Python Interview Series - Part 1

Q. : What is a variable in Python?
Ans : A variable in Python is a name that refers to a value stored in memory. Unlike other languages, you don't need to declare its type — Python infers it automatically. Example:

```
x = 10
name = "Deepak"
```

Here, x is an integer and name is a string. Python variables are references to objects, not containers that hold data directly.

Q. : How does Python manage variable memory?
Ans : When you assign a value to a variable, Python creates an object in memory and binds the variable name to it. If you assign the same value to another variable, both can point to the same memory location (for immutable types).
Example:

```
a = 100
b = a
print(id(a), id(b))  # same id → both refer to same object
```

Mutable objects like lists behave differently — if modified, their memory doesn't change.

Q. : What are Python's built-in data types?
Ans :
Numeric → int, float, complex
Sequence → str, list, tuple
Mapping → dict
Set → set, frozenset
Boolean → bool
Binary → bytes, bytearray, memoryview

Example:

```
num = 10
pi = 3.14
name = "Python"
is_valid = True
```

Q. : What's the difference between mutable and immutable data types?
Ans :
Mutable → Can be changed after creation (list, dict, set)
Immutable → Cannot be changed after creation (int, float, str, tuple)
Example:

```
x = [1, 2, 3]
x.append(4)  # modifies list

y = "hello"
y.upper()    # creates new string, doesn't modify original
```

Q. : What is type casting or type conversion in Python?
Ans : It's the process of converting one data type into another.
Implicit → Python converts automatically

```
x = 5
y = 2.0
print(x + y)  # 7.0
```

Explicit → You convert manually

```
int("10"), float("5.5"), str(25)
```

Q. : Interviewer:* What are operators in Python?
Ans : Operators are symbols that perform operations on variables and values.

Arithmetic → +, -, *, /, %, //, **
Comparison → ==,!=,>, <,>=, <=
Logical → and, or, not
Assignment → =, +=, -=
Membership → in, not in
Identity → is, is not
Bitwise → &, |, ^, ~, <<,>>

Q. : Difference between is and == operators?
Ans : == compares values, 'is' compares memory location.
Example:

```
a = [1, 2]
b = [1, 2]
print(a == b)  # True
print(a is b)  # False
```

Q. : Difference between / and // operators?
Ans : / → true division (float), // → floor division (int)
Example:

```
print(7 / 2)   # 3.5
print(7 // 2)  # 3
```

Q. : How does Python handle dynamic typing?
Ans : Python uses dynamic typing — variable types are determined at runtime and can change.
Example:

```
x = 10       # int
x = "Hello"  # now str
```

Q. : Use of type() and id() functions?
Ans : type() → returns data type, id() → returns memory address
Example:

```
x = 5
print(type(x))  # <class 'int'>
print(id(x))    # unique memory id
```

Python Interview Series - Part 2

Q : What are conditional statements in Python?

Ans :
Conditional statements allow us to execute specific blocks of code based on certain conditions. Python uses if, elif, and else to control decision-making.

```
age = 18
if age < 18:
    print("Minor")
elif age == 18:
    print("Just eligible")
else:
    print("Adult")
```

✅ Only one block executes depending on the condition that evaluates to True.

Q : Can you explain the difference between if and elif?
Ans :
if starts the conditional chain.
elif (short for else if) allows checking multiple conditions sequentially.
If none are True, the else block runs.

```
x = 0
if x> 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

Q : Is there a way to write a single-line if statement in Python?
Ans :
Yes, we can use the ternary (conditional) expression.

```
result = "Even" if num % 2 == 0 else "Odd"
```

This makes the code concise for simple conditions.

Q : What are loops in Python?
Ans :
Loops allow repeating a block of code multiple times.
Python supports two main loops:

- for loop – used to iterate over a sequence (like list, tuple, dict, string).
- while loop – runs as long as a condition is True.

Q : Can you explain how a for loop works in Python?
Ans :
A for loop iterates over any iterable object (like a list or string).

Example:

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

Here, Python automatically fetches each item from the list one by one — no index or counter is required (unlike C or Java).

Q : How does the range() function work in loops?
Ans :
range() generates a sequence of numbers and is often used for looping a fixed number of times.
Syntax : range(start, stop, step)

Example:

```
for i in range(1, 6, 2):
    print(i)  # 1, 3, 5
```

Default values → start=0, step=1.
It doesn't create a list; it returns a range object (saves memory).

Q : What's the difference between for and while loops?
Ans :
- for loop: Used when we know how many times to iterate.
- while loop: Used when we don't know the number of iterations — runs until the condition becomes false.

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

# while loop
i = 0
while i < 5:
    print(i)
    i += 1
```

Q : What is the difference between break, continue, and pass statements?
Ans :

break – Exits the loop immediately
continue – Skips the current iteration and moves to the next
pass – Does nothing (placeholder for future code)

```
for i in range(5):
    if i == 2:
        continue  # skips 2
    if i == 4:
        break     # stops loop
    print(i)
```

Q : What is a nested loop? Give an example.
Ans :
A nested loop means having a loop inside another loop.
Used for matrix traversal or pattern printing.

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

It executes the inner loop completely for each iteration of the outer loop.

Q : Can you use an else clause with loops?
Ans :
Yes. In Python, a loop can have an else clause that runs only if the loop completes normally (not terminated by break).

Example:

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

If break is used, the else block is skipped.

Q : How do we iterate through a dictionary?
Ans :

```
data = {"a": 1, "b": 2}
for key, value in data.items():
    print(key, value)
```

Python Interview Series - Part 3

Topic: Functions, Arguments, and Scope in Python

Q : What is a function in Python and why do we use it?
Ans :
A function is a reusable block of code that performs a specific task.
It helps organize code, reduce repetition, and improve readability.
Functions make large programs modular and easy to debug.

We define a function using the `def` keyword:

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

Q : What's the difference between print() and return inside a function?
Ans :
`print()` → Displays output to the console.
`return` → Sends a value back to the caller for further use.

```
def add(a, b):
    print(a + b)     # prints result
    return a + b     # returns result
```

Q : What are different types of function arguments in Python?
Ans :
There are four main types of arguments:

Positional — `add(2, 3)` — Order matters
Keyword — `add(a=2, b=3)` — Order doesn't matter
Default — `def add(a, b=5)` — Default used if value missing
Variable-length — `*args, **kwargs` — Used when argument count varies

```
def details(name, *skills, **info):
    print(name)
    print(skills)
    print(info)

details("Alex", "Python", "SQL", age=25, city="NY")
```

Output:

```
Alex
('Python', 'SQL')
{'age': 25, 'city': 'NY'}
```

Q : Can you explain *args and **kwargs with a simple example?
Ans :

*args → accepts multiple non-keyword arguments as a tuple.
**kwargs → accepts keyword arguments as a dictionary.

Q : What are lambda functions?
Ans :
Lambda functions are small anonymous functions defined in one line using the lambda keyword.
They're often used for short tasks like sorting or filtering.

```
square = lambda x: x**2
print(square(5))  # Output: 25
```

- *They can take any number of arguments but only one expression.*

Q : Can you explain variable scope in Python?
Ans :
Yes. Python uses the *LEGB* Rule to determine scope:
L → Local
E → Enclosing (nested FUNCTIONS)
G → Global
B → Built-in

'global' & 'nonlocal' keywords

```
x = 10  # Global variable
```

```
def outer():
    x = 20  # Enclosing variable
    def inner():
        x = 30  # Local variable
        print(x)
    inner()
    print(x)

outer()
print(x)
```

Output:

```
30
20
10
```

Q : How do you modify a global variable inside a function?

Ans :

We use the global keyword.

```
x = 5

def update():
    global x
    x = 10

update()
print(x)  # Output: 10
```

Without global , Python treats x as a new local variable inside the function.

Q : Can a function return multiple values?

Ans :

Yes! In Python, a function can return multiple values as a *tuple.*

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

add, sub, mul = stats(5, 3)
print(add, sub, mul)
```

Output:

```
8 2 15
```

Q : What is recursion in Python?
Ans :
Recursion is when a function calls itself to solve a smaller instance of a problem.
A base case is essential to stop infinite recursion.

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

🗓️ *Python Interview Series - Part 4*

Topics: Lists, Tuples, Sets & Dictionaries

Q : Can you explain the difference between a list and a tuple in Python?

Both store ordered collections, but the key difference is *mutability*.

- 🔹 *List* → Mutable (can be changed)
- 🔹 *Tuple* → Immutable (cannot be changed once created)

```
# List
my_list = [1, 2, 3]
my_list.append(4)  # ✅ Works

# Tuple
my_tuple = (1, 2, 3)
my_tuple[0] = 10   # ❌ Error
```

🧠 *Use lists when data can change*, and tuples when it must remain constant.
Tuples are also slightly faster due to immutability.

Q : What are sets in Python? How are they different from lists?
Ans :
A *set* is an unordered collection of *unique* elements.
Unlike lists, sets *remove duplicates* and *don't maintain order*.

• Order: List ✅ | Set ❌
• Duplicates: List ✅ | Set ❌

• Indexing: List ✅ | Set ❌

Q : How do sets handle mathematical operations?

Ans : Sets support *union*, *intersection*, *difference*, and *symmetric difference*.

A = {1, 2, 3}
B = {3, 4, 5}

print(A | B) # Union → {1, 2, 3, 4, 5}
print(A & B) # Intersection → {3}
print(A - B) # Difference → {1, 2}

Q : What is a dictionary in Python and when do you use it?

Ans :
A *dictionary* is an unordered collection of *key-value pairs*.
Used when you need to *map unique keys to values*.

student = {"name": "Alice", "age": 22, "grade": "A"}
print(student["name"]) # Alice

Q : Can you give a real-world example where you'd prefer a dictionary over a list?

Ans : To store employees with their IDs:

employees = {101: "John", 102: "Emma", 103: "David"}
print(employees[102]) # Emma

🧠 Use dictionaries when you want *O(1) lookup* using *unique keys*

Python Interview Series - Part 5

Topic: Comprehensions in Python (List, Set, Dict)

Q : what is a list comprehension in Python?

A list comprehension is a compact way to create lists in one line instead of using multiple lines with for loops.

Syntax:
[expression for item in iterable if condition]

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
print(squares)  # [1, 4, 9, 16, 25]
```

🧠 It makes code cleaner, faster, and more readable — a must-know for interviews.

Q : can we add a condition inside a list comprehension?

we can use an if condition to filter items.

```
even_numbers = [n for n in range(10) if n % 2 == 0]
print(even_numbers)  # [0, 2, 4, 6, 8]
```

🧠 This is called a conditional list comprehension, often used for data filtering or cleaning tasks.

Q : Can you give an example where list comprehension replaces nested loops?

Nested comprehensions can replace multiple loops.

```
matrix = [[1, 2], [3, 4], [5, 6]]
flat = [num for row in matrix for num in row]
print(flat)  # [1, 2, 3, 4, 5, 6]
```

🧠 This is used to flatten 2D lists, a common interview pattern.

Q : what is a set comprehension?

A set comprehension works like list comprehension but returns a set — meaning it automatically removes duplicates.

```
nums = [1, 2, 2, 3, 4, 4]
unique = {n for n in nums}
print(unique)  # {1, 2, 3, 4}
```

🧠 Use case: Removing duplicates, quick filtering, or membership-based operations.

Q : What about dictionary comprehension?

Dictionary comprehension creates dictionaries using a single line of code.

Syntax:
{key_expression: value_expression for item in iterable}

```
nums = [1, 2, 3]
squares = {n: n**2 for n in nums}
print(squares)  # {1: 1, 2: 4, 3: 9}
```

🧠 Why it's useful: Quickly building lookup tables or mapping datasets.

Q : Can you include a condition in dictionary comprehension too?

Yes, just like in lists.

```
even_squares = {n: n**2 for n in range(10) if n % 2 == 0}
print(even_squares)
# {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

🧠 This is great for building filtered key-value maps efficiently.

Q : can you create a dictionary by swapping its keys and values using comprehension?

```
data = {'a': 1, 'b': 2, 'c': 3}
swapped = {v: k for k, v in data.items()}
print(swapped)  # {1: 'a', 2: 'b', 3: 'c'}
```

🧠 Used often in real-world data transformations — e.g., reversing ID-name mappings.

Q :what's the difference between list comprehension and generator expressions?

The main difference is memory efficiency.

List comprehension → Creates the full list in memory
Generator expression → Produces items one by one *(lazy evaluation) (next(g))*

```
# List comprehension
nums = [x**2 for x in range(5)]

# Generator expression
nums_gen = (x**2 for x in range(3))
next(g) //0
next(g) //1
next(g) //2
next(g) // StopIteration Error
```

🧠 Generators are better when dealing with large datasets — they save memory and improve performance.

Python Interview Series - Part 6

Sorting, Serching & Nested Structures

Q : Can you explain how sorting works in Python and what methods are commonly used?

Python provides two main ways to sort data:
 sorted(list) – returns a new sorted list
list.sort() – sorts the list in place

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

print(sorted(nums))  # [1, 2, 3, 4]
nums.sort()
print(nums)          # [1, 2, 3, 4]
```

🧠 Both support `reverse=True` and `key=` for custom sorting

```
words = ['apple', 'banana', 'kiwi']
print(sorted(words, key=len))  # ['kiwi', 'apple', 'banana']
```

Q : Can we sort complex data like a list of dictionaries?

Yes! Use a lambda function with `key=`

```
students = [
    {'name': 'Alice', 'score': 85},
    {'name': 'Bob', 'score': 92},
```

```
    {'name': 'Charlie', 'score': 78}
]

sorted_students = sorted(students, key=lambda x: x['score'], reverse=True)
print(sorted_students)
```

🧠 Common interview pattern when working with JSON-like data

Q : tell me about searching in lists or other data structures

Lists – use `in` or `.index()`

```
nums = [10, 20, 30, 40]
print(30 in nums)      # True
print(nums.index(30))  # 2
```

Dictionaries – search keys or values

```
student = {'name': 'Alice', 'age': 22}
print('name' in student)        # True
print(22 in student.values())   # True
```

Sets – *fastest for membership tests (O(1))*

```
ids = {101, 102, 103}
print(102 in ids)  # True
```

Q : Can you explain nested structures and how we handle them in Python?

Nested structures = data inside data (lists, dicts, etc.)

Nested List

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2])  # 6
```

Nested Dictionary

```
students = {
    'A101': {'name': 'Alice', 'score': 85},
    'A102': {'name': 'Bob', 'score': 90}
}
```

```
print(students['A101']['score'])  # 85
```

List of Dictionaries

```
data = [{'city': 'NY', 'temp': 25}, {'city': 'LA', 'temp': 30}]
for record in data:
    print(record['city'], record['temp'])
```

🧠 Great for tabular or hierarchical data (like JSON)

Q : How do you flatten a nested list like `[[1,2],[3,4]] → [1,2,3,4]`?

Use list comprehension

```
nested = [[1, 2], [3, 4]]
flat = [num for sublist in nested for num in sublist]
print(flat)  # [1, 2, 3, 4]
```

Python Interview Series — Part 7

Practice Round: Lists | Tuples | Sets | Dictionaries

Q : Count frequency of each element in a list

- We need to count occurrences of each number.
- A dictionary is best for mapping number → frequency.
- We can loop and increment count each time we see that number.
- Or use Counter from collections (built-in shortcut).

```
from collections import Counter
nums = [1, 2, 2, 3, 3, 3, 4]
freq = Counter(nums)
print(freq)
// Counter({3: 3, 2: 2, 1: 1, 4: 1})
```

Without Counter:

```
freq = {}
for i in nums:
    freq[i] = freq.get(i, 0) + 1
print(freq)
```

Q : Remove duplicates but keep order

A set() removes duplicates easily but doesn't maintain order.
So I will use a loop + list to check whether the element has appeared before.

```
nums = [1, 2, 2, 3, 1, 4, 3]
unique = []
for i in nums:
    if i not in unique:
        unique.append(i)
print(unique)

// Output: [1, 2, 3, 4]
```

If order doesn't matter:

```
unique = list(set(nums))
```

Q : Find max and min without using max()/min()

I will start by assuming the first element is both max and min. Then, Traverse the list and update values if I find bigger/smaller ones.

```
nums = [5, 8, 2, 9, 1, 7]
max_num = min_num = nums[0]
for n in nums:
    if n> max_num: max_num = n
    if n < min_num: min_num = n
print("Max:", max_num)
print("Min:", min_num)
```

Q : Find unique common elements between two lists

I will use a set to perform intersection. Intersection gives elements present in both sets.

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 6, 7, 8]
common = list(set(a) & set(b))
print(common)
```

🧠 *Output:* `[4, 5]`
& operator finds intersection of sets quickly in O(n) time.

Preserve order of a:

common = [x for x in a if x in b]

Q : Return second largest unique element

- Remove duplicates using set().
- Sort the list in ascending order.
- Pick the second last element.

```
nums = [10, 20, 4, 45, 99, 99, 10]
unique_nums = sorted(set(nums))
second_largest = unique_nums[-2]
print(second_largest)
```

Q : Count character frequency in a string

```
from collections import Counter
text = "python interview"
freq = Counter(text)
print(freq)

// Counter({'t':2, 'e':2, 'n':2, 'r':2, 'p':1, 'y':1, 'h':1, 'i':1, 'v':1, 'w':1})
```

Counter automatically counts all occurrences and stores results in a dictionary-like format.

```
Sorted by frequency
for ch, count in freq.most_common():
    print(ch, ":", count)
```

Python Interview Series - Part 8

Functions & Arguments in Python

Q : What is a function in Python and why do we use it?*

A *function* is a reusable block of code that performs a specific task.
It helps avoid repetition, improves code readability, and makes debugging easier.

```
def greet(name):
    return f"Hello, {name}!"
print(greet("Deepak"))  # Output: Hello, Deepak!
```

🧠 *Functions* are defined using the *def* keyword and can take inputs (*parameters*) and return outputs.

Q : Can you explain different types of arguments Python functions can have?

Yes, Python supports several types:

1️⃣ *Positional Arguments* – Passed in order.

```
def add(a, b):
    return a + b
print(add(3, 4))  # 7
```

2️⃣ *Keyword Arguments* – Passed by name.

```
def greet(name, age):
    print(f"{name} is {age} years old.")
greet(age=25, name="John")  # Works fine
```

3️⃣ *Default Arguments* – Have a preset value.

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

4️⃣ *Variable-Length Arguments* (*args) – For multiple positional values.

```
def add_all(*args):
    return sum(args)
print(add_all(1, 2, 3, 4))  # 10
```

5️⃣ *Keyword Variable-Length Arguments* (**kwargs) – For multiple key-value pairs.

```
def show_details(**kwargs):
    for key, value in kwargs.items():
        print(key, ":", value)

show_details(name="Deepak", age=27, city="Delhi")
```

🧠 *args collects values as a tuple; **kwargs collects values as a dictionary.

Q : Nice explanation. What's the difference between *args and **kwargs in a real-world use case?

Let's take an example of a *restaurant order* function:

```
def order(food_type, *items, **details):
    print(f"Food Type: {food_type}")
    print("Items Ordered:", items)
    print("Details:", details)

order("Veg", "Paneer", "Dal", "Roti", table=5, waiter="Ravi")
```

Output:
Food Type: Veg
Items Ordered: ('Paneer', 'Dal', 'Roti')
Details: {'table': 5, 'waiter': 'Ravi'}

So, *items lets you accept multiple dishes, and **details lets you store extra order info flexibly.

Q : Can you tell me what a lambda function is and when you use it?

A *lambda function* is a small anonymous function (no name) used for short, simple operations.

Syntax:
lambda arguments: expression

```
square = lambda x: x ** 2
print(square(5))  # 25
```

It's often used with *map()*, *filter()*, and *reduce()*.

Q : Explain map(), filter(), and reduce() with examples.

✅ *map()* → Applies a function to every element of a sequence.

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

✅ *filter()* → Filters items that meet a condition.

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

✅ *reduce()* → Applies a rolling computation (must import from functools).

```
from functools import reduce
total = reduce(lambda x, y: x + y, nums)
print(total)  # 10
```

🧠 These are useful for *functional-style programming* — clean and concise.

Q : If you define a function inside another function, what do you call it?

That's called a *nested function*.
It's often used in *closures* or when one function is only needed inside another.

```
def outer():
    def inner():
        return "Hello from inner!"
    return inner()

print(outer())
```

🧠 *Quick Summary:*

• *args → multiple positional values
• **kwargs → multiple keyword values
• lambda → one-line anonymous function
• map() → apply function to all
• filter() → select based on condition
• reduce() → combine to single result

Python Interview Series - Part 8

File Handling & CSV Operations in Python

Q : Can you explain how file handling works in Python?

We use `open()` to read/write files.
Syntax:
`file = open("filename.txt", "mode")`

Modes:
'r' → Read
'w' → Write (overwrites)
'a' → Append
'r+' → Read & write
'b' → Binary mode

Use `file.close()` or better:

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

Q : What's the difference between read(), readline(), and readlines()?

read() → Entire file as string
readline() → One line
readlines() → List of lines

```
with open("demo.txt", "r") as f:
    print(f.read())
    print(f.readline())
    print(f.readlines())
```

Q : How do you write data into a file?

Use `write()` or `writelines()`

```
with open("output.txt", "w") as f:
    f.write("Python is great!\n")
    f.writelines(["Data\n", "Analytics\n", "Interview\n"])
```

Q : What happens if you open a file in write mode that already exists?

It overwrites the file. Use `'a'` to append:

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

Q : What if the file doesn't exist and we try to read it?

It raises `FileNotFoundError`. Use try-except:

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

Q : How do you handle CSV files in Python?

Using `csv` module:

```
import csv

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

with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Deepak", 27])
```

Q : How to read a CSV using pandas?

```
import pandas as pd

df = pd.read_csv("data.csv")
print(df.head())

df.to_csv("new_data.csv", index=False)
```

Q : How to handle large CSV files?

Use `chunksize` in pandas:

```
for chunk in pd.read_csv("large_file.csv", chunksize=10000):
    print(chunk.shape)
```

Q : How to check if a file exists?

```
import os

if os.path.exists("data.csv"):
    print("File found!")
else:
    print("File not found!")
```

Mini Practice Questions

✅ *Q1. Count lines in a text file*

```
with open("sample.txt", "r") as f:
    lines = f.readlines()
print(len(lines))
```

✅ *Q2. Copy contents from one file to another*

```
with open("source.txt", "r") as src, open("target.txt", "w") as tgt:
    tgt.write(src.read())
```

✅ *Q3. Print names of people above 25 from CSV*

```
import csv

with open("people.csv", "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        if int(row["Age"])> 25:
            print(row["Name"])
```

✅ *Q4. Find average of a column using pandas*

```
import pandas as pd
df = pd.read_csv("sales.csv")
print(df["Revenue"].mean())
```

✅ *Q5. Append user input to a file*

```
text = input("Enter text: ")
with open("notes.txt", "a") as f:
    f.write(text + "\n")
```

Python Interview Series – Part 9

Modules & Imports in Python

Q : What is a module in Python?

A module is a `.py` file containing Python code—functions, classes, or variables—that can be reused in other programs using `import`.

Q : How do you import a module?

Use `import` or `from... import...`

```
import math
```

```
print(math.sqrt(16))

from math import pi
print(pi)
```

Q : What's the difference between a module and a package?

- *Module* = single `.py` file
- *Package* = folder with multiple modules + `__init__.py` file

Q : How do you create a custom module?

Write functions in a `.py` file (e.g., `utils.py`)
Then import it:

```
import utils
utils.say_hello()
```

Q : How do you explore module contents?

Use `dir()` and `help()`

```
import math
print(dir(math))
help(math.sqrt)
```

Q : What is `__name__ == "__main__"` used for?

It checks if the script is run directly or imported.

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

Q : How do you install external modules?

Use pip:
pip install requests

Q : How do you handle import errors?

Use try-except:

```
try:
    import unknown_module
except ImportError:
    print("Module not found")
```

Q : What are some commonly used built-in modules?

- `math`
- `os`
- `sys`
- `datetime`
- `random`

Mini Practice Questions

✅ *Q1. Create a module with a function that returns factorial*

```
# mymath.py
def factorial(n):
    return 1 if n==0 else n * factorial(n-1)
```

✅ *Q2. Import and use your custom module*

```
import mymath
print(mymath.factorial(5))
```

✅ *Q3. List all functions in the `os` module*

```
import os
print(dir(os))
```

✅ *Q4. Use `random` to generate 5 random numbers*

```
import random
for _ in range(5):
    print(random.randint(1, 100))
```

✅ *Q5. Use `datetime` to print today's date*

```
from datetime import date
print(date.today())
```

🧠 *Concept Recap:*

- import module_name → imports the whole module
- from module import function → imports specific parts
- ```__name__ == "__main__"``` → helps differentiate between script and import execution
- Modules help write cleaner, reusable, and maintainable code

Python Interview Series — Part 10

Practice Round: Custom Functions | File Handling | Error Handling

Q : Write a function to check if a number is prime

- A prime number is only divisible by 1 and itself
- Check divisibility from 2 to √n

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

print(is_prime(11))  # True
```

Q : Read a text file and print its contents

✅ *Using `with` (best practice):*

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

✅ *Read line by line:*

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

🧠 _Using `with` ensures the file is closed automatically._

Q : Write a function to count lines, words, and characters in a file

```
def file_stats(filename):
    with open(filename, "r") as f:
        lines = f.readlines()
        num_lines = len(lines)
        num_words = sum(len(line.split()) for line in lines)
        num_chars = sum(len(line) for line in lines)
    return num_lines, num_words, num_chars

print(file_stats("sample.txt"))
```

Q : Handle division by zero using try-except

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input.")
```

🧠 *Always catch specific exceptions first, then general ones.*

Q : Raise a custom exception if age < 18

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be at least 18")
    return "Access granted"

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

Python Interview Series – Part 11

Topics: Classes, Objects, Polymorphism & Inheritance

Q : What is a class in Python?

A class is a blueprint for creating objects. It defines attributes and methods that describe the behavior and state of the object.

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

    def greet(self):
        print(f"Hello, I'm {self.name}")
```

Q : What is an object?

An object is an instance of a class. It holds the actual data and can use the methods defined in the class.

```
p = Person("Alice")
p.greet()  # Output: Hello, I'm Alice
```

Q : Explain inheritance in Python

Inheritance allows a class to inherit properties and methods from another class. Promotes code reuse.

```
class Employee(Person):
    def work(self):
        print(f"{self.name} is working")

e = Employee("Bob")
e.greet()
e.work()
```

Q : What is polymorphism?

Polymorphism allows different classes to define methods with the same name but different behavior.

```
class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

def animal_sound(animal):
    animal.speak()

animal_sound(Dog())  # Woof!
animal_sound(Cat())  # Meow!
```

Q : What is method overriding?

When a subclass provides its own version of a method defined in the parent class.

```
class Person:
    def greet(self):
        print("Hello!")

class Student(Person):
    def greet(self):
        print("Hi, I'm a student")
```

Q : What is `super()` used for?

`super()` is used to call a method from the parent class.

```
class Student(Person):
    def greet(self):
        super().greet()
        print("Hi, I'm a student")
```

Mini Practice Questions

✅ Q1. Create a class `Car` with attributes `brand` and `speed`
✅ Q2. Create a subclass `ElectricCar` that adds `battery_capacity`
✅ Q3. Override a method to display car details
✅ Q4. Use polymorphism to call `drive()` method from different vehicle classes

🧠 *Concept Recap*

– Class = blueprint
– Object = instance of class
– Inheritance = reuse code from parent class
– Polymorphism = same method name, different behavior
– `super()` = access parent methods

Python Interview Series – Part 12

Topics: Encapsulation & Abstraction

Q : What is encapsulation in Python?

Encapsulation is the practice of hiding internal object details and exposing only necessary parts. It protects data by restricting direct access and allows controlled interaction through methods.

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

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

    def get_balance(self):
        return self.__balance

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

Q : What does the double underscore `__` mean?

It makes an attribute private by name mangling. It can't be accessed directly from outside the class.

Q: What is abstraction in Python?

Abstraction means hiding complex implementation details and showing only essential features. It simplifies usage and focuses on what an object does, not how it does it.

Q: How do you implement abstraction?

Using abstract classes and methods via the `abc` module.

```
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())  # Output: 78.5
```

Q : How is abstraction different from encapsulation?

– *Encapsulation* is about restricting access
– *Abstraction* is about hiding complexity

Mini Practice Questions with Answers

✅ *Q1. Create a class with private attributes and public methods*

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

    def get_info(self):
        return f"{self.__name}, Age: {self.__age}"

s = Student("Ravi", 21)
print(s.get_info())  # Output: Ravi, Age: 21
```

✅ *Q2. Use `@property` to create getter/setter methods*

```
class Product:
    def __init__(self, price):
        self.__price = price

    @property
```

```
    def price(self):
        return self.__price

    @price.setter
    def price(self, value):
        if value> 0:
            self.__price = value

p = Product(100)
p.price = 150
print(p.price)  # Output: 150
```

✅ *Q3. Create an abstract class `Vehicle` with abstract method `drive()`*

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def drive(self):
        pass

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

c = Car()
c.drive()  # Output: Car is driving
```

✅ *Q4. Implement `drive()` in subclasses like `Car` and `Bike`*

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

vehicles = [Car(), Bike()]
for v in vehicles:
    v.drive()
# Output:
# Car is driving
# Bike is riding
```

🧠 *Concept Recap*

– *Encapsulation* = hide internal data, expose via methods
– *Abstraction* = hide complexity, show only essentials
– Use `__` for private attributes
– Use `abc` module for abstract classes
– `@property` helps manage access to private data

Python Interview Series – Part 13

Topics: Magic Methods – `__init__`, `__str__`

Q : What are magic methods in Python?

Magic methods (also called dunder methods) are special methods with double underscores, like `__init__`, `__str__`, `__len__`, etc. They customize object behavior for built-in operations.

Q : What is `__init__` used for?

`__init__` is the constructor method. It runs automatically when an object is created and is used to initialize attributes.

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

p = Person("Alice", 30)
print(p.name)  # Output: Alice
```

Q : What is `__str__` used for?

`__str__` defines how an object is represented as a string. It's called when you use `print()` or `str()` on an object.

```
class Person:
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, Age: {self.age}"

p = Person("Bob", 25)
print(p)  # Output: Bob, Age: 25
```

Q : What happens if `__str__` is not defined?

Python shows the default object representation, like `<__main__.Person object at 0x...>`

Mini Practice Questions with Answers

✅ Q1. Create a class `Book` with `__init__` and `__str__`

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

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

b = Book("1984", "George Orwell")
print(b)  # Output: '1984' by George Orwell
```

✅ Q2. Create a class `Point` that prints coordinates nicely

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

    def __str__(self):
        return f"Point({self.x}, {self.y})"

p = Point(3, 4)
print(p)  # Output: Point(3, 4)
```

✅ Q3. Use `__str__` to display a shopping cart item

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

    def __str__(self):
        return f"{self.name} - ${self.price:.2f}"

i = Item("Laptop", 999.99)
print(i)  # Output: Laptop - $999.99
```

🧠 *Concept Recap*

– `__init__` = constructor, sets up object state
– `__str__` = string representation for printing
– Magic methods start and end with double underscores
– Customize how objects behave with built-in functions

Python Interview Series – Part 14

Practice Round: Simple Classes – BankAccount | StudentSystem

Q : Build a class `BankAccount` with deposit and withdraw methods

– Use `__init__` to set up account holder and balance
– Add methods for deposit, withdraw, and balance check

```
class BankAccount:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ₹{amount}")

    def withdraw(self, amount):
        if amount> self.balance:
            print("Insufficient funds")
        else:
```

```
            self.balance -= amount
            print(f"Withdrew ₹{amount}")

    def get_balance(self):
        return f"Balance: ₹{self.balance}"

acc = BankAccount("Ravi", 1000)
acc.deposit(500)
acc.withdraw(300)
print(acc.get_balance())  # Output: Balance: ₹1200
```

Q : Build a class `StudentSystem` to store student info and grades

– Use `__init__` to store name and ID
– Add method to update and display grades

```
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self.grades = {}

    def add_grade(self, subject, grade):
        self.grades[subject] = grade

    def show_report(self):
        print(f"Report for {self.name} ({self.student_id})")
        for subject, grade in self.grades.items():
            print(f"{subject}: {grade}")

s = Student("Aisha", "S101")
s.add_grade("Math", 92)
s.add_grade("Science", 88)
s.show_report()
```

🧠 *Concept Recap*

– Use `__init__` to initialize object state
– Define methods to perform actions
– Use dictionaries to store dynamic data like grades
– Keep logic clean and modular for reusability

Python Interview Series – Part 15

Topics: Exception Handling & Logging

Q : What is exception handling in Python?

Exception handling lets you manage runtime errors gracefully using `try`, `except`, `finally`, and `else`. It prevents your program from crashing.

```
try:
    x = int(input("Enter a number: "))
    print("You entered:", x)
except ValueError:
    print("Invalid input! Please enter a number.")
```

Q : Can you catch multiple exceptions?

Yes, using a tuple:

```
try:
    risky_code()
except (ValueError, ZeroDivisionError) as e:
    print("Caught:", e)
```

Q : What is the use of `finally`?

The `finally` block runs no matter what—used for cleanup actions.

```
try:
    f = open("data.txt")
except FileNotFoundError:
    print("File not found")
finally:
    print("Execution complete")
```

Q: How do you raise custom exceptions?

Use `raise` with a custom message:

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18+")
```

Q: How do you log errors in Python?

Use the `logging` module:

```
import logging

logging.basicConfig(level=logging.ERROR)
try:
    1 / 0
except ZeroDivisionError as e:
    logging.error("Error occurred: %s", e)
```

Q : What are logging levels?

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

Mini Practice Questions with Answers

✅ _Q1. Handle file read error with logging_

```
import logging
try:
    with open("missing.txt") as f:
        data = f.read()
except FileNotFoundError as e:
```

```
    logging.error("File error: %s", e)
```

✅ _Q2. Raise exception if password is too short_

```
def validate_password(pwd):
    if len(pwd) < 6:
        raise ValueError("Password too short")
```

✅ _Q3. Use `finally` to close a file_

```
try:
    f = open("data.txt")
    # process file
finally:
    f.close()
```

✅ _Q4. Log a warning if disk space is low_

```
import logging
logging.warning("Disk space below 10%!")
```

✅ _Q5. Catch all exceptions and log them_

```
import logging
try:
    risky_code()
except Exception as e:
    logging.exception("Unexpected error")
```

🧠 *Concept Recap*

– `try-except` handles errors gracefully
– `finally` ensures cleanup
– `raise` triggers custom exceptions
– `logging` tracks errors, warnings, and info
– Avoid bare `except:` — always catch specific exceptions

Python Interview Series – Part 16

Topics: Decorators, Generators & Iterators

Q: *What is a decorator in Python?*

A *decorator* is a function that modifies the behavior of another function without changing its code. It's often used for logging, access control, and timing.

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

@decorator
def greet():
    print("Hello!")

greet()
```

Q: *Can decorators take arguments?*

Yes, by nesting functions:

```
def repeat(n):
    def decorator(func):
        def wrapper():
            for _ in range(n):
                func()
        return wrapper
    return decorator

@repeat(3)
def say_hi():
    print("Hi!")
```

Q : *What is a generator?*

A *generator* is a function that yields values one at a time using `yield`, allowing lazy evaluation and memory efficiency.

```
def countdown(n):
    while n> 0:
        yield n
        n -= 1

for num in countdown(3):
    print(num)
```

Q : *How is a generator different from a list?*

Generators don’t store all values in memory. They compute values on the fly, making them ideal for large datasets or infinite sequences.

Q : What is an iterator?

An *iterator* is an object with `__iter__()` and `__next__()` methods. It lets you traverse elements one at a time.

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current> self.high:
            raise StopIteration
        self.current += 1
        return self.current - 1

for num in Counter(1, 3):
    print(num)
```

Mini Practice Questions with Answers

✅ *Q1. Create a decorator that logs function calls*

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

✅ *Q2. Write a generator for even numbers up to n*

```
def evens(n):
    for i in range(n + 1):
        if i % 2 == 0:
            yield i
```

✅ *Q3. Build a custom iterator for Fibonacci numbers*

```
class Fibonacci:
    def __init__(self, limit):
        self.a, self.b = 0, 1
        self.limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        if self.a> self.limit:
            raise StopIteration
        value = self.a
        self.a, self.b = self.b, self.a + self.b
        return value
```

🧠 *Concept Recap*

– *Decorators* wrap functions to extend behavior

– *Generators* yield values lazily
– *Iterators* implement `__iter__()` and `__next__()`
– Use generators for performance, decorators for clean code, and iterators for custom traversal
- `yield` creates values one-by-one
`next()` retrieves them one-by-one

Top Python OOPs Interview Questions with Answers

1. What is Object-Oriented Programming (OOP)?
OOP is a programming paradigm based on the concept of “objects” that contain data (attributes) and code (methods).
It helps in code reusability, scalability, and maintainability.

2. What are the key OOP principles in Python?
1. *Encapsulation* – Binding data and methods in a class
2. *Abstraction* – Hiding complex implementation details
3. *Inheritance* – Reusing code from parent classes
4. *Polymorphism* – Same method name with different behavior

3. What is a Class and Object in Python?
- *Class*: A blueprint for creating objects.
- *Object*: An instance of a class.

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

my_car = Car("Toyota")
```

4. What is `_init_()` method?
It’s a constructor method in Python that gets called when an object is created. It initializes the object.

```
def _init_(self, name):
    self.name = name
```

5. What is Inheritance?
Allows a class (child) to inherit properties of another class (parent).

```
class Animal:
    def speak(self):
```

```
        print("Makes sound")

class Dog(Animal):
    def speak(self):
          print("Barks")
```

6. What is Encapsulation?

Restricting direct access to some components using private members.

```
class Person:
    def _init_(self):
        self.__age = 25  # private attribute
```

7. What is Polymorphism?

Means “many forms” – same method behaves differently depending on context.

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

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

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

8. What is Method Overriding?

Child class redefines a method from the parent class.
Supports runtime polymorphism.

9. What is `super()` in Python?

Used to call methods of the parent class.

```
class Child(Parent):
    def _init_(self):
        super()._init_()
```

10. Difference between Class and Instance Variables?

- *Class Variable* → Shared by all objects
- *Instance Variable* → Unique for each object

Top Python Exception Handling Interview Q&A

1️⃣ What is exception handling in Python?

Exception handling is a way to handle runtime errors and prevent the program from crashing. Python uses `try-except` blocks to catch and manage exceptions.

2️⃣ What is the syntax of try-except in Python?

```
try:
    # risky code
except ExceptionType:
    # handle error
```

You can also use `finally` and `else`:

```
try:
    ...
except:
    ...
else:
    ...
finally:
    ...
```

3️⃣ What is the difference between SyntaxError and Exception?

- *SyntaxError*: Occurs when the code is invalid Python (caught at compile time).
- *Exception*: Caught at runtime, e.g., `ZeroDivisionError`, `TypeError`.

4️⃣ How do you raise a custom exception?

You can define and raise custom exceptions:

```
class MyError(Exception):
    pass

raise MyError("Something went wrong")
```

5️⃣ What does the `finally` block do?

`finally` always executes, whether an exception occurred or not. It's used for cleanup tasks like closing files or releasing resources.

6️⃣ What is the difference between `raise` and `assert`?*

- `raise` explicitly throws an exception.
- `assert` is used for debugging; it stops execution if a condition is false:
```python
assert x > 0, "x must be positive"
```

7️⃣ Can you have multiple except blocks?*
Yes, to handle different error types:

```
try:
    ...
except ValueError:
    ...
except TypeError:
    ...
```

8️⃣ What happens if you don't handle an exception?*

If unhandled, Python will terminate the program and print a traceback showing the error and where it occurred.

Python Strings & Files – Interview Questions with Answers

1️⃣ How are strings defined in Python?*
Strings are sequences of characters enclosed in single (`'...'`), double (`"..."`), or triple quotes (`"""..."""`).
Examples: `'Hello'`, `"World"`, `"""Multiline string"""`

2️⃣ Are strings mutable in Python?*
No. Strings are *immutable*, meaning once created, their contents cannot be changed. Any modification creates a new string.

3️⃣ What are some common string methods?*

- `lower()`, `upper()` – Case conversion
- `strip()` – Removes leading/trailing whitespace
- `replace(old, new)` – Replaces substrings
- `split(delimiter)` – Splits string into a list
- `find(substring)` – Returns index of first occurrence

4️⃣ What is string slicing?*
Slicing allows access to parts of a string using index ranges.

```
text = "Python"
text[0:3]  # 'Pyt'
text[-1]   # 'n'
```

5️⃣ What are f-strings in Python?*
F-strings provide a concise way to embed expressions inside string literals.

```
name = "Alice"
print(f"Hello, {name}!")  # Output: Hello, Alice!
```

6️⃣ How do you read and write files in Python?*

```
# Reading
with open("file.txt", "r") as f:
    content = f.read()

# Writing
with open("file.txt", "w") as f:
    f.write("Hello, world!")
```

7️⃣ What is a context manager and why use `with`?*
Answer: The `with` statement ensures proper resource handling (like closing files). It simplifies file operations and avoids memory leaks.

8️⃣ How do you handle JSON data in Python?*

```
import json

# Convert dict to JSON string
data = {"name": "Bob", "age": 25}
json_str = json.dumps(data)

# Convert JSON string to dict
parsed = json.loads(json_str)
```

Python Strings & Files – Interview Questions with Answers [Part-2]

1️⃣ Write a Python program to count the number of vowels in a string*

```
def count_vowels(s):
    return sum(1 for char in s.lower() if char in 'aeiou')

print(count_vowels("Interview Ready"))  # Output: 5
```

2️⃣ How do you check if a string is a palindrome*

```
def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("madam"))  # Output: True
```

3️⃣ Write a program to read a file and print its contents line by line*

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

4️⃣ How do you append text to an existing file*

```
with open("log.txt", "a") as file:
    file.write("New log entry\n")
```

5️⃣ Write a program to count the number of words in a file*

```
with open("data.txt", "r") as file:
    text = file.read()
    word_count = len(text.split())
    print("Word count:", word_count)
```

Advanced Python Interview Q&A

1. What is a decorator in Python?*

A decorator is a function that modifies another function's behavior without changing its source code. I've used decorators to add logging and access control in Flask APIs without repeating code.

2. What's the difference between a generator and an iterator?

Generators are a simpler way to create iterators. They yield values one by one, saving memory. I used a generator to stream large CSV files row by row in a data processing script.

3. How does `yield` work?

`yield` pauses the function and returns a value. When called again, the function resumes where it left off. It helped me build a custom pagination tool without holding the entire dataset in memory.

4. What's the difference between a module and a package?

A module is a single `.py` file. A package is a directory with an `_init_.py` file that can include multiple modules. For example, I structured a project into `auth`, `db`, and `routes` packages for clarity.

5. Why do we use virtual environments?

To keep project dependencies isolated. I used `venv` in a Django project so I could use specific versions of packages without affecting other projects.

6. Difference between list, set, and dict in terms of performance?

Lists are slow for lookups (O(n)). Sets and dicts are fast (O(1)) because of hashing. I use sets when I need to check existence quickly, like filtering out duplicates in logs.

7. Shallow vs deep copy—what's the real difference?

Shallow copy copies references, so changes affect the original. Deep copy creates new objects entirely. I use deep copy when cloning nested data like config dictionaries to avoid side effects.

8. Mutable vs Immutable—how does that affect real code?

Mutable types like lists can change, immutables like strings can't. I avoid using mutable types as default arguments in functions because it can lead to bugs across calls.

9. What does writing 'Pythonic' code mean to you?

It means writing readable, clean code using built-in functions and idioms. I prefer list comprehensions and unpacking because they're concise and clear.

10. Why do you use Git for Python projects?

For version control and collaboration. It helps me track changes, experiment safely, and collaborate via pull requests. I never deploy without pushing code to GitHub first.

Advanced Python Interview Q&A

1. What will be the output of the following code?

```
def func():
    for i in range(3):
        yield i

g = func()
print(next(g))
print(next(g))
```

0
1
Each call to `next()` resumes the generator from where it left off.

2. How do you reverse a dictionary in Python?

```
original = {'a': 1, 'b': 2}
reversed_dict = {v: k for k, v in original.items()}
```

3. What's the difference between `is` and `==`?

`is` checks identity (same memory location)
`==` checks equality (same value)

4. What does this code do?

```
def f(x=[]):
    x.append(1)
    return x

print(f())
print(f())
```

[1]
[1, 1]
Default mutable arguments retain state across calls. Use `None` and initialize inside the function to avoid this.

5. How do you flatten a nested list?

```
from itertools import chain
nested = [[1, 2], [3, 4]]
flat = list(chain.from_iterable(nested))
```

6. What is the output of this code?

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
```

[1, 2, 3, 4]
`a` and `b` reference the same list.

7. How do you check if a string is a palindrome?

```
def is_palindrome(s):
    return s == s[::-1]
```

8. What's the difference between `@staticmethod` and `@classmethod`?

`@staticmethod`: no access to class or instance
`@classmethod`: takes `cls` and can access class-level data

9. How do you handle missing keys in a dictionary?

Use `.get()` or `defaultdict`

```
value = my_dict.get('key', 'default')
```

10. What is the use of `__slots__` in a class?

`__slots__` restricts dynamic attribute creation and saves memory

```
class MyClass:
    __slots__ = ['x', 'y']
```

Python Libraries & Tools – Interview Q&A (Part 1: NumPy & Pandas)

1. What is NumPy and why is it used?
NumPy (Numerical Python) is a library used for efficient numerical computations, especially with arrays and matrices.
It offers high-performance array operations and broadcasting capabilities.

2. What is the difference between a Python list and a NumPy array?
- Lists are flexible and can hold mixed data types.
- NumPy arrays are more efficient, support vectorized operations, and require homogeneous data types.

3. How do you create a NumPy array?

```
import numpy as np
arr = np.array([1, 2, 3])
```

4. What are some common NumPy functions?
- `np.zeros()`, `np.ones()`, `np.arange()`, `np.linspace()`
- Mathematical: `np.mean()`, `np.sum()`, `np.dot()`, `np.std()`

5. What is Pandas and why is it important?
Pandas is a data analysis library built on top of NumPy.
It provides `Series` and `DataFrame` structures for handling labeled and tabular data.

6. Difference between Series and DataFrame?
- `Series`: 1D labeled array
- `DataFrame`: 2D labeled data structure with rows and columns

7. How do you create a DataFrame in Pandas?

```
import pandas as pd
df = pd.DataFrame({'Name': ['A', 'B'], 'Age': [25, 30]})
```

8. What are some common DataFrame operations?
- `df.head()`, `df.tail()`, `df.describe()`, `df.info()`
- Filtering: `df[df['Age'] > 25]`
- Sorting: `df.sort_values('Age')`

9. How do you handle missing data in Pandas?
- `df.isnull()` to detect
- `df.dropna()` to remove
- `df.fillna(value)` to replace

10. How do NumPy and Pandas work together?
Pandas is built on NumPy. You can pass NumPy arrays to DataFrames and vice versa for efficient processing.

Python Libraries & Tools – Interview Q&A (Part 2: Regular Expressions)

1. What is the `re` module in Python used for?
The `re` module provides support for working with *regular expressions*, allowing you to search, match, and manipulate strings using patterns.

2. What is a regular expression?
A regular expression is a sequence of characters that defines a search pattern. It's used for pattern matching in strings — such as email validation, text extraction, etc.

3. How do you search for a pattern in a string?

```
import re
re.search(r'pattern', 'your text')
```

Returns a match object if found, else `None`.

4. What's the difference between `search()` and `match()`?

- `re.match()` checks for a match *only at the beginning* of the string.
- `re.search()` scans *through the entire string* for a match.

5. How do you find all occurrences of a pattern?

```
re.findall(r'\d+', 'There are 3 cats and 5 dogs')
```
Output: ['3', '5']

6. What does `re.sub()` do?
It replaces all occurrences of a pattern in a string.

re.sub(r'\s+', '-', 'Hello   World')
Output: 'Hello-World'

7. How can you compile a regex pattern for reuse?
pattern = re.compile(r'\d+')
pattern.findall('123 and 456')

Useful for performance in repeated matching.

8. Common regex symbols used in Python:
- `.` – Any character
- `^` – Start of string
- `$` – End of string
- `\d` – Digit
- `\w` – Word character
- `\s` – Whitespace
- `+`, `*`, `?` – Quantifiers
- `[a-z]` – Character range
- `()` – Capture group
- `|` – OR

9. How do you use groups in regex?
match = re.search(r'(\d+)-(\d+)', 'Phone: 123-4567')
print(match.group(1)) # 123
print(match.group(2)) # 4567

10. When should regex be avoided?
If simple string methods (`split()`, `replace()`, `in`, etc.) are enough, they are faster and more readable than regex.

Python Interview: Set vs Frozenset

1️⃣ Are Sets and Frozensets mutable?

- *Set*: ✅ Mutable — elements can be added or removed
- *Frozenset*: ❌ Immutable — cannot be changed after creation

2️⃣ Can they store mixed data types?*

Yes — both can hold different types like `{1, "Alice", True}`

3️⃣ Are Frozensets hashable?

Yes — can be used as dictionary keys or set elements
Sets are unhashable and cannot be used as keys

4️⃣ Do they support set operations?*

Both support union, intersection, difference, etc.
Example: `a.union(b)`, `a & b`, `a - b`

5️⃣ When should you use Frozenset over Set?*

Use *Frozenset* when:
- You need an immutable set
- You want to use it as a dictionary key
- You want to ensure data integrity

6️⃣ Can you add/remove elements?*

- *Set*: Yes — use `.add()`, `.remove()`
- *Frozenset*: No — no methods to modify

7️⃣ How do you convert between them?*

- `frozenset(my_set)` → Set to Frozenset
- `set(my_frozenset)` → Frozenset to Set

8️⃣ Do they preserve order?*

No — both are unordered collections

9️⃣ Can Frozensets contain Sets?*

No — because Sets are unhashable
But Frozensets can contain other Frozensets

*🔟 **Which is better for constant data?***

Frozenset is ideal for fixed, read-only sets used in configurations or as keys

Python Interview: Shallow Copy vs Deep Copy

1️⃣ What is a Shallow Copy?

A *shallow copy* creates a new object but copies references of nested objects. Changes to nested elements affect both copies.

```
import copy
a = [[1, 2], [3, 4]]
b = copy.copy(a)
```

2️⃣ What is a Deep Copy?

A *deep copy* creates a completely independent copy of the object *and all nested objects*.

```
b = copy.deepcopy(a)
```

3️⃣ When do Shallow and Deep copies differ?

When the object has *nested mutable elements* (e.g., lists inside lists), a shallow copy shares inner objects, while a deep copy replicates them.

4️⃣ What modules/functions are used?

Use the `copy` module:
- `copy.copy()` → Shallow copy
- `copy.deepcopy()` → Deep copy

5️⃣ Can simple assignments copy an object?

No — `b = a` just creates a new reference to the same object (no actual copying).

6️⃣ How to test if a copy is deep or shallow?

Modify a nested element. If both objects reflect the change → shallow. If only one does → deep.

7️⃣ Are strings or integers affected?

No — they're immutable. Copies behave the same regardless of shallow or deep.

8️⃣ Which is faster?
Shallow copy is faster since it doesn't recursively copy nested structures.

PYTHON PATTERNS YOU SHOULD KNOW

Right-Angled Triangle

```
rows = 5
for i in range(1, rows+1):
    for j in range(1, i+1):
        print(j, end=" ")
    print()
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Left-Angled Triangle

```
rows = 5
for i in range(1, rows+1):
    print(" " * (rows - i), end="")
    for j in range(1, i+1):
        print(j, end=" ")
    print()
```

```
    1
   1 2
  1 2 3
 1 2 3 4
1 2 3 4 5
```

Pyramid

```
rows = 5
for i in range(1, rows+1):
    print(" " * (rows - i), end="")
    for j in range(1, i+1):
        print(j, end=" ")
    print()
```

```
    1
   1 2
  1 2 3
 1 2 3 4
1 2 3 4 5
```