

PYTHON FUNDAMENTALS

for Data Science Interviews

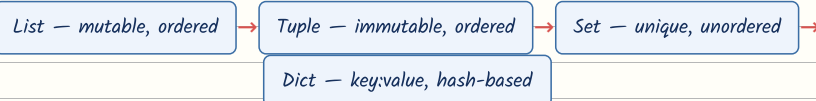


- 1. Data Structures & Time Complexity
- 2. List / Dict Comprehensions
- 3. Functions — default, keyword, *args, **kwargs
- 4. OOP — class, inheritance, magic methods
- 5. Decorators
- 6. Generators & Iterators
- 7. Exception Handling
- 8. File I/O

Phase 0 • Placement-Oriented Notes • Not Exam-Based

Data Structures & Time Complexity

► The 4 Core Structures



List []

Mutable, ordered, allows duplicates. Backed by a dynamic array → index access is instant, but inserting in the middle shifts everything.

Tuple ()

Same as list but immutable → hashable → can be used as a dict key / set element. Slightly faster than list due to fixed size.

Dict {}

Key-value store built on a hash table. Key is hashed → mapped to a bucket → near-instant lookup regardless of size. Python 3.7+ dicts preserve insertion order.

Set {}

Like dict but only keys (no values). Also hash-based → great for de-duplication and $O(1)$ membership checks (`x in my_set`).

► Time Complexity Cheat-Table

Operation	List	Tuple	Dict	Set
Index / Access	$O(1)$	$O(1)$	$O(1)$ avg	—
Search (value)	$O(n)$	$O(n)$	$O(1)$ avg (key)	$O(1)$ avg
Insert (end)	$O(1)$ amortized	N/A	$O(1)$ avg	$O(1)$ avg
Insert (middle)	$O(n)$	N/A	—	—
Delete	$O(n)$	N/A	$O(1)$ avg	$O(1)$ avg

Worst case for dict/set is $O(n)$ on hash collisions — but this is rare and interviewers usually just want the average case.

🔗 Common Interview Questions

- Why is dict/set lookup $O(1)$ on average? → Hashing maps key directly to a memory bucket, no scanning needed.
- When would you use a tuple over a List? → Fixed data that shouldn't change, or when it needs to be a dict key / set element (hashable).
- Why can't a list be a dictionary key? → Lists are mutable → unhashable. Mutating it would break the hash-bucket mapping.
- List vs Array (NumPy)? → List can hold mixed types, no vectorized ops. NumPy array is fixed-type, contiguous memory → fast vectorized math.

List / Dict Comprehensions

Syntax: [expr for item in iterable if condition] — a compact way to build a list/dict/set in one line instead of a for-loop + append.

Comprehensions are preferred in DS code because they're faster than an equivalent for-loop (C-optimized internally) and read closer to math/set-builder notation.

```
# List comprehension
squares = [x**2 for x in range(10) if x % 2 == 0]

# Dict comprehension
prices = {"apple": 40, "mango": 80}
discounted = {k: v * 0.9 for k, v in prices.items()}

# Nested comprehension (flatten a matrix)
matrix = [[1, 2], [3, 4]]
flat = [x for row in matrix for x in row]

# Set comprehension
unique_lens = {len(w) for w in ["data", "ai", "ml", "sql"]}
```

► Comprehension vs map/filter

Aspect	Comprehension	map / filter
Readability	Usually cleaner	Needs lambda, gets cluttered
Speed	Comparable / slightly faster	Comparable
Multiple conditions	Easy (chain if's)	Needs nested filter

🔗 Common Interview Questions

- When does a comprehension hurt readability? → When nested more than 2 levels deep — switch to a normal loop for clarity.
- Comprehension vs generator expression? → Comprehension builds the full list in memory now; generator expression (parentheses instead of []) is lazy — computed one item at a time.
- Can you write a dict comprehension that swaps keys and values? → `{{v: k for k, v in d.items()}}`

Functions — default, keyword, *args, **kwargs

Default vs Keyword Arguments

Default = a fallback value if caller doesn't pass one. Keyword = calling by name instead of position, e.g.

`f(lr=0.01)` — order doesn't matter then.

*args and **kwargs

*args collects any extra positional arguments into a tuple. **kwargs collects extra keyword arguments into a dict.

Used heavily in ML wrapper functions (e.g. sklearn pipelines, decorators) to forward arguments without knowing them in advance.

```
def train_model(data, epochs=10, *args, lr=0.01, **kwargs):
    # data          -> positional (required)
    # epochs=10     -> default/keyword argument
    # *args         -> extra positional args, collected as a tuple
    # lr=0.01       -> keyword-only argument (after *args)
    # **kwargs      -> extra keyword args, collected as a dict
    print(data, epochs, args, lr, kwargs)

train_model("dataset", 20, "extra1", "extra2", lr=0.05, batch_size=32)
# -> dataset 20 ('extra1', 'extra2') 0.05 {'batch_size': 32}

# Argument-order rule when DEFINING a function:
# positional -> *args -> keyword/default -> **kwargs
```

► The Mutable Default Argument Trap

A very common interview gotcha — default arguments are evaluated once, at function definition time, not on every call.

```
# Classic gotcha: mutable default argument
def add_item(item, bucket=[]): # BAD - list is created ONCE, shared across calls
    bucket.append(item)
    return bucket

print(add_item(1)) # [1]
print(add_item(2)) # [1, 2] <-- unexpected!

# Fix:
def add_item(item, bucket=None):
    if bucket is None:
        bucket = []
    bucket.append(item)
    return bucket
```

🕒 Common Interview Questions

- Why is a mutable default argument dangerous? → It's created once at def-time and shared/reused across all calls that don't override it.
- Difference between *args and **kwargs? → *args = extra positional (tuple); **kwargs = extra keyword (dict).
- Can you have arguments after **kwargs? → No, **kwargs must always be last in the function signature.
- What does `f(*my_list)` do at a CALL site? → Unpacks the list into separate positional arguments.

OOP — class, inheritance, magic methods

self = reference to the current instance (like this in other languages). Class variable = shared across all objects. Instance variable = unique per object.

```
class Model:
    total_models = 0 # class variable (shared)

    def __init__(self, name, accuracy):
        self.name = name # instance variable
        self.accuracy = accuracy
        Model.total_models += 1

    def __repr__(self): # for developers / debugging
        return f"Model({self.name!r}, acc={self.accuracy})"

    def __str__(self): # for end users / print()
        return f"{self.name} -> {self.accuracy*100:.1f}% accurate"

    def __eq__(self, other):
        return self.accuracy == other.accuracy

    @classmethod
    def from_dict(cls, d): # alternate constructor
        return cls(d["name"], d["accuracy"])

    @staticmethod
    def is_valid(accuracy): # no self/cls needed
        return 0 <= accuracy <= 1

class BoostedModel(Model): # inheritance
    def __init__(self, name, accuracy, n_estimators):
        super().__init__(name, accuracy) # call parent's __init__
        self.n_estimators = n_estimators
```

► Method Types

Type	First arg	Use case
Instance method	self	Normal behaviour tied to one object
@classmethod	cls	Alternate constructors, factory methods
@staticmethod	none	Utility logic that doesn't touch object/class state

► Magic (Dunder) Methods

__init__ → constructor

__str__ → print()

__repr__ → debug/repl

__eq__ → ==

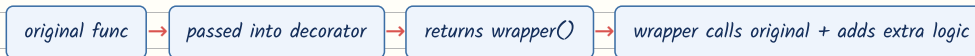
__len__ → len()

🔗 Common Interview Questions

- __str__ vs __repr__? → __str__ is user-facing (print), __repr__ is developer-facing (unambiguous, used in debugger/REPL). If __str__ is missing, Python falls back to __repr__.
- classmethod vs staticmethod? → classmethod receives the class (cls) and can create instances (factory pattern); staticmethod is just a regular function grouped inside the class for organization.
- What is MRO (Method Resolution Order)? → The order Python searches parent classes for a method in multiple inheritance — left-to-right, depth-first (C3 linearization). Check via ClassName.__mro__.
- Why use super().__init__() instead of calling the parent class directly? → super() respects the MRO, so it works correctly even in multiple/diamond inheritance.

Decorators

A decorator is a function that takes another function, wraps extra behaviour around it, and returns the wrapped version — without changing the original function's code.



```

import functools
import time

def timer(func):
    @functools.wraps(func)          # preserves func's name/docstring
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time()-start:.4f}s")
        return result
    return wrapper

@timer                             # equivalent to: train = timer(train)
def train(data):
    time.sleep(1)
    return "model trained"

train(data=[1, 2, 3])
  
```

Used constantly in real DS/ML codebases: logging, timing model training, caching (@functools.lru_cache), authentication in APIs (Flask/FastAPI use @app.route), retry logic.

🔗 Common Interview Questions

- Why use a decorator instead of editing the function directly? → Separation of concerns — reusable cross-cutting logic (timing, logging, auth) without touching business logic.
- What does functools.wraps do and why is it needed? → Copies over __name__/__doc__ from the original function to the wrapper; without it, introspection/debugging shows 'wrapper' instead of the real function name.
- Can a function have multiple decorators? → Yes, applied bottom-up (closest to the function runs first).
- What's @lru_cache used for in a DS context? → Caches expensive function results (e.g. repeated feature computation) so repeated calls with same input are instant.

Generators & Iterators

Iterator = any object implementing `__iter__` + `__next__`. Generator = a shortcut to build an iterator using `yield` instead of manually managing state.

```
# Iterator protocol: needs __iter__ and __next__
class Counter:
    def __init__(self, limit):
        self.i, self.limit = 0, limit
    def __iter__(self):
        return self
    def __next__(self):
        if self.i >= self.limit:
            raise StopIteration
        self.i += 1
        return self.i

# Generator function -- same behaviour, far less code
def counter(limit):
    i = 1
    while i <= limit:
        yield i          # pauses here, resumes on next call
        i += 1

# Generator expression (lazy version of list comprehension)
squares_gen = (x**2 for x in range(1_000_000))  # no memory used yet
squares_list = [x**2 for x in range(1_000_000)] # entire list in memory NOW
```

► Why this matters for Data Science

When reading huge CSV/log files or streaming batches for model training, a generator processes one record at a time — constant memory — instead of loading the entire dataset into RAM like a list would.

List → loads ALL data into RAM

vs

Generator → yields ONE item at a time

🕒 Common Interview Questions

- yield vs return? → return exits the function permanently; yield pauses it, saving all local state, and resumes exactly there on the next call.
- Why prefer a generator over a list for large datasets? → Memory efficiency — $O(1)$ memory vs $O(n)$ for a full list, critical when data doesn't fit in RAM.
- Can you iterate over a generator twice? → No — once exhausted, it's done. Lists can be iterated any number of times.
- Where have you used generators practically? → Reading large files line-by-line, batch-loading data for model training, building ETL/data pipelines.

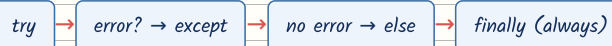
Exception Handling

```
class InvalidAccuracyError(Exception):    # custom exception
    pass

def validate(accuracy):
    if not (0 <= accuracy <= 1):
        raise InvalidAccuracyError(f"Got {accuracy}, must be in [0,1]")
    return True

try:
    validate(1.5)
except InvalidAccuracyError as e:
    print("Validation failed:", e)
except (ValueError, TypeError) as e:    # multiple exception types
    print("Bad input:", e)
else:
    print("All good, no exception raised") # runs only if NO exception
finally:
    print("Cleanup / logging always runs") # runs no matter what
```

► try block flow



🕒 Common Interview Questions

- else vs finally? → else runs only if the try block succeeded with no exception; finally always runs regardless of exception, typically used for cleanup (closing files/DB connections).
- When would you write a custom exception class? → When you want to distinguish domain-specific errors (e.g. InvalidAccuracyError) from generic Python errors, for cleaner error handling upstream.
- Why catch specific exceptions instead of a bare except? → Bare except hides bugs by silently swallowing everything, including things like KeyboardInterrupt — always catch specific types.

File I/O

```
# 'with' = context manager -> auto-closes the file even if an error occurs
with open("data.csv", "r") as f:
    for line in f:
        # reads line-by-line, memory efficient
        process(line)

# Without 'with', you'd have to manually do:
f = open("data.csv", "r")
try:
    data = f.read()
finally:
    f.close()
    # easy to forget -> file handle / memory leak

# Common modes: 'r' read, 'w' write(overwrite), 'a' append, 'rb'/'wb' binary
```

► Mode Cheat-Sheet

Mode	Meaning
r	Read (file must exist)
w	Write — overwrites/creates file
a	Append to end of file
rb / wb	Binary read/write (images, pickled models)

🕒 Common Interview Questions

- Why use 'with open()' instead of open()/close()? → Guarantees the file is closed automatically, even if an exception is raised mid-read — prevents resource/memory leaks.
- How do you read a very large file without running out of memory? → Iterate line-by-line (for line in f) or in chunks, instead of f.read() which loads the entire file into memory.
- How would you save/load a trained ML model to disk? → Binary mode with pickle/joblib: open('model.pkl', 'wb') to dump, 'rb' to load.