AI-ML ENGINEER INTERVIEW

Complete Q&A Guide

PART 1: Python & FastAPI

PYTHON FUNDAMENTALS

Q1	What is Python?
ANS	Python is a high-level, general-purpose programming language known for its clean, readable syntax. Created by Guido van Rossum in 1991, it supports multiple programming paradigms including procedural, object-oriented, and functional programming. It is widely used in web development, data science, AI/ML, automation, and scripting.

Q2	Why is Python an interpreted language?
ANS	Python is interpreted because its code is executed line-by-line by the Python interpreter (CPython) at runtime, rather than being compiled into machine code beforehand. The interpreter reads the source code, converts it to bytecode (.pyc files), and the Python Virtual Machine (PVM) executes it. This allows for rapid development, easier debugging, and platform independence, but generally makes it slower than compiled languages like C++.

Q3	What are the specialties of Python?
ANS	Python's key specialties include: simple and readable syntax (close to English), dynamic typing, extensive standard library, massive third-party ecosystem (PyPI), strong community support, cross-platform compatibility, and first-class support for AI/ML with libraries like NumPy, PyTorch, and TensorFlow. It also supports both scripting and large-scale application development.

Q4	What makes Python special compared to other languages?
ANS	Python stands out for: (1) Readability - enforced indentation makes code clean. (2) Versatility - used for web, AI/ML, scripting, automation. (3) Rapid prototyping - less boilerplate than Java or C++. (4) Dynamic typing - no need to declare types. (5) Huge ecosystem - over 400,000 packages on PyPI. (6) Interpreted nature - test code instantly without compilation. (7) Community - largest active developer community in AI/ML.

Q5 **What is the difference between compiled and interpreted languages?**

ANS

Compiled languages (C, C++, Go) translate the entire source code to machine code before execution, resulting in faster runtime performance. Interpreted languages (Python, JavaScript) translate and execute code line-by-line at runtime. Python uses a hybrid approach: source code is first compiled to bytecode, then interpreted by the PVM, offering a balance between development speed and portability.

Q6 **What are Python's key data types?**

ANS

Python's built-in data types include: int (integers), float (decimals), complex (complex numbers), str (strings), bool (True/False), list (ordered mutable sequence), tuple (ordered immutable sequence), set (unordered unique elements), frozenset (immutable set), dict (key-value pairs), bytes and bytearray (binary data), and NoneType (None). Everything in Python is an object.

Q7 **What is the difference between a list, tuple, set, and dictionary?**

ANS

List: ordered, mutable, allows duplicates → [1, 2, 3]. Tuple: ordered, immutable, allows duplicates → (1, 2, 3). Set: unordered, mutable, NO duplicates → {1, 2, 3}. Dict: unordered key-value pairs, mutable, keys are unique → {'a': 1}. Use tuples for fixed data (coordinates), sets for membership testing, dicts for structured data, and lists for general sequences.

Q8 **What is list comprehension and when should you use it?**

ANS

List comprehension is a concise, Pythonic way to create lists in a single line.
Example: squares = [x**2 for x in range(10) if x % 2 == 0]
This is equivalent to a for loop with an if condition. Use it for simple transformations or filters. Avoid it for complex logic (use regular loops instead for readability). It is faster than traditional for loops because it's optimized at the C level in CPython.

Q9 **What are Python generators and iterators?**

ANS

An iterator is any object with __iter__() and __next__() methods, allowing sequential access to elements. A generator is a special function that uses the 'yield' keyword to return values one at a time, pausing execution between yields. Generators are memory efficient because they produce values lazily (on demand) rather than storing them all in memory. Perfect for large datasets:

```
def count_up(n):
    for i in range(n): yield i
```

Q10 What is a lambda function?

ANS

A lambda is an anonymous, single-expression function defined inline using the lambda keyword. Syntax: lambda arguments: expression. Example: double = lambda x: x * 2. Commonly used with map(), filter(), and sorted(). Avoid for complex logic — use regular def functions instead. They cannot contain multiple statements or type annotations.

Q11 What are decorators in Python and how do they work?

ANS

Decorators are functions that wrap another function to extend or modify its behavior without changing its source code. They use the @decorator syntax. Internally, @my_decorator above a function is equivalent to func = my_decorator(func).

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f'Calling {func.__name__}')
        return func(*args, **kwargs)
    return wrapper
```

Common uses: logging, authentication, timing, caching (@lru_cache).

Q12 What is the difference between *args and **kwargs?

ANS

*args collects extra positional arguments into a tuple. **kwargs collects extra keyword arguments into a dictionary. Use *args when you don't know how many positional arguments a function will receive. Use **kwargs for named optional parameters. Order: regular args → *args → keyword-only args → **kwargs.

```
def func(a, *args, **kwargs): pass
```

Example call: func(1, 2, 3, name='AI')

Q13 What is OOP in Python? Explain the four pillars.

ANS

OOP (Object-Oriented Programming) organizes code around objects (instances of classes). The four pillars are:

1. Encapsulation: Bundling data and methods together; hiding internals with _ or __
2. Inheritance: A child class inherits properties/methods from a parent class
3. Polymorphism: Same method name behaves differently in different classes
4. Abstraction: Hiding complex implementation, exposing only what's needed

Python uses class keyword to define classes.

Q14 What is the difference between @staticmethod and @classmethod?

ANS

@staticmethod: No access to class or instance. It's a regular utility function inside a class. No self or cls parameter. @classmethod: Receives cls (the class itself) as the first argument. Can access and modify class-level state. Used for alternative constructors.

```
class MyClass:
    @classmethod
```

```
def from_string(cls, s): return cls()
@staticmethod
def helper(): return 42
```

Q15 What is Python's GIL and how does it affect multithreading?

ANS The GIL (Global Interpreter Lock) is a mutex in CPython that allows only one thread to execute Python bytecode at a time, even on multi-core processors. This means Python threads don't achieve true parallelism for CPU-bound tasks. However, the GIL is released during I/O operations, so threading is still useful for I/O-bound tasks (network requests, file reading). For CPU-bound parallelism, use multiprocessing instead, which bypasses the GIL by using separate processes.

Q16 What is the difference between multithreading and multiprocessing in Python?

ANS Multithreading: Multiple threads share the same memory space. Limited by GIL for CPU tasks. Good for I/O-bound operations (file reads, HTTP calls). Uses threading module. Multiprocessing: Multiple processes each have their own memory. True parallelism achieved. Good for CPU-bound tasks (ML training, image processing). Uses multiprocessing module. For ML workloads, Python often uses multiprocessing or delegates to C extensions (NumPy, PyTorch) that release the GIL.

Q17 What are Python virtual environments and why do we use them?

ANS A virtual environment is an isolated Python environment with its own packages and interpreter. It prevents version conflicts between projects. Created with: python -m venv myenv and activated with source myenv/bin/activate (Linux/Mac) or myenv\Scripts\activate (Windows). Each project maintains its own requirements.txt. Tools like pipenv, poetry, and conda offer more advanced environment management popular in ML projects.

Q18 What is the difference between shallow copy and deep copy?

ANS Shallow copy creates a new object but copies references to nested objects (not the nested objects themselves). Changes to nested mutable objects affect both. Deep copy creates a completely independent copy including all nested objects. Use copy.copy() for shallow, copy.deepcopy() for deep.

```
import copy
orig = [[1,2],[3,4]]
shallow = copy.copy(orig)   # inner lists shared
deep = copy.deepcopy(orig) # fully independent
```

Q19 What are Python context managers and the 'with' statement?

ANS Context managers manage resources (files, DB connections, locks) automatically, ensuring cleanup even if errors occur. The with statement calls __enter__() on entry and __exit__() on exit. Example: with open('file.txt') as f: data = f.read() — file is closed automatically. You can

	create custom context managers using classes with __enter__/__exit__ or with the @contextmanager decorator from contextlib.

Q20	What is exception handling in Python?
ANS	Python uses try/except/else/finally blocks for error handling. try: code that may fail. except ExceptionType: handle the error. else: runs if no exception occurred. finally: always runs (cleanup). You can catch multiple exceptions, raise custom ones (class MyError(Exception): pass), and chain exceptions. Best practice: catch specific exceptions, never bare except:, use logging in except blocks, clean up in finally.

Q21	What are dunder/magic methods in Python?
ANS	Dunder (double underscore) methods are special methods Python calls implicitly for built-in operations. Common examples: __init__ (constructor), __str__ (string representation for print), __repr__ (debug representation), __len__ (len()), __add__ (+ operator), __eq__ (== comparison), __iter__ and __next__ (iteration), __enter__/__exit__ (context manager), __call__ (make instance callable). They enable operator overloading and Pythonic interfaces.

Q22	What is PEP8 and why does it matter?
ANS	PEP8 is Python's official style guide defining coding conventions: 4-space indentation, max 79-character lines, snake_case for variables/functions, PascalCase for classes, 2 blank lines between top-level functions, proper import ordering (stdlib → third-party → local). It matters because consistent style improves readability and maintainability in teams. Tools like flake8, pylint, and black (auto-formatter) enforce PEP8 automatically.

Q23	What is memory management in Python?
ANS	Python uses automatic memory management via reference counting and a cyclic garbage collector. Every object has a reference count; when it hits zero, memory is freed immediately. For circular references (A → B → A), the garbage collector (gc module) detects and frees them. Python also uses memory pools (pymalloc) for small objects. In ML, explicit memory management matters — del large_tensor, torch.cuda.empty_cache() for GPU memory.

FASTAPI, FLASK & DJANGO

Q24 What is FastAPI?

ANS

FastAPI is a modern, high-performance Python web framework for building APIs, created by Sebastián Ramírez in 2018. It is built on top of Starlette (ASGI framework) and Pydantic (data validation). Key features: automatic OpenAPI/Swagger documentation, native async/await support, Python type hints for input validation, dependency injection, and extremely fast performance — comparable to Node.js and Go.

Q25 Why is FastAPI better than Flask and Django?

ANS

FastAPI vs Flask: Flask is synchronous by default (though you can add async). FastAPI is async-first, faster, and has auto data validation. Flask has no built-in API docs; FastAPI auto-generates OpenAPI/Swagger.

FastAPI vs Django: Django is a full-stack framework (ORM, admin, templates) — great for traditional web apps. FastAPI is API-focused and lightweight. FastAPI is ~3-10x faster in benchmarks. Choose FastAPI for microservices, AI/ML APIs, and high-performance REST APIs.

Q26 What is asynchronization and why should we use it?

ANS

Asynchronization (async programming) allows a program to start an operation and continue doing other work while waiting for that operation to complete, rather than blocking (waiting idle). In Python, this uses async/await keywords with asyncio. Critical for I/O-bound tasks: while waiting for a database query or HTTP response, the server can handle other requests. In AI APIs, async allows handling thousands of simultaneous LLM API calls efficiently. Sync code wastes CPU cycles waiting; async code uses that time productively.

Q27 What is the difference between async and sync in Python?

ANS

Synchronous (sync): Code executes line by line. Each line waits for the previous to complete. Simple but inefficient under load.

Asynchronous (async): Uses event loop. Operations that would block (I/O) are awaited — control returns to the event loop to process other tasks while waiting.

```
# Sync — blocks for 1 second:
```

import time; time.sleep(1)

Async — yields control:

import asyncio; await asyncio.sleep(1)

For ML APIs calling external LLMs, async can dramatically increase throughput.

Q28 What is asyncio in Python?

ANS

asyncio is Python's built-in library for writing concurrent code using async/await. It provides an event loop that manages and schedules coroutines. Key components: event loop (asyncio.run()), coroutines (async def), tasks (asyncio.create_task()), awaitables (coroutines,

tasks, futures), and utilities like asyncio.gather() for running multiple coroutines concurrently. FastAPI uses asyncio under the hood via Starlette and Uvicorn.

Q29 **What is Uvicorn and why is it used with FastAPI?**

ANS

Uvicorn is a lightning-fast ASGI (Asynchronous Server Gateway Interface) server for Python. FastAPI is an ASGI application, so it needs an ASGI server to run. Uvicorn handles incoming HTTP connections and passes them to FastAPI asynchronously. For production, Uvicorn is often run behind Nginx (reverse proxy) and managed by Gunicorn as a process manager.

```
uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4
```

Q30 **What is Pydantic and how does FastAPI use it for validation?**

ANS

Pydantic is a Python library for data validation using Python type annotations. You define a model class inheriting from BaseModel, and Pydantic automatically validates incoming data, coerces types, and raises detailed errors for invalid input. FastAPI uses Pydantic models for request body validation, response serialization, and query parameter validation. This eliminates the need for manual validation boilerplate.

```
from pydantic import BaseModel
```

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

Q31 **What is CORS and how do you configure it in FastAPI?**

ANS

CORS (Cross-Origin Resource Sharing) is a browser security mechanism that blocks web pages from making requests to a different domain than the one that served the page. You must configure CORS on the server to allow cross-origin requests.

```
from fastapi.middleware.cors import CORSMiddleware
```

```
app.add_middleware(CORSMiddleware,
    allow_origins=['https://yourfrontend.com'],
    allow_methods=['*'],
    allow_headers=['*'])
```

In development, allow_origins=['*'] is used, but never in production.

Q32 **What is middleware in FastAPI?**

ANS

Middleware is code that runs before every request reaches your route handler and/or after every response leaves it. Use cases: authentication token verification, CORS handling, logging, rate limiting, request timing, compression. In FastAPI, middleware is added using app.add_middleware() or @app.middleware('http') decorator. Execution order: middleware runs in reverse order of registration for responses (outermost first for requests).

Q33 **How do you handle authentication in FastAPI (JWT, OAuth2)?**

ANS

FastAPI has built-in OAuth2 support via fastapi.security. For JWT auth: 1) User logs in with credentials. 2) Server validates and returns a signed JWT token. 3) Client sends token in Authorization: Bearer <token> header. 4) Server verifies token on every request using a dependency.

```
from fastapi.security import OAuth2PasswordBearer
```

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl='token')
async def get_current_user(token: str = Depends(oauth2_scheme)):
    # verify JWT token here
    pass
```

Q34 **What is dependency injection in FastAPI?**

ANS

Dependency injection (DI) is a design pattern where components receive their dependencies from an external source rather than creating them. In FastAPI, the Depends() function declares dependencies that FastAPI automatically resolves and injects into route handlers. Common uses: database sessions, authentication, shared configuration, caching. DI promotes code reuse, testability, and separation of concerns.

```
from fastapi import Depends
```

```
def get_db(): yield SessionLocal()
@app.get('/items')
def read(db=Depends(get_db)): ...
```