name, delay):
    await asyncio.sleep(delay)
    print(f"Task {name} completed")

async def main():
    await asyncio.gather(
        task("A", 2),
        task("B", 1),
        task("C", 3)
    )

asyncio.run(main())
```

📌 Runs tasks concurrently → output order depends on delay.

---

5. 🔹 Comparing Approaches

Feature	Threading	Multiprocessing	AsyncIO
Best for	I/O-bound	CPU-bound	I/O-bound (scalable)
Parallelism	Limited (GIL)	Yes	Cooperative
Complexity	Easy	Moderate	Requires async/await
Memory	Shared	Separate memory	Shared

---

6. 🔹 Real-World Use Cases

- **Threading** → Downloading multiple files, handling requests, GUI apps
- **Multiprocessing** → Image processing, ML model training, simulations
- **AsyncIO** → Web scraping, chat servers, API calls

---

7. 🔹 Interview Notes

1. **What is GIL?**
   - Global Interpreter Lock → only one thread runs at a time in Python interpreter.
   - Affects **CPU-bound threads**, but **not I/O-bound tasks**.
2. **When to use threading vs multiprocessing?**
   - Threading → I/O-bound (web requests, file I/O)
   - Multiprocessing → CPU-bound (math, ML, image processing)
3. **Difference between concurrency and parallelism?**
   - Concurrency = tasks handled together (time-sharing).
   - Parallelism = tasks run simultaneously (multi-core).
4. **What is asyncio?**
   - Async framework in Python for non-blocking tasks using `async/await`.

✅ Now you know:

- Multithreading (I/O-bound tasks)
- Multiprocessing (CPU-bound tasks, Pool)
- AsyncIO (`async/await`)
- GIL & Interview insights

📘 Python Notes – Part 12 (Important Built-in Modules: `collections`, `itertools`, `functools`)

---

1. 🔹 `collections` Module

Provides **specialized data structures** beyond basic lists, sets, and dicts.

📌 Common Classes

1. **Counter** → Counts frequency of elements.

```
from collections import Counter
```

```
nums = [1, 2, 2, 3, 3, 3]
count = Counter(nums)
print(count)        # Counter({3: 3, 2: 2, 1: 1})
print(count.most_common(1))  # [(3, 3)]
```

---

2. **defaultdict** → Provides default values for missing keys.

```
from collections import defaultdict

dd = defaultdict(int)
dd["a"] += 1
dd["b"] += 2
print(dd)  # defaultdict(<class 'int'>, {'a': 1, 'b': 2})
```

---

3. **deque** → Fast append/pop from both ends.

```
from collections import deque

d = deque([1, 2, 3])
d.appendleft(0)
d.append(4)
print(d)  # deque([0, 1, 2, 3, 4])
```

---

4. **namedtuple** → Tuple with named fields.

```
from collections import namedtuple

Point = namedtuple("Point", "x y")
p = Point(10, 20)
print(p.x, p.y)  # 10 20
```

---

5. **OrderedDict** → Maintains insertion order (Python 3.7+ dict already does this).

```
from collections import OrderedDict

od = OrderedDict()
od["a"] = 1
od["b"] = 2
print(od)  # OrderedDict([('a', 1), ('b', 2)])
```

---

2. 🔶 `itertools` Module

Helps with **iterators & combinatorics**.

📌 Common Functions

1. **count()** → Infinite counter

```
from itertools import count

for i in count(5, 2):  # start=5, step=2
    if i > 15: break
    print(i, end=" ")  # 5 7 9 11 13 15
```

---

2. **cycle()** → Cycles through values infinitely

```
from itertools import cycle

c = 0
for item in cycle("AB"):
    print(item, end=" ")
    c += 1
    if c == 6: break   # A B A B A B
```

---

3. **repeat()**

```
from itertools import repeat

for x in repeat(5, 3):
    print(x)   # 5 5 5
```

---

4. **combinations() & permutations()**

```
from itertools import combinations, permutations

items = [1, 2, 3]
print(list(combinations(items, 2)))   # [(1, 2), (1, 3), (2, 3)]
print(list(permutations(items, 2)))   # [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

---

5. **product()**

```
from itertools import product

print(list(product([1, 2], ['a', 'b'])))
# [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

---

6. **accumulate()**

```
from itertools import accumulate

nums = [1, 2, 3, 4]
print(list(accumulate(nums)))   # [1, 3, 6, 10]
```

---

3. `functools` Module

Provides **higher-order functions** (functions that work with other functions).

📌 Common Functions

1. **reduce()** → Reduces iterable to a single value.

```
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x*y, nums)
print(product)  # 24
```

---

2. **lru_cache()** → Memoization for optimization.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n < 2: return n
    return fib(n-1) + fib(n-2)

print(fib(30))  # Faster due to caching
```

---

3. **partial()** → Fixes certain arguments of a function.

```
from functools import partial

def power(base, exp):
    return base ** exp

square = partial(power, exp=2)
print(square(5))  # 25
```

---

4. **cmp_to_key()** → Converts old-style compare functions.

```
from functools import cmp_to_key

def compare(a, b):
    return a - b

nums = [5, 2, 9]
print(sorted(nums, key=cmp_to_key(compare)))  # [2, 5, 9]
```

---

4. 🔹 Interview Notes

- `collections.Counter` is best for frequency counting.
- `itertools` helps in **permutation/combination problems**.
- `functools.lru_cache` is asked often in **DP optimization**.
- Know when to use `deque` vs `list` (deque is O(1) at both ends).

✅ Now you know the **powerful standard libraries** (`collections`, `itertools`, `functools`) that appear in **coding interviews and CP problems**.

📓 Python Notes – Part 13 (File Handling: Text, Binary, CSV, JSON)

---

1. 🔹 Why File Handling?

- Programs often need to **store data permanently** (not just in RAM).
- File handling allows **reading & writing** data from/to files.
- Python supports **Text, Binary, CSV, JSON** file handling.

---

2. 🔹 Basic File Operations

Python provides a built-in `open()` function.

Syntax

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

Common Modes

Mode	Description
`'r'`	Read (default, file must exist)
`'w'`	Write (creates new or overwrites existing file)
`'a'`	Append (adds at end of file)
`'b'`	Binary mode (e.g., images, videos)
`'+'`	Read & write

👉 Always close the file using `f.close()` or use `with open(...) as f`.

---

3. 🔹 Working with Text Files

Write

```
f = open("data.txt", "w")
```

```
f.write("Hello, Prince!\n")
f.write("Python file handling is fun.")
f.close()
```

Read

```
f = open("data.txt", "r")
print(f.read())       # Reads entire file
f.close()
```

Read line by line

```
with open("data.txt", "r") as f:
    for line in f:
        print(line.strip())
```

---

4. � File Pointer Functions

- `f.read(n)` → Reads `n` characters
- `f.readline()` → Reads one line
- `f.readlines()` → Returns list of lines
- `f.tell()` → Current position of pointer
- `f.seek(pos)` → Moves pointer to position

---

5. � Append Mode

```
with open("data.txt", "a") as f:
    f.write("\nThis line is appended.")
```

---

6. � Working with Binary Files

Binary files store data as **bytes** (e.g., images, videos).

Write binary

```
with open("image_copy.png", "wb") as f:
    f.write(b"Hello Binary")  # writing raw bytes
```

Read binary

```
with open("image.png", "rb") as f:
    data = f.read()
    print(type(data))   # <class 'bytes'>
```

---

7. � Working with CSV Files

Python has a `csv` module.

Write CSV

```
import csv

rows = [["Name", "Age"], ["Prince", 21], ["Aman", 22]]
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Read CSV

```
import csv

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

---

8. � Working with JSON Files

Python has a `json` module for structured data (key-value pairs).

Write JSON

```
import json

data = {"name": "Prince", "age": 21, "skills": ["Python", "MERN"]}
with open("data.json", "w") as f:
    json.dump(data, f, indent=4)
```

Read JSON

```
import json

with open("data.json", "r") as f:
    data = json.load(f)
print(data["name"])  # Prince
```

---

9. � Exception Handling in Files

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

---

10. 🔹 Interview Notes

1. **Difference between text & binary mode?**
   - Text = characters, encodes to Unicode.
   - Binary = raw bytes.
2. **Why use `with open()`?**
   - It auto-closes file (context manager).
3. **How to read large files?**
   - Use `readline()` or iterate line by line (not `read()`).
4. **Difference between CSV & JSON?**
   - CSV = tabular data.
   - JSON = structured hierarchical data.

✅ Now you know:

- Text, Binary, CSV, JSON file handling
- Read, Write, Append modes
- File pointer (`seek`, `tell`)
- Exception handling with files

Prince, do you want me to go **next → Python with Databases (SQLite + MySQL)** or **next → Object-Oriented Programming (OOP in Python: Classes, Inheritance, Polymorphism, etc.)**?

You said:

next

ChatGPT said:

Perfect Prince 👍 Now let's move into a **major core concept** –
👉 **Object-Oriented Programming (OOP) in Python** – which is very important for interviews, projects, and real-world coding.

📘 Python Notes – Part 14 (OOP in Python)

---

1. 🔹 What is OOP?

- OOP = **Object-Oriented Programming** → focuses on objects (real-world entities) instead of just functions.
- In Python, **everything is an object** (int, str, list, function, etc.).

---

2. 🔹 Class and Object

- **Class** → Blueprint/template for objects.
- **Object** → Instance of a class.

Example

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

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

s1 = Student("Prince", 21)   # Object creation
s1.display()
```

---

3. 🔹 `__init__` Method (Constructor)

- Special method called when object is created.
- Used for initializing attributes.

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

---

4. 🔹 Attributes and Methods

- **Instance Attributes** → Belong to objects.
- **Class Attributes** → Shared among all objects.

```
class Dog:
    species = "Canine"    # Class attribute

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

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

print(dog1.species, dog1.name)  # Canine Tommy
print(dog2.species, dog2.name)  # Canine Sheru
```

5. ◈ Encapsulation

- Bundling **data + methods** together.
- Private members: `__var` (name mangling).

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

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

    def get_balance(self):
        return self.__balance

b = Bank(1000)
b.deposit(500)
print(b.get_balance())  # 1500
```

---

6. ◈ Inheritance

- One class **inherits** properties of another.

Example

```
class Animal:
    def speak(self):
        print("Animal speaks")

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

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

Types of Inheritance

- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

---

7. ◈ Polymorphism

- **Same function, different behavior**.

Method Overriding

```
class Bird:
    def sound(self):
        print("Chirp")

class Parrot(Bird):
    def sound(self):
        print("Squawk")

p = Parrot()
p.sound()   # Squawk
```

Duck Typing

```
class Cat:
    def sound(self): print("Meow")

class Dog:
    def sound(self): print("Bark")

def animal_sound(animal):
    animal.sound()

animal_sound(Cat())   # Meow
animal_sound(Dog())   # Bark
```

---

8. ◈ Abstraction

- Hiding internal details, showing only essentials.
- Achieved using **abstract classes** in Python (`abc` module).

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r): self.r = r
    def area(self): return 3.14 * self.r * self.r

c = Circle(5)
print(c.area())  # 78.5
```

---

9. ◈ Operator Overloading

- Redefining operators for objects.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

```
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)  # 4 6
```

---

10. 🔹 Class Methods & Static Methods

Class Method

- Operates on **class** instead of instance.
- Uses `@classmethod`.

```
class Employee:
    company = "Google"

    @classmethod
    def change_company(cls, new_company):
        cls.company = new_company

e = Employee()
print(e.company)      # Google
Employee.change_company("Microsoft")
print(e.company)      # Microsoft
```

Static Method

- Independent function inside class.
- Uses `@staticmethod`.

```
class Math:
    @staticmethod
    def add(x, y):
        return x + y

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

---

11. 🔹 Important OOP Interview Notes

1. **Four pillars of OOP** → Encapsulation, Inheritance, Polymorphism, Abstraction.
2. **Difference between class & object?**
   - Class = Blueprint, Object = Instance.
3. **`@staticmethod` vs `@classmethod`?**
   - `staticmethod` → doesn't access class/object.
   - `classmethod` → works on class, accesses `cls`.
4. **Does Python support multiple inheritance?** → Yes.

✅ Now you know:

- Classes, Objects, Attributes
- Encapsulation, Inheritance, Polymorphism, Abstraction
- Operator Overloading, Static & Class methods

📘 Python Notes – Part 15 (Exception Handling in Python)

---

1. 🔹 What is an Exception?

- An **exception** is an error that occurs during program execution.
- Instead of crashing, Python lets you **handle errors gracefully**.

Example (without handling):

```
x = 10
y = 0
print(x / y)   # ZeroDivisionError
```

---

2. 🔹 Exception Handling with `try-except`

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

📌 Prevents program from crashing.

---

3. 🔹 `try-except-else-finally`

- `try` → Code that may raise error
- `except` → Handles error
- `else` → Runs if no exception
- `finally` → Runs always (cleanup, closing files)

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

```
    print("Number is:", num)
except ValueError:
    print("Invalid input")
else:
    print("Conversion successful")
finally:
    print("Execution finished")
```

---

4. ◆ Catching Multiple Exceptions

```
try:
    a = int("abc")
except (ValueError, TypeError) as e:
    print("Error:", e)
```

---

5. ◆ Raising Exceptions (`raise`)

You can **manually raise errors**.

```
x = -5
if x < 0:
    raise ValueError("x cannot be negative")
```

---

6. ◆ Custom Exceptions

Define your own exception class (inherit from `Exception`).

```
class NegativeNumberError(Exception):
    pass

def check_number(n):
    if n < 0:
        raise NegativeNumberError("Negative not allowed")
    else:
        print("Valid:", n)

try:
    check_number(-10)
except NegativeNumberError as e:
    print("Custom Error:", e)
```

---

7. ◆ Common Python Exceptions

Exception	When it occurs
`ValueError`	Wrong type/value for an operation
`TypeError`	Incompatible types
`IndexError`	Index out of range

Exception	When it occurs
`KeyError`	Key not found in dict
`ZeroDivisionError`	Divide by zero
`FileNotFoundError`	File not found
`AttributeError`	Attribute not found
`ImportError`	Import module fails

---

8. 🔹 Best Practices

✅ Always catch **specific exceptions** (not just `except:`).
✅ Use `finally` for cleanup (e.g., closing files/db).
✅ Use `logging` instead of `print()` in production.
✅ Define **custom exceptions** for domain-specific errors.

---

9. 🔹 Interview Notes

1. **Difference between Error vs Exception?**
   - Error → Problem in code (syntax, logic).
   - Exception → Runtime problem (can be handled).
2. **What does `finally` do?**
   - Always executes, used for cleanup.
3. **Why prefer specific exceptions?**
   - To avoid hiding unexpected bugs.
4. **Can you catch multiple exceptions in one block?**
   - Yes: `except (ValueError, TypeError):`.

✅ Now you know:

- `try-except-else-finally`
- Raising & Custom exceptions
- Common Python exceptions
- Best practices & interview notes

📘 Python Notes – Part 16 (Python with Databases)

---

1. 💠 Why Database with Python?

- Storing data permanently beyond files (structured, queryable).
- Python supports many databases: SQLite, MySQL, PostgreSQL, MongoDB.
- We'll focus on **SQLite (built-in)** and **MySQL (popular in real apps).**

---

2. 💠 Working with SQLite (built-in in Python)

SQLite is a **lightweight, file-based DB** (no server needed).
We use Python's `sqlite3` module.

📌 Steps to use SQLite

1. Connect to DB
2. Create cursor
3. Execute SQL queries
4. Commit changes
5. Close connection

Example: Create Table & Insert

```
import sqlite3

# 1. Connect (creates file if not exists)
conn = sqlite3.connect("students.db")

# 2. Cursor
cur = conn.cursor()

# 3. Create table
cur.execute("""CREATE TABLE IF NOT EXISTS student (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT,
                age INTEGER)""")

# 4. Insert data
cur.execute("INSERT INTO student (name, age) VALUES (?, ?)", ("Prince",
21))

# 5. Commit & close
conn.commit()
conn.close()
```

📌 Read Data

```
import sqlite3

conn = sqlite3.connect("students.db")
cur = conn.cursor()
```

```
cur.execute("SELECT * FROM student")
rows = cur.fetchall()
for row in rows:
    print(row)

conn.close()
```

📌 Update & Delete

```
cur.execute("UPDATE student SET age=? WHERE name=?", (22, "Prince"))
cur.execute("DELETE FROM student WHERE name=?", ("Prince",))
```

---

3. 🔷 Working with MySQL

For larger apps, MySQL is widely used.
We use `mysql-connector-python` package.

Install

```
pip install mysql-connector-python
```

📌 Connect to MySQL

```
import mysql.connector

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

📌 Create Table

```
cur.execute("""CREATE TABLE IF NOT EXISTS employee (
                id INT AUTO_INCREMENT PRIMARY KEY,
                name VARCHAR(50),
                salary INT)""")
```

📌 Insert Data

```
sql = "INSERT INTO employee (name, salary) VALUES (%s, %s)"
val = ("Prince", 50000)
cur.execute(sql, val)
conn.commit()
print(cur.rowcount, "record inserted.")
```

📌 Read Data

```
cur.execute("SELECT * FROM employee")
for row in cur.fetchall():
    print(row)
```

📌 Update & Delete

```
cur.execute("UPDATE employee SET salary=%s WHERE name=%s", (60000,
"Prince"))
cur.execute("DELETE FROM employee WHERE name=%s", ("Prince",))
conn.commit()
```

4. 🔹 Best Practices

- Always use **parameterized queries (`?` or `%s`)** to prevent SQL Injection.
- Always `commit()` after insert/update/delete.
- Use **connection pooling** for large apps.
- For ORMs (Object Relational Mapping) → use `SQLAlchemy` or `Django ORM`.

5. 🔹 Interview Notes

1. **Difference between SQLite and MySQL?**
   - SQLite → lightweight, file-based, single-user apps.
   - MySQL → client-server, handles multiple users, large-scale apps.
2. **Why parameterized queries?**
   - Prevents SQL Injection attacks.
3. **Steps in Python DB connection?**
   - Connect → Cursor → Execute → Commit → Close.
4. **What is ORM?**
   - Object Relational Mapper → map Python objects to DB tables (e.g., SQLAlchemy, Django ORM).

✅ Now you know how to:

- Use **SQLite** (built-in DB) with Python.
- Connect Python with **MySQL** (real-world apps).
- Perform CRUD (Create, Read, Update, Delete).
- Follow best practices & interview notes.