👨‍💻 **Author: Rashid Chaudhary** 🔗 **GitHub: https://github.com/rashid-star**

## 1. Python Basics

**What is Python?**
→ Python is a high-level, interpreted, object-oriented programming language used for web, data science, AI, automation.

**Features of Python**

- Easy syntax (readable)
- Interpreted language
- Object-oriented
- Large libraries (NumPy, Pandas, etc.)

**Compiled vs Interpreted**

- Python is interpreted → runs line by line
- No need compilation like C/C++

**Keywords**

- Reserved words: if, else, for, while, def, class, True, False

---

## 2. Variables & Data Types

**Variable**
→ Used to store data (no datatype declaration needed)

```
x = 10

name = "Rashid"
```

**Main Data Types**

- int → numbers
- float → decimal
- str → text
- bool → True/False
- list → ordered mutable
- tuple → ordered immutable
- set → unique values
- dict → key-value pair

## 3. Type Conversion

**Implicit conversion**
→ Python automatically converts type

**Explicit conversion**

int("10")

float(5)

str(100)

---

## 4. Operators

**Arithmetic** → + - * / % // **
**Comparison** → == != > <
**Logical** → and, or, not
**Assignment** → = += -=

---

## 5. Conditional Statements (Decision Making)

Conditional statements are used when we want Python to take decisions based on conditions.

### 👉 if statement

Used when we want to run code only if condition is true.

```
age = 18

if age >= 18:
    print("You can vote")
```

If condition is true → code runs
If false → skipped

---

### 👉 if-else statement

Used when we have two choices.

```
age = 16

if age >= 18:
    print("Eligible")

else:
    print("Not eligible")
```

If condition true → if block runs
If false → else block runs

### 👉 if-elif-else (multiple conditions)

Used when we check many conditions.

```
marks = 75


if marks >= 90:
   print("A grade")
elif marks >= 60:
   print("B grade")
else:
   print("Fail")
```

Python checks from top to bottom.
First true condition executes.

---

### 👉 Nested if

if inside another if.

```
age = 20
if age >= 18:
   if age >= 21:
      print("Adult")
```

---

**Interview Questions**

**Q: Difference between if and elif?**
→ if checks condition separately, elif checks only if above condition false.

**Q: Can we write multiple if?**
→ Yes, but all will check. elif stops after true.

---

## 6. Loops in Python

Loops are used to repeat code many times.

### 👉 for loop

Used when we know number of iterations.

```
for i in range(5):
   print(i)
```

Output: 0 1 2 3 4

range(5) → 0 to 4

---

**for loop with list**

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

for f in fruits:

  print(f)

---

👉 **while loop**

Used when condition-based loop.

i = 1

while i <= 5:

  print(i)

  i += 1

Runs until condition becomes false.

---

👉 **break statement**

Stops loop immediately.

for i in range(5):

  if i == 3:

    break

  print(i)

Output: 0 1 2

👉 **continue statement**

Skips current iteration.

for i in range(5):

  if i == 2:

    continue

  print(i)

Output: 0 1 3 4

👉 **pass statement**

Does nothing (placeholder).

```
for i in range(5):
    pass
```

**Q: Difference for vs while?**
→ for when iterations known, while when condition based.

**Q: Infinite loop?**
→ When loop never ends.

---

## 7. Functions in Python

Function is reusable block of code.

Used to avoid repetition.

👉 **Creating function**

```
def greet():
    print("Hello")
```

Calling function:

```
greet()
```

👉 **Function with parameters**

```
def add(a, b):
    print(a+b)
add(5,3)
```

a and b = parameters

👉 **Function with return**

Return sends value back.

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

👉 **Default parameters**

```
def greet(name="Guest"):
    print(name)
greet()
greet("Rashid")
```

### 👉 Lambda function (Important)

One line function without name.

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

**Used for short operations.**

👉 **Types of functions**

1. Built-in → print(), len()
2. User defined → created by user
3. Lambda → one line function

👉 ***args and **kwargs (interview favorite)**

Used for multiple arguments.

```
def add(*nums):
    print(sum(nums))
add(1,2,3,4)
```

kwargs = dictionary arguments.

---

## 8. Python Data Structures (Definition, Features, Methods)

Data structures are used to **store and organize data** in Python.

Main 4 types:

1. List
2. Tuple
3. Set
4. Dictionary

**1. LIST**

👉 **Definition**

List is an **ordered, mutable (changeable)** collection that allows duplicate values.

l = [1,2,3.6,"rashid"]

👉 **Features of List**

- Ordered (index available)
- Mutable (can change value)
- Allows duplicates
- Can store different data types
- Written in [ ]

👉 **Important List Methods**

**append()** → add element at end => l.append(50)

**insert()** → add at specific index => l.insert(1,100)

**remove()** → remove value => l.remove(20)

**pop()** → remove last element => l.pop()

**sort()** → sort list => l.sort()

**reverse()** → reverse list => l.reverse()

**len()** → length => len(l)

**2. TUPLE -**

Tuple is an **ordered and immutable** collection.

t = (1,2,3)

👉 **Features of Tuple**

- Ordered
- Immutable (cannot change)
- Allows duplicates
- Faster than list
- Written in ( )

👉 **Tuple Methods**

Only 2 methods (because immutable):

**count()** → count values => t.count(10)

**index()** → find index => t.index(20)

---

**Interview Difference: List vs Tuple**

List mutable, Tuple immutable
List slower, Tuple faster
List [], Tuple ()

---

## 3. SET

👉 **Definition**

Set is an **unordered collection of unique elements**.

s = {1,2,3,4}

👉 **Features of Set**

- Unordered (no index)
- No duplicates allowed
- Mutable
- Written in { }
- Fast operations

👉 **Important Set Methods**

**add()** → add element => s.add(10)

**remove()** → remove element => s.remove(2)

**union()** → combine sets => a.union(b)

**intersection()** → common elements => a.intersection(b)

**len()** → length

---

**4. DICTIONARY**

👉 **Definition**

Dictionary is a **key-value pair** data structure.

d = {"name":"Rashid","age":22}

"name" = key
"Rashid" = value

### 👉 Features of Dictionary

- Key-value pair
- Mutable
- Keys unique
- Fast access

### 👉 Important Dictionary Methods

**keys()** → show keys => d.keys()

**values()** → show values => d.values()

**items()** → key + value => d.items()

**update()** → update value => d.update({"age":25})

**pop()** → remove element => d.pop("age")

**get()** → safe access => d.get("name")

---

## 9. OOPs (Object Oriented Programming)

**Object Oriented Programming (OOP)** is a programming concept that uses **classes and objects** to structure code and represent real-world entities, making programs reusable, secure, and easy to maintain.

### 👉 Simple meaning

Instead of writing everything in one file, we divide code into **objects like real world**.

Example:

- Car → class
- BMW → object
- Student → class
- Rashid → object

---

**CLASS**

A class is a **blueprint or template** used to create objects.

It defines:

- variables (data)
- functions (methods)

👉 **Why class used?**

To organize code and represent real-world entities.

**Example:**

```
class Student:
  name = "Rashid"
```

Class = design
Object = real thing

---

**OBJECT**

An object is an **instance of a class**.
It is a real-world entity created from class.

**Example:**

```
class Student:
  pass

s1 = Student()
```

Student = class
s1 = object

👉 **Why object used?**

To access class data and methods.

---

**CONSTRUCTOR (init)**

A constructor is a **special method automatically called when object is created**.

In Python constructor = __init__

Used to initialize object data.

**Example:**

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

s1 = Student("Rashid")
```

👉 **Why used?**

To assign values when object created.

Interview line:
→ Constructor initializes object automatically.

---

**self Keyword -**

self represents the **current object of the class**.

Used to access:

- instance variables
- methods inside class

**Example:**

```
class A:
  def show(self):
    print("Hello")
```

Without self → cannot access class data.

Interview line:
→ self refers to current object.

---

**FOUR PILLARS OF OOP**

---

**1. ENCAPSULATION**

Encapsulation is the process of **wrapping data and methods into a single unit (class)** and restricting direct access to data.

👉 **Simple meaning**

Data hiding + security.

**Example:**

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

Here balance and function inside class = encapsulation.

👉 **Why used?**

- Data security
- Prevent direct modification
- Clean code

**Interview line:**

→ Encapsulation = data hiding and binding data with methods.

---

**2. INHERITANCE**

Inheritance is a mechanism where one class **acquires properties and methods of another class**.
Child class uses parent class code.

```
class Parent:
  def show(self):
    print("Parent")
class Child(Parent):
  pass
c = Child()
c.show()        => Child using parent method.
```

👉 **Why used?**

- Code reuse
- Reduce repetition
- Easy maintenance

**Real-life example**

Father → Parent class
Son → Child class

**Interview line:**

→ Inheritance allows code reusability.

---

**3. POLYMORPHISM**

Polymorphism means **one function, many forms**.

Same function behaves differently.

**Example:**

```
print(len("abc"))
print(len([1,2,3]))
```

Same len() works differently.

**Method overriding example:**

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


class B(A):
  def show(self):
    print("B")
```

👉 **Why used?**

- Flexibility
- Same function for different tasks

**Interview line:**

→ Polymorphism = same function different behavior.

**4. ABSTRACTION**

Abstraction means **hiding internal implementation and showing only essential features**.

**Simple meaning:**

User only sees important things, not internal code.

**Real-life example:**

ATM machine
We use ATM but don't know internal code.

👉 **Why used?**

- Reduce complexity
- Increase security
- Show only necessary data

**Interview line:**

→ Abstraction hides internal details and shows only functionality.

---

**Access Modifiers**

Used to control access of variables.

**Public**

Accessible everywhere

self.name

**Protected (_)**

self._age

**Private (__)**

self.__salary => Cannot access directly outside class.

---

**Why OOP used in real projects?**

Use this answer in interview:

OOP is used because:

- Makes code reusable
- Easy to manage large projects
- Provides security (encapsulation)
- Helps in real-world modeling
- Used in ML, web development, software

## 10. Comprehensions in Python

**Definition**

Comprehension is a **short and easy way to create list, dictionary, or set using single line code**.

Used to make code:

- shorter
- faster
- readable

---

**1. List Comprehension**

👉 **Definition**

List comprehension is a **compact way to create lists using loop in one line**.

**Normal method:**

```
l=[]
for i in range(5):
  l.append(i)
```

**List comprehension:**

l = [i for i in range(5)]

Both same output.

---

👉 **With condition**

l = [i for i in range(10) if i%2==0]

Output: even numbers

Interview line:
→ List comprehension is one-line method to create list.

---

**2. Dictionary Comprehension**

👉 **Definition**

Used to create dictionary in one line.

d = {x:x*x for x in range(5)}

Output:
0:0, 1:1, 2:4...

**3. Set Comprehension**

s = {x for x in range(5)}

Creates set.

---

**Interview Questions**

**Q: Why comprehension used?**
→ To write short and efficient code.

**Q: Difference loop vs comprehension?**
→ Comprehension shorter and faster.

---

## 11. Exception Handling

Exception handling is used to **handle errors and prevent program from crashing**.

When error occurs → program stops
Exception handling → program continues

---

👉 **Example without exception**

print(10/0)

Error → program stops

---

👉 **try-except**

```
try:
  print(10/0)
except:
  print("Error handled")
```

Program will not crash.

---

**try block**

Contains risky code.

**except block**

Handles error.

**finally block**

Always executes.

```
try:
  x=5
except:
  print("Error")
finally:
  print("Done")
```

---

**Specific exception**

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

---

**Multiple exception**

```
try:
  x=int("abc")
```

except ValueError:

print("Value error")

---

**Why exception handling used?**

- Prevent crash
- Handle errors
- Maintain flow
- User-friendly program

Interview line:
→ Exception handling manages runtime errors.

---

**Most asked interview questions**

**Q: What is exception?**
→ Error that occurs during runtime.

**Q: try vs except?**
→ try contains code, except handles error.

**Q: finally?** → Always executed block.

**12. File Handling in Python**

File handling is used to **read, write, and manage files** in Python.

Used to store data permanently.

---

**Opening file**

f = open("file.txt","r")

Modes:

- r → read
- w → write
- a → append

---

**Read file**

f=open("file.txt","r")

print(f.read())

---

**Write file**

f=open("file.txt","w")

f.write("Hello")

---

**Append file**

f=open("file.txt","a")

f.write("New line")

---

**Close file**

f.close()

---

**with open (best method)**

Automatically closes file.

with open("file.txt","r") as f:

  data = f.read()

  print(data)

**13. Important Built-in Functions in Python**

Built-in functions are **predefined functions already available in Python**, so we don't need to create them.

They make coding faster and easier.

Example: print(), len(), type()

---

**1. print()**

👉 **Definition**

Used to display output on screen.

print("Hello")

Interview line:
→ print() is used to display output.

---

**2. len()**

👉 **Definition**

Returns length of string, list, tuple, dictionary.

len("python")  # 6

```
len([1,2,3])   # 3
```

---

**3. type()**

👉 **Definition**

Returns datatype of variable.

```
x = 10

print(type(x))
```

Output: int

---

**4. range()**

👉 **Definition**

Generates sequence of numbers, used in loops.

```
for i in range(5):
   print(i)
```

Output: 0 to 4

**5. input()**

👉 **Definition**

Takes input from user.

```
name = input("Enter name:")
```

Always returns string.

---

**6. sum()**

👉 **Definition**

Returns sum of all elements.

```
sum([1,2,3,4])
```

Output: 10

---

**7. max() and min()**

👉 **Definition**

Returns maximum and minimum value.

```
max([1,5,2])   # 5

min([1,5,2])   # 1
```

---

**8. sorted()**

👉 **Definition**

Returns sorted list.

sorted([3,1,2])

Output: [1,2,3]

---

**9. map()**

👉 **Definition**

Applies function to all elements in list.

nums = [1,2,3]

result = list(map(lambda x:x*2, nums))

print(result)

Output: [2,4,6]

**10. filter()**

👉 **Definition**

Filters elements based on condition.

nums=[1,2,3,4]

even=list(filter(lambda x:x%2==0,nums))

print(even)

Output: [2,4]

---

**11. zip()**

👉 **Definition**

Combines two lists.

a=[1,2]

b=["a","b"]

print(list(zip(a,b)))

Output: (1,a), (2,b)

---

**12. enumerate()**

👉 **Definition**

Gives index + value.

```
l=["a","b","c"]
for i,v in enumerate(l):
    print(i,v)
```

---

## 16. Iterator & Generator

👉 **Iterator Definition**

Iterator is an object that allows us to **traverse (loop) through elements one by one**.

Example:

```
l=[1,2,3]
for i in l:
    print(i)
```

Internally uses iterator.

Functions:

- iter()
- next()

```
x = iter([1,2,3])
print(next(x))
```

---

👉 **Generator Definition**

Generator is a function that returns values **one by one using yield** instead of return.

```
def count():
    for i in range(3):
        yield i
for i in count():
    print(i)
```

**Why generator used?**

- Memory efficient
- Faster

- Used in large data

Interview line:
→ Generator uses yield and returns values one by one.

---

## 17. Decorators

👉 **Definition**

Decorator is a function that **modifies another function** without changing its code.

Used in:

- Logging
- Authentication
- Flask/FastAPI

Simple example:

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

Interview line:
→ Decorator modifies function behavior.

---

## 18. Virtual Environment

👉 **Definition**

Virtual environment is used to **create separate Python environment for each project**.

Why used:

- Avoid library conflicts
- Project isolation

Command:

```
python -m venv env
```

## 19. PIP

👉 **Definition**

pip is Python package manager used to install libraries.

Command:

pip install numpy

pip install pandas

Interview line:
→ pip installs Python packages.

---

## 21. Multithreading & Multiprocessing (basic)

👉 **Definition**

Used to run multiple tasks simultaneously.

Threading → lightweight
Multiprocessing → heavy tasks

Used in:

- Web scraping
- Data processing

Basic idea enough for interview.

---

## 🔵 22. name == "main"

👉 **Definition**

Used to check file running directly or imported.

```
if __name__ == "__main__":
    print("Run directly")
```

---

**Shallow Copy -**

Shallow copy creates a **new object but references (links) same inner data**.

Means: outer copy new, inner data same.

---

**Deep Copy -**

Deep copy creates **completely independent copy** of object and all nested objects.

---

**== operator**

== compares **values** of two objects.

---

**is operator**

is compares **memory location (object identity)**.

---

⭐ If these notes help you,  star this repository.