**-> Advanced Internals & Performance**

**1. How Python Manages Memory Internally**

Python uses a **private heap space** to store objects.  
Memory management is handled by:

* **Reference counting** (each object tracks how many references point to it)
* **Garbage Collector (GC)** for cleaning up cyclic references.  
  Python also uses **memory pools (PyMalloc)** to efficiently allocate small objects and reduce OS-level allocations.

**2. Difference between id(), hash(), and is**

* **id(obj)** → Returns the memory address (unique identifier) of the object.
* **hash(obj)** → Returns an integer hash value (used in dict/set lookups).
* **is** → Compares **identity** (checks if two variables point to the same object in memory).

**3. How Python’s Garbage Collector Works**

Python primarily uses **reference counting** to free objects when their count drops to 0.  
For **cyclic references**, Python has a **cyclic garbage collector** that runs periodically and finds objects forming cycles (using generations) and frees them.

**4. Reference Cycles & How Python Handles Them**

A **reference cycle** occurs when objects reference each other, preventing their reference count from ever reaching zero.  
Example:

a = []

a.append(a) # cycle

Python detects these cycles using its **cyclic GC** and breaks them to free memory.

**5. Difference Between deepcopy and pickle**

* **deepcopy** → Creates a completely independent copy of an object in memory (used for duplication inside the same program).
* **pickle** → Serializes the object to bytes (so it can be stored/sent) and then deserialized back into a Python object.

**6. Difference Between Shallow Copy, Deep Copy, and Assignment**

* **Assignment** → Just creates a new reference to the same object (no copy).
* **Shallow copy** → Creates a new object, but references the inner objects (changes in inner lists affect both).
* **Deep copy** → Creates a new object and recursively copies all inner objects (fully independent).

**7. How Python Lists are Implemented Internally**

Python lists are **dynamic arrays** (resizable arrays in memory).

* Over-allocates memory to minimize reallocations.
* Appending is usually **O(1) amortized** because resizing happens infrequently.
* Indexing is **O(1)** since it uses array indexing under the hood.

**8. How Python Implements Dictionaries (Hash Tables)**

Dictionaries use **open addressing with probing** in a hash table.

* Keys are hashed → index determined.
* If collision occurs → Python finds next free slot (linear probing).
* Resizes automatically when load factor gets high.
* Since Python 3.7+, dicts preserve **insertion order**.

**9. Difference Between OrderedDict and Normal dict in Python 3.7+**

* Before Python 3.7 → OrderedDict was needed to preserve insertion order.
* Python 3.7+ → Regular dict also preserves insertion order by default.
* OrderedDict still provides extra features (like move\_to\_end(), equality checks by order).

**10. Profiling Python Code for Performance Bottlenecks**

You can use:

* **cProfile** (built-in): python -m cProfile myscript.py
* **timeit**: For micro-benchmarking small code snippets.
* **line\_profiler / memory\_profiler**: For detailed line-by-line CPU/memory usage.
* **Concurrency, Parallelism & Async**

**11. Global Interpreter Lock (GIL)**

The **GIL** is a mutex in CPython that allows only **one thread to execute Python bytecode at a time**.  
It exists to make memory management (reference counting) thread-safe.  
It simplifies interpreter design but limits true parallel execution on multiple cores for CPU-bound code.

**12. Difference: Multithreading vs Multiprocessing**

* **Multithreading:** Multiple threads within the same process share memory. Limited by GIL (CPU-bound tasks won’t run truly in parallel). Good for I/O-bound tasks.
* **Multiprocessing:** Spawns separate processes with their own memory space → true parallelism. Slightly higher overhead due to inter-process communication.

**13. When to Use threading vs multiprocessing**

* Use **threading** for I/O-bound tasks (network requests, file reads/writes) since they spend time waiting.
* Use **multiprocessing** for CPU-bound tasks (heavy computation, data processing) since it bypasses GIL and utilizes multiple CPU cores.

**14. What is asyncio? How It Differs from Threads**

asyncio is a library for **asynchronous, single-threaded, cooperative multitasking**.  
It uses an **event loop** and coroutines to switch tasks when they’re waiting (non-blocking I/O).  
Unlike threads, it does not run code in parallel — it just interleaves tasks efficiently.

**15. Coroutines vs Generators**

* **Generators** yield values one at a time and can be iterated over.
* **Coroutines** are generalizations of generators that can consume data (send()), pause, and resume execution — designed for cooperative multitasking.

**16. ThreadPoolExecutor vs ProcessPoolExecutor**

* **ThreadPoolExecutor:** Runs tasks in a pool of threads, good for I/O-bound tasks.
* **ProcessPoolExecutor:** Runs tasks in separate processes, ideal for CPU-bound tasks as it avoids GIL and runs in parallel on multiple cores.

**17. Cooperative vs Preemptive Multitasking**

* **Cooperative multitasking:** Tasks voluntarily yield control (e.g., await in async code).
* **Preemptive multitasking:** The OS scheduler interrupts tasks to give CPU time to others (used in threads/processes).

**18. Async Function Example (Concurrent URL Fetching)**

import asyncio

import aiohttp

async def fetch(session, url):

async with session.get(url) as response:

return await response.text()

async def fetch\_all(urls):

async with aiohttp.ClientSession() as session:

tasks = [fetch(session, url) for url in urls]

return await asyncio.gather(\*tasks)

# Example Usage:

# asyncio.run(fetch\_all(["https://example.com", "https://python.org"]))

**19. Race Conditions & Deadlocks**

* **Race Condition:** Multiple threads/processes access shared data simultaneously → unpredictable results.
* **Deadlock:** Two or more threads wait indefinitely for each other’s resources → program freeze.

**Handling in Python:**  
Use **threading.Lock()**, **RLock()**, or higher-level synchronization (Queue, Condition, Semaphore) to avoid data corruption and design carefully to prevent circular waits.

**20. Inter-Process Communication (IPC) in Python**

Python provides several IPC mechanisms:

* **multiprocessing.Queue & Pipe** → for message passing between processes.
* **shared memory (multiprocessing.Value/Array)** → share data safely.
* **managers** → share complex objects.
* **sockets** or external systems (Redis, message queues) for distributed processes.
* **Memory Optimization & Design**

**21. What are Python Memory Views?**

A **memoryview** is a built-in object that allows direct access to the memory of objects supporting the buffer protocol (e.g., bytes, bytearray, NumPy arrays) **without copying** the data.  
This is useful for slicing and manipulating large binary data efficiently.

**22. Difference between bytes, bytearray, and memoryview**

* **bytes** → Immutable sequence of bytes.
* **bytearray** → Mutable sequence of bytes (can be modified in place).
* **memoryview** → A view object that references the underlying buffer of bytes/bytearray without creating a copy — useful for zero-copy operations.

**23. Reducing Memory Usage for Large Datasets**

* Use **generators/iterators** instead of loading everything into lists.
* Use **NumPy arrays** or array module (store compact binary representation).
* Use **chunked processing** (read/write data in small batches).
* Use **slots** in classes to save attribute memory.
* Use **compression** (gzip, bz2) for large data files.

**24. Slots (\_\_slots\_\_) in Python Classes**

\_\_slots\_\_ is a special attribute that defines a fixed set of attributes a class can have.  
Benefits:

* Saves memory by avoiding per-instance \_\_dict\_\_.
* Improves attribute access speed slightly.  
  Trade-off: Cannot dynamically add new attributes outside of \_\_slots\_\_.

**25. Implementing an LRU Cache Manually**

You can use **OrderedDict** to implement an LRU cache:

from collections import OrderedDict

class LRUCache:

def \_\_init\_\_(self, capacity):

self.cache = OrderedDict()

self.capacity = capacity

def get(self, key):

if key not in self.cache:

return -1

self.cache.move\_to\_end(key)

return self.cache[key]

def put(self, key, value):

self.cache[key] = value

self.cache.move\_to\_end(key)

if len(self.cache) > self.capacity:

self.cache.popitem(last=False)

**26. How functools.lru\_cache Works Internally**

* Wraps a function with a **dictionary + doubly linked list** under the hood.
* Stores recent results keyed by arguments (hashable).
* Automatically evicts the **least recently used** entry when cache size is exceeded.
* Thread-safe and highly optimized in C.

**27. Weak References (weakref)**

A **weak reference** lets you reference an object **without increasing its reference count**.  
When the object is garbage-collected, the weak reference becomes None.  
Useful for caches or object registries where you don’t want to prevent GC.

**28. Debugging Memory Leaks in Python**

* Use **gc module** (gc.collect(), gc.garbage) to inspect unreachable objects.
* Use **tracemalloc** (built-in) to track memory allocations.
* Use external tools: objgraph, memory\_profiler, pympler.
* Check for **reference cycles** and lingering global variables.

**29. Generators vs Lists for Memory Efficiency**

* **Generators:** Produce values lazily (on demand), so they **don’t store all values in memory** → very memory-efficient for large data.
* **Lists:** Store all elements at once, faster for random access but use more memory.

**30. Handling Huge Data (GBs) Efficiently**

* Use **iterators/generators** and stream processing.
* Process data in **chunks** (avoid loading all into RAM).
* Use **Pandas with chunked reading** (chunksize).
* Use **memory-mapped files** (mmap) for large binary data.
* Offload to **databases** or **big data frameworks** (Dask, Spark) if dataset is too big.
* **Advanced OOP & Design Patterns**

**31. Metaclasses in Python**

Metaclasses are **classes of classes** — they define how classes behave.  
Every class is an instance of a metaclass (default: type).  
Used for customizing class creation (e.g., enforcing naming conventions, registering classes automatically).

**32. Difference between type() and class**

* type(obj) → returns the class of an object.
* type(name, bases, dict) → dynamically creates a new class at runtime.
* class keyword → statically defines a class in code.

**33. Singleton Class**

Ensures only **one instance** exists:

class Singleton:

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not cls.\_instance:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

**34. Factory Design Pattern**

Encapsulates object creation:

class Shape: pass

class Circle(Shape): pass

class Square(Shape): pass

def shape\_factory(shape\_type):

return {"circle": Circle, "square": Square}.get(shape\_type, Shape)()

**35. Monkey Patching**

Dynamically modifying a class or module at runtime:

import math

math.sqrt = lambda x: x \*\* 0.5 # replaces built-in sqrt

**36. Mixins**

Small classes used to add reusable functionality via multiple inheritance:

class JSONMixin:

def to\_json(self):

import json

return json.dumps(self.\_\_dict\_\_)

class Person(JSONMixin):

def \_\_init\_\_(self, name):

self.name = name

**37. Duck Typing**

If an object **acts like a duck**, we treat it as one.  
Python cares about behavior, not type:

def quack(obj):

obj.quack()

class Duck:

def quack(self): print("Quack!")

class Person:

def quack(self): print("Person imitates duck")

quack(Duck()); quack(Person())

**38. Multiple Inheritance & MRO**

Python uses **C3 linearization (MRO)** to determine method resolution order.  
Use Class.mro() or Class.\_\_mro\_\_ to inspect order.  
It ensures consistent, left-to-right lookup without duplication.

**39. Abstract Base Classes (abc module)**

ABC enforces interface contracts:

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def speak(self): pass

Any subclass must implement speak().

**40. Dependency Injection**

Inject dependencies instead of hardcoding:

class Engine:

def run(self): print("Engine running")

class Car:

def \_\_init\_\_(self, engine): # injected dependency

self.engine = engine

def drive(self): self.engine.run()

Car(Engine()).drive()

* **Data Structures & Algorithms**

**41. Trie Implementation**

class TrieNode:

def \_\_init\_\_(self):

self.children, self.is\_end = {}, False

class Trie:

def \_\_init\_\_(self): self.root = TrieNode()

def insert(self, word):

node = self.root

for ch in word:

node = node.children.setdefault(ch, TrieNode())

node.is\_end = True

def search(self, word):

node = self.root

for ch in word:

if ch not in node.children: return False

node = node.children[ch]

return node.is\_end

**42. Thread-Safe Queue**

import queue

q = queue.Queue()

q.put(1); print(q.get())

**43. Custom Linked List**

class Node:

def \_\_init\_\_(self, data): self.data, self.next = data, None

class LinkedList:

def \_\_init\_\_(self): self.head = None

def insert(self, data):

new = Node(data)

new.next, self.head = self.head, new

def search(self, key):

cur = self.head

while cur:

if cur.data == key: return True

cur = cur.next

return False

def delete(self, key):

cur, prev = self.head, None

while cur:

if cur.data == key:

if prev: prev.next = cur.next

else: self.head = cur.next

return

prev, cur = cur, cur.next

**44. Priority Queue**

import heapq

pq = []

heapq.heappush(pq, (1, "task1"))

heapq.heappush(pq, (3, "task3"))

print(heapq.heappop(pq)) # lowest priority first

**45. Optimize Sorting Large Datasets**

* Use **key functions** (avoid expensive comparisons).
* Use **NumPy / Pandas sort** for large numeric data (C-optimized).
* Use **external merge sort** for data that doesn't fit into memory (chunk + merge).

**46. Balanced BST**

(Pseudocode-level for interviews — AVL/Red-Black Trees are complex.)  
Explain that Python’s built-in bisect or sortedcontainers can maintain sorted order efficiently without manually coding AVL.

**47. Graph Traversal**

def bfs(graph, start):

visited, q = set([start]), [start]

while q:

node = q.pop(0)

for neigh in graph[node]:

if neigh not in visited:

visited.add(neigh); q.append(neigh)

def dfs(graph, start, visited=None):

if visited is None: visited = set()

visited.add(start)

for neigh in graph[start]:

if neigh not in visited:

dfs(graph, neigh, visited)

**48. Cycle Detection (Directed Graph)**

def has\_cycle(graph):

visited, stack = set(), set()

def dfs(node):

if node in stack: return True

if node in visited: return False

visited.add(node); stack.add(node)

for neigh in graph[node]:

if dfs(neigh): return True

stack.remove(node)

return False

return any(dfs(n) for n in graph)

**49. Rate Limiter**

import time

class RateLimiter:

def \_\_init\_\_(self, rate):

self.rate, self.tokens, self.last = rate, rate, time.time()

def allow(self):

now = time.time()

self.tokens += (now - self.last) \* self.rate

self.tokens = min(self.tokens, self.rate)

self.last = now

if self.tokens >= 1:

self.tokens -= 1

return True

return False

**50. Distributed Counter**

Use **Redis** or **multiprocessing.Manager()**:

from multiprocessing import Manager

manager = Manager()

counter = manager.Value('i', 0)

def increment():

with counter.get\_lock():

counter.value += 1

* **Advanced Functional Programming**

**51. Closures**

Functions that capture variables from enclosing scope:

def outer(x):

def inner(y): return x + y

return inner

add5 = outer(5); print(add5(10))

**52. Currying**

Transform function taking multiple args into nested single-arg functions:

def curry\_add(x):

return lambda y: x + y

print(curry\_add(3)(4)) # 7

**53. Partial Functions**

Fix some arguments of a function:

from functools import partial

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

square = partial(power, exp=2)

**54. Memoization vs Caching**

* **Memoization:** Stores results of function calls based on inputs (used in recursion).
* **Caching:** Broader term — storing computed results for reuse (may expire or be size-limited).

**55. Functional Pipelines**

data = [1, 2, 3, 4]

result = list(map(lambda x: x\*2, filter(lambda x: x%2==0, data)))

**56. Tail Recursion Optimization**

Python doesn’t support TCO natively. Convert recursion to loop:

def factorial(n):

result = 1

while n > 0:

result \*= n

n -= 1

return result

**57. Higher-Order Functions**

Functions that take functions as arguments or return them:

def apply(func, data): return [func(x) for x in data]

print(apply(lambda x: x\*2, [1,2,3]))

**58. map/filter/reduce vs Comprehensions**

* **map/filter/reduce:** Functional, work with iterators, sometimes faster.
* **Comprehensions:** More Pythonic, readable, and lazy with generator expressions.

**59. Function Composition**

Combine functions to create pipelines:

def compose(f, g):

return lambda x: f(g(x))

double = lambda x: x\*2

inc = lambda x: x+1

f = compose(double, inc)

print(f(3)) # (3+1)\*2 = 8

**60. Monads in Python**

Monads are a way to **chain operations** while handling context (like Maybe, IO in FP).  
In Python, you can model them with wrapper classes:

class Maybe:

def \_\_init\_\_(self, value): self.value = value

def bind(self, func):

return Maybe(func(self.value)) if self.value is not None else self

* **Testing, Packaging & Deployment**

**61. Writing Unit Tests (unittest vs pytest)**

* **unittest:** Built-in, class-based, more verbose.
* **pytest:** Third-party, simple function-based tests, better fixtures, parameterization, and plugins.

# unittest

import unittest

class TestMath(unittest.TestCase):

def test\_add(self): self.assertEqual(1+1, 2)

# pytest

def test\_add():

assert 1+1 == 2

**62. Mocking**

Mocking replaces real objects with dummy ones to isolate code under test.  
Example: Mocking database calls so tests don’t hit production DB.

**63. Patching Dependencies (unittest.mock.patch)**

from unittest.mock import patch

@patch("module.requests.get")

def test\_api(mock\_get):

mock\_get.return\_value.status\_code = 200

# call your function that uses requests.get

**64. Integration vs Unit Tests**

* **Unit tests:** Test smallest code units (functions, classes).
* **Integration tests:** Test multiple components together (API + DB + Cache).

**65. Parameterized Tests**

import pytest

import math

@pytest.mark.parametrize("x,expected", [(2,4),(3,9),(4,16)])

def test\_square(x, expected):

assert math.pow(x,2) == expected

**66. tox vs pytest**

* **pytest:** Runs tests.
* **tox:** Automates testing across multiple Python environments (e.g., 3.8, 3.9, 3.10).

**67. Build & Publish Package to PyPI**

python setup.py sdist bdist\_wheel

twine upload dist/\*

Or use pyproject.toml (modern way).

**68. Managing Virtual Environments**

* venv or virtualenv → create isolated envs.
* pip freeze > requirements.txt → capture deps.
* pip install -r requirements.txt → install deps.

**69. Wheels (.whl)**

Wheels are **prebuilt binary distributions** of Python packages for faster installation (no source compilation needed).

**70. Poetry**

Modern dependency manager & build tool:

* Handles dependencies, virtualenvs, packaging.
* Uses pyproject.toml for configuration.

**> Advanced Networking & System Design**

**71. Simple REST API without Flask/Django**

from http.server import BaseHTTPRequestHandler, HTTPServer

class Handler(BaseHTTPRequestHandler):

def do\_GET(self):

self.send\_response(200)

self.end\_headers()

self.wfile.write(b"Hello World")

HTTPServer(("localhost", 8000), Handler).serve\_forever()

**72. WebSockets**

Use websockets library:

import asyncio, websockets

async def echo(ws):

async for msg in ws: await ws.send(msg)

asyncio.run(websockets.serve(echo, "localhost", 8765))

**73. WSGI**

WSGI = Web Server Gateway Interface.  
Standard that connects Python web apps (Flask, Django) with servers (Gunicorn, uWSGI).

**74. Scaling to Millions of Requests**

* Use **async frameworks** (FastAPI), caching (Redis), DB connection pooling.
* Load balancers + multiple app instances.
* Horizontal scaling with Kubernetes or serverless.

**75. Message Queues**

* **RabbitMQ**, **Kafka**, **Redis Pub/Sub**, **AWS SQS** — used for decoupling services, background tasks, event streaming.

**76. Distributed Task Queue**

Celery example:

from celery import Celery

app = Celery("tasks", broker="redis://localhost:6379/0")

@app.task

def add(x, y): return x+y

**77. Sync vs Async Web Frameworks**

* **Sync:** One request per thread/process (Flask, Django).
* **Async:** Non-blocking I/O, handles many requests per thread (FastAPI, Starlette).

**78. Retries & Backoff**

import time, random

def retry(func, retries=3):

for i in range(retries):

try: return func()

except Exception:

time.sleep(2\*\*i + random.random())

**79. uvicorn & gunicorn**

* **uvicorn:** ASGI server for async apps (FastAPI).
* **gunicorn:** WSGI process manager. Can use uvicorn workers for async.

**80. API Rate Limiting**

Use token bucket or leaky bucket algorithm, or middleware:

from collections import defaultdict, deque

import time

requests = defaultdict(deque)

def allow\_request(ip):

now = time.time()

dq = requests[ip]

while dq and now - dq[0] > 60: dq.popleft()

if len(dq) < 10:

dq.append(now)

return True

return False

* **Expert-Level Tricky & Practical**

**81. Context Manager without with**

class MyContext:

def \_\_enter\_\_(self): print("Enter")

def \_\_exit\_\_(self, exc\_type, exc, tb): print("Exit")

ctx = MyContext()

ctx.\_\_enter\_\_()

# do work

ctx.\_\_exit\_\_(None, None, None)

**82. Decorator Class with Parameters**

class Repeat:

def \_\_init\_\_(self, n): self.n = n

def \_\_call\_\_(self, func):

def wrapper(\*a, \*\*kw):

for \_ in range(self.n): func(\*a, \*\*kw)

return wrapper

@Repeat(3)

def hello(): print("Hi")

**83. \_\_new\_\_ vs \_\_init\_\_**

* \_\_new\_\_ → creates instance (static method, called first).
* \_\_init\_\_ → initializes instance after creation.

**84. Operator Overloading**

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)

**85. Lazy Evaluation**

Compute only when needed:

class Lazy:

def \_\_init\_\_(self, func): self.func, self.\_value = func, None

@property

def value(self):

if self.\_value is None: self.\_value = self.func()

return self.\_value

**86. eval() vs exec()**

* eval(expr) → evaluates and returns value (e.g., eval("2+3") → 5).
* exec(code) → executes statements (no return).  
  Dangerous → can run arbitrary malicious code.

**87. Sandbox Untrusted Code**

Use libraries like RestrictedPython, or run code in a container/VM to isolate process.

**88. Secure Against Code Injection**

* Validate and sanitize user input.
* Avoid eval()/exec() on untrusted data.
* Use parameterized queries for DB access.

**89. Optimize for Multi-core CPUs**

* Use **multiprocessing** (bypasses GIL).
* Use **C-extensions** or **NumPy** (release GIL in C code).
* Consider **PyPy** or async for I/O-bound tasks.

**90. Custom Event Loop**

import selectors, socket

selector = selectors.DefaultSelector()

def accept(sock):

conn, \_ = sock.accept()

conn.setblocking(False)

selector.register(conn, selectors.EVENT\_READ, read)

def read(conn):

data = conn.recv(1024)

if data: conn.send(data)

else: selector.unregister(conn); conn.close()

sock = socket.socket()

sock.bind(("localhost", 8080)); sock.listen()

sock.setblocking(False)

selector.register(sock, selectors.EVENT\_READ, accept)

while True:

for key, \_ in selector.select():

key.data(key.fileobj)