

INTERVIEW PREPARATION GUIDE | 2026 EDITION

100 Python Interview Questions and Answers

For freshers, intermediate and experienced developers.
Every code example tested on Python 3.13, with real output.

40

Freshers questions

35

Intermediate questions

25

Experienced questions

● **Code and Debug**

codeanddebug.in | [YouTube: @codeanddebug](https://www.youtube.com/@codeanddebug) | Free to share for learning

How to use this guide

The 100 questions are grouped by level: **Freshers** (1 to 40), **Intermediate** (41 to 75) and **Experienced** (76 to 100). Freshers should master the first section completely and read the second. Experienced candidates are expected to answer all three.

Do not memorise answers word for word. Interviewers follow up with "why?" and "what happens if...?". Run every code example yourself, change it, and break it on purpose.

Every code example in this guide was executed on Python 3.13 and the output shown is the real output. Outputs such as memory sizes can differ slightly on other versions.

The interactive version of this guide, with flashcards, practice mode, progress tracking and a predict-the-output game, is at codeanddebug.in/blog/python-interview-questions.

Contents

Freshers (1 to 40)

1. What is Python and what are its key features?
2. Is Python compiled or interpreted?
3. What is PEP 8 and why does it matter?
4. What does it mean that Python is dynamically typed?
5. What are Python's built-in data types?
6. What is the difference between mutable and immutable objects?
7. What is the difference between a list and a tuple?
8. What is a dictionary in Python, and what can be a key?
9. What is a set and when would you use one?
10. What is the difference between `is` and `==`?
11. What are `*args` and `**kwargs`?
12. Why should you not use a mutable default argument?
13. What is a lambda function?
14. What are list, dict and set comprehensions?
15. How do you reverse a string in Python, and how does slicing work?
16. Why is indentation important in Python?
17. What is the difference between `append()` and `extend()`?
18. What do `pass`, `break` and `continue` do?
19. What is `None` in Python?
20. Are strings mutable? Name some useful string methods.
21. What is type conversion (casting) in Python?
22. What is the difference between `/`, `//` and `%`?
23. How does Python manage memory?
24. What are local and global variables? When do you need the `global` keyword?
25. What is the difference between a module and a package?
26. What does `if __name__ == "__main__":` do?
27. How does exception handling work in Python?
28. How do you read and write files in Python? Why use `with`?
29. What do `enumerate()` and `zip()` do?
30. What is `range()` and why is it memory efficient?
31. What is the difference between a shallow copy and a deep copy?
32. What is a docstring?
33. What are namespaces and the LEGB rule?
34. How do you remove duplicates from a list while keeping the order?
35. Write a program to check whether a string is a palindrome.
36. How do you count the frequency of each character or word?
37. How do you swap two variables in Python?
38. Find the largest and second largest number in a list.
39. Write a function to check whether a number is prime.
40. Print the Fibonacci series up to `n` terms.

Intermediate (41 to 75)

41. What is the difference between an iterable and an iterator?
42. What is a generator and how does `yield` work?

43. What is the difference between a list comprehension and a generator expression?
44. What is a decorator in Python?
45. How do you write a decorator that takes arguments?
46. What is a closure?
47. Explain map(), filter() and reduce().
48. What are the four pillars of OOP, and how does Python support them?
49. What is the difference between class variables and instance variables?
50. What are __init__ and self?
51. How does inheritance work, and what does super() do?
52. What is the Method Resolution Order (MRO)?
53. What is the difference between instance methods, class methods and static methods?
54. What are magic (dunder) methods? Give examples.
55. How does encapsulation work in Python? What is name mangling?
56. What is an abstract class in Python?
57. What does the @property decorator do?
58. What is duck typing?
59. How do you create and raise a custom exception?
60. What is a context manager and how do you create one?
61. What is the difference between sort() and sorted()?
62. How do get(), setdefault() and defaultdict help with dictionaries?
63. What are namedtuple, deque and Counter in the collections module?
64. What is unpacking, and what is the walrus operator?
65. What is a frozenset, and what makes an object hashable?
66. What are f-strings, and how do you format numbers with them?
67. What are type hints? Does Python enforce them?
68. What is pickling and unpickling?
69. How do you work with JSON in Python?
70. How would you process a file that is too large to fit in memory?
71. What is a virtual environment, and why should you use one?
72. Two Sum: find two numbers in a list that add up to a target.
73. Check whether two strings are anagrams.
74. Flatten a nested list of any depth.
75. Find the missing number in a list containing 0 to n.

Experienced (76 to 100)

76. What is the GIL (Global Interpreter Lock)?
77. When would you use threading, multiprocessing or asyncio?
78. How does async and await work? Give an example.
79. What is a race condition and how do you prevent it in Python?
80. How does Python's garbage collector handle reference cycles?
81. What are __slots__ and when should you use them?
82. What is a metaclass?
83. What is a descriptor?
84. How are Python dictionaries implemented internally?
85. What is the time complexity of common list, dict and set operations?
86. What is the late binding closure problem?
87. What is integer caching and string interning?
88. What does functools.lru_cache do?
89. Name some useful functions from the itertools module.
90. What are dataclasses?
91. What do send() and yield from do in generators?
92. What is structural pattern matching (match-case)?
93. What are some important features added in recent Python versions?
94. How do you write unit tests in Python?
95. How do you find and fix slow Python code?
96. Is Python pass-by-value or pass-by-reference?
97. Why is NumPy faster than Python lists for numerical work?
98. How do you handle missing values in a Pandas DataFrame?
99. How do groupby and merge work in Pandas?
100. Implement an LRU cache.

Freshers level

Core Python concepts and basic coding questions asked in campus placements and first job interviews.

Q1. What is Python and what are its key features?

FRESHERS · Basics

Python is a high-level, general-purpose programming language created by Guido van Rossum and first released in 1991. It is known for readable code that looks close to plain English, which is why it is often the first language people learn and the default choice for data science, machine learning, automation and backend development.

- **Easy to read and write:** indentation instead of braces, and very little boilerplate.
- **Interpreted:** you run code directly without a separate compile step.
- **Dynamically typed:** you do not declare variable types.
- **Multi-paradigm:** supports procedural, object-oriented and functional styles.
- **Huge standard library and ecosystem:** NumPy, Pandas, Django, FastAPI, PyTorch and thousands more.
- **Cross-platform and open source.**

Q2. Is Python compiled or interpreted?

FRESHERS · Basics

Both, in a way. When you run a `.py` file, CPython (the standard implementation) first compiles the source code into **bytecode**, a low-level set of instructions, and caches it in `.pyc` files inside the `__pycache__` folder. The **Python Virtual Machine (PVM)** then interprets that bytecode line by line.

Because the compile step is automatic and invisible, Python is usually called an interpreted language. You can see the bytecode with the `dis` module:

CODE

```
import dis

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

dis.dis(add)
```

OUTPUT

3	RESUME	0
4	LOAD_FAST_LOAD_FAST	1 (a, b)
	BINARY_OP	0 (+)
	RETURN_VALUE	

Q3. What is PEP 8 and why does it matter?

FRESHERS · Basics

PEP 8 is the official style guide for Python code. PEP stands for Python Enhancement Proposal. It covers naming, indentation, line length and spacing so that Python code written by different people looks consistent. Key rules:

- Use 4 spaces per indentation level.
- Keep lines to 79 characters (many teams allow up to 88 or 100).
- `snake_case` for functions and variables, `PascalCase` for classes, `UPPER_CASE` for constants.

- Two blank lines around top-level functions and classes.
- Imports at the top, one per line.

Tools like [black](#), [ruff](#) and [flake8](#) check or auto-format code to PEP 8. Interviewers ask this to see whether you write code that a team can maintain.

Q4. What does it mean that Python is dynamically typed?

FRESHERS · Basics

In a dynamically typed language, the type belongs to the value, not to the variable name. You never declare a type, and the same variable can point to an integer now and a string later. Types are checked at runtime, when the code actually runs.

Python is also **strongly typed**: it will not silently convert between unrelated types. Adding a number to a string raises an error instead of guessing what you meant.

CODE

```
x = 10
print(type(x))
x = "ten"
print(type(x))

try:
    print("Age: " + 25)
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
<class 'int'>
<class 'str'>
TypeError: can only concatenate str (not "int") to str
```

Q5. What are Python's built-in data types?

FRESHERS · Data types

The main built-in types are:

- **Numeric:** [int](#), [float](#), [complex](#)
- **Text:** [str](#)
- **Sequence:** [list](#), [tuple](#), [range](#)
- **Mapping:** [dict](#)
- **Set:** [set](#), [frozenset](#)
- **Boolean:** [bool](#) (a subclass of [int](#))
- **Binary:** [bytes](#), [bytearray](#), [memoryview](#)
- **None type:** [NoneType](#), which has a single value, [None](#)

CODE

```
values = [42, 3.14, 2 + 3j, "code", [1, 2], (1, 2), range(3),
          {"a": 1}, {1, 2}, frozenset({1}), True, b"hi", None]
for v in values:
    print(type(v).__name__, end=" ")
```

OUTPUT

```
int float complex str list tuple range dict set frozenset bool bytes NoneType
```

Q6. What is the difference between mutable and immutable objects?

FRESHERS · Data types

A **mutable** object can be changed in place after it is created. An **immutable** object cannot: any "change" creates a brand new object.

- Mutable: `list`, `dict`, `set`, `bytearray`, most user-defined objects.
- Immutable: `int`, `float`, `str`, `tuple`, `frozenset`, `bytes`, `bool`.

This matters because two variables can point to the same mutable object, so changing it through one name changes what the other sees. Only immutable objects can be dictionary keys or set members.

CODE

```
a = [1, 2, 3]
b = a          # b points to the SAME list
b.append(4)
print("a:", a)

t = (1, 2, 3)
try:
    t[0] = 99
except TypeError as e:
    print("TypeError:", e)

s = "code"
print(id(s) == id(s + "debug"))  # concatenation makes a new string
```

OUTPUT

```
a: [1, 2, 3, 4]
TypeError: 'tuple' object does not support item assignment
False
```

Q7. What is the difference between a list and a tuple?

FRESHERS · Data types

Both are ordered sequences that can hold any type, but:

- A **list** is mutable and written with square brackets: `[1, 2, 3]`.
- A **tuple** is immutable and written with parentheses: `(1, 2, 3)`.
- Tuples use less memory and are slightly faster to create and iterate.
- Tuples are hashable (if their items are), so they can be dictionary keys. Lists cannot.

Use a list for a collection that will change, like a shopping cart. Use a tuple for a fixed record, like coordinates `(lat, lon)` or a function returning several values.

CODE

```
import sys
lst = [1, 2, 3, 4, 5]
tup = (1, 2, 3, 4, 5)
print("list bytes:", sys.getsizeof(lst))
print("tuple bytes:", sys.getsizeof(tup))

locations = {(28.61, 77.21): "Delhi"}  # tuple as a dict key
print(locations[(28.61, 77.21)])
```

OUTPUT

```
list bytes: 104
tuple bytes: 80
Delhi
```

Q8. What is a dictionary in Python, and what can be a key?

FRESHERS · Data types

A dictionary ([dict](#)) stores key-value pairs and gives average $O(1)$ lookup, insertion and deletion, because it is built on a hash table. Since Python 3.7, dictionaries keep the order in which keys were inserted.

A key must be **hashable**: its hash value must never change. Strings, numbers, booleans and tuples of immutable items work as keys. Lists, dicts and sets do not.

CODE

```
student = {"name": "Riya", "branch": "CSE", "cgpa": 8.7}
student["city"] = "Pune"          # add
print(student.get("phone", "N/A")) # safe lookup with a default
for key, value in student.items():
    print(key, "->", value)

try:
    bad = {[1, 2]: "list key"}
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
N/A
name -> Riya
branch -> CSE
cgpa -> 8.7
city -> Pune
TypeError: unhashable type: 'list'
```

Q9. What is a set and when would you use one?

FRESHERS · Data types

A set is an unordered collection of **unique**, hashable items. Membership checks (`x in s`) are $O(1)$ on average, compared with $O(n)$ for a list. Use a set to remove duplicates, to check membership quickly, or for mathematical operations like union and intersection.

CODE

```
python_students = {"Aman", "Riya", "Kabir", "Sneha"}
java_students = {"Kabir", "Sneha", "Arjun"}

# sets are unordered, so sorted() is used only to print them neatly
print("Both:", sorted(python_students & java_students))
print("Either:", sorted(python_students | java_students))
print("Only Python:", sorted(python_students - java_students))
print("Unique marks:", sorted(set([90, 85, 90, 70, 85])))
```

OUTPUT

```
Both: ['Kabir', 'Sneha']
Either: ['Aman', 'Arjun', 'Kabir', 'Riya', 'Sneha']
Only Python: ['Aman', 'Riya']
Unique marks: [70, 85, 90]
```

Q10. What is the difference between is and ==?

FRESHERS · Basics

`==` checks **equality**: do the two objects have the same value? `is` checks **identity**: are they the exact same object in memory (same `id()`)?

Use `==` to compare values. Use `is` only for singletons, most commonly `if x is None`.

CODE

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(a == b)    # same values
print(a is b)    # different objects
print(a is c)    # same object

x = None
print(x is None)
```

OUTPUT

```
True
False
True
True
```

Q11. What are *args and **kwargs?

FRESHERS · Functions

They let a function accept a variable number of arguments.

- `*args` collects extra **positional** arguments into a tuple.
- `**kwargs` collects extra **keyword** arguments into a dictionary.

The names `args` and `kwargs` are only convention. The `*` and `**` do the work. The same symbols also unpack a list or dict when *calling* a function.

CODE

```
def register(name, *subjects, **details):
    print("Name:", name)
    print("Subjects:", subjects)
    print("Details:", details)

register("Riya", "DSA", "Python", city="Pune", year=3)

nums = [3, 7, 1]
print(max(*nums))    # unpacking a list into arguments
```

OUTPUT

```
Name: Riya
Subjects: ('DSA', 'Python')
Details: {'city': 'Pune', 'year': 3}
7
```

Q12. Why should you not use a mutable default argument?

FRESHERS · Functions

Default argument values are evaluated **once**, when the function is defined, not every time it is called. If the default is a mutable object like a list, every call that relies on the default shares the same list, so data leaks between calls. This is one of the most common Python interview traps.

The fix is to use `None` as the default and create a new list inside the function.

CODE

```
def add_item_bad(item, cart=[]):
    cart.append(item)
    return cart

print(add_item_bad("pen"))
print(add_item_bad("book"))    # surprise: pen is still there

def add_item(item, cart=None):
    if cart is None:
        cart = []
    cart.append(item)
    return cart

print(add_item("pen"))
print(add_item("book"))
```

OUTPUT

```
['pen']
['pen', 'book']
['pen']
['book']
```

Q13. What is a lambda function?

FRESHERS · Functions

A lambda is a small anonymous function written in one expression: `lambda arguments: expression`. It returns the value of that expression automatically. Lambdas are handy as short throwaway functions, most often as the `key` for sorting or with `map` and `filter`.

If the logic needs more than one expression, or you want to reuse it, write a normal `def` function instead. PEP 8 discourages assigning a lambda to a name.

CODE

```
students = [("Aman", 78), ("Riya", 92), ("Kabir", 85)]
students.sort(key=lambda s: s[1], reverse=True)
print(students)

square = lambda x: x * x
print(square(9))
```

OUTPUT

```
[('Riya', 92), ('Kabir', 85), ('Aman', 78)]
81
```

Q14. What are list, dict and set comprehensions?

FRESHERS · Data types

Comprehensions build a new collection from an iterable in a single readable line. They are usually faster than an equivalent `for` loop with `append`, because the loop runs in optimised C code.

The pattern is `[expression for item in iterable if condition]`. Use curly braces with `key: value` for a dict, and curly braces without a colon for a set.

CODE

```
nums = range(1, 11)
squares = [n * n for n in nums]
evens = [n for n in nums if n % 2 == 0]
lengths = {word: len(word) for word in ["code", "and", "debug"]}
first_letters = {word[0] for word in ["apple", "avocado", "banana"]}

print(squares)
print(evens)
print(lengths)
print(sorted(first_letters))  # sets have no fixed order
```

OUTPUT

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[2, 4, 6, 8, 10]
{'code': 4, 'and': 3, 'debug': 5}
['a', 'b']
```

Q15. How do you reverse a string in Python, and how does slicing work?

FRESHERS · Strings

The simplest way is slicing with a step of -1: `s[::-1]`. Slicing uses `sequence[start:stop:step]`. `start` is included, `stop` is excluded, and any part can be omitted. Negative indices count from the end, so `s[-1]` is the last character.

Slicing works on strings, lists and tuples, and always returns a new object.

CODE

```
s = "codeanddebug"
print(s[::-1])      # reversed
print(s[0:4])       # 'code'
print(s[-5:])       # last 5 characters
print(s[::2])       # every second character
print("".join(reversed(s)))  # another way
```

OUTPUT

```
gubeddnædoc
code
debug
cdædeu
gubeddnædoc
```

Q16. Why is indentation important in Python?

FRESHERS · Basics

Python uses indentation to define code blocks, where languages like C++ and Java use curly braces. The body of an `if`, a loop, a function or a class is whatever is indented under it. Wrong indentation is not a style issue: it is a syntax error, or worse, valid code that does something different from what you intended.

PEP 8 recommends 4 spaces per level. Never mix tabs and spaces.

CODE

```
marks = 72
if marks >= 40:
    print("Passed")
    print("Certificate issued")    # inside the if
print("Result processed")        # outside the if, always runs
```

OUTPUT

```
Passed
Certificate issued
Result processed
```

Q17. What is the difference between `append()` and `extend()`?

FRESHERS · Data types

`append(x)` adds `x` as a **single** element at the end of the list, even if `x` is itself a list.

`extend(iterable)` adds **each element** of the iterable one by one. `insert(i, x)` adds an element at a specific position.

CODE

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

b = [1, 2]
b.extend([3, 4])
print(b)

c = [1, 2]
c.insert(0, 0)
print(c)
```

OUTPUT

```
[1, 2, [3, 4]]
[1, 2, 3, 4]
[0, 1, 2]
```

Q18. What do `pass`, `break` and `continue` do?

FRESHERS · Basics

- `pass` does nothing. It is a placeholder where Python requires a statement, like an empty function or class you will fill in later.
- `break` exits the nearest loop immediately.
- `continue` skips the rest of the current iteration and moves to the next one.

CODE

```
def todo():
    pass # implement later

for n in range(1, 10):
    if n % 2 == 0:
        continue # skip even numbers
    if n > 7:
        break # stop the loop
    print(n, end=" ")
```

OUTPUT

```
1 3 5 7
```

Q19. What is None in Python?

FRESHERS · Basics

None is a special constant that represents the absence of a value. It is the only instance of **NoneType**. Functions that do not explicitly return anything return **None**. It is falsy in boolean context, but it is not the same as **0**, **False** or an empty string.

Always check for it with **is None**, not **== None**.

CODE

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

result = greet("Riya")
print(result)
print(result is None)
print(bool(None), None == 0, None == False)
```

OUTPUT

```
Hello Riya
None
True
False False False
```

Q20. Are strings mutable? Name some useful string methods.

FRESHERS · Strings

No, strings are immutable. Every string method returns a **new** string and leaves the original unchanged. Commonly asked methods:

CODE

```
s = " Code and Debug "
print(s.strip())
print(s.lower(), "|", s.upper())
print(s.strip().split(" "))
print("-".join(["2026", "10", "17"]))
print("python".replace("py", "Py"))
print("data.csv".endswith(".csv"), "Riya".startswith("R"))
print("banana".count("a"), "banana".find("n"))
print("42".isdigit(), "abc".isalpha())
```

OUTPUT

```
Code and Debug
code and debug | CODE AND DEBUG
['Code', 'and', 'Debug']
2026-10-17
Python
True True
3 2
True True
```

Q21. What is type conversion (casting) in Python?

FRESHERS · Basics

Type conversion changes a value from one type to another. **Implicit** conversion happens automatically when it is safe, for example `int + float` gives a `float`. **Explicit** conversion (casting) uses constructor functions like `int()`, `float()`, `str()`, `list()`, `tuple()` and `set()`. Invalid conversions raise a `ValueError`.

CODE

```
print(5 + 2.5)           # implicit: int -> float
print(int("42") + 1)
print(float("3.5"))
print(int(9.99))         # truncates, does not round
print(list("abc"), tuple([1, 2]), sorted(set("hello")))
try:
    int("forty")
except ValueError as e:
    print("ValueError:", e)
```

OUTPUT

```
7.5
43
3.5
9
['a', 'b', 'c'] (1, 2) ['e', 'h', 'l', 'o']
ValueError: invalid literal for int() with base 10: 'forty'
```

Q22. What is the difference between /, // and %?

FRESHERS · Basics

- `/` is true division and always returns a float.
- `//` is floor division: it divides and rounds **down** to the nearest whole number.
- `%` is modulo: the remainder.

The tricky part is negative numbers. Floor division rounds towards negative infinity, not towards zero, so `-7 // 2` is `-4`. The result of `%` always has the same sign as the divisor.

CODE

```
print(7 / 2, 7 // 2, 7 % 2)
print(-7 / 2, -7 // 2, -7 % 2)
print(divmod(17, 5))      # (quotient, remainder) in one call
```

OUTPUT

```
3.5 3 1
-3.5 -4 1
(3, 2)
```

Q23. How does Python manage memory?

FRESHERS · Memory

Python manages memory automatically through a private heap that the interpreter controls. Two mechanisms free memory you no longer use:

- **Reference counting:** every object tracks how many references point to it. When the count drops to zero, the memory is freed immediately.
- **Cyclic garbage collector:** reference counting cannot free objects that refer to each other in a cycle, so the `gc` module periodically finds and cleans up such cycles.

Small objects are allocated by an optimised allocator called `pymalloc`. As a developer you rarely manage memory yourself, but you should avoid holding references to large objects longer than needed.

CODE

```
import sys, gc

data = [1, 2, 3]
print(sys.getrefcount(data))    # includes the temporary reference from the call
alias = data
print(sys.getrefcount(data))    # one more reference now
del alias
print(sys.getrefcount(data))
print("GC enabled:", gc.isenabled())
```

OUTPUT

```
2
3
2
GC enabled: True
```

Q24. What are local and global variables? When do you need the global keyword?

FRESHERS · Functions

A variable created inside a function is **local** to it. A variable created at the top level of a module is **global**. A function can *read* a global variable freely, but if it *assigns* to that name, Python treats it as a new local variable unless you declare it with `global`. For variables in an enclosing function, use `nonlocal`.

Modifying globals makes code hard to test, so prefer returning values instead.

CODE

```
count = 0

def increment_wrong():
    try:
        count += 1
    except UnboundLocalError as e:
        print("UnboundLocalError:", e)

def increment():
    global count
    count += 1

increment_wrong()
increment()
increment()
print("count =", count)
```

OUTPUT

```
UnboundLocalError: cannot access local variable 'count' where it is not associated with a value
count = 2
```

Q25. What is the difference between a module and a package?

FRESHERS · Modules

A **module** is a single `.py` file containing functions, classes and variables. A **package** is a folder of modules, which can contain sub-packages. Traditionally a package has an `__init__.py` file, which runs when the package is imported. Since Python 3.3, folders without it can also work as "namespace packages".

A **library** is a broader term for a collection of packages, like NumPy. You import them with `import module`, `from package import module` or `import numpy as np`.

CODE

```
import math
from collections import Counter
import json as js

print(math.sqrt(144))
print(Counter("mississippi").most_common(2))
print(js.dumps({"ok": True}))
```

OUTPUT

```
12.0
[('i', 4), ('s', 4)]
{"ok": true}
```

Q26. What does `if __name__ == "__main__"` do?

FRESHERS · Modules

Every module has a built-in variable `__name__`. When you run a file directly, Python sets it to `"__main__"`. When the file is imported by another file, `__name__` is the module's name instead.

Putting code under `if __name__ == "__main__":` makes it run only when the file is executed directly, not on import. This lets one file work both as a reusable module and as a script, and it keeps test or demo code from running when someone imports your functions.

CODE

```
def calculate_gst(amount, rate=0.18):
    return round(amount * rate, 2)

if __name__ == "__main__":
    print("Running directly, __name__ is", __name__)
    print("GST on 1000:", calculate_gst(1000))
```

OUTPUT

```
Running directly, __name__ is __main__
GST on 1000: 180.0
```

Q27. How does exception handling work in Python?

FRESHERS · Exceptions

Exceptions are handled with `try`, `except`, `else` and `finally`:

- `try`: code that might fail.
- `except`: runs if a matching exception occurs. Catch specific exceptions, not a bare `except:`.
- `else`: runs only if no exception occurred.
- `finally`: always runs, used for clean-up like closing a connection.

Use `raise` to throw an exception yourself.

CODE

```
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Cannot divide by zero")
        return None
    except TypeError as e:
        print("Bad input:", e)
        return None
    else:
        print("Division successful")
        return result
    finally:
        print("Done")

print(safe_divide(10, 2))
print(safe_divide(10, 0))
```

OUTPUT

```
Division successful
Done
5.0
Cannot divide by zero
Done
None
```

Q28. How do you read and write files in Python? Why use with?

FRESHERS · Files

Use the built-in `open(path, mode)` with modes like `"r"` (read), `"w"` (write, overwrites), `"a"` (append) and `"b"` for binary. The `with` statement is a context manager that **automatically closes the file**, even if an error occurs, so you never leak file handles.

Always pass `encoding="utf-8"` for text files so behaviour is the same on Windows and Linux.

CODE

```
with open("notes.txt", "w", encoding="utf-8") as f:
    f.write("Python\nDSA\nSQL\n")

with open("notes.txt", encoding="utf-8") as f:
    for line_no, line in enumerate(f, start=1):
        print(line_no, line.strip())

print(f.closed)  # True: closed automatically
```

OUTPUT

```
1 Python
2 DSA
3 SQL
True
```

Q29. What do enumerate() and zip() do?

FRESHERS · Basics

`enumerate(iterable, start=0)` gives you the index and the item together, so you do not need a manual counter. `zip(a, b, ...)` walks through several iterables in parallel and yields tuples. It stops at the shortest one (use `itertools.zip_longest` to go to the longest, or `strict=True` in Python 3.10+ to raise an error on unequal lengths).

CODE

```
names = ["Aman", "Riya", "Kabir"]
marks = [78, 92, 85]

for rank, name in enumerate(names, start=1):
    print(rank, name)

for name, mark in zip(names, marks):
    print(f"{name}: {mark}")

print(dict(zip(names, marks)))
```

OUTPUT

```
1 Aman
2 Riya
3 Kabir
Aman: 78
Riya: 92
Kabir: 85
{'Aman': 78, 'Riya': 92, 'Kabir': 85}
```

Q30. What is range() and why is it memory efficient?

FRESHERS · Basics

`range(start, stop, step)` represents a sequence of integers. It does not store all the numbers. It calculates each value when needed, so `range(10**9)` takes the same tiny amount of memory as `range(10)`. It supports indexing, slicing, `len()` and fast membership tests.

CODE

```
import sys
r = range(0, 20, 3)
print(list(r))
print(r[2], len(r), 9 in r)
print(sys.getsizeof(range(10)), sys.getsizeof(range(10**9)))
print(sys.getsizeof(list(range(1000))))
```

OUTPUT

```
[0, 3, 6, 9, 12, 15, 18]
6 7 True
48 48
8056
```

Q31. What is the difference between a shallow copy and a deep copy?

FRESHERS · Data types

A **shallow copy** creates a new outer object but fills it with **references** to the same inner objects. A **deep copy** recursively copies everything, so the new object is completely independent. The difference only matters for nested mutable objects, like a list of lists.

Shallow copy: `list.copy()`, `list(a)`, `a[:]` or `copy.copy()`. Deep copy: `copy.deepcopy()`.

CODE

```
import copy
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0].append(99)
print("original:", original)
print("shallow: ", shallow)    # inner list is shared
print("deep:    ", deep)      # fully independent
```

OUTPUT

```
original: [[1, 2, 99], [3, 4]]
shallow:  [[1, 2, 99], [3, 4]]
deep:     [[1, 2], [3, 4]]
```

Q32. What is a docstring?

FRESHERS · Functions

A docstring is a string literal written as the first statement of a module, function, class or method. It documents what the code does and is stored in the object's `__doc__` attribute, so tools like `help()`, IDEs and documentation generators can read it. Use triple quotes, and describe the purpose, parameters and return value.

CODE

```
def bmi(weight_kg, height_m):
    """Return body mass index rounded to 1 decimal.

    weight_kg: weight in kilograms
    height_m: height in metres
    """
    return round(weight_kg / height_m ** 2, 1)

print(bmi(68, 1.72))
print(bmi.__doc__.splitlines()[0])
```

OUTPUT

```
23.0
Return body mass index rounded to 1 decimal.
```

Q33. What are namespaces and the LEGB rule?

FRESHERS · Functions

A namespace is a mapping from names to objects. When Python looks up a name, it searches four scopes in order, known as **LEGB**:

1. **Local:** inside the current function.
2. **Enclosing:** inside any outer function, for nested functions.
3. **Global:** the module level.
4. **Built-in:** names like `len` and `print`.

The first match wins. That is why shadowing a built-in, such as naming a variable `list`, breaks later code that calls `list()`.

CODE

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)
    inner()
    print(x)

outer()
print(x)
print(len("LEGB"))  # found in built-ins
```

OUTPUT

```
local
enclosing
global
4
```

Q34. How do you remove duplicates from a list while keeping the order?

FRESHERS · Coding

`set(lst)` removes duplicates but does not keep the original order. Since dictionaries preserve insertion order, `dict.fromkeys(lst)` is the cleanest one-liner that keeps order. A manual loop with a `seen` set also works and is easy to explain in an interview. Both run in $O(n)$.

CODE

```
cities = ["Pune", "Delhi", "Pune", "Surat", "Delhi", "Goa"]

print(list(dict.fromkeys(cities)))

seen, result = set(), []
for c in cities:
    if c not in seen:
        seen.add(c)
        result.append(c)
print(result)
```

OUTPUT

```
['Pune', 'Delhi', 'Surat', 'Goa']
['Pune', 'Delhi', 'Surat', 'Goa']
```

Q35. Write a program to check whether a string is a palindrome.

FRESHERS · Coding

A palindrome reads the same forwards and backwards, like "madam" or "racecar". The simplest check compares the string with its reverse. For sentences, first remove non-alphanumeric characters and ignore case. The two-pointer version uses $O(1)$ extra space and is often what interviewers want next. We cover the number version in detail in our [palindrome program in Python](#) guide.

CODE

```
def is_palindrome(s):
    clean = "".join(ch.lower() for ch in s if ch.isalnum())
    return clean == clean[::-1]

def is_palindrome_two_pointer(s):
    i, j = 0, len(s) - 1
    while i < j:
        if s[i] != s[j]:
            return False
        i += 1
        j -= 1
    return True

print(is_palindrome("A man, a plan, a canal: Panama"))
print(is_palindrome_two_pointer("racecar"), is_palindrome_two_pointer("python"))
```

OUTPUT

```
True
True False
```

Q36. How do you count the frequency of each character or word?

FRESHERS · Coding

Use [collections.Counter](#), which is a dict subclass built for counting. It also gives you [most_common\(n\)](#). You can do the same with a plain dict and [get\(\)](#), which interviewers sometimes ask for to see that you understand the logic.

CODE

```
from collections import Counter

text = "to be or not to be"
print(Counter(text.split()))
print(Counter("hello").most_common(1))

freq = {}
for ch in "banana":
    freq[ch] = freq.get(ch, 0) + 1
print(freq)
```

OUTPUT

```
Counter({'to': 2, 'be': 2, 'or': 1, 'not': 1})
[('l', 2)]
{'b': 1, 'a': 3, 'n': 2}
```

Q37. How do you swap two variables in Python?

FRESHERS · Coding

Python lets you swap in one line with tuple packing and unpacking: `a, b = b, a`. The right side builds a tuple first, then unpacks it into the names on the left, so no temporary variable is needed. Interviewers may also ask for the arithmetic or XOR trick, which only works for numbers.

CODE

```
a, b = 5, 10
a, b = b, a
print(a, b)

x, y = 3, 7
x = x + y
y = x - y
x = x - y
print(x, y)
```

OUTPUT

```
10 5
7 3
```

Q38. Find the largest and second largest number in a list.

FRESHERS · Coding

The easy way is `sorted(set(nums))`, but that is $O(n \log n)$. The single-pass approach tracks the top two values as you go, in $O(n)$ time and $O(1)$ space. Handle duplicates and lists with fewer than two distinct values.

CODE

```
def top_two(nums):
    first = second = float("-inf")
    for n in nums:
        if n > first:
            first, second = n, first
        elif first > n > second:
            second = n
    return first, (second if second != float("-inf") else None)

print(top_two([12, 45, 2, 41, 45, 30]))
print(top_two([7, 7, 7]))
```

OUTPUT

```
(45, 41)
(7, None)
```

Q39. Write a function to check whether a number is prime.

FRESHERS · Coding

A prime number is greater than 1 and divisible only by 1 and itself. You only need to test divisors up to the square root of n , because any factor larger than the square root pairs with one smaller than it. That brings the time down to $O(\sqrt{n})$. Skipping even numbers after 2 halves the work again.

CODE

```
import math

def is_prime(n):
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False
    for d in range(3, math.isqrt(n) + 1, 2):
        if n % d == 0:
            return False
    return True

print([n for n in range(1, 40) if is_prime(n)])
print(is_prime(97), is_prime(1_000_003))
```

OUTPUT

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
True True
```

Q40. Print the Fibonacci series up to n terms.

FRESHERS · Coding

Each Fibonacci number is the sum of the two before it: 0, 1, 1, 2, 3, 5, 8 and so on. The iterative solution runs in $O(n)$ time and $O(1)$ extra space. The naive recursive version is $O(2^n)$ and is a classic example of why memoisation matters (see the [lru_cache](#) question later). A generator is a neat Pythonic version.

CODE

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

print(list(fibonacci(12)))
```

OUTPUT

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Intermediate level

Iterators, generators, decorators, OOP, context managers, the collections module and coding rounds.

Q41. What is the difference between an iterable and an iterator?

INTERMEDIATE · Iterators & generators

An **iterable** is any object you can loop over, such as a list, string, dict or file. It has an `__iter__()` method that returns an iterator. An **iterator** is the object that actually produces values one at a time through `__next__()`, and raises `StopIteration` when it runs out. Iterators remember their position and can only be consumed once.

A `for` loop calls `iter()` on the iterable, then calls `next()` repeatedly until `StopIteration`.

CODE

```
nums = [10, 20, 30]          # iterable
it = iter(nums)              # iterator
print(next(it), next(it), next(it))
try:
    next(it)
except StopIteration:
    print("StopIteration: iterator exhausted")

class Countdown:
    def __init__(self, start):
        self.current = start
    def __iter__(self):
        return self
    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
        return self.current + 1

print(list(Countdown(5)))
```

OUTPUT

```
10 20 30
StopIteration: iterator exhausted
[5, 4, 3, 2, 1]
```

Q42. What is a generator and how does yield work?

INTERMEDIATE · Iterators & generators

A generator is a function that uses `yield` instead of `return`. Calling it does not run the body. It returns a generator object, which is an iterator. Each `next()` runs the function until the next `yield`, hands back that value, and **pauses**, keeping all local variables. The next call resumes right after the `yield`.

Generators produce values lazily, one at a time, so they can handle huge or even infinite sequences with almost no memory.

CODE

```
def read_in_batches(items, size):
    for i in range(0, len(items), size):
        print(f" (producing batch starting at {i})")
        yield items[i:i + size]

gen = read_in_batches(list(range(1, 8)), 3)
print(type(gen).__name__)
for batch in gen:
    print(batch)
```

OUTPUT

```
generator
(producing batch starting at 0)
[1, 2, 3]
(producing batch starting at 3)
[4, 5, 6]
(producing batch starting at 6)
[7]
```

Q43. What is the difference between a list comprehension and a generator expression?

INTERMEDIATE · Iterators & generators

They look almost identical, but a list comprehension uses square brackets and builds the **entire list in memory** right away. A generator expression uses parentheses and produces items **lazily**, one at a time, using a tiny fixed amount of memory. Use a generator expression when you only need to iterate once, for example inside `sum()`, `max()` or `any()`.

CODE

```
import sys
squares_list = [n * n for n in range(1_000_000)]
squares_gen = (n * n for n in range(1_000_000))

print("list bytes:", sys.getsizeof(squares_list))
print("generator bytes:", sys.getsizeof(squares_gen))
print(sum(n * n for n in range(1_000_000)))
```

OUTPUT

```
list bytes: 8448728
generator bytes: 200
333332833333500000
```

Q44. What is a decorator in Python?

INTERMEDIATE · Decorators

A decorator is a function that takes another function and returns a new function that adds behaviour before or after it, without changing the original code. Writing `@decorator` above a function is shorthand for `func = decorator(func)`. Decorators are used for logging, timing, authentication checks, caching and input validation. Flask's `@app.route` and Python's `@property` are decorators.

Always use `functools.wraps` inside your decorator so the wrapped function keeps its name and docstring.

CODE

```

from functools import wraps

def log_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@log_call
def total_price(price, qty, discount=0):
    return price * qty - discount

total_price(250, 3, discount=50)
print(total_price.__name__)

```

OUTPUT

```

Calling total_price with (250, 3) {'discount': 50}
total_price returned 700
total_price

```

Q45. How do you write a decorator that takes arguments?

INTERMEDIATE · Decorators

You add one more level of nesting. The outer function receives the decorator's arguments and returns the actual decorator, which then receives the function. So `@retry(times=3)` first calls `retry(times=3)`, and the result is applied to the function.

CODE

```

from functools import wraps

def retry(times):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, times + 1):
                try:
                    return func(*args, **kwargs)
                except ConnectionError as e:
                    print(f"Attempt {attempt} failed: {e}")
                    raise RuntimeError(f"Failed after {times} attempts")
            return wrapper
        return decorator

calls = {"n": 0}

@retry(times=3)
def fetch_data():
    calls["n"] += 1
    if calls["n"] < 3:
        raise ConnectionError("server busy")
    return "data received"

print(fetch_data())

```

OUTPUT

```

Attempt 1 failed: server busy
Attempt 2 failed: server busy
data received

```

Q46. What is a closure?

INTERMEDIATE · Functions

A closure is an inner function that remembers variables from the enclosing function's scope, even after the outer function has finished running. Three conditions: there is a nested function, it refers to a variable from the outer function, and the outer function returns the inner one.

Closures let you keep private state without writing a class, and they are the mechanism that makes decorators work.

CODE

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

c1 = make_counter()
c2 = make_counter()
print(c1(), c1(), c1())
print(c2())
print(c1.__closure__[0].cell_contents)
```

OUTPUT

```
1 2 3
1
3
```

Q47. Explain map(), filter() and reduce().

INTERMEDIATE · Functions

- `map(func, iterable)` applies a function to every item.
- `filter(func, iterable)` keeps only the items for which the function returns `True`.
- `reduce(func, iterable)` from `functools` combines items into a single value by applying the function cumulatively.

`map` and `filter` return lazy iterators in Python 3. In most code, list comprehensions are considered more readable than `map` and `filter` with lambdas.

CODE

```
from functools import reduce

prices = [120, 450, 80, 999, 300]
with_gst = list(map(lambda p: round(p * 1.18, 2), prices))
expensive = list(filter(lambda p: p > 200, prices))
total = reduce(lambda a, b: a + b, prices)

print(with_gst)
print(expensive)
print(total)
```

OUTPUT

```
[141.6, 531.0, 94.4, 1178.82, 354.0]
[450, 999, 300]
1949
```

Q48. What are the four pillars of OOP, and how does Python support them?

INTERMEDIATE · OOP

- **Encapsulation:** bundling data and methods in a class and restricting direct access. Python uses naming conventions (`_protected`, `__private`) and properties, not strict access modifiers.
- **Abstraction:** exposing only what the user needs and hiding details. Python provides abstract base classes through the `abc` module.
- **Inheritance:** a class can reuse and extend another class. Python supports single, multiple and multilevel inheritance.
- **Polymorphism:** the same interface works for different types, through method overriding, operator overloading with dunder methods, and duck typing.

CODE

```
class Payment:
    def pay(self, amount):
        raise NotImplementedError

class UPI(Payment):
    def pay(self, amount):
        return f"Paid Rs {amount} via UPI"

class Card(Payment):
    def pay(self, amount):
        return f"Paid Rs {amount} via card"

for method in [UPI(), Card()]:
    print(method.pay(499))    # same call, different behaviour
```

OUTPUT

```
Paid Rs 499 via UPI
Paid Rs 499 via card
```

Q49. What is the difference between class variables and instance variables?

INTERMEDIATE · OOP

A **class variable** is defined directly in the class body and is shared by all instances. An **instance variable** is set on `self`, usually in `__init__`, and each object gets its own copy. Assigning to a class variable through an instance creates a new instance variable that hides the class one, which is a common source of bugs.

CODE

```
class Student:
    college = "IIT Bombay"    # class variable
    def __init__(self, name):
        self.name = name      # instance variable

a, b = Student("Aman"), Student("Riya")
print(a.college, b.college)
Student.college = "IIT Delhi"
print(a.college, b.college)
a.college = "NIT Surat"      # creates an instance variable on a only
print(a.college, b.college, Student.college)
```

OUTPUT

```
IIT Bombay IIT Bombay
IIT Delhi IIT Delhi
NIT Surat IIT Delhi IIT Delhi
```

Q50. What are `__init__` and `self`?

INTERMEDIATE · OOP

`__init__` is the initialiser method that runs automatically right after a new object is created. It sets up the object's initial state. Strictly speaking, the constructor that creates the object is `__new__`, and `__init__` then initialises it.

`self` is the first parameter of every instance method and refers to the current object. Python passes it automatically, so `obj.method(x)` is really `Class.method(obj, x)`. The name `self` is only a convention.

CODE

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

    def deposit(self, amount):
        self.balance += amount
        return self.balance

acc = BankAccount("Kabir", 1000)
print(acc.deposit(500))
print(BankAccount.deposit(acc, 250))  # what Python does behind the scenes
```

OUTPUT

```
1500
1750
```

Q51. How does inheritance work, and what does `super()` do?

INTERMEDIATE · OOP

Inheritance lets a child class reuse the attributes and methods of a parent class and override or extend them. `super()` returns a proxy to the next class in the method resolution order, so you can call the parent's version of a method, most commonly the parent's `__init__`, without hard-coding the parent's name.

CODE

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def describe(self):
        return f"{self.name} earns {self.salary}"

class Manager(Employee):
    def __init__(self, name, salary, team_size):
        super().__init__(name, salary)
        self.team_size = team_size
    def describe(self):
        return super().describe() + f" and manages {self.team_size} people"

print(Manager("Sneha", 150000, 8).describe())
print(issubclass(Manager, Employee))
```

OUTPUT

```
Sneha earns 150000 and manages 8 people
True
```

Q52. What is the Method Resolution Order (MRO)?

INTERMEDIATE · OOP

The MRO is the order in which Python searches classes for a method or attribute, which matters with multiple inheritance. Python uses the **C3 linearisation** algorithm, which guarantees that a child comes before its parents and that the order of parents in the class definition is respected. This solves the "diamond problem", where two parents share a common grandparent. You can inspect it with `ClassName.__mro__` or `ClassName.mro()`.

CODE

```
class A:
    def hello(self): return "A"
class B(A):
    def hello(self): return "B"
class C(A):
    def hello(self): return "C"
class D(B, C):
    pass

print(D().hello())
print([cls.__name__ for cls in D.__mro__])
```

OUTPUT

```
B
['D', 'B', 'C', 'A', 'object']
```

Q53. What is the difference between instance methods, class methods and static methods?

INTERMEDIATE · OOP

- **Instance method:** takes `self` and works with a specific object's data.
- **Class method** (`@classmethod`): takes `cls`, the class itself. Often used as an alternative constructor, like `from_string`.
- **Static method** (`@staticmethod`): takes neither. It is a plain function that lives in the class because it logically belongs there.

CODE

```
class Date:
    def __init__(self, day, month, year):
        self.day, self.month, self.year = day, month, year

    def display(self):
        # instance method
        return f"{self.day:02d}/{self.month:02d}/{self.year}"

    @classmethod
    def from_string(cls, text):
        # alternative constructor
        d, m, y = map(int, text.split("-"))
        return cls(d, m, y)

    @staticmethod
    def is_leap(year):
        # utility
        return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

print(Date.from_string("17-10-2026").display())
print(Date.is_leap(2028), Date.is_leap(1900))
```

OUTPUT

```
17/10/2026
True False
```

Q54. What are magic (dunder) methods? Give examples.

INTERMEDIATE · OOP

Dunder (double underscore) methods let your classes work with Python's built-in syntax and functions. For example, `__len__` makes `len(obj)` work, `__add__` defines `+`, and `__eq__` defines `==`. `__str__` returns a readable string for end users (used by `print`), while `__repr__` returns an unambiguous developer-facing string, ideally one that could recreate the object (used in the REPL and in lists). If only `__repr__` is defined, `print` falls back to it.

CODE

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)
    def __len__(self):
        return 2
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"
    def __str__(self):
        return f"({self.x}, {self.y})"

v = Vector(1, 2) + Vector(3, 4)
print(v)
print([v])
print(v == Vector(4, 6), len(v))
```

OUTPUT

```
(4, 6)
[Vector(4, 6)]
True 2
```

Q55. How does encapsulation work in Python? What is name mangling?

INTERMEDIATE · OOP

Python has no true `private` keyword. It relies on conventions:

- `_name` (single underscore) means "internal, please do not touch". It is not enforced.
- `__name` (double underscore) triggers **name mangling**: Python renames it to `_ClassName__name`. This prevents accidental overriding in subclasses, but determined code can still reach it.

The Pythonic way to control access is with `@property`.

CODE

```
class Wallet:
    def __init__(self):
        self._currency = "INR"
        self.__balance = 500

    def balance(self):
        return self.__balance

w = Wallet()
print(w._currency)
try:
    print(w.__balance)
except AttributeError as e:
    print("AttributeError:", e)
print(w._Wallet__balance)    # name-mangled attribute
print(w.balance())
```

OUTPUT

```
INR
AttributeError: 'Wallet' object has no attribute '__balance'
500
500
```

Q56. What is an abstract class in Python?

INTERMEDIATE · OOP

An abstract class defines methods that subclasses **must** implement, and cannot be instantiated itself. In Python you create one by inheriting from [ABC](#) and marking methods with [@abstractmethod](#). If a subclass forgets to implement an abstract method, Python raises a [TypeError](#) when you try to create it. This enforces a common interface across many classes.

CODE

```
from abc import ABC, abstractmethod

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

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return round(3.14159 * self.r ** 2, 2)

print(Circle(3).area())
try:
    Shape()
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
28.27
TypeError: Can't instantiate abstract class Shape without an implementation for abstract method 'area'
```

Q57. What does the @property decorator do?

INTERMEDIATE · OOP

[@property](#) lets you access a method like an attribute, so you can add validation or computed values without changing how callers use the class. Combine it with [@name.setter](#) to control assignment. This is the Pythonic replacement for Java-style [get_x\(\)](#) and [set_x\(\)](#) methods.

CODE

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price          # goes through the setter

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
        self._price = value

    @property
    def price_with_gst(self):
        return round(self._price * 1.18, 2)

p = Product("Keyboard", 1500)
print(p.price, p.price_with_gst)
try:
    p.price = -10
except ValueError as e:
    print("ValueError:", e)
```

OUTPUT

```
1500 1770.0
ValueError: Price cannot be negative
```

Q58. What is duck typing?

INTERMEDIATE · OOP

"If it walks like a duck and quacks like a duck, it is a duck." Python does not check an object's type before calling a method. It only cares whether the object has the method you call. Any object with a [read\(\)](#) method can be used where a file is expected, for example. This makes code flexible and is a form of polymorphism that does not need inheritance.

CODE

```
class PDFReport:
    def export(self):
        return "Exported as PDF"

class ExcelReport:
    def export(self):
        return "Exported as Excel"

def download(report):
    print(report.export())    # no type check, only the method matters

download(PDFReport())
download(ExcelReport())
```

OUTPUT

Q59. How do you create and raise a custom exception?

INTERMEDIATE · Exceptions

Create a class that inherits from `Exception` (or a more specific built-in exception), then `raise` it. Custom exceptions make errors self-explanatory and let callers catch your specific errors separately. You can add extra attributes and use `raise ... from ...` to keep the original cause.

CODE

```
class InsufficientBalanceError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Need Rs {amount}, but balance is Rs {balance}")

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientBalanceError(balance, amount)
    return balance - amount

try:
    withdraw(1000, 2500)
except InsufficientBalanceError as e:
    print("Error:", e)
    print("Short by:", e.amount - e.balance)
```

OUTPUT

```
Error: Need Rs 2500, but balance is Rs 1000
Short by: 1500
```

Q60. What is a context manager and how do you create one?

INTERMEDIATE · Context managers

A context manager sets something up when a `with` block starts and guarantees clean-up when it ends, even if an exception is raised. Files, locks and database connections are typical examples. You can build one with a class that defines `__enter__` and `__exit__`, or more simply with the `@contextmanager` decorator from `contextlib`, where the code before `yield` is setup and the code after it is clean-up.

CODE

```

from contextlib import contextmanager
import time

class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self
    def __exit__(self, exc_type, exc, tb):
        print("Block finished, exception:", exc_type)
        return False # do not suppress exceptions

@contextmanager
def connection(name):
    print(f"Opening {name}")
    try:
        yield name.upper()
    finally:
        print(f"Closing {name}")

with Timer():
    sum(range(1000))

with connection("db") as conn:
    print("Using", conn)

```

OUTPUT

```

Block finished, exception: None
Opening db
Using DB
Closing db

```

Q61. What is the difference between sort() and sorted()? INTERMEDIATE · Data types

INTERMEDIATE · Data types

- `list.sort()` sorts the list **in place** and returns `None`. It only exists on lists.
- `sorted(iterable)` returns a **new** sorted list and works on any iterable, including tuples, strings and dicts.

Both accept `key` and `reverse`, and both use Timsort, which runs in $O(n \log n)$ and is **stable**: items that compare equal keep their original order. Stability lets you sort by several keys in steps.

CODE

```

marks = [("Aman", 85), ("Riya", 92), ("Kabir", 85), ("Sneha", 78)]

by_marks = sorted(marks, key=lambda s: s[1], reverse=True)
print(by_marks) # Aman stays before Kabir (stable)

nums = [5, 2, 9]
print(nums.sort()) # returns None
print(nums)
print(sorted("python"))
print(sorted(marks, key=lambda s: (-s[1], s[0]))) # marks desc, then name

```

OUTPUT

```

[('Riya', 92), ('Aman', 85), ('Kabir', 85), ('Sneha', 78)]
None
[2, 5, 9]
['h', 'n', 'o', 'p', 't', 'y']
[('Riya', 92), ('Aman', 85), ('Kabir', 85), ('Sneha', 78)]

```

Q62. How do get(), setdefault() and defaultdict help with dictionaries?

INTERMEDIATE · Data types

Accessing a missing key with `d[key]` raises a `KeyError`. Three ways to handle that:

- `d.get(key, default)` returns a default without changing the dict.
- `d.setdefault(key, default)` returns the value, inserting the default first if the key is missing.
- `collections.defaultdict(factory)` automatically creates a default value for any missing key, which is ideal for grouping.

CODE

```
from collections import defaultdict
```

```
stock = {"apple": 10}
print(stock.get("mango", 0))
```

```
groups = {}
groups.setdefault("A", []).append("Aman")
print(groups)
```

```
by_branch = defaultdict(list)
for name, branch in [("Riya", "CSE"), ("Kabir", "ECE"), ("Sneha", "CSE")]:
    by_branch[branch].append(name)
print(dict(by_branch))
```

OUTPUT

```
0
{'A': ['Aman']}
{'CSE': ['Riya', 'Sneha'], 'ECE': ['Kabir']}
```

Q63. What are namedtuple, deque and Counter in the collections module?

INTERMEDIATE · Data types

- `namedtuple`: a tuple with named fields, so `p.x` is clearer than `p[0]`. Lightweight and immutable.
- `deque`: a double-ended queue with $O(1)$ appends and pops at both ends. A list is $O(n)$ for `pop(0)`. Use it for queues and BFS.
- `Counter`: counts hashable items, with `most_common()` and arithmetic between counters.
- Others worth knowing: `defaultdict`, `OrderedDict` and `ChainMap`.

CODE

```
from collections import namedtuple, deque, Counter
```

```
Point = namedtuple("Point", ["x", "y"])
p = Point(3, 4)
print(p.x, p.y, p)
```

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

```
print(Counter("aabbbc") + Counter("abc"))
```

OUTPUT

```
3 4 Point(x=3, y=4)
0 deque([1, 2, 3, 4])
Counter({'b': 4, 'a': 3, 'c': 2})
```

Q64. What is unpacking, and what is the walrus operator?

INTERMEDIATE · Basics

Unpacking assigns items of an iterable to several names in one line. A starred name collects "the rest" into a list. The **walrus operator** `:=` (Python 3.8+) assigns a value as part of an expression, which avoids computing something twice, for example inside a `while` condition or a comprehension.

CODE

```
first, *middle, last = [10, 20, 30, 40, 50]
print(first, middle, last)

name, (city, pin) = ("Riya", ("Pune", 411001))
print(name, city, pin)

data = [3, 8, 1, 9, 4]
if (n := len(data)) > 3:
    print(f"List is long ({n} items)")

values = [y for x in data if (y := x * 10) > 30]
print(values)
```

OUTPUT

```
10 [20, 30, 40] 50
Riya Pune 411001
List is long (5 items)
[80, 90, 40]
```

Q65. What is a frozenset, and what makes an object hashable?

INTERMEDIATE · Data types

A `frozenset` is an immutable version of a set. Because it cannot change, it is hashable, so it can be a dictionary key or an element of another set, which a normal set cannot.

An object is **hashable** if it has a hash value that never changes during its lifetime (`__hash__`) and can be compared with `__eq__`. Equal objects must have equal hashes. All immutable built-ins are hashable; tuples are hashable only if everything inside them is.

CODE

```
skills = frozenset({"python", "sql"})
teams = {skills: "Data team"}
print(teams[frozenset({"sql", "python"})])

print(hash((1, 2)) == hash((1, 2)))
try:
    hash((1, [2, 3]))
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
Data team
True
TypeError: unhashable type: 'list'
```

Q66. What are f-strings, and how do you format numbers with them?

INTERMEDIATE · Strings

f-strings (Python 3.6+) embed expressions directly inside string literals: `f"Hello {name}"`. They are faster and more readable than `%` formatting or `str.format()`. A format specifier after a colon controls decimals, padding, alignment and separators. Adding `=` prints the expression too, which is handy for debugging.

CODE

```
name, salary, score = "Riya", 1250000.5, 0.8734
print(f"{name} earns Rs {salary:,.2f}")
print(f"Accuracy: {score:.1%}")
print(f"|{name:<8}|{name:>8}|{name:^8}|")
print(f"{42:05d} {255:x} {255:b}")
x = 7
print(f"{x * 6 = }")
```

OUTPUT

```
Riya earns Rs 1,250,000.50
Accuracy: 87.3%
|Riya   |   Riya|  Riya  |
00042 ff 11111111
x * 6 = 42
```

Q67. What are type hints? Does Python enforce them?

INTERMEDIATE · Basics

Type hints (PEP 484) annotate the expected types of variables, parameters and return values, like `def area(r: float) -> float`. Python does **not** enforce them at runtime: a wrong type still runs. Their value comes from tools: IDE autocompletion, and static checkers like mypy or Pyright that catch type errors before the code runs. Large codebases rely on them heavily.

CODE

```
def average(marks: list[float]) -> float:
    return sum(marks) / len(marks)

print(average([80, 90, 70]))
print(average.__annotations__)

def double(x: int) -> int:
    return x * 2

print(double("ab")) # a str is passed, Python runs it anyway
```

OUTPUT

```
80.0
{'marks': list[float], 'return': <class 'float'>}
abab
```

Q68. What is pickling and unpickling?

INTERMEDIATE · Files

Pickling converts a Python object into a byte stream (serialisation) with `pickle.dump` or `pickle.dumps`, so it can be saved to disk or sent over a network. **Unpickling** rebuilds the object with `pickle.load` or `pickle.loads`. It handles almost any Python object, including custom classes, and is often used to save trained ML models.

Never unpickle data from an untrusted source: a crafted pickle can execute arbitrary code. Use JSON for data exchange with other systems.

CODE

```
import pickle

model_config = {"model": "logistic", "C": 1.0, "features": ["age", "income"]}
blob = pickle.dumps(model_config)
print(type(blob).__name__, len(blob), "bytes")

restored = pickle.loads(blob)
print(restored == model_config, restored)
```

OUTPUT

```
bytes 78 bytes
True {'model': 'logistic', 'C': 1.0, 'features': ['age', 'income']}
```

Q69. How do you work with JSON in Python?

INTERMEDIATE · Files

The built-in `json` module converts between JSON text and Python objects. `json.dumps` turns an object into a JSON string and `json.loads` parses a string. `json.dump` and `json.load` do the same with files. JSON objects map to dicts, arrays to lists, `true/false` to `True/False`, and `null` to `None`.

CODE

```
import json

api_response = '{"user": "riya", "active": true, "skills": ["python", "sql"], "manager": null}'
data = json.loads(api_response)
print(data["skills"], data["active"], data["manager"])

data["skills"].append("pandas")
print(json.dumps(data, indent=2))
```

OUTPUT

```
['python', 'sql'] True None
{
  "user": "riya",
  "active": true,
  "skills": [
    "python",
    "sql",
    "pandas"
  ],
  "manager": null
}
```

Q70. How would you process a file that is too large to fit in memory?

INTERMEDIATE · Files

Never call `read()` or `readlines()` on a huge file, because they load everything at once. Instead:

- Iterate over the file object directly: it reads **one line at a time**.
- Wrap the processing in generators so each step handles one record at a time.
- For binary files, read fixed-size chunks with `f.read(chunk_size)`.
- For CSVs, use `pandas.read_csv(..., chunksize=100_000)`.

CODE

```

with open("sales.log", "w", encoding="utf-8") as f:
    for i in range(1, 10001):
        f.write(f"order_{i},{i % 7 * 100}\n")

def read_amounts(path):
    with open(path, encoding="utf-8") as f:
        for line in f:
            # one line in memory at a time
            yield int(line.strip().split(",")[1])

total = sum(read_amounts("sales.log"))
big_orders = sum(1 for amt in read_amounts("sales.log") if amt >= 500)
print(total, big_orders)

```

OUTPUT

2999800 2856

Q71. What is a virtual environment, and why should you use one?

INTERMEDIATE · Modules

A virtual environment is an isolated folder with its own Python interpreter and installed packages. Each project gets its own versions of libraries, so project A can use Django 4 while project B uses Django 5 without conflicts, and you never pollute the system Python.

Typical workflow:

- `python -m venv .venv` to create it.
- `source .venv/bin/activate` on Linux and macOS, or `.venv\Scripts\activate` on Windows.
- `pip install -r requirements.txt` to install dependencies.
- `pip freeze > requirements.txt` to record exact versions.

Tools like `poetry`, `pipenv`, `conda` and `uv` build on the same idea.

CODE

```

import sys
print("Running inside a venv:", sys.prefix != sys.base_prefix)

```

OUTPUT

Running inside a venv: False

Q72. Two Sum: find two numbers in a list that add up to a target.

INTERMEDIATE · Coding

The brute-force approach checks every pair in $O(n^2)$. The optimal approach uses a dictionary that maps each number to its index. For each number, check whether `target - number` has already been seen. That gives $O(n)$ time and $O(n)$ space, and it is one of the most asked coding questions in Python interviews.

CODE

```

def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        need = target - num
        if need in seen:
            return [seen[need], i]
        seen[num] = i
    return []

print(two_sum([2, 7, 11, 15], 9))
print(two_sum([3, 2, 4], 6))

```

OUTPUT

```
[0, 1]
[1, 2]
```

Q73. Check whether two strings are anagrams.

INTERMEDIATE · Coding

Two strings are anagrams if they contain the same characters with the same counts, like "listen" and "silent". Sorting both strings works in $O(n \log n)$. Comparing two [Counter](#) objects works in $O(n)$. See our [Valid Anagram](#) walkthrough for more approaches.

CODE

```
from collections import Counter

def is_anagram(a, b):
    return Counter(a) == Counter(b)

def is_anagram_sorted(a, b):
    return sorted(a) == sorted(b)

print(is_anagram("listen", "silent"), is_anagram("rat", "car"))
print(is_anagram_sorted("dusty", "study"))
```

OUTPUT

```
True False
True
```

Q74. Flatten a nested list of any depth.

INTERMEDIATE · Coding

Recursion fits naturally: walk through each item, and if it is a list, flatten it too, otherwise yield it. A generator with [yield from](#) keeps the code short and memory-friendly. Interviewers may follow up by asking for an iterative version using a stack, which avoids Python's recursion limit on very deep nesting.

CODE

```
def flatten(items):
    for item in items:
        if isinstance(item, list):
            yield from flatten(item)
        else:
            yield item

nested = [1, [2, [3, [4, 5]], 6], [[7]], 8]
print(list(flatten(nested)))

def flatten_iterative(items):
    stack, result = [items], []
    while stack:
        current = stack.pop()
        if isinstance(current, list):
            stack.extend(reversed(current))
        else:
            result.append(current)
    return result

print(flatten_iterative(nested))
```

OUTPUT

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Q75. Find the missing number in a list containing 0 to n.

INTERMEDIATE · Coding

The list has n distinct numbers taken from 0 to n, so exactly one is missing. The sum of 0 to n is $n(n+1)/2$, so the missing number is that total minus the actual sum: O(n) time and O(1) space. The XOR trick avoids any risk of overflow in other languages. We solve this step by step in our [Missing Number \(LeetCode 268\)](#) post.

CODE

```
def missing_number(nums):
    n = len(nums)
    return n * (n + 1) // 2 - sum(nums)

def missing_number_xor(nums):
    result = len(nums)
    for i, num in enumerate(nums):
        result ^= i ^ num
    return result

print(missing_number([3, 0, 1]))
print(missing_number_xor([9, 6, 4, 2, 3, 5, 7, 0, 1]))
```

OUTPUT

```
2
8
```

Experienced level

Concurrency, memory management, advanced OOP, performance, testing, modern Python and data libraries.

Q76. What is the GIL (Global Interpreter Lock)?

EXPERIENCED · Concurrency

The GIL is a lock in CPython that allows only **one thread to execute Python bytecode at a time**, even on a multi-core machine. It exists because CPython's memory management (reference counting) is not thread-safe, and a single lock was a simple, fast way to protect it.

- For **I/O-bound** work (network calls, file and database access), threads still help, because a thread releases the GIL while it waits.
- For **CPU-bound** work (heavy calculations), threads do not run in parallel, so use [multiprocessing](#), or libraries like NumPy that do the work in C and release the GIL.

Python 3.13 introduced an experimental "free-threaded" build (PEP 703) that can run without the GIL. It is optional and not yet the default, so the standard answer still applies to most production code.

CODE

```
import sys
print(sys.version.split()[0])
check = getattr(sys, "_is_gil_enabled", None)
print("GIL enabled:", check() if check else True)
```

OUTPUT

```
3.13.16
GIL enabled: True
```

Q77. When would you use threading, multiprocessing or asyncio?

EXPERIENCED · Concurrency

- **threading:** I/O-bound tasks with a moderate number of concurrent operations, like downloading 20 files. Threads share memory, so protect shared data with locks.
- **multiprocessing:** CPU-bound tasks, like processing images or crunching numbers. Each process has its own interpreter and its own GIL, so work truly runs in parallel on several cores. The cost is higher memory use and data must be pickled to move between processes.
- **asyncio:** very large numbers of I/O-bound tasks, like thousands of network requests or websocket connections, in a single thread with an event loop. Requires async-aware libraries ([aiohttp](#), [asyncpg](#)).

[concurrent.futures](#) gives the same simple API ([ThreadPoolExecutor](#), [ProcessPoolExecutor](#)) for the first two.

CODE

```

from concurrent.futures import ThreadPoolExecutor
import time

def fake_download(n):
    time.sleep(0.2)          # simulates waiting on the network
    return f"file_{n} done"

start = time.perf_counter()
with ThreadPoolExecutor(max_workers=5) as pool:
    results = list(pool.map(fake_download, range(5)))
elapsed = time.perf_counter() - start
print(results)
print("5 downloads finished in under 0.5 s:", elapsed < 0.5)

```

OUTPUT

```

['file_0 done', 'file_1 done', 'file_2 done', 'file_3 done', 'file_4 done']
5 downloads finished in under 0.5 s: True

```

Q78. How does async and await work? Give an example.

EXPERIENCED · Concurrency

`async def` defines a **coroutine**. Calling it creates a coroutine object, and nothing runs until it is awaited or scheduled on the event loop. `await` pauses the current coroutine until the awaited operation finishes, and while it waits, the event loop runs other coroutines. `asyncio.gather` runs several coroutines concurrently and returns their results in the order you passed them, not the order they finish.

CODE

```

import asyncio

async def fetch(name, delay):
    print(f"start {name}")
    await asyncio.sleep(delay)      # non-blocking wait
    print(f"end {name}")
    return f"{name}:{delay}s"

async def main():
    results = await asyncio.gather(
        fetch("users", 0.3), fetch("orders", 0.1), fetch("stock", 0.2))
    print(results)

asyncio.run(main())

```

OUTPUT

```

start users
start orders
start stock
end orders
end stock
end users
['users:0.3s', 'orders:0.1s', 'stock:0.2s']

```

Q79. What is a race condition and how do you prevent it in Python?

EXPERIENCED · Concurrency

A race condition happens when several threads read and modify shared data at the same time, and the result depends on unpredictable timing. Even `counter += 1` is not atomic: it is a read, an add and a write, and a thread switch can happen between them. Protect shared state with a `threading.Lock`, ideally used as a context manager, or avoid shared state entirely by using a `queue.Queue`.

CODE

```
import threading

balance = 0
lock = threading.Lock()

def deposit_many(times):
    global balance
    for _ in range(times):
        with lock:
            # only one thread updates at a time
            balance += 1

threads = [threading.Thread(target=deposit_many, args=(100_000,)) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print("Final balance:", balance)
```

OUTPUT

```
Final balance: 400000
```

Q80. How does Python's garbage collector handle reference cycles?

EXPERIENCED · Memory

Reference counting frees an object as soon as nothing refers to it, but two objects that refer to each other keep each other's count above zero forever. CPython's **generational garbage collector** (the `gc` module) periodically scans container objects, finds groups that are only reachable from each other, and frees them. Objects are grouped into generations: new objects are checked often, and objects that survive several collections are checked less often, since they tend to live long.

To avoid creating cycles in the first place, for example parent-child links or caches, use the `weakref` module.

CODE

```
import gc

class Node:
    def __init__(self, name):
        self.name = name
        self.partner = None

gc.collect()
a, b = Node("a"), Node("b")
a.partner, b.partner = b, a      # reference cycle
del a, b                        # refcounts never reach zero
print("Unreachable objects found:", gc.collect() > 0)
print("Thresholds:", gc.get_threshold())
```

OUTPUT

```
Unreachable objects found: True
Thresholds: (2000, 10, 10)
```

Q81. What are `__slots__` and when should you use them?

EXPERIENCED · Memory

By default every instance stores its attributes in a per-object `__dict__`, which is flexible but costs memory. Declaring `__slots__ = ("x", "y")` tells Python to reserve fixed space for exactly those attributes and skip the `__dict__`. Instances use noticeably less memory and attribute access is slightly faster. The trade-off is that you cannot add new attributes dynamically. Use it when you create millions of small objects. `@dataclass(slots=True)` does this for you in Python 3.10+.

CODE

```
import sys

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

class PointSlots:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x, self.y = x, y

a, b = PointDict(1, 2), PointSlots(1, 2)
print("with __dict__: ", sys.getsizeof(a) + sys.getsizeof(a.__dict__), "bytes")
print("with __slots__: ", sys.getsizeof(b), "bytes")
try:
    b.z = 3
except AttributeError as e:
    print("AttributeError:", e)
```

OUTPUT

```
with __dict__: 344 bytes
with __slots__: 48 bytes
AttributeError: 'PointSlots' object has no attribute 'z' and no __dict__ for setting new attributes
```

Q82. What is a metaclass?

EXPERIENCED · Advanced OOP

In Python, classes are objects too, and a metaclass is the "class of a class": it controls how classes are created. The default metaclass is `type`. A custom metaclass (inheriting from `type` and overriding `__new__` or `__init__`) can automatically register classes, enforce naming rules, or add methods. Django models and SQLAlchemy use metaclasses internally.

They are powerful but rarely needed in application code. Class decorators and `__init_subclass__` usually solve the same problems more simply, and saying so is a good interview answer.

CODE

```
class PluginRegistry(type):
    plugins = {}
    def __new__(mcs, name, bases, namespace):
        cls = super().__new__(mcs, name, bases, namespace)
        if bases:
            # skip the base class itself
            mcs.plugins[name.lower()] = cls
        return cls

class Plugin(metaclass=PluginRegistry):
    pass

class CsvExporter(Plugin): pass
class PdfExporter(Plugin): pass

print(type(Plugin))
print(list(PluginRegistry.plugins))
```

OUTPUT

```
<class '__main__.PluginRegistry'>
['csvexporter', 'pdfexporter']
```

Q83. What is a descriptor?

EXPERIENCED · Advanced OOP

A descriptor is an object that defines `__get__`, `__set__` or `__delete__` and is stored as a class attribute. When you access that attribute on an instance, Python calls the descriptor's methods instead of reading a plain value. Descriptors are the machinery behind `@property`, `@classmethod`, `@staticmethod` and ORM fields. They are useful for reusable validation across many attributes.

CODE

```
class Positive:
    def __set_name__(self, owner, name):
        self.name = "_" + name
    def __get__(self, obj, objtype=None):
        return getattr(obj, self.name)
    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.name[1:]} must be positive")
        setattr(obj, self.name, value)

class Order:
    quantity = Positive()
    price = Positive()
    def __init__(self, quantity, price):
        self.quantity, self.price = quantity, price

o = Order(3, 499)
print(o.quantity * o.price)
try:
    o.price = -5
except ValueError as e:
    print("ValueError:", e)
```

OUTPUT

```
1497
ValueError: price must be positive
```

Q84. How are Python dictionaries implemented internally?

EXPERIENCED · Performance

A dict is a **hash table**. When you insert a key, Python computes `hash(key)`, uses it to pick a slot, and stores the key, value and hash there. On a collision (two keys mapping to the same slot), CPython uses **open addressing**: it probes other slots in a pseudo-random sequence until it finds a free one. When the table gets about two-thirds full, it is resized to keep lookups fast.

Since Python 3.6, CPython stores entries in a compact array in insertion order, with a separate sparse index table pointing into it. That saves memory and is why dicts preserve insertion order (guaranteed from 3.7). Average lookup, insert and delete are $O(1)$. The worst case is $O(n)$ if many keys collide.

CODE

```
print(hash("python") == hash("python"))
print(hash(42), hash(True), hash(1.0) == hash(1))
d = {1: "int", True: "bool"}      # 1 == True and same hash, so same key
print(d)
```

OUTPUT

```
True
42 1 True
{1: 'bool'}
```

Q85. What is the time complexity of common list, dict and set operations?

EXPERIENCED · Performance

Knowing these is essential for writing efficient code and for coding interviews. For the basics of Big O, see our [time and space complexity guide](#).

Operation	list	dict / set
Access by index or key	O(1)	O(1) average
Membership test (<code>x in c</code>)	O(n)	O(1) average
Append / add	O(1) amortised	O(1) average
Insert or delete at the start	O(n)	n/a
Pop from the end	O(1)	n/a
Delete by key	n/a	O(1) average
Sort	O(n log n)	n/a
Iterate	O(n)	O(n)

The classic interview follow-up: if you check `x in my_list` inside a loop, convert the list to a set first.

CODE

```
import timeit
data_list = list(range(100_000))
data_set = set(data_list)
t_list = timeit.timeit(lambda: 99_999 in data_list, number=200)
t_set = timeit.timeit(lambda: 99_999 in data_set, number=200)
print("set lookup is faster:", t_set < t_list)
```

OUTPUT

```
set lookup is faster: True
```

Q86. What is the late binding closure problem?

EXPERIENCED · Functions

Closures look up variables when the inner function is **called**, not when it is created. So functions created in a loop all see the loop variable's final value. This trips up even experienced developers. Fix it by binding the current value as a default argument, or by using [functools.partial](#).

CODE

```
multipliers = [lambda x: x * i for i in range(1, 4)]
print([f(10) for f in multipliers])          # all use i = 3

fixed = [lambda x, i=i: x * i for i in range(1, 4)]
print([f(10) for f in fixed])
```

OUTPUT

```
[30, 30, 30]
[10, 20, 30]
```

Q87. What is integer caching and string interning?

EXPERIENCED · Memory

CPython pre-creates and reuses integer objects from **-5 to 256**, so any two variables holding, say, 100 point to the same object. Larger integers created at runtime are usually separate objects. Similarly, short strings that look like identifiers are often **interned** (stored once and reused), and you can force it with `sys.intern`.

These are implementation details. The lesson for interviews: never use `is` to compare numbers or strings. Use `==`.

CODE

```
a = int("256"); b = int("256")
print(a is b)          # cached small integer
c = int("257"); d = int("257")
print(c is d)          # separate objects
print(c == d)          # always compare values with ==

import sys
s1 = sys.intern("code and debug")
s2 = sys.intern("code and debug")
print(s1 is s2)
```

OUTPUT

```
True
False
True
True
```

Q88. What does `functools.lru_cache` do?

EXPERIENCED · Performance

`@lru_cache` memoises a function: it stores results for arguments it has already seen and returns the cached result instead of recomputing. LRU means Least Recently Used: when the cache is full (`maxsize`), the oldest unused entries are dropped. `@cache` (Python 3.9+) is the unbounded version. The arguments must be hashable. It turns the naive recursive Fibonacci from exponential time into linear time.

CODE

```
from functools import lru_cache

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

print(fib(100))
print(fib.cache_info())
```

OUTPUT

```
354224848179261915075
CacheInfo(hits=98, misses=101, maxsize=None, cursize=101)
```

Q89. Name some useful functions from the itertools module.

EXPERIENCED · Performance

`itertools` provides fast, memory-efficient building blocks that return iterators:

- `chain`: join several iterables end to end.
- `islice`: slice any iterator, even an infinite one.
- `groupby`: group consecutive items with the same key. Sort by that key first.
- `combinations`, `permutations`, `product`: combinatorics without nested loops.
- `accumulate`: running totals.
- `count`, `cycle`, `repeat`: infinite iterators.

CODE

```
from itertools import chain, groupby, combinations, accumulate, islice, count

print(list(chain([1, 2], (3, 4), "ab")))
print(list(combinations("ABC", 2)))
print(list(accumulate([100, 200, 50])))
print(list(islice(count(10, 5), 4)))

sales = [("Pune", 300), ("Pune", 200), ("Surat", 150)]
for city, rows in groupby(sales, key=lambda r: r[0]):
    print(city, sum(amount for _, amount in rows))
```

OUTPUT

```
[1, 2, 3, 4, 'a', 'b']
[('A', 'B'), ('A', 'C'), ('B', 'C')]
[100, 300, 350]
[10, 15, 20, 25]
Pune 500
Surat 150
```

Q90. What are dataclasses?

EXPERIENCED · Advanced OOP

The `@dataclass` decorator (Python 3.7+) generates `__init__`, `__repr__` and `__eq__` automatically from the class's type-annotated fields, which removes a lot of boilerplate for classes that mainly hold data. Options include `frozen=True` for immutable, hashable instances, `order=True` to generate comparison methods, and `slots=True` for lower memory use. Use `field(default_factory=list)` for mutable defaults.

CODE

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Student:
    cgpa: float
    name: str
    skills: list = field(default_factory=list)

a = Student(8.7, "Riya", ["python"])
b = Student(9.1, "Kabir")
print(a)
print(a == Student(8.7, "Riya", ["python"]))
print(max(a, b).name)
```

OUTPUT

```
Student(cgpa=8.7, name='Riya', skills=['python'])
True
Kabir
```

Q91. What do send() and yield from do in generators?

EXPERIENCED · Iterators & generators

`yield from iterable` delegates to another generator or iterable, yielding all of its values, which makes it easy to compose generators. `send(value)` resumes a paused generator and makes the current `yield` expression evaluate to `value`, so data flows *into* the generator. This two-way communication is what Python's coroutines were originally built on. Prime a generator with `next()` before the first `send()`.

CODE

```
def running_average():
    total, count, average = 0, 0, None
    while True:
        value = yield average
        total += value
        count += 1
        average = total / count

avg = running_average()
next(avg)          # prime the generator
for mark in [80, 90, 70]:
    print(avg.send(mark))

def all_items():
    yield from ["a", "b"]
    yield from range(3)

print(list(all_items()))
```

OUTPUT

```
80.0
85.0
80.0
['a', 'b', 0, 1, 2]
```

Q92. What is structural pattern matching (match-case)?

EXPERIENCED · Modern Python

Introduced in Python 3.10, `match` compares a value against **patterns**, not just equal values. It can unpack sequences and dictionaries, check types, bind variables and add `if` guards, which makes parsers, command handlers and API response handling much cleaner than long `if/elif` chains. The underscore `_` is the wildcard that matches anything.

CODE

```
def handle(event):
    match event:
        case {"type": "click", "x": x, "y": y}:
            return f"Click at ({x}, {y})"
        case {"type": "key", "key": "q" | "Q"}:
            return "Quit"
        case [first, *rest]:
            return f"List starting with {first}, {len(rest)} more"
        case int(n) if n < 0:
            return "Negative number"
        case _:
            return "Unknown event"

print(handle({"type": "click", "x": 10, "y": 20}))
print(handle({"type": "key", "key": "Q"}))
print(handle([5, 6, 7]))
print(handle(-3), "|", handle(3.5))
```

OUTPUT

```
Click at (10, 20)
Quit
List starting with 5, 2 more
Negative number | Unknown event
```

Q93. What are some important features added in recent Python versions?

EXPERIENCED · Modern Python

- **3.8:** walrus operator `:=`, positional-only parameters with `/`, and `f"{x=}"` debugging.
- **3.9:** dict merge operators `|` and `|=`, built-in generics like `list[int]`, and `str.removeprefix()`.
- **3.10:** structural pattern matching (`match`), the `X | Y` union type syntax, and clearer error messages.
- **3.11:** large interpreter speed-ups, exception groups with `except*`, and `tomllib`.
- **3.12:** new type parameter syntax (`def first[T](items: list[T]) -> T`), the `type` statement, and more flexible f-strings.
- **3.13:** a new interactive REPL, an experimental free-threaded build without the GIL, and an experimental JIT compiler.

CODE

```
defaults = {"theme": "light", "lang": "en"}
user = {"theme": "dark"}
print(defaults | user)                                # 3.9 dict merge

def first[T](items: list[T]) -> T:                    # 3.12 generics syntax
    return items[0]
print(first(["python", "sql"]))

print("report_final.pdf".removeprefix("report_"))
value = 42
print(f"{value = }")
```

OUTPUT

```
{'theme': 'dark', 'lang': 'en'}
python
final.pdf
value = 42
```

Q94. How do you write unit tests in Python?

EXPERIENCED · Testing

The two main options are the built-in `unittest` module and the third-party **pytest**, which most teams prefer because tests are plain functions with plain `assert` statements, and it has powerful fixtures and parametrisation. A good test is fast, independent, and checks one behaviour, including edge cases and expected errors. Use `unittest.mock` to replace external dependencies.

With `pytest` you would write `def test_discount(): assert apply_discount(1000, 10) == 900` in a file named `test_*.py` and run `pytest`. The same idea with `unittest`:

CODE

```
import unittest

def apply_discount(price, percent):
    if not 0 <= percent <= 100:
        raise ValueError("percent must be between 0 and 100")
    return price * (100 - percent) / 100

class TestDiscount(unittest.TestCase):
    def test_normal(self):
        self.assertEqual(apply_discount(1000, 10), 900)
    def test_zero(self):
        self.assertEqual(apply_discount(500, 0), 500)
    def test_invalid(self):
        with self.assertRaises(ValueError):
            apply_discount(500, 120)

suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestDiscount)
result = unittest.TextTestRunner(verbosity=0).run(suite)
print("Tests run:", result.testsRun, "| Failures:", len(result.failures))
```

OUTPUT

```
Tests run: 3 | Failures: 0
```

```
-----
Ran 3 tests in 0.000s
```

```
OK
```

Q95. How do you find and fix slow Python code?

EXPERIENCED · Performance

Measure first, then optimise. Guessing usually targets the wrong code.

1. **Profile:** `python -m cProfile -s cumtime app.py` shows which functions take the most time. `line_profiler` goes line by line, and `tracemalloc` tracks memory.
2. **Benchmark** small alternatives with `timeit`.
3. **Fix the algorithm first:** a set instead of a list for lookups, or avoiding nested loops, beats any micro-optimisation.
4. **Use built-ins and libraries:** `sum`, `sorted`, comprehensions and NumPy run in C.
5. **Cache** repeated work with `lru_cache`.
6. **Go parallel** with multiprocessing for CPU-bound work, or use PyPy, Cython or Numba for hot loops.

CODE

```
import cProfile, pstats, io

def slow_unique(items):
    result = []
    for x in items:
        if x not in result:      # O(n) check inside a loop: O(n^2) total
            result.append(x)
    return result

profiler = cProfile.Profile()
profiler.enable()
slow_unique(list(range(3000)) * 2)
profiler.disable()

stats = pstats.Stats(profiler, stream=io.StringIO()).sort_stats("tottime")
top = stats.fcn_list[0]
print("Function using the most time:", top[2])
```

OUTPUT

Q96. Is Python pass-by-value or pass-by-reference?

EXPERIENCED · Functions

Neither, exactly. Python uses **pass-by-object-reference** (also called call-by-sharing). The function receives a reference to the same object the caller passed. What happens next depends on mutability:

- If you **mutate** a mutable object (append to a list), the caller sees the change.
- If you **rebind** the parameter to a new object (`x = x + 1` or `lst = [...]`), only the local name changes, and the caller's variable is unaffected.

CODE

```
def modify(nums, count):
    nums.append(4)      # mutates the caller's list
    count += 1          # rebinds a local name only
    nums = [0]          # rebinds, caller unaffected

my_list, my_count = [1, 2, 3], 10
modify(my_list, my_count)
print(my_list, my_count)
```

OUTPUT

```
[1, 2, 3, 4] 10
```

Q97. Why is NumPy faster than Python lists for numerical work?

EXPERIENCED · Data (NumPy & Pandas)

A Python list stores pointers to separate Python objects scattered in memory, and every operation goes through the interpreter one element at a time. A NumPy array stores raw numbers of a single type in one **contiguous block**, and operations are **vectorised**: one call runs a compiled C loop over the whole array, often using CPU SIMD instructions. That makes numerical code commonly 10 to 100 times faster and far more memory-efficient. NumPy also adds broadcasting, so arrays of different shapes can be combined without writing loops.

CODE

```
import numpy as np, sys

marks = [56, 78, 91, 45, 88]
arr = np.array(marks)

print([m * 1.1 for m in marks])      # list: explicit loop
print(arr * 1.1)                     # NumPy: vectorised
print(arr[arr > 60])                 # boolean filtering
print(arr.mean(), arr.std().round(2))

big_list = list(range(1_000_000))
list_bytes = sys.getsizeof(big_list) + sum(sys.getsizeof(x) for x in big_list)
print("list memory (MB):", round(list_bytes / 1e6, 1))
print("array memory (MB):", round(np.arange(1_000_000).nbytes / 1e6, 1))
```

OUTPUT

```
[61.600000000000001, 85.800000000000001, 100.10000000000001, 49.500000000000001, 96.80000000000001]
[ 61.6  85.8 100.1  49.5  96.8]
[78 91 88]
71.6 18.1
list memory (MB): 36.0
array memory (MB): 8.0
```

Q98. How do you handle missing values in a Pandas DataFrame?

EXPERIENCED · Data (NumPy & Pandas)

First find them with `df.isna().sum()`. Then choose a strategy based on the column and how much is missing:

- **Drop** rows or columns with `dropna()` when very little is missing or the column is useless.
- **Fill** with `fillna()`: the median for skewed numbers, the mean for roughly normal ones, the mode or "Unknown" for categories.
- **Forward or backward fill** (`ffill`, `bfill`) for time series.
- **Interpolate** for smoothly changing numeric data.

For machine learning, compute fill values on the training data only, to avoid data leakage. Our [Pandas roadmap](#) covers data cleaning in more depth.

CODE

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "name": ["Aman", "Riya", "Kabir", "Sneha"],
    "city": ["Pune", None, "Surat", "Pune"],
    "salary": [52000, 61000, np.nan, 58000],
})
print(df.isna().sum().to_dict())
df["salary"] = df["salary"].fillna(df["salary"].median())
df["city"] = df["city"].fillna("Unknown")
print(df.to_string(index=False))
```

OUTPUT

```
{'name': 0, 'city': 1, 'salary': 1}
  name  city  salary
Aman  Pune  52000.0
Riya  Unknown  61000.0
Kabir  Surat  58000.0
Sneha  Pune  58000.0
```

Q99. How do groupby and merge work in Pandas?

EXPERIENCED · Data (NumPy & Pandas)

`groupby` follows **split, apply, combine**: split rows into groups by one or more columns, apply an aggregation such as `sum`, `mean` or `count` to each group, and combine the results into a new table. `agg` lets you compute several aggregations at once.

`merge` joins two DataFrames on common columns, like a SQL join. The `how` parameter selects "inner", "left", "right" or "outer".

CODE

```
import pandas as pd

orders = pd.DataFrame({"customer_id": [1, 2, 1, 3, 2],
                       "amount": [500, 1200, 300, 800, 400]})
customers = pd.DataFrame({"customer_id": [1, 2, 4],
                          "name": ["Riya", "Kabir", "Sneha"]})

summary = orders.groupby("customer_id")["amount"].agg(["sum", "count"])
print(summary)

joined = orders.merge(customers, on="customer_id", how="left")
print(joined.to_string(index=False))
```

OUTPUT

	sum	count
customer_id		
1	800	2
2	1600	2
3	800	1

customer_id	amount	name
1	500	Riya
2	1200	Kabir
1	300	Riya
3	800	NaN
2	400	Kabir

Q100. Implement an LRU cache.

EXPERIENCED · Coding

An LRU (Least Recently Used) cache has a fixed capacity. When it is full and a new key arrives, it evicts the item that was used least recently. Both `get` and `put` must run in $O(1)$. In Python, `collections.OrderedDict` makes this simple: `move_to_end()` marks a key as recently used, and `popitem(last=False)` removes the oldest one. In a C++ or Java interview you would build the same thing from a hash map plus a doubly linked list, and it helps to explain that structure too.

CODE

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.data = OrderedDict()

    def get(self, key):
        if key not in self.data:
            return -1
        self.data.move_to_end(key)      # mark as recently used
        return self.data[key]

    def put(self, key, value):
        if key in self.data:
            self.data.move_to_end(key)
        self.data[key] = value
        if len(self.data) > self.capacity:
            self.data.popitem(last=False)  # evict least recently used

cache = LRUCache(2)
cache.put("a", 1); cache.put("b", 2)
print(cache.get("a"))
cache.put("c", 3)          # evicts "b"
print(cache.get("b"), cache.get("c"), list(cache.data))
```

OUTPUT

```
1
-1 3 ['a', 'c']
```

Keep practising

Revise one level at a time, and explain each answer out loud as if you were in the interview.

Pair this guide with daily coding practice. Code and Debug publishes step-by-step solutions to hundreds of DSA problems in Python at codeanddebug.in/blog.

Watch the video explanations on YouTube: [@codeanddebug](https://www.youtube.com/@codeanddebug).