

The Complete Python Cheat Sheet

Every Python concept you need — from the basics to OOP, concurrency, testing, data, web APIs, GenAI apps and interview prep. Short explanations, real examples, tested on Python 3.12.

By **Priyabrata Dalbehera** · GenAI engineer

17 pages

11 parts

300+ tested examples

3.12 Python

What's inside

- | | |
|--|--|
| 1 Core Python
types, numbers, strings, f-strings, conditions, match, loops, functions, scope
p. 2–3 | 2 Data structures
list, tuple, set, dict, copying, comprehensions, sorting, collections, itertools, Big-O
p. 4–5 |
| 3 Object-oriented Python
classes, properties, inheritance, MRO, ABCs, magic methods, dataclasses, enums
p. 6–7 | 4 Advanced functions
iterators, generators, decorators, closures, functools, context managers
p. 8 |
| 5 Errors, logging & files
exceptions, chaining, exception groups, logging, pathlib, CSV, JSON
p. 9 | 6 Standard library
regex, datetime, time zones, os, subprocess, argparse, random, math
p. 10 |
| 7 Concurrency
GIL, threads, locks, processes, asyncio, TaskGroup, timeouts
p. 11 | 8 Typing, tooling & testing
type hints, generics, pyproject, project layout, debugging, pytest, mocks
p. 12–13 |
| 9 Data & web
NumPy, pandas, charts, HTTP, FastAPI, Pydantic, SQLite
p. 14–15 | 10 Python for GenAI
LLM calls, structured output, tokens & cost, chunking, embeddings, RAG
p. 16 |
| 11 Interview prep
memory model, gotchas, quick answers, complexity reference
p. 17 | |

How to read it

Code shows real output in comments:
`len("hi")` # 2. Grey italic = explanation.

Tip & Watch out

Green boxes are good habits. Amber boxes are mistakes people make in real code and interviews.

Learn more

Each topic has a full, free note with exercises at learn.aiwithpriyabrata.com

Python version

Examples were run on **Python 3.12.7**. Most work on 3.10+; anything newer is marked 3.10+, 3.11+ or 3.12+ where it appears.

Packages

Parts 1–8 use only the standard library (plus `pytest` in Part 8). Parts 9–10 need:
`pip install numpy pandas matplotlib httpx fastapi uvicorn pydantic pytest openai python-dotenv`
On Windows, time-zone examples also need `tzdata`. Calls to the OpenAI API need your own key.

1.1 Variables & types

```
name = "Asha"           # str  - text
age = 25                 # int  - whole number
score = 9.5              # float - decimal
ok = True                # bool - True / False
nothing = None           # NoneType - "no value"
type(age)                # <class 'int'>
isinstance(age, int)     # True
int("42"), float("3.5"), str(7), bool(0) # 42, 3.5, '7', False
x, y = 1, 2              # assign several at once
x, y = y, x              # swap
a = b = 0                # same value to both
```

Variables are **names that point to objects**; a name can point to any type later (dynamic typing).

1.2 Numbers

```
# 3.5, 3, 1 (true div, floor div, remainder)
7 / 2, 7 // 2, 7 % 2
-7 // 2 # -4 (floors toward minus infinity)
# 1024, 1000000000000000000 (ints never overflow)
2 ** 10, 10 ** 20
divmod(17, 5) # (3, 2)
round(3.14159, 2) # 3.14
round(2.5), round(3.5) # 2, 4 (rounds half to even!)
abs(-5), pow(2, 10, 1000) # 5, 24 (3-arg pow = modulo)
1_000_000 # 1000000 (underscores for readability)
int("ff", 16), bin(5), hex(255) # 255, '0b101', '0xff'
0.1 + 0.2 == 0.3 # False - floats are approximate
```

```
import math
from decimal import Decimal
from fractions import Fraction
math.isclose(0.1 + 0.2, 0.3) # True
Decimal("0.1") + Decimal("0.2") == Decimal("0.3") # True (money)
# Fraction(1, 2)
Fraction(1, 3) + Fraction(1, 6)
```

Watch out: use `Decimal` (or integer paise) for money — never float.

1.3 Operators

Kind	Operators	Example
Arithmetic	+ - * / // % **	5 % 3 → 2
Comparison	== != < > = >=	0 < x < 10 (chained)
Logical	and or not	short-circuit
Identity	is, is not	x is None
Membership	in, not in	"a" in "cat"
Assignment	= += -= *= /= //= **=	n += 1
Walrus	:=	if (n := len(s)) > 3:
Bitwise	& ^ ~ << >>	1 << 3 → 8

```
True or False and False # True (and binds tighter than or)
# 'default' (or returns the first truthy value)
"" or "default"
[] and "never evaluated" # []
```

1.4 Strings — basics

```
s = "GenAI"
s[0], s[-1], s[1:3], s[::-1] # 'G', 'I', 'en', 'IAneG'
len(s), "AI" in s           # 5, True
"ha" * 3, "a" + "b"         # 'hahaha', 'ab'
multi = """Line 1
Line 2"""                   # triple quotes: multi-line
path = r"C:\new\docs"        # raw string: backslashes kept
"tab\tthere", "new\nline"   # escapes: \t \n \\ \' \'
s[0] = "g"                   # TypeError - strings are immutable
```

1.5 String methods

```
" hi ".strip(), "xxhixx".strip("x") # 'hi', 'hi'
"GenAI".lower(), "genai".upper()    # 'genai', 'GENAI'
"hello world".title()               # 'Hello World'
"a,b,c".split(",")                  # ['a', 'b', 'c']
# ['one', 'two'] (any whitespace)
"one two".split()
"-. ".join(["a", "b", "c"])         # '-a-b-c'
# 'heLlo' (max 1 replacement)
"hello".replace("l", "L", 1)
# 2, -1 (index() raises instead)
"hello".find("l"), "hello".find("z")
"hello".count("l")                  # 2
"report.pdf".endswith((".pdf", ".md")) # True (tuple of options)
"key=value".partition("=")          # ('key', '=', 'value')
"a\nb".splitlines()                 # ['a', 'b']
"7".zfill(3), "py".center(6, "*")   # '007', '***py**'
"123".isdigit(), "abc".isalpha()     # True, True
# b'\xe2\x82\x99' (str → bytes)
"₹".encode("utf-8")
# 'hi' (bytes → str)
b"hi".decode("utf-8")
```

1.6 f-strings & formatting

```
name, cost, n = "gpt-4o-mini", 0.123456, 1234567
f"{name}: ₹{cost:.2f}" # 'gpt-4o-mini: ₹0.12'
f"{n:,}", f"{n:}"      # '1,234,567', '1_234_567'
f"{0.875:.1%}"         # '87.5%'
f"{42:5}|{42:<5}|{42:^5}|" # ' 42|42 | 42 |'
f"{7:03d}", f"{255:x}", f"{5:b}" # '007', 'ff', '101'
f"{3.14159:8.2f}"       # ' 3.14' (width 8, 2 decimals)
f"{1e6:, .0f}"          # '1,000,000'
# "name='gpt-4o-mini', 'n' * 2 = 2469134"
f"{name=}", f"{n * 2 =}"
f"{name!r}"              # "'gpt-4o-mini'" (repr)
f"{{literal braces}}"    # '{literal braces}'
```

Tip: `f"{x=}"` prints the name **and** the value — the fastest way to debug.

1.7 Input & output

```
print("a", "b", sep=" | ", end="!\n") # a | b!
age = int(input("Age? "))              # input() always returns a str
print(*[1, 2, 3])                      # 1 2 3 (unpack into arguments)
```

1.8 Conditions & truthiness

```
if score >= 90:
    grade = "A"
elif score >= 75:
    grade = "B"
else:
    grade = "C"

label = "pass" if score >= 40 else "fail" # one-line if/else
if 0 <= score <= 100: ... # chained comparison
if not items: ... # empty list / dict / "" / 0 / None
if value is None: ... # use "is" for None — never ==
```

Falsy	False None 0 0.0 "" [] {} () set() range(0)
Truthy	everything else — including "0", " ", [0]

1.9 match — structural pattern matching (3.10+)

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

def describe(event):
    match event:
        case "quit" | "exit": # literal values
            return "bye"
        case {"type": "click", "x": x, "y": y}: # dict shape
            return f"click at {x},{y}"
        case [first, *rest]: # sequence shape
            return f"list starting with {first}"
        case Point(x=0, y=0): # class + fields
            return "origin"
        case int(n) if n < 0: # type + guard
            return "negative int"
        case _: # anything else
            return "other"

describe({"type": "click", "x": 3, "y": 4}) # 'click at 3,4'
# 'origin', 'negative int'
describe(Point(0, 0)), describe(-5)
```

1.10 Loops

```
for i in range(3): ... # 0 1 2
for i in range(10, 0, -3): ... # 10 7 4 1
for i, item in enumerate(items, start=1): ...
# 3.10+: error if lengths differ
for a, b in zip(names, ages, strict=True): ...
for key, value in prices.items(): ...
for item in reversed(items): ...
for item in sorted(items, key=len): ...

while attempts < 3:
    attempts += 1

for n in numbers:
    if n < 0:
        continue # skip to the next item
    if n == 0:
        break # Leave the loop
else:
    print("no zero found") # runs only if the loop didn't break
```

Watch out: don't add/remove items of a list while looping over it — loop over a copy or build a new list.

1.11 Functions

```
def greet(name: str, greeting: str = "Hello") -> str:
    """Return a greeting (this is the docstring)."""
    return f"{greeting}, {name}!"

# 'Hello, Asha!', 'Hi, Ravi!'
greet("Asha"), greet("Ravi", greeting="Hi")

def stats(*nums, **options): # *args -> tuple, **kwargs -> dict
    return sum(nums), options
stats(1, 2, 3, unit="km") # (6, {'unit': 'km'})

# a,b positional-only · d keyword-only
def f(a, b, /, c, *, d): ...
f(1, 2, 3, d=4) # OK
args, kw = [1, 2, 3], {"d": 4}
# unpack a list and a dict into a call
f(*args, **kw)

def min_max(nums):
    return min(nums), max(nums) # returns a tuple
lo, hi = min_max([4, 1, 9]) # 1, 9

square = lambda x: x * x # small anonymous function
def nothing(): pass # returns None
```

Watch out: def f(items=[]) shares one list across calls — use items=None.

1.12 Scope (LEGB), global, nonlocal, closures

```
count = 0 # Global
def bump():
    global count # rebind the global name
    count += 1

def make_counter(): # Enclosing function
    n = 0
    def step():
        nonlocal n # rebind the enclosing variable
        n += 1
        return n
    return step # a closure: remembers n

counter = make_counter()
counter(), counter() # 1, 2
```

Name lookup order: Local → Enclosing — Global → Built-in.

1.13 Built-in functions

```
len("abc"), sum([1, 2, 3], 10) # 3, 16 (start value)
min(3, 1, 2), max(["bb", "a"], key=len) # 1, 'bb'
sorted([3, 1, 2], reverse=True) # [3, 2, 1]
any([0, "", 5]), all([1, "x"]) # True, True
list(map(int, ["1", "2"])) # [1, 2]
list(filter(str.isdigit, ["1", "a"])) # ['1']
list(zip("ab", [1, 2])) # [('a', 1), ('b', 2)]
list(enumerate("ab")) # [(0, 'a'), (1, 'b')]
chr(65), ord("A") # 'A', 65
abs(-2), round(2.675, 2) # 2, 2.67 (float repr!)
isinstance(3, (int, float)) # True
id(x), type(x), dir(x), vars(obj) # inspect objects
help(str.split) # docs in the terminal
```

2.1 list — ordered, changeable

```

nums = [3, 1, 2]
nums.append(4)           # [3, 1, 2, 4]
nums.extend([5, 6])      # [3, 1, 2, 4, 5, 6]
nums.insert(0, 0)         # [0, 3, 1, 2, 4, 5, 6]
nums.pop()               # 6 (removes last)
# 0 (removes index 0 — slow on big lists)
nums.pop(0)
nums.remove(2)           # removes first 2 → [3, 1, 4, 5]
nums.index(4), nums.count(4) # 2, 1
nums.sort()              # in place → [1, 3, 4, 5]; returns None!
nums.reverse()           # in place → [5, 4, 3, 1]
del nums[0]              # [4, 3, 1]
nums.clear()             # []
[1, 2] + [3], [0] * 3     # [1, 2, 3], [0, 0, 0]
len(nums), 3 in [1, 3]   # 0, True

```

Watch out: `nums = nums.sort()` sets `nums` to `None`. Use `sorted(nums)` for a new list.

2.2 Slicing `seq[start:stop:step]`

```

a = [0, 1, 2, 3, 4, 5]
a[1:4], a[:3], a[3:]    # [1, 2, 3], [0, 1, 2], [3, 4, 5]
a[-2:], a[:-2]          # [4, 5], [0, 1, 2, 3]
a[::2], a[::-1]         # [0, 2, 4], [5, 4, 3, 2, 1, 0]
a[1:3] = ["x"]          # replace a slice → [0, 'x', 3, 4, 5]
del a[::2]              # delete every 2nd → ['x', 4]
copy = a[:]             # shallow copy

```

Stop is **excluded**. Out-of-range slices never raise — `a[10:]` is `[]`.

2.3 tuple — fixed record

```

point = (3, 4)
single = (5,)           # the comma makes the tuple, not ()
x, y = point             # unpacking
point.count(3), point.index(4) # 1, 1
{(1, 2): "ok"}          # tuples can be dict keys (lists can't)

```

```

from collections import namedtuple
User = namedtuple("User", "name age")
u = User("Asha", 25)
# 'Asha', 25, {'name': 'Asha', 'age': 25}
u.name, u[1], u._asdict()

```

2.4 set — unique items, fast lookup

```

a, b = {1, 2, 3}, {2, 3, 4}
a | b, a & b             # {1, 2, 3, 4}, {2, 3}
a - b, a ^ b             # {1}, {1, 4}
a <= {1, 2, 3, 9}        # True (subset)
a.add(5); a.discard(9)   # discard: no error if missing
a.remove(5)             # remove: KeyError if missing
empty = set()           # {} is an empty DICT
set("banana")           # {'a', 'b', 'n'}
frozenset([1, 2])       # immutable set → can be a dict key
list(dict.fromkeys([3, 1, 3, 2])) # [3, 1, 2] dedupe, keep order

```

2.5 dict — key → value

```

user = {"name": "Asha", "age": 25}
user["city"] = "Pune"    # add or update
user["name"]            # 'Asha' (KeyError if missing)
user.get("email")        # None
user.get("email", "n/a") # 'n/a'
"age" in user            # True (checks keys)
user.pop("age")          # 25 — remove and return
user.pop("missing", None) # None — no error
user.setdefault("tags", []).append("ai")
user.update({"city": "Delhi"}, role="admin")
list(user)              # keys, in insertion order
user.keys(), user.values(), user.items() # Live views

```

```

defaults = {"model": "mini", "temp": 0.2}
settings = defaults | {"temp": 0.7} # merge → new dict (3.9+)
defaults |= {"top_p": 1}            # merge in place
dict(zip(["a", "b"], [1, 2]))        # {'a': 1, 'b': 2}
dict.fromkeys(["x", "y"], 0)         # {'x': 0, 'y': 0}
{v: k for k, v in {"a": 1}.items()}  # {1: 'a'} invert
max(scores, key=scores.get)         # key with the biggest value
for k, v in sorted(scores.items(), key=lambda kv: kv[1]): ...

```

Dicts keep insertion order (3.7+). Keys must be **hashable**: str, int, tuple — not list or dict.

2.6 Copying: reference, shallow, deep

```

import copy
a = [[1, 2], [3]]
b = a                    # same object — no copy at all
c = a.copy()            # shallow: new outer list, same inner lists
d = copy.deepcopy(a)    # deep: copies everything inside
c[0].append(9)
a                        # [[1, 2, 9], [3]] — changed via c!
d                        # [[1, 2], [3]] — independent

```

Tip: shallow copies: `lst.copy()`, `lst[:]`, `dict(d)`, `d.copy()`, `set(s)`.

2.7 Which structure?

Need	Use
ordered items, duplicates OK	list
fixed record / dict key	tuple, namedtuple
look up by key	dict
uniqueness, fast in	set
queue / sliding window	<code>collections.deque</code>
counting	<code>collections.Counter</code>
group into lists	<code>collections.defaultdict(list)</code>
top-k / priority queue	heapq
sorted inserts / binary search	bisect

2.8 Comprehensions

```

nums = [1, 2, 3, 4]
[x * 2 for x in nums]           # [2, 4, 6, 8]
[x for x in nums if x % 2 == 0] # [2, 4]           (filter)
["even" if x % 2 == 0 else "odd" for x in nums[:3]]
# → ['odd', 'even', 'odd']      (if/else inside a comprehension)
{x: x ** 2 for x in range(3)}   # {0: 0, 1: 1, 2: 4}
{len(w) for w in ["a", "bb", "cc"]} # {1, 2}
grid = [[1, 2], [3, 4]]
[n for row in grid for n in row] # [1, 2, 3, 4]      (flatten)
# [[1, 3], [2, 4]] (transpose)
[[r[i] for r in grid] for i in range(2)]
# 30 (generator — no list built)
sum(x * x for x in nums)

```

Tip: if a comprehension needs a comment to explain it, write a normal loop.

2.9 Unpacking

```

first, *rest = [1, 2, 3, 4]      # 1, [2, 3, 4]
*init, last = [1, 2, 3]         # [1, 2], 3
head, _, tail = "python"       # 'p', 'n' (_ = ignore)
(a, b), c = (1, 2), 3           # nested
merged = [*[1, 2], *[3]]       # [1, 2, 3]
combined = **{"a": 1, "b": 2}   # {'a': 1, 'b': 2}

```

2.10 Sorting

```

people = [{"name": "Ravi", "age": 31}, {"name": "Asha", "age": 25},
          {"name": "Meera", "age": 31}]
sorted(people, key=lambda p: p["age"]) # youngest first
sorted(people, key=lambda p: (-p["age"], p["name"]))
# age desc, then name asc → Meera, Ravi, Asha
sorted(["b", "A", "c"])                # ['A', 'b', 'c']
sorted(["b", "A", "c"], key=str.lower) # ['A', 'b', 'c']
from operator import itemgetter, attrgetter
sorted(people, key=itemgetter("name"))  # by dict key
sorted(users, key=attrgetter("age"))    # by attribute

```

Python's sort is **stable** (ties keep their order) and $O(n \log n)$.

2.11 collections

```

from collections import (ChainMap, Counter, OrderedDict,
                        defaultdict, deque)

# Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
c = Counter("hello")
# 2, 0 (missing → 0, no KeyError)
c["l"], c["z"]
c.most_common(1), c.total() # [('l', 2)], 5 total(): 3.10+
Counter(a=3) - Counter(a=1) # Counter({'a': 2})

groups = defaultdict(list) # missing key → new empty list
for name, team in [("Asha", "AI"), ("Ravi", "AI")]:
    groups[team].append(name) # {'AI': ['Asha', 'Ravi']}

q = deque([1, 2, 3], maxlen=3) # fixed size: oldest drops off
q.append(4)                    # deque([2, 3, 4])
q.appendleft(1); q.popleft()   # O(1) at both ends
q.rotate(1)                    # deque([4, 2, 3])

lru = OrderedDict(a=1, b=2)
lru.move_to_end("a")          # mark "a" as recently used
lru.popitem(last=False)       # ('b', 2) — evict the oldest

settings = ChainMap({"temp": 0.7}, {"temp": 0.2, "model": "mini"})
# 0.7, 'mini' (first map wins)
settings["temp"], settings["model"]

```

2.12 heapq & bisect

```

import heapq, bisect
nums = [5, 1, 4, 2]
heapq.heapify(nums)           # min-heap in place, O(n)
heapq.heappush(nums, 0)
heapq.heappop(nums)           # 0 — smallest first
# [1, 4], [5, 4]
heapq.nsmallest(2, [5, 1, 4]), heapq.nlargest(2, [5, 1, 4])
heapq.heappush(h := [], -10) # max-heap trick: push negatives
tasks = []
heapq.heappush(tasks, (2, "write"))
heapq.heappush(tasks, (1, "plan"))
# (1, 'plan') — tuple = priority queue
heapq.heappop(tasks)

scores = [10, 20, 30]
bisect.bisect_left(scores, 25) # 2 — where 25 would go
bisect.insort(scores, 25)      # [10, 20, 25, 30] stays sorted

```

2.13 itertools

```

from itertools import (accumulate, batched, chain, combinations,
                      count, groupby, islice, pairwise, permutations, product,
                      zip_longest)
list(chain([1, 2], [3]))          # [1, 2, 3]
list(islice(count(10), 3))       # [10, 11, 12]
# [1, 3, 6] running total
list(accumulate([1, 2, 3]))
list(pairwise([1, 2, 3]))        # [(1, 2), (2, 3)] 3.10+
# [(0, 1), (2, 3), (4,)] 3.12+
list(batched(range(5), 2))
list(zip_longest([1, 2], [3], fillvalue=0)) # [(1, 3), (2, 0)]
len(list(product("ab", repeat=2)))         # 4 (aa ab ba bb)
len(list(permutations("abc", 2)))           # 6 (order matters)
len(list(combinations("abc", 2)))           # 3 (order doesn't)
[(k, len(list(g))) for k, g in groupby("aabbbc")]
# → [('a', 2), ('b', 3), ('c', 1)]

```

Watch out: groupby only groups **adjacent** items — sort by the same key first.

2.14 Big-O of common operations

Operation	Cost
list: <code>a[i]</code> , <code>append</code> , <code>pop()</code> , <code>len</code>	$O(1)$
list: <code>insert(0, x)</code> , <code>pop(0)</code> , <code>x in a</code> , <code>remove</code>	$O(n)$
list: slice <code>a[i:j]</code> , <code>copy</code>	$O(k) / O(n)$
dict / set: <code>get</code> , <code>set</code> , <code>in</code> , <code>delete</code>	$O(1)$ average
deque: <code>append</code> / <code>pop</code> at either end	$O(1)$
heapq: <code>push</code> / <code>pop</code> · <code>heapify</code>	$O(\log n) \cdot O(n)$
<code>sorted()</code> , <code>.sort()</code>	$O(n \log n)$
<code>bisect</code> search · <code>insort</code>	$O(\log n) \cdot O(n)$
string <code>+=</code> in a loop	$O(n^2) \rightarrow$ use <code>"".join(parts)</code>

3.1 Class anatomy

```
class Account:
    interest = 0.04 # class attribute - shared

    def __init__(self, owner, balance=0):
        self.owner = owner # instance attribute
        # _name = "internal" by convention
        self._balance = balance

    # read like a field: a.balance
    @property
    def balance(self):
        return self._balance

    # a.balance = 10 runs validation
    @balance.setter
    def balance(self, value):
        if value < 0:
            raise ValueError("balance can't be negative")
        self._balance = value

    # instance method: gets self
    def deposit(self, amount):
        self.balance += amount
        # return self -> chainable calls
        return self

    # gets the class: alternative constructor
    @classmethod
    def from_dict(cls, data):
        return cls(data["owner"], data.get("balance", 0))

    # plain function grouped in the class
    @staticmethod
    def is_valid(amount):
        return amount > 0

    # shown in the REPL / debugger
    def __repr__(self):
        return f"Account({self.owner!r}, {self.balance})"

a = Account("Asha").deposit(100).deposit(50)
a.balance, a # 150, Account('Asha', 150)
Account.from_dict({"owner": "Ravi"}) # Account('Ravi', 0)
Account.is_valid(-5) # False
# ValueError: balance can't be negative
a.balance = -1
```

3.2 Method types at a glance

Kind	First arg	Use it for
instance method	self	work with one object's data
@classmethod	cls	alternative constructors, factories
@staticmethod	—	helper that needs neither
@property	self	computed / validated attribute

3.3 Visibility & name mangling

```
class Vault:
    def __init__(self):
        self.public = 1
        # convention: "don't touch from outside"
        self._internal = 2
        self.__secret = 3 # renamed to _Vault__secret

v = Vault()
# 3 - mangling avoids clashes, it isn't security
v._Vault__secret
```

3.4 Inheritance & super()

```
class Agent:
    def __init__(self, name):
        self.name = name
    def run(self, task):
        return f"{self.name} -> {task}"

class ToolAgent(Agent): # inherits everything
    def __init__(self, name, tools):
        # run the parent's __init__
        super().__init__(name)
        self.tools = tools
    def run(self, task): # override
        return "[tool] " + super().run(task)

ToolAgent("bot", ["search"]).run("find") # '[tool] bot -> find'
isinstance(ToolAgent("b", []), Agent) # True
issubclass(ToolAgent, Agent) # True
```

3.5 Multiple inheritance & MRO

```
class A:
    def hi(self): return "A"
class B(A):
    def hi(self): return "B" + super().hi()
class C(A):
    def hi(self): return "C" + super().hi()
class D(B, C): pass

D().hi() # 'BCA'
# ['D', 'B', 'C', 'A', 'object']
[k.__name__ for k in D.__mro__]
```

super() follows the **MRO** (method resolution order), not just "the parent" — that's why B's super() reaches C.

3.6 Abstract base classes

```
from abc import ABC, abstractmethod

class Tool(ABC):
    @abstractmethod
    def run(self, query: str) -> str:
        """Every tool must implement run()."""

class Search(Tool):
    def run(self, query):
        return f"results for {query}"

Search().run("rag") # 'results for rag'
# TypeError: Can't instantiate abstract class Tool
Tool()
```

3.7 Composition over inheritance

```
class Retriever:
    def search(self, q): return [f"doc about {q}"]

# HAS a retriever - not IS a retriever
class ChatBot:
    def __init__(self, retriever):
        # easy to swap / fake in tests
        self.retriever = retriever
    def answer(self, q):
        return self.retriever.search(q)[0]

ChatBot(Retriever()).answer("refunds") # 'doc about refunds'

Tip: inherit for true "is-a" relationships; otherwise pass objects in (composition).
```

3.8 Magic (dunder) methods

Method	Called by
<code>__init__(self, ...)</code>	<code>Cls(...)</code> — set up a new object
<code>__repr__</code> / <code>__str__</code>	<code>repr(x)</code> , <code>REPL</code> / <code>str(x)</code> , <code>print(x)</code>
<code>__eq__</code> , <code>__lt__</code> , ...	<code>==</code> , <code><</code> (sorting)
<code>__hash__</code>	dict keys, sets — define with <code>__eq__</code>
<code>__len__</code> , <code>__bool__</code>	<code>len(x)</code> , if <code>x</code> :
<code>__getitem__</code> , <code>__setitem__</code>	<code>x[i]</code> , <code>x[i] = v</code>
<code>__contains__</code>	item in <code>x</code>
<code>__iter__</code> , <code>__next__</code>	for item in <code>x</code>
<code>__call__</code>	<code>x(...)</code> — object used like a function
<code>__enter__</code> , <code>__exit__</code>	with <code>x</code> :
<code>__add__</code> , <code>__mul__</code> , ...	<code>+</code> , <code>*</code> (operator overloading)

3.9 A value object with operators

```
from functools import total_ordering

# fills in < > >= from == and <
@total_ordering
class Money:
    def __init__(self, paise):
        self.paise = paise
    def __repr__(self):
        return f"Money({self.paise})"
    def __str__(self):
        return f"₹{self.paise / 100:.2f}"
    def __eq__(self, other):
        return self.paise == other.paise
    def __lt__(self, other):
        return self.paise < other.paise
    def __hash__(self):
        return hash(self.paise)
    def __add__(self, other):
        return Money(self.paise + other.paise)
    def __bool__(self):
        return self.paise != 0

str(Money(1250) + Money(50))    # ₹13.00'
Money(5) >= Money(3)           # True
sorted([Money(9), Money(2)])   # [Money(2), Money(9)]
bool(Money(0))                 # False
```

3.10 A custom container

```
class Playlist:
    def __init__(self, songs):
        self._songs = list(songs)
    def __len__(self):
        return len(self._songs)
    # also enables slicing and for-loops
    def __getitem__(self, i):
        return self._songs[i]
    def __contains__(self, song):
        return song in self._songs
    def __call__(self):
        return f"Playing {self._songs[0]}"

p = Playlist(["a", "b", "c"])
# 3, 'c', ['a', 'b'], True, 'Playing a'
len(p), p[-1], p[:2], "b" in p, p()
list(p)                                # ['a', 'b', 'c']
```

3.11 dataclasses

```
from dataclasses import dataclass, field, asdict, replace

@dataclass
class Chunk:
    text: str
    source: str
    score: float = 0.0
    # never tags=[]
    tags: list[str] = field(default_factory=list)

    # validate after __init__
    def __post_init__(self):
        if not 0 <= self.score <= 1:
            raise ValueError("score must be 0-1")

c = Chunk("Refunds in 30 days", "faq.md")
c # → Chunk(text='Refunds in 30 days', source='faq.md',
#         score=0.0, tags=[])
asdict(c)["source"] # 'faq.md'
replace(c, score=0.9).score # 0.9 — copy with changes

@dataclass(order=True, frozen=True, slots=True)
# order: < > · frozen: read-only + hashable
class Version:
    major: int # slots (3.10+): less memory
    minor: int = 0

# Version(major=1, minor=5)
sorted([Version(2), Version(1, 5)])[0]
Version(1).major = 2 # FrozenInstanceError
```

3.12 Enums

```
from enum import Enum, IntEnum, StrEnum, auto

class Status(Enum):
    ACTIVE = auto()
    BLOCKED = auto()

Status.ACTIVE.name, Status.ACTIVE.value # ('ACTIVE', 1)
# Look up by name / value
Status["BLOCKED"], Status(2)
[s.name for s in Status]                # ['ACTIVE', 'BLOCKED']

class Role(StrEnum):
    USER = auto() # members ARE strings (3.11+)
    Role.USER == "user" # value 'user' # True

class Priority(IntEnum):
    LOW = 1
    HIGH = 3
    Priority.HIGH > Priority.LOW # True
```

3.13 __slots__ & Protocols

```
class Point:
    # no per-object __dict__: Less memory
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x, self.y = x, y
    # AttributeError — fixed attributes
    Point(1, 2).z = 3

from typing import Protocol

# "anything with a run(str) method"
class SupportsRun(Protocol):
    def run(self, query: str) -> str: ...

# duck typing, checked by mypy
def use(tool: SupportsRun) -> str:
    return tool.run("hello")
```

A Protocol needs no inheritance — any class with a matching `run` fits.

4.1 Iterators

```
it = iter([10, 20])           # any iterable → iterator
next(it), next(it)           # 10, 20
# 'done' (default instead of StopIteration)
next(it, "done")
```

```
class Countdown:              # the iterator protocol
    def __init__(self, start):
        self.n = start
    def __iter__(self):
        return self
    def __next__(self):
        if self.n <= 0:
            raise StopIteration
        self.n -= 1
        return self.n + 1
```

```
list(Countdown(3))           # [3, 2, 1]
```

Iterable = can give an iterator (`__iter__`). Iterator = has `__next__` and is used up once.

4.2 Decorators

```
import functools, time

def timed(fn):
    # keep fn's name and docstring
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = fn(*args, **kwargs)
        print(f"{fn.__name__}: {time.perf_counter() - start:.3f}s")
        return result
    return wrapper

# same as: slow = timed(slow)
@timed
def slow():
    time.sleep(0.1)

# decorator WITH arguments:
def repeat(n):
    # one more level of nesting
    def decorator(fn):
        @functools.wraps(fn)
        def wrapper(*args, **kwargs):
            return [fn(*args, **kwargs) for _ in range(n)]
        return wrapper
    return decorator

@repeat(2)
def hi():
    return "hi"

hi(), hi.__name__           # ['hi', 'hi'], 'hi'
```

```
# class-based: keeps state
class CountCalls:
    def __init__(self, fn):
        functools.update_wrapper(self, fn)
        self.fn, self.calls = fn, 0
    def __call__(self, *args, **kwargs):
        self.calls += 1
        return self.fn(*args, **kwargs)
```

```
@CountCalls
def ping():
    return "pong"

ping(); ping(); ping.calls  # 2
```

Stacked `@a` over `@b` means `a(b(f))` — the one nearest the function wraps first.

4.3 Generators — lazy sequences

```
def countdown(n):
    while n > 0:
        yield n           # pause here, hand back n, resume later
        n -= 1
list(countdown(3))        # [3, 2, 1]

# pipelines: each step pulls one item at a time (huge files OK)
def non_empty(lines):
    # generator expression
    return (l for l in lines if l.strip())
def numbered(lines):
    for i, line in enumerate(lines, 1):
        yield f"{i}: {line}"
list(numbered(non_empty(["a", "", "b"])))  # ['1: a', '2: b']

def flatten(items):
    for item in items:
        if isinstance(item, list):
            # delegate to a sub-generator
            yield from flatten(item)
        else:
            yield item
list(flatten([1, [2, [3, 4]]]))           # [1, 2, 3, 4]

squares = (x * x for x in range(3))
# [0, 1, 4], [] — used up!
list(squares), list(squares)
```

4.4 functools

```
from functools import lru_cache, partial, reduce, singledispatch

from_hex = partial(int, base=16)           # pre-fill arguments
from_hex("ff")                             # 255
reduce(lambda a, b: a * b, [1, 2, 3, 4])   # 24

@lru_cache(maxsize=128)                    # @cache = unlimited size
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)
# 23416728348467685 — instant
fib(80)
fib.cache_info().currsize                  # 81

# choose code by argument type
@singledispatch
def show(x):
    return f"object {x}"
@show.register
def _(x: int):
    return f"int {x}"
show(3), show("a")                        # ('int 3', 'object a')
```

Watch out: cache arguments must be hashable — pass tuples, not lists.

4.5 Context managers

```
class Timer:                               # class-based
    def __enter__(self):
        self.start = time.perf_counter()
        return self           # bound to "as t"
    def __exit__(self, exc_type, exc, tb):
        self.elapsed = time.perf_counter() - self.start
        # True would SWALLOW the exception
        return False

with Timer() as t:
    sum(range(10**6))
t.elapsed > 0                           # True

from contextlib import contextmanager, suppress

@contextmanager
def opened(path):                       # generator-based: shorter
    f = open(path, encoding="utf-8")
    try:
        yield f                   # the with-block runs here
    finally:
        f.close()                 # always runs, even on error

with suppress(FileNotFoundError):       # ignore one specific error
    os.remove("maybe-missing.tmp")
```

5.1 try / except / else / finally

```
import json
try:
    data = json.loads(raw)
except (json.JSONDecodeError, TypeError) as err:  # several types
    print(f"bad payload: {err}")
    data = {}
else:
    print("parsed OK")  # only if no exception
finally:
    # cleanup: close files, release locks
    print("always runs")

raise ValueError("score must be between 0 and 1")
# dev check - disabled by python -O
assert x >= 0, "x must be positive"
```

Watch out: never write bare `except:` or `except Exception: pass` — it hides real bugs.

5.2 Exception hierarchy (most used)

```
BaseException
├── KeyboardInterrupt, SystemExit (don't catch these)
├── Exception
│   ├── ValueError      right type, bad value: int("abc")
│   ├── TypeError       wrong type: "a" + 1
│   ├── LookupError    KeyError, IndexError
│   ├── AttributeError  missing attribute (often None.x)
│   ├── NameError      undefined name
│   ├── ArithmeticError - ZeroDivisionError
│   ├── OSError        FileNotFoundError, PermissionError, TimeoutError
│   ├── ImportError    ModuleNotFoundError
│   └── RuntimeError    RecursionError, NotImplementedError
```

5.3 Custom exceptions & chaining

```
class APIError(Exception):
    """Raised when the LLM provider returns an error."""
    def __init__(self, status, message):
        super().__init__(f"{status}: {message}")
        self.status = status

try:
    {}["key"]
except KeyError as err:
    raise APIError(404, "not found") from err  # keeps the cause

str(APIError(429, "slow down"))  # '429: slow down'
APIError(429, "x").status        # 429
```

`raise ... from err` shows both errors in the traceback ("The above exception was the direct cause...").

5.4 Exception groups (3.11+)

```
try:
    errors = [ValueError("a"), TypeError("b")]
    raise ExceptionGroup("batch failed", errors)
except* ValueError as group:
    # value errors: 1
    print("value errors:", len(group.exceptions))
except* TypeError:
    print("type errors handled")  # also runs
```

Used by `asyncio.TaskGroup` when several tasks fail at once.

5.5 logging

```
import logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s: %(message)s",
)
log = logging.getLogger(__name__)  # one logger per module

log.debug("hidden at INFO level")
log.info("loaded %d documents", 12)  # lazy %-formatting
log.warning("retrying (attempt %d)", 2)
try:
    1 / 0
except ZeroDivisionError:
    log.exception("division failed")  # ERROR + full traceback
```

Level DEBUG 10 · INFO 20 · WARNING 30 · ERROR 40 · CRITICAL 50

Tip: log IDs and sizes, never API keys or full user prompts.

5.6 Files & pathlib

```
from pathlib import Path
p = Path("data") / "notes.txt"  # / joins paths on any OS
p.parent.mkdir(parents=True, exist_ok=True)
p.write_text("hello\nworld", encoding="utf-8")
p.read_text(encoding="utf-8").splitlines()  # ['hello', 'world']
# 'notes.txt', 'notes', '.txt', True
p.name, p.stem, p.suffix, p.exists()
p.with_suffix(".md").name  # 'notes.md'
# rglob() = include sub-folders
sorted(Path("data").glob("*.txt"))
Path.home(), Path.cwd(), p.resolve()  # absolute paths
# 11 bytes (12 on Windows: \n → \r\n)
p.stat().st_size
p.unlink(missing_ok=True)  # delete a file

# closes automatically
with open(p, "a", encoding="utf-8") as f:
    f.write("more\n")
```

Mode r read · w overwrite · a append · x create-only · b binary · + read & write

Watch out: always pass `encoding="utf-8"` — Windows' default encoding differs.

5.7 CSV & JSON

```
import csv, json
from datetime import datetime

with open("people.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["name", "age"])
    writer.writeheader()
    writer.writerow({"name": "Asha", "age": 25})

with open("people.csv", newline="", encoding="utf-8") as f:
    # [{"name": 'Asha', 'age': '25'}]
    rows = list(csv.DictReader(f))  # values are always str!

json.dumps({"a": 1}, indent=2)  # pretty string
json.dumps({"name": "सोम"}, ensure_ascii=False)  # keep Unicode
json.dumps({"at": datetime(2026, 1, 1)}, default=str)
# → '{"at": "2026-01-01 00:00:00"}' (custom types)
with open("out.json", "w", encoding="utf-8") as f:
    json.dump(rows, f)  # json.load(f) to read back
```

Watch out: never `pickle.load` data from untrusted sources — it can run code.

6.1 Regular expressions (re)

```
import re
text = "Order #4821 on 2026-01-15, order #77 on 2026-02-01"
re.findall(r"#(\d+)", text) # ['4821', '77']
m = re.search(r"(?P<year>\d{4})-(?P<month>\d{2})", text)
# '2026', '2026-01', (15, 22)
m.group("year"), m.group(0), m.span()
# None — match() checks the START only
re.match(r"\d", "a1")
# matches the WHOLE string (PIN code)
re.fullmatch(r"\d{6}", "560001")
re.sub(r"s+", " ", "a b\n c") # 'a b c'
re.sub(r"(\w+)@", r"[\1]@", "me@x.com") # '[me]@x.com'
re.split(r"[,;]\s*", "a, b;c") # ['a', 'b', 'c']
EMAIL = re.compile(r"[\w.+-]+@[ \w-]+\.[\w.]+", re.IGNORECASE)
bool(EMAIL.fullmatch("Me@Site.com")) # True
[m.start() for m in re.finditer("o", "foo")] # [1, 2]
```

<code>\d \w \s</code>	digit · word char · whitespace (<code>\D \W \S</code> = NOT)
<code>.</code> <code>^</code> <code>\$</code>	any char · start · end
<code>* + ? {2,5}</code>	0+ · 1+ · 0 or 1 · 2 to 5 — add <code>?</code> for lazy: <code>.*?</code>
<code>[a-z]</code> <code>[\^0-9]</code>	any of · none of
<code>(...)</code> <code>(?:...)</code> <code>(?P<n>...)</code>	group · non-capturing · named group
<code>a b</code> <code>\b</code>	either · word boundary

Tip: always use raw strings `r"..."` for patterns.

6.4 Command-line arguments

```
import argparse
parser = argparse.ArgumentParser(description="Summarise a folder")
parser.add_argument("folder") # required
parser.add_argument("--limit", type=int, default=5)
parser.add_argument("-v", "--verbose", action="store_true")
# () = real CLI args
args = parser.parse_args(["docs", "--limit", "3"])
# ('docs', 3, False)
args.folder, args.limit, args.verbose
```

python summarise.py --help prints usage automatically.

6.5 random, math, statistics

```
import random, math, statistics
random.seed(42) # same "random" numbers every run
random.randint(1, 6) # 1-6 inclusive
random.random() # 0.0 ≤ x < 1.0
random.choice(["a", "b", "c"]) # one item
random.sample(range(100), 3) # 3 unique items
random.shuffle(deck) # in place
math.sqrt(16), math.ceil(2.1), math.floor(2.9) # 4.0, 3, 2
# 1.0, 3.0, 3.14159...
math.log(math.e), math.log10(1000), math.pi
math.inf, math.isnan(float("nan")) # inf, True
statistics.mean([1, 2, 3, 4]) # 2.5
statistics.median([3, 1, 2]) # 2
statistics.stdev([2, 4, 4, 4, 5, 5, 7, 9]) # 2.138...
```

6.2 Dates & times

```
from datetime import datetime, date, timedelta, timezone
# Windows: pip install tzdata
from zoneinfo import ZoneInfo

d = datetime(2026, 1, 15, 9, 30)
d + timedelta(days=7, hours=2) # 2026-01-22 11:30:00
d.strftime("%d %b %Y, %H:%M") # '15 Jan 2026, 09:30'
d.isoformat() # '2026-01-15T09:30:00'
datetime.strptime("15/01/2026", "%d/%m/%Y").date() # 2026-01-15
# parse ISO text
datetime.fromisoformat("2026-01-15T09:30")
(date(2026, 3, 1) - date(2026, 2, 1)).days # 28
date(2026, 1, 15).weekday() # 3 (Mon = 0)

# aware: knows its time zone
now = datetime.now(timezone.utc)
utc = datetime(2026, 1, 1, tzinfo=timezone.utc)
# '05:30'
utc.astimezone(ZoneInfo("Asia/Kolkata")).strftime("%H:%M")
```

<code>%Y %m %d</code>	2026 · 01 · 15	<code>%H %M %S</code>	09 · 30 · 00
<code>%b %B</code>	Jan · January	<code>%a %A</code>	Thu · Thursday

Watch out: store and compare times in UTC; convert to local time only to display.

6.6 IDs, hashes, secrets, timing

```
import uuid, hashlib, secrets, time
str(uuid.uuid4()) # random unique id
hashlib.sha256(b"hello").hexdigest()[12:] # '2cf24dba5fb0'
# tokens / passwords (not random!)
secrets.token_urlsafe(16)
# seconds since 1970 (wall clock)
time.time()
# use this to measure durations
start = time.perf_counter()
time.sleep(0.5) # pause (blocking)
```

6.3 os, sys, shutil, subprocess

```
import os, sys, shutil, subprocess
# environment variable or None
os.environ.get("OPENAI_API_KEY")
# current folder, its entries
os.getcwd(), os.listdir(".")
os.makedirs("out/logs", exist_ok=True)
# ['script.py', 'arg1', ...]
sys.argv
sys.version_info >= (3, 10) # True on 3.10+
sys.exit(1) # stop with an error code
shutil.copy("a.txt", "b.txt"); shutil.rmtree("build")

result = subprocess.run(["git", "status", "--short"],
                        capture_output=True, text=True, check=True)
# output text, 0 = success
result.stdout, result.returncode
```

Watch out: pass a **list** to `subprocess.run` — avoid `shell=True` with user input.

7.1 Choose the right tool

Your code is mostly...	Use
waiting — API calls, downloads, DB queries	asyncio (best) or threads
computing — maths, parsing, image work	processes (ProcessPoolExecutor)
waiting, but the library is blocking	threads, or asyncio.to_thread
a few simple steps	plain code — don't add concurrency

The **GIL** lets only one thread run Python bytecode at a time, so threads speed up **waiting**, not **computing**. (3.13+ offers an experimental free-threaded build.)

7.2 Thread pool — many blocking calls

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import time

def fetch(i):
    # stands in for a blocking HTTP call
    time.sleep(0.2)
    return i * 10

with ThreadPoolExecutor(max_workers=5) as pool:
    # ~0.2s instead of 1s
    results = list(pool.map(fetch, range(5)))
    # [0, 10, 20, 30, 40] — input order
    results

with ThreadPoolExecutor() as pool:
    futures = {pool.submit(fetch, i): i for i in range(3)}
    # as each one finishes
    for future in as_completed(futures):
        print(futures[future], future.result())
```

7.3 Shared state needs a lock

```
import threading
counter, lock = 0, threading.Lock()

def add():
    global counter
    for _ in range(10_000):
        with lock:
            counter += 1

threads = [threading.Thread(target=add) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
# wait for all to finish
# 40000 — always, thanks to the lock
counter
```

Tip: prefer passing results back (futures, queues) over sharing variables.

7.4 Process pool — CPU-heavy work

```
from concurrent.futures import ProcessPoolExecutor

def heavy(n):
    return sum(i * i for i in range(n))

# required: workers re-import this file
if __name__ == "__main__":
    # one worker per CPU core
    with ProcessPoolExecutor() as pool:
        print(list(pool.map(heavy, [10**6] * 4)))
```

Arguments and results are pickled between processes — keep them small and simple.

7.5 asyncio — thousands of waits on one thread

```
import asyncio

# "async def" makes a coroutine
async def fetch(i):
    # await = pause, let others run
    await asyncio.sleep(0.2)
    return i

async def main():
    # run together; results come back in input order
    results = await asyncio.gather(*(fetch(i) for i in range(5)))

    # 3.11+ TaskGroup: waits for all; cancels the rest on failure
    async with asyncio.TaskGroup() as tg:
        t1 = tg.create_task(fetch(1))
        t2 = tg.create_task(fetch(2))

    # give up after a time limit (3.11+)
    try:
        async with asyncio.timeout(0.1):
            await fetch(99)
    except TimeoutError:
        print("too slow")

    # at most 3 running at once (API rate limits)
    limit = asyncio.Semaphore(3)
    async def limited(i):
        async with limit:
            return await fetch(i)
    done = await asyncio.gather(*(limited(i) for i in range(6)))

    # run blocking code without freezing the event loop
    await asyncio.to_thread(time.sleep, 0.1)
    return results, t1.result() + t2.result(), len(done)

asyncio.run(main())
# ([0, 1, 2, 3, 4], 3, 6)
```

```
for next_done in asyncio.as_completed([fetch(i) for i in range(3)]):
    # in finishing order (inside async def)
    print(await next_done)
```

7.6 async mistakes

Mistake	Fix
calling <code>fetch()</code> without <code>await</code> → a coroutine object, nothing runs	<code>await fetch()</code>
<code>time.sleep</code> / requests inside <code>async def</code> blocks everything	<code>await asyncio.sleep</code> , <code>httpx.AsyncClient</code>
<code>await</code> in a loop → still one at a time	<code>gather</code> / <code>TaskGroup</code>
gathering 1,000 API calls at once → rate limited	<code>asyncio.Semaphore(n)</code>
<code>asyncio.run()</code> inside Jupyter fails	just <code>await main()</code> in the cell

8.1 Type hints

```
def top_k(items: list[str], k: int = 3) -> list[str]:
    return items[:k]

scores: dict[str, float] = {}
point: tuple[int, int] = (1, 2)
names: tuple[str, ...] = ("a", "b") # any length
maybe: str | None = None # = Optional[str]
value: int | str = 5 # = Union[int, str]

from typing import Any, Callable, Final, Literal, TypedDict
# function taking str -> str
Handler = Callable[[str], str]
Mode = Literal["fast", "accurate"] # only these values
# constant - don't reassign
MAX_TOKENS: Final = 4096

class Message(TypedDict): # dict with known keys
    role: Literal["system", "user", "assistant"]
    content: str

# generic function (3.12+)
def first[T](items: list[T]) -> T:
    return items[0]
type Vector = list[float] # type alias (3.12+)
```

Hints are **not enforced at run time**. Check them with `mypy src/` or your editor; validate data with Pydantic.

8.2 Environments & packages

```
# built-in venv + pip
python -m venv .venv
# Windows (macOS/Linux: source .venv/bin/activate)
.venv\Scripts\activate
pip install "openai>=1.40" pydantic
pip freeze > requirements.txt # pin exact versions
pip install -r requirements.txt

# uv - one fast tool for all of it
uv init --no-package my-app # project + pyproject.toml
uv add openai pydantic # install + record + lock
uv add --dev pytest ruff # dev-only tools
uv run main.py # run inside the project env
uv sync --locked # CI / Docker: exact lockfile
```

8.3 pyproject.toml

```
[project]
name = "rag-bot"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = ["openai>=1.40", "pydantic>=2"]

[dependency-groups]
dev = ["pytest", "ruff", "mypy"]

[tool.ruff]
line-length = 100

[tool.pytest.ini_options]
testpaths = ["tests"]
```

8.4 Project layout

```
rag-bot/
├── pyproject.toml
├── .env # secrets - in .gitignore
├── src/rag_bot/
│   ├── __init__.py # makes it a package
│   ├── config.py # settings from env vars
│   ├── retrieval.py
│   └── main.py # entry point
├── tests/
│   ├── conftest.py # shared fixtures
│   └── test_retrieval.py

```

```
from rag_bot.retrieval import search # absolute import
# relative (inside the package)
from .config import settings

def main(): ...
# only when run directly,
if __name__ == "__main__":
    main() # not when imported
```

8.5 Code quality tools

```
ruff format . # format every file
ruff check . --fix # lint: bugs, unused imports, style
mypy src/ # static type check
pytest -q # run tests
```

Tip: turn on "format on save" in your editor and run the same commands in CI.

8.6 Debugging

```
print(f"{user=} {type(user)=}") # quickest look at a value
# pause here and open the debugger
breakpoint()
# print the current error
import traceback; traceback.print_exc()
```

pdb command	Does
n / s	next line / step into a call
c / q	continue / quit
p expr · pp expr	print a value · pretty-print
l · w	show code here · show the call stack
b 42	breakpoint at line 42

Read a traceback **bottom-up**: the last line says *what*, the lowest line in *your* file says *where*.

8.7 pytest basics

```
# calc.py
def average(nums):
    if not nums:
        raise ValueError("empty list")
    return sum(nums) / len(nums)

# test_calc.py - files and functions start with "test_"
import pytest
from calc import average

def test_average():
    assert average([1, 2, 3]) == 2 # plain assert

def test_empty_list_raises():
    with pytest.raises(ValueError, match="empty"):
        average([])

def test_floats():
    assert 0.1 + 0.2 == pytest.approx(0.3) # never == on floats

@pytest.mark.parametrize("nums, expected", [
    ([4], 4),
    ([1, 2], 1.5),
    ([-1, 1], 0),
])
def test_many_cases(nums, expected): # runs once per row
    assert average(nums) == expected
```

8.8 Running tests

pytest -q	quiet summary
pytest -k average	only tests whose name matches
pytest -x · --lf	stop at first failure · rerun last failures
pytest -vv -s	verbose · show print output
pytest --cov=src	coverage (needs pytest-cov)

8.9 Fixtures — reusable setup

```
import pytest

@pytest.fixture
# tmp_path: a fresh temp folder
def docs_dir(tmp_path):
    doc = tmp_path / "faq.md"
    doc.write_text("Refunds in 30 days", encoding="utf-8")
    yield tmp_path # the test runs here
    # code after yield = teardown (runs even if the test fails)

def test_loads_docs(docs_dir): # ask for it by name
    files = list(docs_dir.glob("*.md"))
    assert len(files) == 1

# build once for all tests
@pytest.fixture(scope="session")
def big_index():
    return {"refunds": [1, 2, 3]}
```

Put shared fixtures in `tests/conftest.py` — every test file can use them without importing.

8.10 Mocking — test without the real API

```
from unittest.mock import MagicMock, patch

def summarise(client, text): # code under test
    reply = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": text}])
    return reply.choices[0].message.content.strip()

def test_summarise_uses_reply():
    # fake client: no key, no cost
    client = MagicMock()
    message = MagicMock(content=" Short summary ")
    client.chat.completions.create.return_value.choices = [
        MagicMock(message=message)]
    assert summarise(client, "long text") == "Short summary"
    client.chat.completions.create.assert_called_once()

def test_failure_path():
    client = MagicMock()
    client.chat.completions.create.side_effect = ConnectionError
    # side_effect = raise this
    with pytest.raises(ConnectionError):
        summarise(client, "x")

def test_no_real_sleep():
    # replace for this block
    with patch("time.sleep") as fake_sleep:
        time.sleep(30) # returns instantly
    fake_sleep.assert_called_once_with(30)
```

Tip: pass clients in as arguments (dependency injection) — then tests need no patching at all.

8.11 monkeypatch — env vars & attributes

```
import os

def test_reads_model_from_env(monkeypatch):
    # undone after the test
    monkeypatch.setenv("LLM_MODEL", "test-model")
    assert os.getenv("LLM_MODEL") == "test-model"

def test_missing_key(monkeypatch):
    monkeypatch.delenv("OPENAI_API_KEY", raising=False)
    assert os.getenv("OPENAI_API_KEY") is None
```

8.12 What to test

- Pure logic first: parsing, chunking, prompt building, scoring.
- Edge cases: empty input, one item, huge input, bad types.
- Error paths: API down, timeout, invalid JSON from the model.
- Fake the LLM in unit tests; run a few real "smoke" calls separately.

9.1 NumPy arrays

```
import numpy as np
a = np.array([1, 2, 3])
np.zeros((2, 3)), np.ones(3), np.full(2, 7) # filled arrays
np.arange(0, 10, 2) # [0 2 4 6 8]
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1. ]
m = np.arange(6).reshape(2, 3) # [[0 1 2]
# [3 4 5]]
m.shape, m.ndim, m.size # (2, 3), 2, 6
m.T.shape # (3, 2) transpose
m[1, 2], m[0], m[:, 0] # 5, [0 1 2], [0 3]
m[m > 2] # [3 4 5] boolean mask
# [2 4 6] [2 4 6] [1 4 9] — no loops
a * 2, a + a, a ** 2
m + np.array([10, 20, 30]) # broadcasting: added to each row
m.sum(), m.sum(axis=0), m.mean(axis=1) # 15, [3 5 7], [1. 4.]
a @ a # 14 (dot product)
np.argsort([30, 10, 20]) # [1 2 0] indices that would sort
np.where(a > 1, a, 0) # [0 2 3]
np.random.default_rng(42).normal(size=3) # reproducible randoms
```

Tip: replace Python loops over numbers with array operations — often 10–100x faster.

9.2 Cosine similarity — the core of embeddings

```
def cosine(a, b):
    return float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b)))

cosine(np.array([1, 0]), np.array([1, 1])) # 0.7071... (45°)

E = np.random.default_rng(0).normal(size=(100, 8)) # 100 vectors
q = E[7] # query vector
scores = E @ q / (np.linalg.norm(E, axis=1) * np.linalg.norm(q))
# top 3 — index 7 first
np.argsort(-scores)[:3]
```

9.3 pandas — DataFrames

```
import pandas as pd
df = pd.DataFrame({
    "name": ["Asha", "Ravi", "Meera", "Kabir"],
    "team": ["AI", "AI", "Data", "Data"],
    "salary": [120, 95, 110, None],
})
# (4, 3), ['name', 'team', 'salary']
df.shape, list(df.columns)
df.head(2); df.info(); df.describe() # Look at the data first

df["salary"] # one column → Series
df[["name", "salary"]] # several columns → DataFrame
df[df["team"] == "AI"] # filter rows
# & | ~ with brackets
df[(df.salary > 100) & (df.team == "AI")]
df.loc[df["salary"] > 100, "name"].tolist() # ['Asha', 'Meera']
df.iloc[0, 0] # 'Asha' (by position)
df.query("team == 'Data'") # filter with a string

# new column (vectorised)
df["bonus"] = df["salary"] * 0.1
# 108.33... (skips missing)
df["salary"].mean()
df.groupby("team")["salary"].agg(["mean", "count"])
#          mean count
# AI      107.5     2
# Data   110.0     1

# 'Asha'
df.sort_values("salary", ascending=False).iloc[0]["name"]
df["team"].value_counts()["AI"] # 2
```

9.4 Cleaning, joining, saving

```
df["salary"].isna().sum() # 1 missing value
df.fillna({"salary": 0}) # fill (returns a new DataFrame)
df.dropna(subset=["salary"]) # drop rows with missing salary
df.drop_duplicates(); df.rename(columns={"name": "employee"})
df["salary"].astype("Int64") # nullable integer type

teams = pd.DataFrame({"team": ["AI", "Data"], "floor": [3, 5]})
df.merge(teams, on="team", how="left") # SQL-style join
pd.concat([df, df]) # stack rows

# also read_json, read_excel, read_parquet
df = pd.read_csv("data.csv")
df.to_csv("out.csv", index=False)
```

Watch out: avoid `df.apply` / loops over rows when a column operation exists — they're much slower.

9.5 Charts with matplotlib

```
import matplotlib.pyplot as plt # pip install matplotlib

fig, (left, right) = plt.subplots(1, 2, figsize=(8, 3))
days, p95 = [1, 2, 3, 4], [220, 180, 240, 160]
left.plot(days, p95, marker="o", label="p95 ms")
left.set(title="Latency", xlabel="day", ylabel="ms")
left.legend()
right.bar(["AI", "Data"], [107.5, 110.0], color="#0284c7")
right.set_title("Avg salary by team")
fig.tight_layout()
# or plt.show() on your machine
fig.savefig("report.png", dpi=150)

df.plot(kind="bar", x="name", y="salary") # pandas shortcut
plt.hist(np.random.default_rng(1).normal(size=500), bins=20)
```

Tip: use `fig, ax = plt.subplots()` and call methods on `ax` — clearer than the global `plt.*` style.

9.6 Calling APIs (httpx / requests)

```
# pip install httpx (requests: same API, sync only)
import httpx

r = httpx.get("https://api.example.com/items",
              params={"page": 2},           # → ?page=2
              headers={"Authorization": f"Bearer {token}"},
              # always set a timeout
              timeout=10)

r.raise_for_status()           # raise on 4xx / 5xx
r.status_code, r.json()       # 200, parsed JSON

r = httpx.post(url, json={"query": "refunds"}, timeout=10)

with httpx.Client(base_url="https://api.example.com",
                  timeout=10) as client:
    client.get("/items")        # reuses connections — faster

# inside async def
async with httpx.AsyncClient(timeout=10) as client:
    responses = await asyncio.gather(*(client.get(u) for u in urls))

Status 200 OK · 201 created · 400 bad request · 401/403 auth · 404 not found ·
422 invalid data · 429 rate limit · 5xx server error
```

9.7 FastAPI — build an API

```
# main.py — pip install fastapi uvicorn
# run it: uvicorn main:app --reload
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel, Field

app = FastAPI(title="RAG API")

class Question(BaseModel):           # request body, validated
    text: str = Field(min_length=3)
    top_k: int = Field(default=3, ge=1, le=10)

class Answer(BaseModel):             # response shape
    answer: str
    sources: list[str]

DOCS = {"refunds": "Refunds within 30 days.",
        "shipping": "Free over ₹500."}

@app.get("/health")
def health():
    return {"status": "ok"}           # dict → JSON

@app.get("/documents/{doc_id}")       # path parameter
def get_document(doc_id: str):
    if doc_id not in DOCS:
        raise HTTPException(status_code=404, detail="not found")
    return {"id": doc_id, "text": DOCS[doc_id]}

# query parameters: ?q=...&limit=...
@app.get("/search")
def search(q: str = Query(min_length=2), limit: int = 5):
    hits = [k for k, v in DOCS.items() if q.lower() in v.lower()]
    return {"hits": hits[:limit]}

@app.post("/ask", response_model=Answer)
# async: can await LLM calls
async def ask(question: Question):
    return Answer(answer=f"You asked: {question.text}",
                  sources=list(DOCS)[: question.top_k])
```

Open <http://127.0.0.1:8000/docs> for free interactive API docs.

9.8 Testing the API — no server needed

```
from fastapi.testclient import TestClient
client = TestClient(app)

# {'status': 'ok'}
client.get("/health").json()
client.get("/documents/refunds").status_code      # 200
client.get("/documents/nope").status_code          # 404
# {'hits': ['shipping']}
client.get("/search", params={"q": "free"}).json()
client.post("/ask", json={"text": "hi"}).status_code  # 422
client.post("/ask", json={"text": "refund?", "top_k": 1}).json()
# → {'answer': 'You asked: refund?', 'sources': ['refunds']}
```

9.9 Streaming responses

```
from fastapi.responses import StreamingResponse

@app.get("/stream")
def stream():
    def tokens():
        for word in ["Hello", " from", " the", " model"]:
            # sent to the client as produced
            yield word
    return StreamingResponse(tokens(), media_type="text/plain")
```

Tip: keep secrets in env vars, add CORS only for your own front-end, and rate-limit public endpoints.

9.10 Pydantic models in depth

```
from pydantic import (BaseModel, Field, ValidationError,
                      field_validator)

class Signup(BaseModel):
    name: str = Field(min_length=2)
    age: int = Field(ge=13)
    email: str
    # safe here: Pydantic copies defaults
    tags: list[str] = []

    @field_validator("email")
    @classmethod
    def normalise_email(cls, v: str) -> str:
        if "@" not in v:
            raise ValueError("invalid email")
        return v.lower()

# "25" is converted
s = Signup(name="Asha", age="25", email="ASHA@X.COM")
s.age, s.email           # 25, 'asha@x.com'
# {'name': 'Asha', 'age': 25, 'email': 'asha@x.com', 'tags': []}
s.model_dump()
s.model_dump_json()      # the same as a JSON string
s.model_copy(update={"age": 26}).age  # 26
Signup.model_json_schema()["required"] # ['name', 'age', 'email']
try:
    Signup(name="A", age=5, email="x")
except ValidationError as err:
    # 3 — every problem reported at once
    err.error_count()
```

9.11 SQL databases with sqlite3

```
import sqlite3
# ":memory:" = throwaway database
con = sqlite3.connect("app.db")
con.row_factory = sqlite3.Row           # rows behave like dicts
con.execute("CREATE TABLE IF NOT EXISTS users "
            "(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)")
# commit, or roll back on error
with con:
    sql = "INSERT INTO users (name, age) VALUES (?, ?)"
    con.execute(sql, ("Asha", 25))
    con.executemany(sql, [("Ravi", 31), ("Meera", 28)])
query = "SELECT name, age FROM users WHERE age > ? ORDER BY age"
rows = con.execute(query, (26,)).fetchall()
[r["name"] for r in rows]              # ['Meera', 'Ravi']
con.close()
```

Watch out: never build SQL with f-strings — always use `?` placeholders (prevents SQL injection).

10.1 Secrets & client setup

```
import os
# pip install python-dotenv
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()  # reads .env → os.environ
# reads OPENAI_API_KEY itself
client = OpenAI(
    timeout=30,  # seconds per request
    # retries 429 / 5xx with back-off
    max_retries=3,
)
MODEL = os.getenv("LLM_MODEL", "gpt-4o-mini")
```

Watch out: never hard-code keys; add .env to .gitignore; revoke a leaked key at once.

10.2 Chat call & streaming

```
messages = [
    {"role": "system", "content": "You are a concise assistant."},
    {"role": "user", "content": "What is RAG?"},
]
reply = client.chat.completions.create(
    model=MODEL, messages=messages, temperature=0.2)
text = reply.choices[0].message.content
# cost tracking
reply.usage.prompt_tokens, reply.usage.completion_tokens

stream = client.chat.completions.create(
    model=MODEL, messages=messages, stream=True)
# print tokens as they arrive
for part in stream:
    print(part.choices[0].delta.content or "", end="", flush=True)

# keep history
messages.append({"role": "assistant", "content": text})
```

10.3 Structured output with Pydantic

```
from pydantic import BaseModel, Field, ValidationError

class Ticket(BaseModel):
    category: str
    priority: int = Field(ge=1, le=5)

reply = client.chat.completions.create(
    model=MODEL,
    response_format={"type": "json_object"},  # JSON mode
    messages=[{"role": "system", "content":
        'Reply as JSON: {"category": str, "priority": 1-5}'},
        {"role": "user", "content": "Payment failed twice!"}])
raw = reply.choices[0].message.content
try:
    ticket = Ticket.model_validate_json(raw)
except ValidationError as err:
    # re-ask the model, including str(err) so it can fix itself
    ...

Ticket.model_validate_json('{"category": "billing", "priority": 4}')
# Ticket(category='billing', priority=4)
```

10.4 Tokens & cost

```
def estimate_tokens(text: str) -> int:
    # = 4 characters per token (English)
    return max(1, len(text) // 4)

def cost_usd(tokens_in, tokens_out, in_per_m, out_per_m):
    return tokens_in / 1e6 * in_per_m + tokens_out / 1e6 * out_per_m

estimate_tokens("Retrieval-augmented generation")  # 7
cost_usd(2_000, 500, in_per_m=0.15, out_per_m=0.60)  # 0.0006 ($)
```

Prices per million tokens are examples — check your provider's pricing page. Exact counts: the tiktoken package.

10.5 Chunking with overlap

```
def chunk(text, size=200, overlap=40):
    """Split into ~size-word chunks that share `overlap` words."""
    words = text.split()
    step = size - overlap
    return [" ".join(words[i:i + size])
            for i in range(0, max(len(words) - overlap, 1), step)]

doc = " ".join(f"w{i}" for i in range(500))
pieces = chunk(doc)
len(pieces), pieces[1].split()[0]  # 3, 'w160'
```

10.6 Embeddings & retrieval

```
import numpy as np
emb = client.embeddings.create(model="text-embedding-3-small",
                               input=pieces)

# one row per chunk
M = np.array([d.embedding for d in emb.data])

def top_k(query_vec, M, k=3):
    scores = M @ query_vec / (np.linalg.norm(M, axis=1)
                              * np.linalg.norm(query_vec))

    # best chunk indices
    return np.argsort(-scores)[:k]

q = client.embeddings.create(model="text-embedding-3-small",
                              input=["How long do refunds take?"])
hits = [pieces[i] for i in top_k(np.array(q.data[0].embedding), M)]
```

10.7 A grounded RAG prompt

```
def build_prompt(question, hits):
    numbered = (f"[{i}] {h}" for i, h in enumerate(hits, 1))
    context = "\n\n".join(numbered)
    return ("Answer ONLY from the context. Cite sources like [1]. "
            "If the answer is not there, say you don't know.\n\n"
            f"<context>\n{context}\n</context>\n\n"
            f"Question: {question}")

build_prompt("Refund time?", ["30 days."]).splitlines()[-1]
# 'Question: Refund time?'
```

10.8 Many calls at once, safely

```
import asyncio
from openai import AsyncOpenAI
aclient = AsyncOpenAI(timeout=30, max_retries=3)
limit = asyncio.Semaphore(5)  # respect rate limits

async def summarise(text):
    async with limit:
        r = await aclient.chat.completions.create(
            model=MODEL,
            messages=[{"role": "user", "content": text}])
        return r.choices[0].message.content

async def summarise_all(texts):
    return await asyncio.gather(*(summarise(t) for t in texts),
                                # keep partial results
                                return_exceptions=True)
```

10.9 Production checklist

- Timeouts + retries on every client; fall back to a second provider if needed.
- Validate model output (Pydantic) before using it.
- Log tokens, latency and cost per request — not prompts with personal data.
- Cache repeated calls; trim long chat history before sending.
- Evaluate with a fixed set of questions before every prompt change.

11.1 Names, objects & mutability

```

a = [1, 2]
b = a                # b is another NAME for the same List
b.append(3)
a                    # [1, 2, 3]
# True, True (same object · equal value)
a is b, a == [1, 2, 3]
# False (equal value, different object)
a is [1, 2, 3]

s = "hi"
t = s
# strings are immutable: t is a NEW object
t += "!"
s, t                 # 'hi', 'hi!'

def add(item, bucket):
    bucket.append(item) # mutates the caller's List
nums = []
add(1, nums); nums   # [1]

```

Immutable	int float bool str tuple frozenset bytes None
-----------	---

Mutable	list dict set bytearray, most custom objects
---------	--

Python passes **references to objects** ("call by object reference"): mutating changes the caller's object, rebinding doesn't.

11.2 Classic gotchas

Trap	Fix
<code>def f(x=[])</code> shares one list	<code>x=None → x = [] if x is None else x</code>
<code>[[0] * 3] * 3</code> → 3 refs to one row	<code>[[0] * 3 for _ in range(3)]</code>
<code>b = a</code> isn't a copy	<code>a.copy()</code> / <code>copy.deepcopy(a)</code>
removing items while looping	loop over a copy / build a new list
<code>[lambda: i for i in range(3)]</code> all give 2	<code>lambda i:i: i</code>
<code>if not x</code> treats <code>0</code> / <code>""</code> as missing	<code>if x is None</code>
<code>is</code> to compare values	<code>==</code> ; <code>is</code> only for <code>None</code>
<code>0.1 + 0.2 == 0.3</code> is False	<code>math.isclose</code> / <code>Decimal</code>
<code>lst = lst.sort()</code> → None	<code>lst.sort()</code> or <code>sorted(lst)</code>
a generator used twice is empty	make a list, or create it again
bare <code>except:</code> hides bugs	catch specific exceptions
<code>round(2.5)</code> is 2	banker's rounding; use <code>Decimal</code> for money

11.3 Quick answers

list vs tuple	list is mutable; tuple is fixed, hashable, slightly faster
shallow vs deep copy	shallow copies the container; deep copies everything inside
iterable vs iterator	iterable gives an iterator; an iterator yields items once
generator vs list	generator is lazy: O(1) memory, single pass
decorator	a function that wraps another to add behaviour
@classmethod vs @staticmethod	gets <code>cls</code> (factories) vs gets nothing (helper)
__init__ vs __new__	<code>__new__</code> creates the object, <code>__init__</code> fills it in
*args / **kwargs	extra positional → tuple; extra keyword → dict
GIL	one thread runs bytecode at a time: threads for I/O, processes for CPU
hashable	has a stable hash; immutable built-ins are, lists aren't
duck typing	"if it has the method, use it" — no inheritance needed
with statement	guaranteed setup/cleanup via <code>__enter__</code> / <code>__exit__</code>
dict order	insertion order is guaranteed since Python 3.7

11.4 Patterns worth memorising

```

from collections import Counter, defaultdict, deque
Counter(words).most_common(3) # top-k frequent
seen, dupes = set(), set()    # duplicates in O(n)
for x in nums:
    if x in seen: dupes.add(x)
    seen.add(x)
pairs = {}                    # two-sum in O(n)
for i, n in enumerate(nums):
    if target - n in pairs: break
    pairs[n] = i
window = deque(maxlen=3)      # sliding window
# adjacency list for BFS / DFS
graph = defaultdict(list)
# BFS queue: popLeft() is O(1)
queue = deque([start])

```

11.5 Interview revision checklist

- ☐ Explain mutable vs immutable and what `b = a` really does
- ☐ List / tuple / set / dict — when to use each, and their Big-O
- ☐ Write a comprehension, a generator and a decorator from memory
- ☐ `*args` / `**kwargs`, default-argument trap, LEGB scope
- ☐ Classes: `@property`, `@classmethod`, inheritance & `super()`, dataclasses
- ☐ Errors: catch specific exceptions, `raise ... from`, `finally`
- ☐ Concurrency: GIL, threads vs processes vs `asyncio` — pick one for a task
- ☐ Testing: a `pytest` test with a fixture and a mock

11.6 Keep learning — free

- **Notes:** Python Basics → Advanced, with exercises
 - **Tutorials:** calling LLMs, uv, Git, error handling, `async`
 - **Templates:** build a RAG chatbot step by step
 - **Architecture:** production RAG and AI agent design
- learn.aiwithpriyabrat.com — new notes every month.