das-purnendu

The Python Interview Handbook

35 slides.
Junior to senior.

Everything that
actually gets asked.

Save it once.
Use it for every
interview.

5+ years writing Python in production.

Distilled from 30+ rounds — what hiring managers actually probe, not textbook trivia.

Save this slide for your next interview 👇

das-purnendu

HOW TO USE THIS HANDBOOK

Slides 3–8: Core data structures & operations

Slides 9–12: **Functional Python**
(comprehensions, itertools, decorators, generators)

Slides 13–16: OOP deep dive

Slides 17–20: Files, OS, datetime, errors, logging

Slides 21–24: Concurrency, async, GIL, performance

Slides 25–28: Testing, typing, packaging, code quality

Slides 29–32: Algorithm patterns + complexity

Slides 33–35: Traps, one-liners, final checklist

Bookmark the section you're weakest at.
Revisit before every round.

Save this slide for your next interview 👇

das-purnendu

DATA STRUCTURES: PICK THE RIGHT ONE

Need	Use	Why	Ordered, mutable
Growable sequence	list	O(1) append, O(n) search	✓ ✓
Unique items	set	O(1) lookup	✗ ✓
Key → value mapping	dict	O(1) lookup, ordered (3.7+)	✓ ✓
Counting occurrences	Counter	Counter(arr) .most_common(k)	✗ ✓
Queue / stack	deque	O(1) appendleft/ popleft	✓ ✓
Default values	defaultdict	No KeyError	✓ ✓
Min/Max heap	heapq	O(log n) push/pop	✗ ✓
Immutable sequence	tuple	Hashable, faster	✓ ✗
Frozen set	frozenset	Hashable set	✗ ✗
Sorted lookup	bisect on list	O(log n) search	✓ ✗

Trap: using **list** where **set** belongs.
Interviewer notices instantly.

Save this slide for your next interview 👇

das-purnendu

LISTS: EVERYTHING YOU'LL NEED

python

```
arr = [3, 1, 4, 1, 5]
```

Basic operations

```
arr.sort()                    # in-place, returns None
sorted(arr, reverse=True)     # new list
arr[::-1]                     # reverse
arr[::2]                      # every 2nd
arr[-3:]                      # last 3
```

Sort by custom key

```
words.sort(key=len)
pairs.sort(key=lambda x: (-x[1], x[0]))    # desc val, asc key
```

Unpacking

```
a, *rest, b = [1, 2, 3, 4, 5]
```

Flatten

```
flat = [x for row in matrix for x in row]
```

In-place vs new

```
arr.reverse()   vs   arr[::-1]
arr.sort()      vs   sorted(arr)
```

Insert / remove

```
arr.insert(0, x); arr.pop(0)    # both O(n)
                                → use deque
arr.remove(x)                   # removes first occurrence
```

Key takeaway: Lists are powerful but not always optimal. Know when to switch to **set**, **dict**, or **deque**.

Save this slide for your next interview 👇

das-purnendu

DICT & SET MASTERY

Counting safely

```
# Counting safely
d[key] = d.get(key, 0) + 1
```

defaultdict / Counter

```
from collections import defaultdict, Counter
groups = defaultdict(list)
groups[key].append(val)
```

Merge dicts (3.9+)

```
merged = d1 | d2
d1 |= d2              # in-place
```

Invert

```
inv = {v: k for k, v in d.items()}
```

Sort dict by value

```
sorted(d.items(), key=lambda x: x[1],
                  reverse=True)
```

Set ops

```
a & b     # intersection
a | b     # union
a - b     # difference
a ^ b     # symmetric difference
```

dict.setdefault — single-line grouping

```
for k, v in pairs:
    d.setdefault(k, []).append(v)
```

`if key in d: d[key]+=1 else: d[key]=1`
→ instant junior signal.

Save this slide for your next interview 👇

das-purnendu

STRINGS: THE SILENT INTERVIEW KILLER

```python
s = "hello world"
```

Basic operations

```
s.split(); "-".join(["a","b"])
s.strip(); s.lower(); s.upper()
s.replace("l", "L")
s.startswith("he"); s.endswith("ld")
s.find("o")              # -1 if not found
s.count("l")
s.isdigit(); s.isalpha();
s.isalnum(); s.isspace()
```

f{} f-strings — use these, always

```
f"{name=}, {score:.2f}, {n:,}, {x:>10}, {d:%Y-%m-%d}"
```

↑	↑	↑	↑	↑
shows name and value	2 decimal places	thousands separator	right align in width 10	date format

Reverse / palindrome / anagram

```
s[::-1]                  # reverse
s == s[::-1]             # palindrome?
sorted(a) == sorted(b)   # anagram?
```

Concat in loop = O(n²)

```
res = ""
for part in parts:
    res += part    # ✗ O(n²)
```

```
"".join(parts)    # ✓
# O(n) — always use this
```

Translation table (fast char replace)

```
s.translate(str.maketrans("abc", "xyz"))
# a->x, b->y, c->z (others unchanged)
```

Know these. Use them fluently.
Otherwise, you look like a beginner — instantly.

Save this slide for your next interview 👇

das-purnendu

TUPLES, NAMEDTUPLES, ENUMS

Tuple

- Immutable
- Hashable
- Faster than list

```
# Tuple — immutable, hashable, faster than list
point = (3, 4)
x, y = point
```

- Can be dict keys
- Safe default
- Slightly faster

NamedTuple

- Readable
- Still immutable

```
# NamedTuple — readable, still immutable
from collections import namedtuple
Point = namedtuple("Point", ["x", "y"])
p = Point(3, 4)
p.x, p.y     # attribute access
```

```
p.x → 3
p.y → 4
```

typing.NamedTuple

- With type hints

```
# typing.NamedTuple — with type hints
from typing import NamedTuple
class Point(NamedTuple):
    x: int
    y: int
```

- Static typing
- IDE support
- Self-documenting

Enum

- For fixed sets of values
- Safer & clearer than strings

```
# Enum — for fixed sets of values
from enum import Enum, auto
class Status(Enum):
    PENDING = auto()
    APPROVED = auto()
    REJECTED = auto()
```

```
Status.PENDING.name   # 'PENDING'
Status.PENDING.value   # 1

Status.APPROVED.value   # 2
Status.REJECTED.value   # 3
```

Use **Enum** instead of magic strings.
Less bugs. More clarity. **Reviewers love it.**

Save this slide for your **next interview** 👇

das-purnendu

COLLECTIONS MODULE (UNDERUSED GOLD)

```python
from collections import (
    Counter, defaultdict, deque,
    OrderedDict, ChainMap, namedtuple
)
```

Why it matters

- Built for performance
- Cleaner code
- Handles edge cases
- Interviewer loves to see it

Counter

```python
c = Counter("mississippi")
c.most_common(2)                    # [('i', 4), ('s', 4)]
c1 + c2; c1 - c2; c1 & c2           # union, diff, intersection
```

deque

O(1) both ends

```python
dq = deque(maxlen=5)                # auto-evicts oldest
dq.appendleft(x); dq.popleft()      # O(1)
dq.rotate(2)                        # rotate right by 2
```

★ deque(maxlen=N) for sliding windows = senior-level move.

defaultdict

```python
from collections import defaultdict
tree = lambda: defaultdict(tree)        # infinite nested dict
t = tree()                              # create the root
t['a']['b']['c'] = 1                    # works!
```

ChainMap

Layered lookups (configs!)

```python
from collections import ChainMap
config = ChainMap(cli_args, env_vars, defaults)
config['timeout']      # checks in order
```

Perfect for configs: CLI args > ENV vars > defaults

OrderedDict

Order + extra methods

```python
from collections import OrderedDict
od = OrderedDict([('a', 1), ('b', 2)])
od.move_to_end('a')                     # move key to end
od.popitem(last=False)                  # pop from beginning
```

namedtuple

Lightweight structs

```python
from collections import namedtuple
Point = namedtuple('Point', 'x y')
p = Point(10, 20)
p.x, p.y                                # fast, readable, immutable
```

Avoid reinvention. collections = clarity + speed.
Use it when the default tools start to feel clumsy.

Save this slide for your next interview 👇

das-purnendu

COMPREHENSIONS, LAMBDAS, MAP/FILTER

Comprehensions

- Concise
- Fast
- Pythonic

```
# Comprehensions
squares = [x*x for x in range(10)]
evens   = [x for x in arr if x % 2 == 0]
d       = {x: x*x for x in range(5)}
unique  = {x.lower() for x in words}
```

Nested

- Multiple for/if

```
# Nested
pairs = [(i, j) for i in range(3) for j in range(3) if i != j]
```

Generator expression

- Lazy, memory-friendly

```
# Generator expression — lazy, memory-friendly
total = sum(x*x for x in range(10**7))
```

Lambda + sorted

- Inline functions

```
# Lambda + sorted
sorted(users, key=lambda u: u["age"])
```

zip & enumerate

- Iterate smarter

```
# zip & enumerate
for i, val in enumerate(arr): ...
for a, b in zip(list1, list2): ...
for i, (a, b) in enumerate(zip(x, y)): ...
```

any / all

- Short-circuit logic

```
# any / all
all(x > 0 for x in arr)
any(w in s for w in banned)
```

Write less. Do more.
These tools separate strong from **standout.**

Save this slide for your **next interview** 👇

das-purnendu

itertools & functools

itertools

- Iterator building blocks
- Memory efficient
- Composables
- Hidden superpower

```python
from itertools import (
    product, permutations, combinations,
    chain, accumulate, groupby, islice,
    cycle, repeat, count, pairwise,     # 3.10+
    takewhile, dropwhile
)
```

product	`product([1,2], [3,4])`	`# cartesian product`
permutations	`permutations([1,2,3], 2)`	`# length-2 permutations`
combinations	`combinations([1,2,3], 2)`	`# length-2 combinations`
chain	`chain([1,2], [3,4])`	`# 1 2 3 4`
accumulate	`accumulate([1,2,3,4])`	`# 1 3 6 10`
groupby	`for k, g in groupby(data): ...`	`# group consecutive`
pairwise	`list(pairwise([1,2,3,4]))`	`# [(1,2), (2,3), (3,4)]`
islice	`islice(infinite_gen, 0, 10)`	`# slice generators`
cycle	`cycle([1,2,3])`	`# 1 2 3 1 2 3 ...`
repeat	`repeat('x', 3)`	`# x x x`
count	`count(10, 2)`	`# 10 12 14 16 ...`
takewhile	`takewhile(lambda x: x < 0, data)`	`# until condition false`
dropwhile	`dropwhile(lambda x: x < 0, data)`	`# drop until condition false`

Think in iterators. Build pipelines. Save memory.

functools

- Higher-order power tools
- Performance boosters
- Cleaner code

```python
from functools import reduce, lru_cache, cache, partial, wraps
```

reduce

```python
from functools import reduce
reduce(lambda a, b: a*b, arr, 1)        # product of all elements
```

lru_cache

```python
from functools import lru_cache
@lru_cache(maxsize=None)
def fib(n):                             # memoize, free DP
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

cache (3.9+)

```python
from functools import cache
@cache
def fib(n):                             # simpler than lru_cache
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

partial

```python
from functools import partial
def add(a, b): return a + b
add5 = partial(add, 5)                  # pre-fill arguments
add5(10)       # 15
```

wraps

```python
from functools import wraps
def deco(fn):
    @wraps(fn)                          # preserve metadata
    def wrapper(*args, **kwargs):         (name, doc, etc.)
        return fn(*args, **kwargs)
    return wrapper
```

pairwise or lru_cache in a coding round = instant senior signal.

Save this slide for your next interview 👇

das-purnendu

DECORATORS (DEEP DIVE)

1. BASIC DECORATOR

- Adds behavior
- No change to original function
- @wraps preserves metadata

```python
from functools import wraps
import time

def timer(fn):
    @wraps(fn)        # preserves metadata
    def wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        print(f"{fn.__name__}: {time.time()-start:.3f}s")
        return result
    return wrapper
```

Why @wraps?

- Preserves __name__
- __doc__
- __module__
- __annotations__
- Critical for debugging & tools

Usage

```python
@timer
def work(): ...
```

2. DECORATOR WITH ARGUMENTS

- Factory that returns a decorator
- Flexible & reusable

```python
def retry(times=3, exc=Exception):
    def deco(fn):
        @wraps(fn)
        def wrapper(*a, **kw):
            for attempt in range(1, times + 1):
                try:
                    return fn(*a, **kw)
                except exc as e:
                    if attempt == times:
                        raise
        return wrapper
    return deco
```

Usage

```python
@retry(times=5)
def flaky(): ...
# Retries up to 5 times
# on Exception
```

3. CLASS AS DECORATOR

- Stateful decorators
- Keeps state between calls
- Use __call__

```python
class CountCalls:
    def __init__(self, fn):
        self.fn = fn
        self.n = 0
    def __call__(self, *a, **kw):
        self.n += 1
        print(f"Called {self.fn.__name__} {self.n} time(s)")
        return self.fn(*a, **kw)
```

Usage

```python
@CountCalls
def add(a, b):
    return a + b

add(1, 2)   # Called 1 time(s)
add(3, 4)   # Called 2 time(s)
print(add.n)  # 2
```

4. MULTIPLE DECORATORS

- Applied bottom-up
- Execution order matters

Bottom-up application

```python
@timer
@retry(times=5)
def call_api():
    ...
```

↓

```
call_api = timer(retry(times=5)(call_api))
```

↓

First @retry wraps the function, then @timer wraps the result.

Execution flow

1. timer.wrapper() starts
2. retry.wrapper() tries up to 5 times
3. call_api() executes
4. Returns → back through retry
5. Returns → back through timer
6. timer logs time and returns

Usage

```python
@timer
@retry(times=5)
def call_api():
    ...

# Clean, composable,
readable!
```

Forgetting @wraps is the #1 decorator bug.
You'll lose metadata, break help(), docs, and some frameworks.

Decorators add power without changing code.
Master them, and your Python looks different.

Save this slide for your next interview 👇

das-purnendu

GENERATORS & ITERATORS

1. GENERATOR FUNCTION

- Uses `yield`
- Returns an iterator
- Constant memory
- Perfect for large data

```
def read_large_file(path):
    with open(path) as f:
        for line in f:
            yield line.strip()

# Constant memory, even on 10 GB files
```

Why use it?

- Handles huge data
- Lazy evaluation
- Efficient pipelines

2. GENERATOR EXPRESSION

- Like list comp, but lazy
- Memory efficient

```
squares = (x*x for x in range(10**9))

# Generates on-demand, one value at a time
```

Example

```
sum(x*x for x in range(10**9))
# No huge list in memory
```

3. yield from (DELEGATION)

- Delegate to a sub-generator
- Cleaner recursion
- Faster (CPython opt.)

```
def flatten(nested):
    for x in nested:
        if isinstance(x, list):
            yield from flatten(x)
        else:
            yield x
```

Example

```
list(flatten([1,[2,[3,4],5],6]))
# [1, 2, 3, 4, 5, 6]
```

4. CUSTOM ITERATOR PROTOCOL

- Implement `__iter__` and `__next__`
- Raise StopIteration when done

```
class Counter:
    def __init__(self, n):
        self.n, self.i = n, 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i >= self.n:
            raise StopIteration
        self.i += 1
        return self.i
```

Usage

```
for i in Counter(3):
    print(i)

# Output:
# 1 2 3
```

5. SEND VALUES INTO GENERATOR

- Bi-directional communication with generators
- Use `send()`

```
def echo():
    while True:
        x = yield
        print(x)
```

```
g = echo()
next(g)          # Prime the generator
g.send("hi")     # hi
```

Flow

1. `next(g)` starts generator until first `yield`
2. `send(value)` resumes and sends a value
3. Generator processes and yields again

Generators are lazy. Iterators control the flow.
Use them to write scalable, memory-efficient Python.

Save this slide for your next interview 👇

das-purnendu

OOP FUNDAMENTALS

```python
class User:
    species = "Homo sapiens"     # class variable

    def __init__(self, name, age):
        self.name = name         # instance variable
        self._age = age          # protected (convention)
        self.__id = id(self)     # name-mangled

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, v):
        if v < 0:
            raise ValueError("age cannot be negative")
        self._age = v

    @classmethod
    def from_dict(cls, d):
        return cls(d["name"], d["age"])

    @staticmethod
    def is_adult(age):
        return age >= 18

# Usage
u = User("Alice", 25)
u.age = 30
User.is_adult(u.age)         # True
u2 = User.from_dict({"name": "Bob", "age": 20})
```

Key Concepts

- Class vs Instance variables
- Encapsulation
- Properties for controlled access
- Name mangling (double underscore)
- Factory methods (@classmethod)
- Utility methods (@staticmethod)

Encapsulation

Use single underscore (_age) to indicate "protected" by convention.

Use double underscore (__id) for name mangling (not truly private).

Pro Tip

Prefer properties over public attributes when you need validation or computed values.

METHOD TYPES COMPARISON

Type	Definition	First Parameter	Access	Use Case	Example
Instance Method	Works with instance data	`self`	`obj.method()`	Operate on instance attributes	`u.age = 30`
@classmethod	Works with class data	`cls`	`Class.method()` `obj.method()`	Alternate constructors, class-level logic	`User.from_dict(d)`
@staticmethod	No access to instance or class	none	`Class.method()`	Utility/helper functions	`User.is_adult(20)`

Difference between @classmethod / @staticmethod / instance method = asked in 70% of mid-level rounds.

Save this slide for your next interview 👇

Good tests save time, bugs, and careers.
Test. Refactor. Repeat.

das-purnendu

INHERITANCE, MRO, super()

```python
class A:
    def greet(self):
        return "A"

class B(A):
    def greet(self):
        return "B -> " + super().greet()

class C(A):
    def greet(self):
        return "C -> " + super().greet()

class D(B, C):    # multiple inheritance
    def greet(self):
        return "D -> " + super().greet()

D().greet()             # 'D -> B -> C -> A'
```

What's happening?

- super() follows the Method Resolution Order (MRO).
- Each class's greet() calls the next class in the MRO.
- This creates a cooperative chain without hardcoding parent names.
- In D(B, C), Python searches left to right, then up the tree using C3 linearization.

Check MRO

```
>>> D.__mro__
(<class '__main__.D'>,
 <class '__main__.B'>,
 <class '__main__.C'>,
 <class '__main__.A'>,
 <class 'object'>)
>>> D.mro()   # same result
```

```python
# Always call super().__init__ in subclasses
class Admin(User):
    def __init__(self, name, age, role):
        super().__init__(name, age)
        self.role = role
```

Key Takeaway

Use super() in all subclasses for:

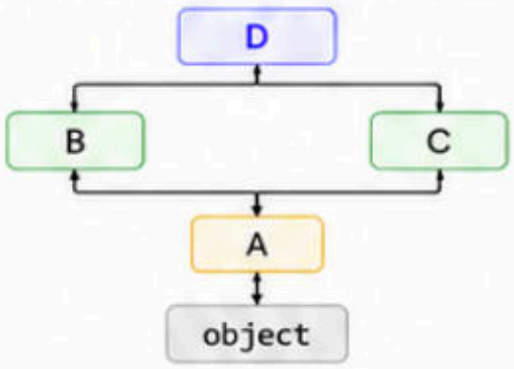
- Proper initialization
- Works with multiple inheritance
- Future-proof (less breakage)

```python
from abc import ABC, abstractmethod
class Storage(ABC):
    @abstractmethod
    def save(self, data):
        ...
```

Abstract Base Class (ABC)

- Defines an interface/contract.
- Subclasses must implement abstract methods.
- Cannot instantiate Storage directly.
- Great for plugins, frameworks, extensibility.

METHOD RESOLUTION ORDER (MRO)



- Python uses C3 Linearization to compute MRO.
- Properties:
  - Monotonicity
  - Local precedence order
  - Extended precedence graph
- Ensures a consistent and predictable method lookup order.

Example MRO

```python
class D(B, C): pass
D.__mro__
# -> (D, B, C, A, object)
class E(C, B): pass
E.__mro__
# -> (E, C, B, A, object)
```

MRO depends on the order of base classes.

Python uses C3 linearization for MRO.
Mention it if probed in interviews. It shows depth.

Save this slide for your next interview 👇

Inheritance + super() + MRO = Senior-level Python thinking.

das-purnendu

DUNDER METHODS & DATACLASSES

1 MANUAL DUNDER METHODS

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

    def __repr__(self):         return f"Point({self.x},{self.y})"
    def __str__(self):          return f"({self.x},{self.y})"
    def __eq__(self, o):        return (self.x,self.y) == (o.x,o.y)
    def __hash__(self):         return hash((self.x, self.y))
    def __lt__(self, o):        return (self.x,self.y) < (o.x,o.y)
    def __len__(self):          return 2
    def __getitem__(self, i):   return (self.x, self.y)[i]
    def __iter__(self):         yield from (self.x, self.y)
    def __add__(self, o):       return Point(self.x+o.x, self.y+o.y)
    def __call__(self):         return (self.x, self.y)
```

USAGE EXAMPLE

```python
p1 = Point(1, 2)
p2 = Point(3, 4)
print(repr(p1))                    # Point(1,2)
print(str(p1))                     # (1,2)
print(p1 == Point(1,2))            # True
print(hash(p1) == hash(Point(1,2))) # True
print(p1 < p2)                     # True
print(len(p1))                     # 2
print(p1[0], p1[1])                # 1 2
print(list(p1))                    # [1, 2]
print(p1 + p2)                     # Point(4,6)
print(p1())                        # (1, 2)
```

COMMON DUNDER METHODS

Method	Purpose	Example
`__repr__(self)`	Official string representation	`Point(1,2)`
`__str__(self)`	User-friendly string	`(1,2)`
`__eq__(self, o)`	Equality (==)	`p1 == p2`
`__hash__(self)`	Hash value (for sets, dict keys)	`hash(p)`
`__lt__(self, o)`	Less than (<) (enables sorting)	`p1 < p2`
`__len__(self)`	Length of object	`len(p)`
`__getitem__(self, i)`	Indexing	`p[0]`
`__iter__(self)`	Iteration	`for x in p`
`__add__(self, o)`	Addition (+)	`p1 + p2`
`__call__(self)`	Callable object	`p()`

QUICK NOTES

- Implement only what you need.
- `__eq__` and `__hash__` should be consistent.
- `__iter__` + `__getitem__` gives sequence behavior.
- `__call__` turns instances into callable objects.

2 DATACLASSES – LET PYTHON DO THE BOILERPLATE

```python
from dataclasses import dataclass, field

@dataclass(frozen=True, order=True, slots=True)   # slots: 3.10+
class Point:
    x: int
    y: int
    tags: list = field(default_factory=list)
```

WHAT YOU GET AUTOMATICALLY

- `__init__`, `__repr__`, `__eq__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, `__hash__*`, `__match_args__`, and more.
- Clean, readable, and less error-prone code.

DATACLASS USAGE

```python
p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1)                      # Point(x=1, y=2, tags=[])
print(p1 == p2)                # True
print(p1 < Point(2, 0))        # True
print(hash(p1))                # Works because frozen=True
p3 = dataclasses.replace(p1, x=10)
print(p3)                      # Point(x=10, y=2, tags=[])
```

PARAMETERS EXPLAINED

- `frozen=True` → Instances are immutable and hashable.
- `order=True` → Comparison methods (<, >, <=, >=) are added.
- `slots=True` → Uses `__slots__` under the hood. Faster attribute access and less memory.
- `field(default_factory=list)` → Ensures each instance gets its own list.

`frozen=True` → **hashable** (can be used in sets, dict keys).
`slots=True` → **less memory usage and faster attribute access** (Python 3.10+).

Save this slide for your next interview 👇

Dunder methods for control.
Dataclasses for productivity.

das-purnendu

COMPOSITION, MIXINS, PROTOCOLS

1 COMPOSITION

Prefer composition over inheritance

- Loose coupling
- Easier to change
- More flexible
- Follows the has-a principle

```python
# Composition > inheritance (in most cases)
class Engine:
    def start(self):
        print("vroom")

class Car:
    def __init__(self):
        self.engine = Engine()     # has-a, not is-a
    def drive(self):
        self.engine.start()
```

```python
car = Car()
car.drive()    # vroom
```

Why Composition?

- Reduces tight coupling
- Objects can be composed at runtime
- Avoids deep inheritance hierarchies
- Easier testing & maintenance

2 MIXINS

Add small, reusable behavior to classes

- Promotes code reuse
- Keeps classes focused
- Combine behaviors easily

```python
# Mixin — small reusable behavior
class JSONMixin:
    def to_json(self):
        import json
        return json.dumps(self.__dict__)

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

```python
u = User("Alice")
print(u.to_json())    # {"name": "Alice"}
```

Mixin Tips

- Don't use mixins when state/behavior conflicts are likely
- Keep mixins small and single-purpose
- Name mixins with "Mixin" suffix

3 PROTOCOLS

Duck typing with type hints (3.8+)

- Define expected structure (not implementation)
- Increase flexibility
- Great for libraries & APIs

```python
# Protocol — duck typing with type hints (3.8+)
from typing import Protocol

class Drawable(Protocol):
    def draw(self) -> None: ...

def render(obj: Drawable):    # any class with .draw() works
    obj.draw()
```

```python
class Circle:
    def draw(self):
        print("Drawing a circle")

render(Circle())    # Drawing a circle
```

Protocol Benefits

- Works with any class that matches the structure
- No inheritance required
- Static type checkers love it (mypy, pyright)

INHERITANCE vs COMPOSITION

Aspect	Inheritance (is-a)	Composition (has-a)
Relationship	`Is-a (Dog is an Animal)`	`Has-a (Car has an Engine)`
Coupling	Tight	Loose
Flexibility	Less flexible	More flexible
Extensibility	Harder to change	Easier to extend/change
Code Reuse	Via subclassing	Via components/mixins
Best For	True is-a + shared interface	Most real-world cases

Senior Takeaway

Prefer composition.
Use inheritance ONLY when:

- There is a clear is-a relationship AND
- You need to share a common interface/behavior
- Polymorphism is required

Everything else → composition.

Composition for flexibility, Mixins for reuse, Protocols for contracts.
Write designs that scale, adapt, and stay clean.

Save this slide for your next interview 👇

Think: "Can this be composed instead of inherited?"

das-purnendu

FILE OPERATIONS

1 READ FILE (ALWAYS USE WITH)

- Auto-closes file
- Exception-safe
- Streaming is memory-efficient

```python
# Always use `with` – auto-close, exception-safe
with open("f.txt", "r", encoding="utf-8") as f:
    data = f.read()             # whole file
    # or
    for line in f:              # streaming, memory-safe
        process(line)
```

```python
# Modes: r, w, a, r+, rb, wb, x (exclusive create)
```

FILE MODES

- `r` Read (default)
- `w` Write (truncate)
- `a` Append (create if not exists)
- `r+` Read & write
- `rb` / `wb` Binary read/write
- `x` Exclusive create (fails if exists)

2 WRITE FILE

- `write()` for string
- `writelines()` for multiple lines

```python
with open("f.txt", "w", encoding="utf-8") as f:
    f.write("hello\n")
    f.writelines(["a\n", "b\n"])
```

TIP

Use newline `\n` for line breaks.

Always specify encoding for text files.

3 JSON

- `json.load()` → read
- `json.dump()` → write

```python
import json
with open("d.json", "r", encoding="utf-8") as f:
    data = json.load(f)

with open("d.json", "w", encoding="utf-8") as f:
    json.dump(data, f, indent=2)
```

JSON TIPS {}

- Use indent for pretty output.
- `json.load()` returns Python objects.
- `json.dump()` serializes them.

4 CSV

- Use DictReader for named columns
- Great for tabular data

```python
import csv
with open("d.csv", newline="", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    rows = list(reader)
```

CSV TIPS

- Always use `newline=""` when opening CSV files.
- `DictReader` → list of dicts (one per row).

5 PICKLE

- Serialize Python objects
- **NEVER** load untrusted data!

```python
import pickle
# Write (serialize)
pickle.dump(obj, open("o.pkl", "wb"))

# Read (deserialize) – NEVER from untrusted source
obj = pickle.load(open("o.pkl", "rb"))
```

SECURITY WARNING

Never unpickle data from untrusted sources.

It can execute arbitrary code.

6 PATHLIB (MODERN, PREFER THIS)

- Object-oriented file paths
- Cleaner & safer than `os.path`

```python
from pathlib import Path
p = Path("data") / "f.txt"

p.read_text(encoding="utf-8")
p.write_text("hi", encoding="utf-8")

p.exists()     # True / False
p.is_file()    # True if file
p.suffix       # '.txt'
p.stem         # 'f'
```

WHY PATHLIB?

- Cross-platform paths
- More readable
- Rich API
- Pretty objects

```python
Path("a") / "b" / "c.txt"
```

Use `with` for safety, the right mode for the job, and the right tool for the data.
Prefer **pathlib** for paths, **csv/json** for data, and **pickle** only when you must.

Save this slide for your next interview 👇

Files are everywhere. Handle them like a pro.

das-purnendu

PYTHON: os, sys, subprocess, pathlib

1 os

```python
import os
os.getcwd(); os.chdir("/tmp")
os.listdir("."); os.environ.get("HOME")
os.path.join("a", "b", "c.txt")
os.makedirs("a/b/c", exist_ok=True)
os.remove("f.txt"); os.rename("a", "b")
os.walk(".")        # recursive directory traversal
```

- Interact with the operating system
- Manage files, dirs, environment
- `os.path` provides path utilities (legacy style)

2 pathlib (modern replacement)

```python
from pathlib import Path
Path.home(); Path.cwd()
list(Path(".").rglob("*.py"))    # recursive glob
Path("dir").mkdir(parents=True, exist_ok=True)
```

- Object-oriented filesystem paths
- Cross-platform
- More readable & easier to use

3 sys

```python
import sys
sys.argv                 # CLI arguments
sys.path                 # Module search paths
sys.exit()               # Exit program with status
sys.stdin.read()         # Read from stdin
sys.stdout.write("x")    # Write to stdout
```

- Access interpreter and runtime info
- Interact with CLI, streams, paths
- Control program execution

4 subprocess — run shell commands safely

```python
import subprocess
result = subprocess.run(
    ["ls", "-la"],
    capture_output=True,
    text=True,
    check=True
)
print(result.stdout)
```

- Run external commands
- `subprocess.run` is safer & more powerful than `os.system`
- Capture output, handle errors, set timeouts, env, etc.

os.path vs pathlib

os.path (Legacy)	pathlib (Modern & Recommended)
`os.path.join("a", "b", "c.txt")`	`Path("a") / "b" / "c.txt"`
`os.path.exists("f.txt")`	`Path("f.txt").exists()`
`os.path.basename("/a/b/c.txt")`	`Path("/a/b/c.txt").name  # "c.txt"`
`os.path.dirname("/a/b/c.txt")`	`Path("/a/b/c.txt").parent  # /a/b`
`os.path.abspath(".")`	`Path(".").resolve()`

QUICK REFERENCE

os — Commonly Used

`os.getcwd()`	Current working directory
`os.chdir(path)`	Change current directory
`os.listdir(path)`	List files in directory
`os.environ[key]`	Environment variable
`os.makedirs(path)`	Create directories
`os.remove(path)`	Delete a file
`os.rename(src, dst)`	Rename/Move a file or dir
`os.walk(path)`	Walk directory tree

pathlib — Commonly Used

`Path.home()`	User's home directory
`Path.cwd()`	Current working directory
`Path("x").exists()`	Check existence
`Path("x").is_file()`	Check file
`Path("x").is_dir()`	Check directory
`Path("x").rglob("*.py")`	Recursive glob
`Path("dir").mkdir(parents=True, exist_ok=True)`	Create nested dirs

sys — Commonly Used

`sys.argv`	List of CLI arguments
`sys.path`	List of import paths
`sys.exit(code)`	Exit program
`sys.stdin`	Standard input
`sys.stdout`	Standard output
`sys.stderr`	Standard error

subprocess.run — Key Params

`args`	Command & arguments (list)
`capture_output=True`	Capture stdout & stderr
`text=True`	Return strings (not bytes)
`check=True`	Raise CalledProcessError on non-zero exit
`cwd=...`	Set working directory
`env=...`	Set environment variables

Best Practices

- Use pathlib over os.path for cleaner, cross-platform code.
- Use subprocess.run over os.system for safety & control.
- Never trust external input when using subprocess.

Save this slide for your next interview 👇

Know the standard library. Write better, safer Python.

das-purnendu

DATETIME, TIME, TIMEZONES

1 IMPORTS

```
from datetime import datetime, date, time, timedelta, timezone
import time as t
```

2 NOW

```
datetime.now()                  # naive (no tz)
datetime.now(timezone.utc)      # aware (always prefer)
```

3 CONSTRUCT

```
dt = datetime(2026, 5, 7, 14, 30)
```

4 FORMAT / PARSE

```
dt.strftime("%Y-%m-%d %H:%M:%S")          # -> '2026-05-07 14:30:00'
datetime.strptime("2026-05-07", "%Y-%m-%d")
                                          # -> datetime(2026, 5, 7, 0, 0)
```

5 ARITHMETIC

```
tomorrow = datetime.now(timezone.utc) + timedelta(days=1)  # + 1 day
diff = dt2 - dt1                          # -> timedelta
diff.total_seconds()                      # total seconds (float)
```

6 TIMEZONES — USE zoneinfo (3.9+)

```
from zoneinfo import ZoneInfo
ist = datetime.now(ZoneInfo("Asia/Kolkata"))    # aware
utc = ist.astimezone(timezone.utc)              # convert
ist.tzinfo                    # -> zoneinfo.ZoneInfo('Asia/Kolkata')
```

7 UNIX TIMESTAMPS

```
t.time()                      # epoch seconds (float)
datetime.fromtimestamp(1700000000, tz=timezone.utc)        # aware
int(datetime.now(timezone.utc).timestamp())                # to epoch int
```

8 PERFORMANCE TIMING

```
start = t.perf_counter()          # high-resolution timer
# ... do work ...
elapsed = t.perf_counter() - start  # seconds (float)
```

COMMON CLASSES

Class	Purpose
date	Calendar date (YYYY-MM-DD)
time	Time of day (HH:MM:SS[.ffffff])
datetime	Date + time
timedelta	Duration / difference
timezone	Fixed offset timezone (e.g., UTC)
ZoneInfo	IANA time zones (e.g., Asia/Kolkata)

BEST PRACTICES

- Always store in UTC in databases, APIs, logs.
- Convert to local timezone only for display to users.
- Use aware datetimes (with tzinfo).
- Prefer zoneinfo over pytz (Python 3.9+).
- Be careful with DST transitions (ambiguous / non-existent times).

TIMEZONE EXAMPLE

```
from zoneinfo import ZoneInfo
ist = datetime(2026, 5, 7, 9, 0,
   tzinfo=ZoneInfo('Asia/Kolkata'))
utc = ist.astimezone(timezone.utc)
ny = ist.astimezone(ZoneInfo('America/New_York'))
print(ist)              # 2026-05-07 09:00:00+05:30
print(utc)              # 2026-05-07 03:30:00+00:00
print(ny)               # 2026-05-06 23:30:00-04:00
```

QUICK REFERENCE

strftime / strptime cheatsheet

%Y	4-digit year	%m	Month (01-12)	%d	Day (01-31)
%H	Hour (00-23)	%M	Minute (00-59)	%S	Second (00-59)
%f	Microsecond (000000-999999)				
%z	UTC offset (+HHMM)	%Z	Timezone name		

timedelta cheatsheet

```
timedelta(days=1, hours=2, minutes=3, seconds=4)
.days, .seconds, .microseconds
.total_seconds()
```

Bare except: catches KeyboardInterrupt and SystemExit.
NEVER do that.

Use specific exceptions.
Don't hide bugs.

Save this slide for your next interview 👇

Always store UTC. Convert to local only on display.

das-purnendu

EXCEPTIONS, LOGGING, ASSERTIONS

1 EXCEPTION HANDLING

```python
try:
    risky()
except (ValueError, KeyError) as e:
    log.warning(e)
except Exception as e:
    log.exception(e)      # logs traceback
    raise                 # re-raise, preserve trace
else:
    commit()              # only if no exception
finally:
    close()               # always
```

KEY POINTS

- Use specific exceptions whenever possible.
- The else block runs only if no exception occurred.
- The finally block runs no matter what.
- log.exception() logs the full traceback.

2 CUSTOM EXCEPTION

```python
class PaymentError(Exception):
    def __init__(self, msg, code):
        super().__init__(msg)
        self.code = code
```

WHY CUSTOM EXCEPTIONS?

- Express domain-specific failures.
- Carry extra context (like error codes).
- Make error handling clearer at higher levels.

3 CHAIN EXCEPTIONS — PRESERVE CAUSE

```python
try:
    parse(x)
except ValueError as e:
    raise PaymentError("bad input", 400) from e
```

WHY CHAIN?

- Preserves the original traceback as the cause.
- Easier debugging and root-cause analysis.
- Shown as "The above exception was the direct cause of the following exception".

4 LOGGING — NEVER USE print() IN PRODUCTION

```python
import logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)s %(name)s: %(message)s"
)
log = logging.getLogger(__name__)

log.info("started")
log.error("failed", exc_info=True)    # logs traceback
```

BEST PRACTICES

- Configure logging once at app startup.
- Use appropriate levels.
- Use meaningful log messages.
- Include context (ids, params, etc).

LOGGING LEVELS (HIGH → LOW)

CRITICAL	System is unusable
ERROR	Something failed
WARNING	Something unexpected
INFO	General information
DEBUG	Detailed diagnostics

WHY NOT print()?

- Can't control verbosity.
- Not structured.
- Harder to filter/search.
- Doesn't include context or levels.

5 ASSERTIONS — DEBUG ONLY

```python
assert x > 0, "x must be positive"
# For internal sanity checks.
# Stripped out when running Python with -O (optimized).
```

WHEN TO USE ASSERT

- Verify assumptions in your code.
- Catch programmer errors early.
- Do NOT use for user input validation or runtime checks.

BARE EXCEPT: CATCHES KEYBOARDINTERRUPT AND SYSTEMEXIT

NEVER DO THAT.

```python
try:
    ...
except:    # BAD
    ...
```

This will catch KeyboardInterrupt, SystemExit and hide real problems. Always catch specific exceptions.

Save this slide for your next interview 👇

Handle exceptions. Log meaningfully. Assert wisely.

das-purnendu

MUTABILITY, COPY, IDENTITY

1 MUTABLE vs IMMUTABLE

- Immutable: int, float, str, tuple, frozenset, bool, bytes
- Mutable: list, dict, set, bytearray, most custom objects

IMMUTABLE (cannot change after creation)

`int` `float` `str` `tuple` `frozenset` `bool` `bytes`

MUTABLE (can be changed)

`list` `dict` `set` `bytearray` `most custom objects`

KEY TAKEAWAYS

- Immutable objects are safer as keys in dicts/sets.
- Mutable objects can change in place.
- Be careful with shared mutable objects.
- Understand identity vs equality to avoid subtle bugs.

2 is vs ==

```
a == b     # value equality (are values the same?)
a is b     # identity (same object in memory?)
```

EXAMPLE

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

c = a
a is c     # True
```

3 DEFAULT ARGUMENT TRAP (CLASSIC)

❌ Bad: shared across calls

```
def add(x, items=[]):
    items.append(x)
    return items

add(1)    # [1]
add(2)    # [1, 2]   <- surprise!
```

✅ Good: create new list when needed

```
def add(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items

add(1)     # [1]
add(2)     # [2]    <- as expected
```

DEFAULT ARGUMENT RULE

Never use a mutable object as a default argument.

Use *None* and create inside the function.

4 SHALLOW vs DEEP COPY

```
import copy
b = a.copy()            # shallow copy (top-level only)
c = copy.deepcopy(a)    # deep copy (recursively copies all)
```

NOTE

- Shallow copy copies the outer container, not nested objects.
- Deep copy copies everything recursively.
- Use deepcopy only when you need it (costly).

EXAMPLE

```
a = [1, [2, 3]]
b = a.copy()
b[1].append(4)
a          # [1, [2, 3, 4]]  (changed!)

c = copy.deepcopy(a)
c[1].append(5)
a          # [1, [2, 3, 4]]  (unchanged)
c          # [1, [2, 3, 4, 5]]
```

5 TUPLE OF LISTS IS MUTABLE INSIDE

```
t = ([1, 2], [3, 4])
t[0].append(9)     # works!
t                  # ([1, 2, 9], [3, 4])
```

Remember

Tuple is immutable, but the objects inside it can be mutable.

6 INTEGER CACHING (CPython IMPLEMENTATION DETAIL)

```
a = 256; b = 256; a is b     # True
a = 257; b = 257; a is b     # False
```

Why?

CPython caches small integers (typically -5 to 256).
Do not rely on this behavior.

HOW TO CHECK?

```
id(x)      # memory address (identity)
type(x)    # type of object
dir(x)     # attributes & methods
help(x)    # help on object
```

DO NOT rely on is for value comparison. Use == for comparing values.

NEVER use bare mutable defaults. **COPY** only when needed. **DEEPCOPY** with care.

Understand what is mutable, what is shared, and what is copied.

Save this slide for your next interview 👇

Know your objects. Avoid surprises. Write correct, predictable Python.

das-purnendu

GIL, THREADS, PROCESSES

1 THE GIL (GLOBAL INTERPRETER LOCK)

- Only ONE thread executes Python bytecode at a time.
- I/O-bound tasks → threading or asyncio (real speedup)
- CPU-bound tasks → multiprocessing (real parallelism)

Rule of thumb

I/O-bound → threads / asyncio
CPU-bound → processes / (native extensions / C / Numpy)

WHY THREADING DOESN'T SPEED UP CPU LOOPS?

- The GIL allows only one thread to run Python bytecode at a time.
- For CPU-bound work, threads just context-switch — no parallel speedup.
- Use processes for true parallelism across CPU cores.

2 THREADS — I/O BOUND (CONCURRENT)

```python
from concurrent.futures import ThreadPoolExecutor

def fetch(url):
    ...  # I/O work (network, disk, DB, etc.)

urls = [...]

with ThreadPoolExecutor(max_workers=8) as ex:
    results = list(ex.map(fetch, urls))
```

Good for

- Network calls
- File I/O
- APIs
- Waiting tasks

THREADS vs PROCESSES

	Threads	Processes
Memory	Shared	Separate
GIL	Yes (one at a time)	No (per process)
Best for	I/O-bound	CPU-bound
Overhead	Low	Higher
Data sharing	Easy	Use Queues/Pipes (for files)

3 PROCESSES — CPU BOUND (PARALLEL)

```python
from concurrent.futures import ProcessPoolExecutor

def crunch(chunk):
    ...  # CPU-heavy work

chunks = [...]

with ProcessPoolExecutor() as ex:
    results = list(ex.map(crunch, chunks))
```

Good for

- Heavy computation
- Data processing
- ML training (pure Python)
- Anything CPU-intensive

BEST PRACTICES

- Use threads for I/O-bound, processes for CPU-bound.
- Keep critical sections small.
- Avoid sharing mutable data; prefer message passing.
- Always use context managers for executors: `with ThreadPoolExecutor(...) as ex:`
- Handle exceptions inside workers.

4 LOCKS — PROTECT SHARED STATE (THREADS)

```python
from threading import Lock
lock = Lock()
shared = []

def add(x):
    with lock:
        shared.append(x)   # critical section
```

Use locks when

- Multiple threads read/modify shared data
- You need atomicity and consistency

COMMON PITFALLS

- Using threads for CPU-heavy loops → no speedup.
- Forgetting locks → race conditions.
- Deadlocks → acquire locks in a consistent order.
- Sending large objects between processes → slow (serialize/deserialize cost).
- Not guarding multiprocessing code with `if __name__ == "__main__":` (on Windows).

5 QUEUE — THREAD-SAFE COMMUNICATION

```python
from queue import Queue
q = Queue()
q.put(1)          # producer
item = q.get()    # consumer (blocks if empty)
```

Good for

- Passing messages
- Producer / Consumer
- Work queues

QUICK REFERENCE

`ThreadPoolExecutor(max_workers=n)`	Run I/O tasks concurrently
`ProcessPoolExecutor(max_workers=n)`	Run CPU tasks in parallel
`threading.Lock()`	Mutual exclusion
`queue.Queue()`	Thread-safe FIFO queue
`multiprocessing.Queue()`	Process-safe FIFO queue

6 MULTIPROCESSING — IPC VIA QUEUE / PIPE (SEPARATE MEMORY)

```python
from multiprocessing import Process, Queue
def worker(q_in, q_out):
    while True:
        item = q_in.get()
        if item is None: break
        q_out.put(process(item))
q_in, q_out = Queue(), Queue()
p = Process(target=worker, args=(q_in, q_out))
p.start()
q_in.put(data)
result = q_out.get()
```

Important

- Each process has its own memory.
- Share data using Queue, Pipe, or files.

THREADING DOESN'T SPEED UP CPU-BOUND PYTHON CODE.
That's the GIL.

I/O → threads/asyncio
CPU → processes

Save this slide for your next interview 👇

Understand the model. Pick the right tool.
Write concurrent code that's correct and fast.

das-purnendu

ASYNCIO ESSENTIALS

1 BASIC HTTP FETCH (aiohttp)

```python
import asyncio
import aiohttp

async def fetch(session, url):
    async with session.get(url) as r:
        return await r.text()

async def main(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, u) for u in urls]
        return await asyncio.gather(*tasks)

results = asyncio.run(main(urls))
```

Why this is fast

All requests are in-flight concurrently without blocking the event loop.

KEY TAKEAWAYS

- Asyncio is single-threaded concurrency using an event loop.
- Great for I/O-bound tasks (network, disk, APIs).
- Use `await` to yield control (cooperative multitasking).
- CPU-bound work? Use `ProcessPoolExecutor` or `asyncio.to_thread()`.
- `async def` without `await` inside = useless. Common mistake!

2 KEY PRIMITIVES

```python
await asyncio.sleep(1)              # non-blocking sleep
asyncio.gather(*tasks)              # run concurrently, collect results
asyncio.create_task(coro())         # schedule + run in background
asyncio.wait_for(coro, 5)           # timeout after 5 seconds
asyncio.Semaphore(10)               # limit concurrency
```

NOTE

Always prefer `await asyncio.sleep()` over `time.sleep()` in async code, or you will block the entire loop.

3 CREATE TASKS & RUN IN BACKGROUND

```python
async def worker(name):
    for i in range(3):
        print(f"{name}: {i}")
        await asyncio.sleep(0.5)

async def main():
    t1 = asyncio.create_task(worker("A"))
    t2 = asyncio.create_task(worker("B"))
    # do other work while tasks run
    await asyncio.gather(t1, t2)
asyncio.run(main())
```

Good for

- Fire-and-forget jobs
- Long running background tasks
- Maintain responsiveness

EXAMPLE OUTPUT (interleaved)

```
A: 0
B: 0
A: 1
B: 1
A: 2
B: 2
```

4 TIMEOUTS

```python
async def fetch_with_timeout(session, url):
    try:
        return await asyncio.wait_for(fetch(session, url), 5)
    except asyncio.TimeoutError:
        return f"Timeout: {url}"
```

DON'T BLOCK THE EVENT LOOP

```python
import time
async def bad():
    time.sleep(2)          # ✗ blocks everything!
    return "done"
# Use await asyncio.sleep(2) instead.
```

5 ASYNC GENERATOR

```python
async def stream():
    for i in range(10):
        await asyncio.sleep(0.1)
        yield i

async def main():
    async for x in stream():
        print(x)

asyncio.run(main())
```

Use cases

- Streaming data
- Reading large files in chunks
- Real-time feeds

6 OFFLOAD BLOCKING (SYNC) CODE

```python
def blocking_fn(x):
    import time
    time.sleep(2)
    return x * 2

async def main():
    result = await asyncio.to_thread(blocking_fn, 21)
    print(result)  # 42

asyncio.run(main())
```

Why to_thread?

Runs blocking code in a thread without blocking the event loop.

QUICK REFERENCE

Primitive	Purpose	Primitive	Purpose
await coro	Wait for coroutine to complete	asyncio.Event()	Wait for an event to be set
asyncio.gather(*aws)	Run coroutines concurrently and gather results	asyncio.Queue()	FIFO queue for producer/consumer
asyncio.create_task(coro())	Schedule coroutine to run in background	asyncio.Lock()	Mutex lock (not thread-safe across loops)
asyncio.wait_for(coro, timeout)	Run with timeout	asyncio.to_thread(fn, *args)	Run blocking code in a thread
asyncio.shield(coro)	Protect coroutine from cancellation	asyncio.CancelledError	Raised when a task is cancelled
asyncio.Semaphore(n)	Limit number of concurrent tasks		

COMMON MISTAKES

- `async def` without `await`
- Using `time.sleep()` in async code
- Creating tasks and forgetting them (no references)
- Not handling exceptions in tasks
- Blocking the event loop with CPU-heavy work
- Too many concurrent tasks (no limits)

Save this slide for your next interview 👇

Use the right primitive. Keep the loop free. Write non-blocking, scalable Python.

das-purnendu

PERFORMANCE & MEMORY

BIG IDEA

Measure first.
Optimize second.
Don't guess.

1 PROFILE — FIND THE SLOW PART FIRST

```
import cProfile
cProfile.run("main()")

# or run from CLI
# python -m cProfile -s tottime script.py
```

Useful options

- `-s tottime` # sort by time
- `-s cumtime` # sort by cum. time
- `-o out.prof` # save stats
- pstats module for analysis

2 TIME SMALL SNIPPETS

```
from timeit import timeit
t = timeit("'-'.join(str(n) for n in range(100))",
           number=10000)
print(f"{t:.6f} sec")
```

TIPS

- Always run many iterations (number=...)
- timeit disables GC by default (good)

3 MEASURE MEMORY

```
import sys
sys.getsizeof(obj)          # shallow size only

import tracemalloc
tracemalloc.start()

...    # run code

current, peak = tracemalloc.get_traced_memory()
print(f"current={current;,}  peak={peak;,} bytes")

# Optional: take snapshots for diff
snap1 = tracemalloc.take_snapshot()
...    # run more code
snap2 = tracemalloc.take_snapshot()
top_stats = snap2.compare_to snap1, 'lineno')
```

MEMORY NOTES

- sys.getsizeof() doesn't include referenced objects.
- Use tracemalloc to see where memory is allocated.
- For real-world apps, profile under realistic load.

4 MORE QUICK WINS (EXAMPLES)

Instead of	Do this
`for i in range(len(lst)):` `  x = lst[i]`	`for x in lst:`

`total = 0`
`for x in lst:`
`  total += x`
	`total = sum(lst)`

`if x not in some_list:`
`  ...`
	`some_set = set(some_list)`
	`if x not in some_set:`
	`  ...`

`s = ""`
`for part in parts:`
`  s += part`
	`s = "".join(parts)`

`big_list = [f(x) for x in huge_iter]`
`# then use one by one`
	`for x in (f(x) for x in huge_iter):`
	`  process(x)`
	`(generator)`

5 BEWARE OF COMMON SLOW PATTERNS

- ✗ Repeated attribute lookups in tight loops (bind to local)
- ✗ Creating objects in hot loops unnecessarily
- ✗ Using list for membership tests
- ✗ String concatenation with += in loops
- ✗ Premature optimization (measure first!)

QUICK WINS — HIGH IMPACT

1. Local variable lookup is faster than global. Bind globals to locals inside hot loops.
2. Use built-ins (`sum`, `min`, `any`, `all`, `max`, `sorted`) — they're in C and highly optimized.
3. Use `sets` for membership tests. >> O(1) average vs O(n) for lists.
4. `"".join(parts)` over `+=` in loops. Avoid quadratic string building.
5. Use generators when you don't need the full list in memory.
6. `__slots__` to reduce per-instance memory and speed up attribute access.

__SLOTS__ EXAMPLE

```
class Point:
    __slots__ = ("x", "y")     # no __dict__
    def __init__(self, x, y):
        self.x, self.y = x, y

# Benefits:
# - No per-instance __dict__
# - Less memory
# - Faster attribute access
```

Compare:

Class	Instance Size (approx)	Has __dict__?
Normal class	~56 bytes	Yes
__slots__ class	~32 bytes	No

MEMBERSHIP TESTS

Container	Test	Time Complexity	Use When
list	`x in lst`	O(n)	small lists / order matters
set	`x in s`	O(1) avg	fast membership
dict	`k in d`	O(1) avg	fast membership on keys

ⓘ Use set for membership checks. Keep list for order.

GENERATORS > LISTS (WHEN POSSIBLE)

```
def squares(n):
    for i in range(n):
        yield i * i

# Consume lazily
for x in squares(10):
    print(x)
```

Why?

- Lower memory
- Can handle large or infinite data
- Composable

MEASURE → UNDERSTAND → IMPROVE

1. Measure (profile / timeit / memory)
2. Understand the bottleneck
3. Apply the right optimization
4. Measure again (verify improvement)

Premature optimization is the root of all evil.
But unmeasured code is a performance bug.

Golden Rule:
Make it work. Make it correct. Make it fast.

Save this slide for your next interview 👇

Profile. Measure. Optimize.
Write fast, memory-efficient Python.

das-purnendu

TESTING WITH PYTEST

1 BASIC TESTS

```python
# test_math.py – pytest auto-discovers test_*.py
import pytest
from app import divide

def test_divide():
    assert divide(10, 2) == 5

def test_divide_zero():
    with pytest.raises(ZeroDivisionError):
        divide(4, 0)
```

What's happening?

- Test functions start with test_
- Use assert for verifications
- Use pytest.raises to check exceptions

2 PARAMETRIZE — TABLE-DRIVEN TESTS

```python
@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),
    (0, 0, 0),
    (-1, 1, 0),
])
def test_add(a, b, expected):
    assert a + b == expected
```

Why parametrize?

- Runs the same test with multiple inputs
- Cleaner, DRYer tests
- Great for edge cases

3 FIXTURE — REUSABLE SETUP

```python
@pytest.fixture
def db():
    conn = connect()        # setup
    yield conn              # provide to test
    conn.close()            # teardown

def test_query(db):
    assert db.query("...") is not None
```

Fixtures are:

- Reusable
- Can be shared across tests
- Support setup and teardown

4 MOCKING — ISOLATE EXTERNALS

```python
from unittest.mock import patch, MagicMock

@patch("app.requests.get")
def test_fetch(mock_get):
    mock_get.return_value.json.return_value = {"id": 1}
    assert fetch_user(1)["id"] == 1
```

Why mock?

- Avoid real HTTP calls
- Fast, reliable tests
- Control responses
- Test error cases

COMMON FIXTURE SCOPES

function (default)	@pytest.fixture(scope="module")	@pytest.fixture(scope="session")	@pytest.fixture(autouse=True)
Isolated	Module scope	Session scope	Autouse
Runs for each test	Once per test module	Once per test session	Applied automatically

5 HANDY ADDITIONS

Skip a test
`pytest.skip("reason")`

Skip now
`pytest.skip_now()`

Expected failure
`@pytest.mark.xfail`

Temporary directory
`tmp_path fixture`

Environment vars
`monkeypatch.setenv("KEY", "VAL")`

Example test session

```
$ pytest -v
=========================================test session starts==========================================
platform darwin -- Python 3.11.6, pytest-8.2.0
rootdir: /project
collected 6 items

tests/test_math.py::test_divide                                   PASSED [  16%]
tests/test_math.py::test_divide_zero                              PASSED [  33%]
tests/test_math.py::test_add[2-3-5]                               PASSED [  50%]
tests/test_math.py::test_add[0-0-0]                               PASSED [  66%]
tests/test_math.py::test_add[-1-1-0]                              PASSED [  83%]
tests/test_db.py::test_query                                      PASSED [ 100%]
================================== 6 passed in 0.18s ===================================
```

KEY TAKEAWAYS

- Pytest auto-discovers test files and functions.
- Write small, focused, independent tests.
- Use fixtures for reusable setup/teardown.
- Parametrize for table-driven tests.
- Mock external dependencies.
- Run with coverage. Don't guess—measure.

RUNNING TESTS

```
pytest -v
```

- -v verbose output
- -q quiet (less output)
- -k "expr" run tests matching expression
- -s stop after first failure
- -x don't capture print output

WITH COVERAGE

```
pytest -v --cov=app --cov-report=term-missing
```

- --cov=app measure coverage for app package
- --cov-report=term-missing show missing lines in terminal

Example coverage output

```
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
app/__init__.py          2      0   100%
app/math.py             18      3    83%   15-16
app/utils.py             9      4    60%   12, 25, 30
--------------------------------------------------
TOTAL                   30      7    77%
```

(lines not covered are listed in "Missing")

GOOD PRACTICES

- Test behavior, not implementation.
- Each test should be independent.
- Use meaningful test names.
- Keep tests fast (~ 100ms ideally).
- Use fixtures instead of repeated code.
- Clean up resources (yield or teardown).
- Aim for high coverage on critical code.

COMMON MISTAKES

- Forgetting to assert → test always passes.
- Tests depend on each other.
- Not mocking externals → slow/flaky tests.
- Over-mocking → tests become meaningless.
- Large tests → hard to debug.
- Not checking edge cases and errors.

PRO TIP

Let tests guide your refactors.
High coverage + good tests = confidence to change fast.

Coverage targets (guideline)

- New code: aim for 90%+
- Overall project: aim for 80%+

Test pyramid

- Many unit tests (fast)
- Some integration tests
- Few E2E tests (slow)

Save this slide for your next interview 👇

Good tests save time, bugs, and careers.
Test. Refactor. Repeat.

das-purnendu

TYPE HINTS (USE THEM)

1 WHY TYPE HINTS?

- Better IDE autocomplete & refactoring
- Catch bugs early (static analysis)
- Self-documenting code
- Safer APIs & easier maintenance

Golden Rule

Add types everywhere it adds clarity.

2 BASIC SYNTAX

```python
from typing import (
    Optional, Union, List, Dict, Tuple, Any,
    Callable, Iterable, Iterator, TypeVar, Generic
)

def greet(name: str, times: int = 1) -> str:
    return f"hi {name}" * times
```

Notes

- Parameter types come after :
- Return type comes after ->
- Default values still work

3 MODERN SYNTAX (PYTHON 3.9+ / 3.10+)

```python
def f(items: list[int], cfg: dict[str, str]) -> None: ...
def g(x: int | None) -> str | None: ...    # 3.10+
# Built-in generics: list, dict, set, tuple, etc.
# Union shorthand with | (3.10+)
```

Versions

- 3.9+ built-in generic types
- 3.10+ | for Union shorthand

4 CALLABLE

```python
Handler = Callable[[int, str], bool]

def process(h: Handler, x: int) -> bool:
    return h(x, 'ok')
```

Callable[[Args...], ReturnType]

Describes function signatures.

5 GENERICS WITH TYPEVAR

```python
T = TypeVar("T")
def first(items: list[T]) -> T:
    return items[0]
def identity(x: T) -> T:
    return x
```

TypeVar

- Represents an unknown type
- Preserves the same type in and out

6 MORE USEFUL TYPES

```python
def sum_all(xs: Iterable[int]) -> int: ...
def read_lines(path: str) -> Iterator[str]: ...
Point = Tuple[float, float]
JSON = Dict[str, Any]
MaybeInt = Optional[int]   # same as int | None
```

Optional[T]

Shortcut for Union[T, None] (or T | None in 3.10+)

CHEAT SHEET: COMMON TYPES

Type	Example
str, int, float, bool	name: str, count: int
list, set, tuple	items: list[int]
dict	cfg: dict[str, str]
Optional[T]	x: Optional[int]
Union[A, B]	x: Union[int, str]
T | None (3.10+)	x: int | None
Callable[[A, B], R]	handler: Callable[[int, str], bool]
Iterable[T]	xs: Iterable[str]
Iterator[T]	it: Iterator[int]
Any	data: Any

PRO TIP

Be as specific as practical, as general as necessary.

Prefer precise types over Any.

7 TYPEDDICT (STRUCTURED DICTS)

```python
from typing import TypedDict

class User (TypedDict):
    name: str
    age: int
    email: str
```

Why?

- Dicts with a fixed schema
- Better than Dict[str, Any]

8 LITERAL & FINAL

```python
from typing import Literal, Final

Mode = Literal["r", "w", "a"]
MAX: Final = 100
```

Why?

- Literal: restricts allowed values
- Final: constant (should not change)

9 CHECK WITH MYPY

```
$ mypy app.py --strict
Success: no issues found in 1 source file

# Good flags to know
--strict                  Enable all strict checks
--warn-redundant-casts    Warn for unnecessary casts
--ignore-missing-imports  Ignore missing type info
--show-error-codes        Show error codes
```

10 BEFORE vs AFTER

BEFORE (harder to read & riskier)

```python
def process(items, cfg):
    total = 0
    for x in items:
        total += x["value"]
    return total
```

AFTER (clearer & safer)

```python
def process(items: list[dict[str, int]],
            cfg: dict[str, int]) -> int:
    total: int = 0
    for x in items:
        total += x["value"]
    return total
```

Win

- Readable
- Self-documenting
- IDE helps
- Static checks catch issues early

COMMON PITFALLS

- Forgetting return types
- Using Any everywhere
- Incorrect Optional (use | None)
- Overly complex types (keep it simple)
- Out-of-date annotations (keep in sync!)

Type hints don't slow you down.
They help you move faster.

Your future self (and teammates) will thank you.
Type it right. Sleep well at night. 🌙

das-purnendu

VIRTUAL ENVS, PIP, PACKAGING

1 VIRTUAL ENVIRONMENTS (venv)

Create

```
python -m venv .venv
```

Activate

Linux / macOS

```
source .venv/bin/activate
```

Windows

```
.venv\Scripts\activate
```

Always work inside a virtual environment.
Keeps dependencies isolated per project.

Tips

- Name it `.venv` (and add to `.gitignore`)
- Recreate anytime
- Never commit the venv directory

2 PIP ESSENTIALS

```
pip install requests                # install a package
pip install -r requirements.txt     # install from file
pip freeze > requirements.txt       # export exact versions
pip install -e .                    # editable install (dev)
pip uninstall requests              # remove a package
pip show requests                   # package info
pip list                            # list installed packages
pip check                           # check for conflicts
pip cache purge                     # clear download cache
```

editable install (`-e .`)

Installs your local package in "editable" mode.
Changes to your code are reflected immediately.

3 REQUIREMENTS FILE

```
# requirements.txt
requests>=2.31
pytest>=7.4
```

Best practices

- Pin minimum versions
- One requirement per line
- Keep it clean

4 PYPROJECT.TOML (PEP 621)

```
[project]
name = "myapp"
version = "0.1.0"
description = "My awesome app"
readme = "README.md"
requires-python = ">=3.9"
dependencies = [
  "requests>=2.31",
]

[project.optional-dependencies]
dev = [
  "pytest",
  "ruff",
  "mypy",
]

[build-system]
requires = ["setuptools>=69", "wheel"]
build-backend = "setputools.build_meta"
```

Notes

- PEP 621 is the modern standard for Python packaging.
- All metadata in one place.
- Works with modern build backends (setuptools, hatchling, flit, ...).

PROJECT LAYOUT (TYPICAL)

```
myapp/
├── src/
│   └── myapp/
│       ├── __init__.py
│       └── core.py
├── tests/
├── pyproject.toml
├── README.md
└── .venv/          # not committed
```

QUICK TIPS

- Use a venv per project.
- Pin your dependencies.
- Keep metadata in `pyproject.toml`.
- Automate with modern tools.
- Ship quality, reproducible builds.

CHEAT SHEET

Task	Command
Create venv	`python -m venv .venv`
Activate (Linux/macOS)	`source .venv/bin/activate`
Activate (Windows)	`.venv\Scripts\activate`
Install package	`pip install <pkg>`
Install from file	`pip install -r requirements.txt`
Freeze deps	`pip freeze > requirements.txt`
Editable install	`pip install -e .`
Uninstall	`pip uninstall <pkg>`
List packages	`pip list`
Check issues	`pip check`

5 MODERN TOOLING (FASTER + BETTER DX)

uv

Ultra-fast Python package installer and virtual environment manager.

```
uv venv
uv add requests
uv pip install -r requirements.txt
uv run python app.py
```

poetry

Dependency management and packaging, batteries included.

```
poetry new myapp
poetry add requests
poetry add --group dev pytest
poetry build
poetry publish
```

pipx

Install and run CLI tools globally in isolated environments.

```
pipx install black
pipx install httpie
pipx list
pipx upgrade-all
```

6 BUILD & DISTRIBUTE

```
python -m build          # build sdist + wheel (uses pyproject.toml)
twine upload dist/*      # upload to PyPI
```

Artifacts created in `./dist/`
Prefer wheels over source distributions for speed.

7 SETUP.PY IS LEGACY

`setup.py` is the old way.
Still works, but new projects should use `pyproject.toml`.

Why pyproject.toml?

- Standardized (PEP 517, PEP 518, PEP 621)
- Better tooling support
- Easier builds and isolation
- Future-proof

ONE-LINERS

Run a module inside venv	`python -m yourmodule`
Install and run a tool with pipx	`pipx install ruff && ruff check .`
Recreate env from scratch	`rm -rf .venv && python -m venv .venv`
Update all packages (carefully)	`pip list --outdated` `pip install -U <pkg>`

Isolate. Install. Package. Ship.
The boring stuff that makes great projects.

Use modern standards.
Your future self (and team) will thank you.

das-purnendu

CODE QUALITY: PYLINT, RUFF, BLACK, MYPY

1 FORMATTER — BLACK

Auto-format your code. Consistent style, zero bikeshedding.

```
$ black app.py
reformatted app.py

$ black --check .      # CI mode

All done! ✨  3 files would be left unchanged.
```

Why Black?

- Opinionated
- Fast
- Stable
- Integrates well with tools & CI

2 LINTER — PYLINT vs RUFF

pylint

- Thorough analysis (deep checks)
- More configuration, slower

```
$ pylint app.py
************* Module app
app.py:10:0: C0114: Missing module docstring (missing-module-docstring)
app.py:23:8: W0612: Unused variable 'tmp' (unused-variable)
Your code has been rated at 7.50/10
```

ruff

- Fast, modern linter written in Rust
- Great defaults (recommended)

```
$ ruff check .
app.py:10:5: F841 Local variable 'tmp' is assigned to but never used
app.py:23:1: E722 Do not use bare except
Found 2 errors.
$ ruff check --fix .
Found 2 errors (2 fixed, 0 remaining).
```

Ruff rules = Pyflakes + pycodestyle + isort + more (and growing)

3 IMPORTS — ISORT (OR RUFF)

Keep imports clean and consistently ordered.

```
$ isort app.py
Fixing /Users/me/app.py
$ ruff check --select I --fix .
# Sort imports via Ruff
```

Tip

Ruff can sort imports too. One tool to rule them all! 💪

4 TYPE CHECKER — MYPY

Catch type errors before runtime.

```
$ mypy app.py --strict
app.py:15: error: Argument 1 to "foo" has incompatible type "int"; expected "str"  [arg-type]
app.py:27: error: Incompatible return value type (got "None", expected "User")  [return-value]
Found 2 errors in 1 file (checked 1 source file)
```

Why mypy?

- Fewer runtime bugs
- Better refactoring
- Self-documenting types
- Great IDE support

5 WHAT PYLINT / RUFF CATCH

- **Unused variables / imports** — Code that does nothing
- **Unreachable code** — Code that never runs
- **Shadowing builtins** — e.g., `list = []`
- **Wrong import order** — Non-standard ordering
- **Mutable default args** — `def f(x, items=[]): ...`
- **Cyclomatic complexity** — Functions too complex
- **Too-broad excepts** — `except:` swallows bugs
- **And many more...** — Dead code, duplicates, etc.

6 PRE-COMMIT — RUN QUALITY CHECKS ON EVERY COMMIT

Automate formatting, linting, and type checking before code is committed.

`.pre-commit-config.yaml`

```
repos:
  - repo:  https://github.com/astral-sh/ruff-pre-commit
    rev:  v0.5.6
    hooks:
      - id: ruff            # run linter
      - id: ruff-format     # format with Ruff (Black-compatible)
  - repo:  https://github.com/pre-commit/mirrors-mypy
    rev:  v1.10.0
    hooks:
      - id:  mypy           # type check
        args:  ["--strict"]
```

How it works

1. You commit changes
2. pre-commit runs the hooks
3. Issues are shown & fixed (if --fix)
4. Only clean code gets committed ✅

Setup

```
$ pip install pre-commit
$ pre-commit install
# Optional: run on all files now
$ pre-commit run --all-files
```

Run everything manually (when needed)

```
$ ruff check . && black --check . && mypy . --strict
```

7 TOOL COMPARISON

Tool	Purpose	Speed	Auto-fix	Strengths	Notes
black	Formatter	⚡ ⚡ ⚡	Yes	Consistent style, zero configs	Use first (format), then lint
ruff	Linter + more	⚡ ⚡ ⚡ ⚡	Yes	Super fast, modern, many rules	Includes import sorting (I)
pylint	Linter	⚡	Partial	Deep analysis, lots of checks	Slower, more config needed
isort	Import sorter	⚡ ⚡ ⚡	Yes	Clean import order	Ruff can do this too
mypy	Type checker	⚡ ⚡	N/A	Static type safety	Use --strict for best results

Recommended workflow

1. Write code
2. `black app.py`
3. `ruff check --fix .`
4. `mypy . --strict`
5. Commit (pre-commit runs automatically)

PRINCIPLE

Clean code is not about you.
It's about your future self and your team.

THE GOAL

- Consistent style
- Fewer bugs
- Easier reviews
- Confident refactors

REMEMBER

Automate it.
Don't rely on memory. Rely on tools.

Automate quality. Ship with confidence.

Better tools. Better code. Better you. 🚀

das-purnendu

ALGORITHM PATTERNS (THE 80/20)

Master these patterns and you can solve ~80% of common coding problems.
Recognize → Choose pattern → Implement → Test → Optimize.

1 TWO POINTERS — Sorted array, pair sum

```python
l, r = 0, len(arr) - 1
while l < r:
    s = arr[l] + arr[r]
    if s == target:
        return (l, r)
    elif s < target:
        l += 1
    else:
        r -= 1
```

When to use
- Sorted array/list
- Find pair with given sum
- Remove duplicates
- Partition problems
- 3-sum (extended)

Complexity
- Time: O(n)
- Space: O(1)

Example

arr = [1, 2, 3, 4, 6, 9], target = 7

1	2	3	4	6	9
↑ l					↑ r

1 + 9 = 10 > 7 → r--

1 + 6 = 7 == 7 ✓

return (0, 4)

2 SLIDING WINDOW — Longest substring w/o repeat

```python
seen, l, best = {}, 0, 0
for r, c in enumerate(s):
    if c in seen and seen[c] >= l:
        l = seen[c] + 1
    seen[c] = r
    best = max(best, r - l + 1)
# best is the answer
```

When to use
- Subarray / substring
- Fixed or variable window
- Constraints on window
- Longest / shortest range

Complexity
- Time: O(n)
- Space: O(min(n, |alphabet|))

Example

s = "abcabcbb"

a	b	c	a	b	c	b	b
0	1	2	3	4	5	6	7

Window moves right (r).
Left (l) jumps past repeats.
Answer = 3 ("abc")

3 BINARY SEARCH — Use bisect (built-in)

```python
from bisect import bisect_left, insort

arr = [1, 2, 4, 4, 6, 8]
x = 4

i = bisect_left(arr, x)   # first index >= x
# arr[i] is the insertion point for x
# Use insort to keep list sorted
# insort(arr, x)
```

When to use
- Sorted data
- Find position / boundary
- Lower bound / upper bound
- Range queries

Complexity
- Time: O(log n)
- Space: O(1)

Example

arr = [1, 2, 4, 4, 6, 8], x = 4

0	1	2	3	4	5
1	2	4	4	6	8
		↑			

bisect_left → 2 (first 4)

bisect_right → 4 (after last 4)

4 PREFIX SUM — Range sum in O(1)

```python
arr = [3, 1, 4, 1, 5]

pre = [0]
for x in arr:
    pre.append(pre[-1] + x)

# range sum [l, r] inclusive
l, r = 1, 3
range_sum = pre[r+1] - pre[l]   #= 1 + 4 + 1 = 6
# pre: [0, 3, 4, 8, 9, 14]
```

When to use
- Range sum queries
- Subarray sums
- 2D prefix sums (grids)
- Count sums with hashmap

Complexity
- Precompute: O(n)
- Query: O(1)
- Space: O(n)

Example

arr = [3, 1, 4, 1, 5]

pre = [0, 3, 4, 8, 9, 14]

index:	0	1	2	3	4	5
pre:	0	3	4	8	9	14
		l = 1			r = 3	

range_sum = pre[r+1] - pre[l]
= pre[4] - pre[1] = 9 - 3 = 6

KEY TAKEAWAYS

Recognize the pattern
Most problems are variations of a few core patterns.

Choose the right tool
Pick the simplest pattern that fits the constraints.

Code carefully
Edge cases, boundaries, and off-by-one errors matter.

Analyze complexity
Optimize only after it works and passes tests.

Practice these patterns daily.
Reuse them. Remember them. Get faster.

Patterns + Practice = Problem Solved.
Keep grinding! 💪

30/35

das-purnendu

GRAPHS, TREES, DP

Core Patterns You'll See Everywhere

These patterns show up everywhere.
Master them once, use them forever.

Model it | Traverse it | Optimize it

1 BFS — SHORTEST PATH IN UNWEIGHTED GRAPH

```python
from collections import deque

def bfs(graph, start):
    q = deque([start])
    seen = {start}
    while q:
        node = q.popleft()
        for nxt in graph[node]:
            if nxt not in seen:
                seen.add(nxt)
                q.append(nxt)
    return seen
```

WHEN TO USE
- Unweighted graph
- Shortest path (min # of edges)
- Level-order traversal

COMPLEXITY
- Time: O(V + E)
- Space: O(V)

Mnemonic
BFS = Breadth = wide before deep. Queue (FIFO).

Why deque?
`list.pop(0)` is O(n). `deque.popleft()` is O(1).

Example — Start at A

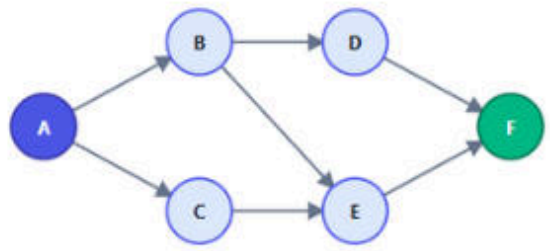
ADJACENCY LIST
A: [B, C]
B: [D, E]
C: [E]
D: [F]
E: [F]
F: []

BFS ORDER
A → B → C → D → E → F

SHORTEST DISTANCE FROM A ✓ FIXED

Node	A	B	C	D	E	F
Dist	0	1	1	2	2	3

GRAPH VISUALIZATION



2 DFS — RECURSIVE

```python
def dfs(node, seen=None):
    if seen is None:
        seen = set()
    seen.add(node)
    for nxt in graph[node]:
        if nxt not in seen:
            dfs(nxt, seen)
    return seen

# Iterative DFS uses a stack
# instead of the call stack.
```

WHEN TO USE
- Explore / traverse
- Connected components
- Cycle detection
- Topological sort
- Backtracking

COMPLEXITY
- Time: O(V + E)
- Space: O(V) recursion

Mnemonic
DFS = Depth = deep before wide. Stack (LIFO).

Recursion limit
Default is 1000. For deep graphs use `sys.setrecursionlimit()` or iterative DFS.

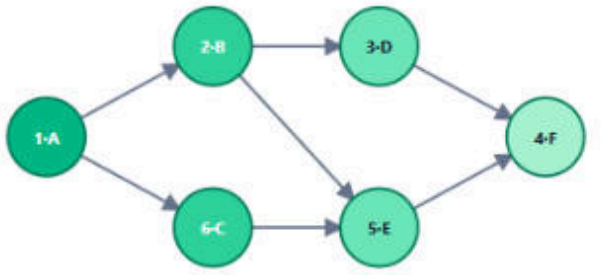
Example (DFS from A)

ADJACENCY LIST (SAME AS BFS)
A: [B, C] B: [D, E]
C: [E] D: [F] E: [F]

ONE POSSIBLE DFS ORDER
A → B → D → F → E → C

Tip — Iterative DFS
Use a stack (LIFO) instead of recursion to avoid Python's recursion-depth limit on deep graphs.

TRAVERSAL PATH (NUMBERED)



3 DIJKSTRA — SHORTEST PATH, WEIGHTED

```python
import heapq

def dijkstra(graph, src):
    dist = {v: float('inf') for v in graph}
    dist[src] = 0
    pq = [(0, src)]   # (dist, node)
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]: continue
        for v, w in graph[u]:
            if dist[v] > d + w:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return dist
```

WHEN TO USE
- Weighted graph
- Non-negative weights
- Shortest path (single source)

COMPLEXITY
- Time: O((V + E) log V)
- Space: O(V)

No negative weights
Use Bellman-Ford for those.

Why a heap?
Always pop the smallest unprocessed distance. heapq gives O(log n) push/pop.

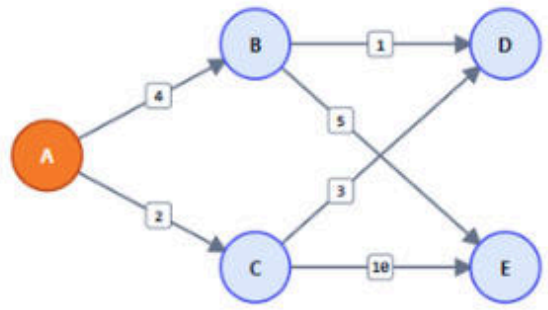
Example — Source: A

WEIGHTED EDGES
A → B (4) A → C (2)
B → D (1) B → E (5)
C → D (3) C → E (10)

SHORTEST DISTANCES FROM A

Node	A	B	C	D	E
Dist	0	4	2	5	9

WEIGHTED GRAPH



Best path A→E goes via B: 4+5 = **9** (not direct 10).

4 DP — MEMOIZATION (TOP-DOWN)

```python
from functools import lru_cache

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

# lru_cache = memoization
# O(n) time, O(n) space
```

WHEN TO USE
- Overlapping subproblems
- Optimal substructure
- Count / optimize / decide

COMPLEXITY
- Time: O(states × transitions)
- Space: O(states) memo

The 3-step recipe
1. Define state 2. Write recurrence
3. Identify base cases.

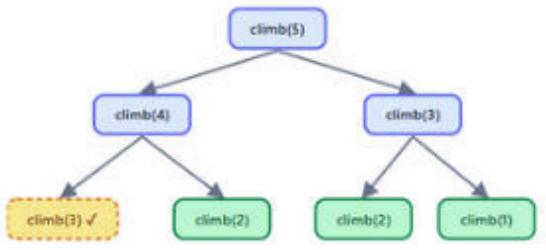
Top-down vs Bottom-up
Memoization recurses + caches. Tabulation builds the answer iteratively from base cases.

Climb Stairs (n = 5) — 1 or 2 steps at a time

n	0	1	2	3	4	5
ways	1	1	2	3	5	8

This is Fibonacci!
Memoization avoids recomputing the same subproblems.

RECURSION TREE (WITH MEMOIZATION)



Yellow dashed = served from cache. Green = base case.

5 TREES QUICK RECAP

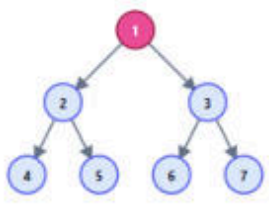
BINARY TREE TRAVERSALS
- Inorder (L, Root, R)
- Preorder (Root, L, R)
- Postorder (L, R, Root)
- Level-order (BFS)

COMMON TREE PROBLEMS
- Height / Depth
- LCA (Lowest Common Ancestor)
- Path sum
- Serialize / Deserialize
- Diameter

USEFUL PATTERNS
- BFS (level order)
- DFS (traversals)
- Backtracking
- Divide & Conquer

DP PATTERNS
- 1D / 2D DP
- Knapsack
- LIS
- Subset / Partition
- Memo vs Tabulation

SAMPLE TREE



das-purnendu

TIME COMPLEXITY CHEATSHEET

Operation	List	Dict	Set	Deque	Heap (min-heap)
Access by index	O(1)	—	—	O(n)	—
x in s (membership test)	O(n)	O(1)	O(1)	O(n)	O(n)
Insert end	O(1)	O(1)	O(1)	O(1)	—
Insert front	O(n)	—	—	O(1)	—
Delete	O(n)	O(1)	O(1)	O(1)	O(log n)
Min / Max	O(n)	—	—	—	O(1) peek

Sort (Timsort) — Time: O(n log n) — Python uses Timsort (stable, adaptive, fast in practice).

Heap push / pop — Time: O(log n) — Push or pop maintains the heap property.

String concat in loop — Time: $O(n^2)$ — Repeated `s = s + x` is $O(n^2)$. Use `"".join(list_of_strings)` → O(n).

bisect (on sorted list) — Search: O(log n) — Binary search to find position is O(log n). Insert: O(n) — Insertion may shift elements → O(n).

Space (Recursion) — Space: O(depth) — Recursion uses O(depth) space on the call stack.

Rule of Thumb

Lower complexity (big-O) is better.
Aim for algorithms that scale.

Complexity order (best → worst):

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

das-purnendu

DESIGN PATTERNS (PYTHONIC VERSIONS)

Pythonic ≠ Ceremony.
Prefer simple, readable, and composable solutions over heavy frameworks.

1 SINGLETON

Most cases, just use a module.

Modules are singletons by import semantics.

```python
# config.py (loaded once)
DB_URL = "postgres://...."
TIMEOUT = 30

# everywhere
import config
print(config.DB_URL)
```

- ✓ No class boilerplate.
- ✓ Instance is created once per interpreter (per process).
- ✓ Simple and explicit.

2 FACTORY

Centralize object creation logic.

```python
def make_shape(kind, **kw):
    return {"circle": Circle, "square": Square}[kind](**kw)

s1 = make_shape("circle", radius=5)
s2 = make_shape("square", side=3)
```

- ✓ Hide construction details.
- ✓ Easy to add new types.
- ✓ Returns the right object.

3 STRATEGY

Pass a function to change behavior.

```python
def sort_by(items, strategy):
    return sorted(items, key=strategy)

by_name = lambda x: x.name.lower()
by_age  = lambda x: x.age

users_by_name = sort_by(users, by_name)
users_by_age  = sort_by(users, by_age)
```

- ✓ Swap algorithms at runtime.
- ✓ Functions > classes (less code, more flexible).
- ✓ Great with key=, reverse=, etc.

4 OBSERVER

Publish / Subscribe.

```python
class Event:
    def __init__(self):
        self.subs = []
    def subscribe(self, fn):
        self.subs.append(fn)
    def emit(self, data):
        for fn in self.subs:
            fn(data)

ev = Event()
ev.subscribe(lambda d: print(f"log → {d}"))
ev.subscribe(lambda d: metrics.increment(d))
ev.emit({"user": "alice"})
```

- ✓ Loose coupling.
- ✓ Any callable can subscribe.
- ✓ Easy fan-out.

5 CONTEXT MANAGER

RAII / Cleanup.

```python
from contextlib import contextmanager
@contextmanager
def db_session():
    s = open_session()
    try:
        yield s    # use the resource
        s.commit()
    except Exception:
        s.rollback()
        raise
    finally:
        s.close()

with db_session() as s:
    s.execute("select 1")
```

- ✓ Deterministic cleanup.
- ✓ Works with `with` statement.
- ✓ Safer and clearer resource management.

6 DECORATOR

Add behavior without modifying the function.

```python
import functools, time
def logged(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        t0 = time.perf_counter()
        try:
            return fn(*args, **kwargs)
        finally:
            dt = (time.perf_counter() - t0) * 1000
            print(f"{fn.__name__} took {dt:.2f} ms")
    return wrapper

@logged
def add(a, b):
    return a + b
```

- ✓ Reusable cross-cutting behavior (logging, retry, auth, caching, timing…).
- ✓ Stackable and composable.

What patterns do I actually use?

- ✓ Composition
- ✓ Context Managers
- ✓ Decorators
- ✓ Strategy via Functions

Simple tools.
Powerful when combined.

das-purnendu

INTERVIEW TRAPS THAT FAIL CANDIDATES

These small mistakes cause big bugs.
Know them. Avoid them. Ace the interview.

1 Late-binding closures

```
fns = [lambda: i for i in range(3)]
print([f() for f in fns])   # [2, 2, 2]
```

Why? Lambdas capture `i` by reference. After the loop, `i == 2` for all.

Fix:

```
fns = [lambda i=i: i for i in range(3)]
print([f() for f in fns])   # [0, 1, 2]
```

2 Mutable default arguments

(See slide 21 for details)

```
def add(x, lst=[]):
    lst.append(x)
    return lst
```

Default list is created once and shared across calls.

Fix:

```
def add(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

3 List multiplication ≠ deep copy

```
grid = [[0]*3]*3    # ✗ shared rows
print(grid)
```

```
[[0, 0, 0],  →
 [0, 0, 0],  →   same object!
 [0, 0, 0]]  →
```

All rows reference the same list object.

Fix:

```
grid = [[0]*3 for _ in range(3)]  # ✓
print(grid)
```

4 == on floats

```
0.1 + 0.2 == 0.3                  # False
import math
math.isclose(0.1 + 0.2, 0.3)  # True
```

Floats have precision errors.

Use `math.isclose(a, b, rel_tol=1e-9, abs_tol=0.0)` for comparison.

5 Iterating + mutating

```
arr = [1, 2, 2, 3]  # two consecutive 2s
for x in arr:
    if x % 2 == 0:
        arr.remove(x)    # ✗ skips items
print(arr)              # [1, 2, 3] (wrong)
```

Removing while iterating skips elements.

Fix:

```
arr = [x for x in arr if x % 2 != 0] # ✓
print(arr)                  # [1, 3
```

6 Truthy / falsy

```
bool([]), bool({}), bool(0), bool("")
# (False, False, False, False)
```

Many values are falsy in Python.

if x:

- Fails for 0, 0.0, "", [], {}, None, False
- Good for "value exists and is truthy"

if x is not None:

- Only filters out None
- Use when 0, "", or [] are valid

7 Chained comparison

```
x = 5
1 < x < 10    # True
```

Correct and efficient!

`1 < x` ✓ **and** `x < 10` ✓

Equivalent to `(1 < x) and (x < 10)`
Evaluated left to right, `x` evaluated once.

8 Generators are one-shot

```
g = (x for x in range(3))
list(g)    # [0, 1, 2]
list(g)    # []
```

Once exhausted, a generator can't be reused.

Fix:

```
g = (x for x in range(3))
result = list(g)
list(result)    # [0, 1, 2]
list(result)    # [0, 1, 2]
```

9 dict.keys() is a view, not a list

```
d = {'a': 1, 'b': 2}
k = d.keys()
type(k)     # <class 'dict_keys'>
print(k)    # dict_keys(['a', 'b'])
```

- It's a dynamic view of the dictionary.
- Not a list. No indexing or slicing.

Fix / Convert if needed:

```
list(d.keys())      # ['a', 'b']
```

Bonus Reminders

- Use `is` for identity (None, singletons)
- Use `==` for value equality
- Beware of hidden mutability (lists, dicts, sets inside objects)
- Read error messages carefully
- Write small tests for edge cases
- Don't overthink built-ins—know their behavior!

Interviews test attention to detail. These traps separate good from great.
Avoid the silly mistakes. Focus on solving the real problem.

das-purnendu

PYTHONIC ONE-LINERS

Write less. Do more. Readable, efficient, Pythonic.
Use these patterns to save time and write elegant code.

1	Swap two variables	`a, b = b, a`	Example: `a, b = 1, 2 → (a, b) = (2, 1)`
2	Conditional assign	`status = "pass" if score >= 50 else "fail"`	Example: `score = 72 → 'pass'`
3	Walrus operator (3.8+)	`if (n := len(arr)) > 10: print(n)`	Example: `arr = [1]*12 → 12`
4	Most frequent element	`max(set(arr), key=arr.count)`	Example: `[1,2,2,3,3,3,2] → 2 or 3`
5	Group consecutive duplicates	`[(k, list(g)) for k, g in groupby(arr)]` `from itertools import groupby`	Example: `[1,1,2,2,2,3,1]` `→ [(1, [1, 1]), (2, [2, 2, 2]), (3, [3]), (1, [1])]`

6	Transpose matrix	`list(zip(*matrix))`	Example:
			`matrix = [[1,2,3],[4,5,6]]`
			`→ [(1,4), (2,5), (3,6)]`

7	Remove duplicates, keep order	`list(dict.fromkeys(arr))`	Example: `[3,1,2,3,2,4] → [3,1,2,4]`
8	Chunk a list	`[arr[i:i+n] for i in range(0, len(arr), n)]`	Example: `arr = [1,2,3,4,5,6,7], n = 3` `→ [[1,2,3], [4,5,6], [7]]`
9	Read file → list of lines	`Path("f.txt").read_text().splitlines()` `from pathlib import Path`	Example: `f.txt = "a\nb\nc\n"` `→ ['a', 'b', 'c']`
10	Merge & sort	`sorted(chain(a, b, c))` `from itertools import chain`	Example: `a=[3,1], b=[2], c=[4,0]` `→ [0,1,2,3,4]`
11	Frequency map → top k	`Counter(arr).most_common(k)` `from collections import Counter`	Example: `arr=[1,1,2,2,2,3], k=2` `→ [(2, 3), (1, 2)]`
12	Flatten one level	`[x for row in matrix for x in row]`	Example: `[[1,2],[3,4],[5]] → [1,2,3,4,5]`
13	Dict from two lists	`dict(zip(keys, values))`	Example: `keys=['a','b'], values=[1,2]` `→ {'a': 1, 'b': 2}`
14	Reverse dict	`{v: k for k, v in d.items()}`	Example: `{'a':1,'b':2} → {1:'a', 2:'b'}` Note: Values must be unique.
15	Sum of truthy values	`sum(bool(x) for x in arr)` `# counts truthy items`	Example: `[0,1,[],[1],{},'x',''] → 3`

Pro Tip

Readability first. Use one-liners when they make the code clearer, not clever.

- ✓ Great for interviews
- ✓ Great for quick scripts
- ✗ Avoid overuse in large codebases

BONUS

das-purnendu

PYTHON QUICK REFERENCE

Handy cheats for interviews & everyday coding

1 DATA STRUCTURES

Type	Literal / Init	Ordered	Mutable	Allows Dups	Lookup
list	`[1, 2, 3]`	✓	✓	✓	O(n)
tuple	`(1, 2, 3)`	✓	✗	✓	O(n)
set	`{1, 2, 3}`	✗	✓	✗	O(1) avg
dict	`{'k': 'v'}`	✓*	✓	Keys: ✗ Values: ✓	O(1) avg
str	`"abc"`	✓	✗	✓	O(1) index

- dicts preserve insertion order (Python 3.7+)

2 COMMON BUILT-INS

`len(x)`	Length / size	`dict(x)`	Convert to dict
`sum(x)`	Sum of items	`str(x)`	Convert to string
`min(x), max(x)`	Minimum / maximum	`int(x), float(x)`	Convert to number
`sorted(x, key=...)`	Sorted list	`abs(x)`	Absolute value
`reversed(x)`	Reverse iterator	`round(x, n)`	Round to n places
`enumerate(x, start=0)`	(index, value) pairs	`divmod(a, b)`	(a//b, a%b)
`zip(a, b, ...)`	Parallel iterator	`ord(c), chr(i)`	Char ⇄ int
`any(x)`	Any truthy?	`map(f, x)`	Apply f to each
`all(x)`	All truthy?	`filter(f, x)`	Filter truthy
`range(n)`	0 to n-1	`eval(expr)`	Evaluate (use carefully!)
`list(x)`	Convert to list	`repr(x)`	Developer string
`set(x)`	Convert to set	`id(x)`	Identity (memory addr)

3 STRING CHEATS

`s[i]`	Indexing
`s[a:b]`	Slice [a, b)
`s[::-1]`	Reverse
`s.lower(), s.upper()`	Case convert
`s.strip(), s.lstrip(), s.rstrip()`	Trim spaces
`s.split(sep)`	Split to list
`sep.join(parts)`	Join list with sep
`s.find(sub)`	First index or -1
`s.startswith(p), s.endswith(p)`	Prefix / suffix
`f"Hello {name}!"`	f-string

4 LOOPING PATTERNS

`for x, y in enumerate(arr):`	Index + value
`for i in range(0, n, step):`	Step loop
`for x in reversed(arr):`	Reverse loop
`for k, v in d.items():`	Dict key, value
`for k in d:`	Iterate keys
`while (line := f.readline()):`	Read until EOF
`for _ in range(n):`	Repeat n times

5 COMPREHENSIONS

`[x*x for x in nums]`	List comp
`{x for x in nums if x%2==0}`	Set comp
`{x: x*x for x in nums}`	Dict comp
`[x for row in matrix for x in row]`	Flatten 2D
`{f(x) for x in data if cond(x)}`	Filter + map

6 EXCEPTIONS

```
try:
    risky()
except ValueError as e:
    handle(e)
except (TypeError, IndexError):
    handle()
else:
    runs if no exception
finally:
    always runs (cleanup)
```

Best practices

- ✓ Be specific with exceptions.
- ✓ Log or add context.
- ✓ Use finally for cleanup.
- ✓ Don't ignore exceptions.

7 FILE I/O (PYTHONIC)

```
from pathlib import Path
p = Path("data.txt")
# Read text
text = p.read_text(encoding="utf-8")
lines = p.read_text().splitlines()
# Write text
p.write_text("hello\n", encoding="utf-8")
# Read / write JSON
import json
data = json.loads(p.read_text())
p.write_text(json.dumps(data, indent=2))
```

8 USEFUL SNIPPETS

Swap:	`a, b = b, a`
Clamp x to [lo, hi]:	`x = max(lo, min(hi, x))`
Is power of 2:	`n > 0 and (n & (n-1)) == 0`
Count set bits:	`bin(n).count("1")`
Unique, sorted:	`sorted(set(arr))`
Last item:	`arr[-1]`
Safe get:	`d.get(key, default)`

9 ALGORITHMIC HELPERS

Prefix sums:	`from itertools import accumulate`
Frequencies:	`from collections import Counter`
Group by key:	`from itertools import groupby`
Combinations:	`from itertools import combinations`
Permutations:	`from itertools import permutations`
Product:	`from itertools import product`
Heap:	`import heapq`

10 TIME & SPACE COMPLEXITY (AT A GLANCE)

Operation	List	Dict	Set	Notes
Access by index	O(1)	–	–	list only
Append / Add	O(1)	O(1) avg	O(1) avg	amortized
Lookup	O(n)	O(1) avg	O(1) avg	–
Insert (middle)	O(n)	–	–	shifts elements
Delete	O(n)	O(1) avg	O(1) avg	list shifts / set via O(1)
Min / Max	O(n)	–	–	use heap for O(1) peek
Sort	O(n log n)	–	–	Timsort (stable)

11 ENV & RUNTIME

Python version:	`import sys; sys.version`
Path of script:	`from pathlib import Path` `Path(__file__).resolve()`
Current time:	`from datetime import datetime` `datetime.now()`
Measure time:	`import time` `t0 = time.perf_counter()`
Memory (rough):	`import sys` `sys.getsizeof(obj)`

12 GOOD HABITS

- ✓ Write small, pure functions.
- ✓ Use meaningful names.
- ✓ Prefer built-ins & standard library.
- ✓ Handle edge cases.
- ✓ Write tests for tricky logic.
- ✓ Profile before optimizing.

REMEMBER

Pythonic code is all about readability, clarity, and leveraging the language.

Practice consistently. Patterns become second nature!

Save this slide for your next interview 👇

Understand the model. Pick the right tool. Write concurrent code that's correct and fast.

das-purnendu

FINAL CHECKLIST + CTA

PRE-INTERVIEW 30-MIN REVIEW:

- [ ] Data structures + complexities (slides 3, 31)
- [ ] Decorators, generators, context managers (10–12)
- [ ] OOP: inheritance, MRO, dataclasses (13–15)
- [ ] GIL: when threads vs processes vs async (22–23)
- [ ] Mutable defaults + late binding (33)
- [ ] One pattern per category: 2-pointer, sliding window, BFS, DP (29–30)

DURING THE ROUND:

- [ ] Clarify input/output before coding
- [ ] State your approach + complexity before typing
- [ ] **Use Counter, defaultdict, bisect, lru_cache when they fit**
- [ ] Test with edge cases: empty, single, duplicates, negatives

You just saved 12+ hours of revision.

Save this.

Send to someone interviewing this month.

Follow Purnendu Das

for more deep technical handbooks like this one — Python, GenAI, system design, RAG.

das-purnendu

- Interview-focused
- Concise & practical
- Used by 100K+ engineers
- Continuously updated

Next handbook:

SQL | System Design | LangChain | FastAPI | Pandas

Comment your pick 👇