

Python Interview Prep 10 LPA



consider all previous data and give python proper interview questions with proper answers deep explanation to crack 10 LPA

Great! Since you're targeting a 10 LPA Python developer role, you'll need strong command over Python fundamentals, OOPs, advanced topics (like decorators, generators, context managers), standard libraries, and backend frameworks (especially Django/Flask), along with problem-solving and system design basics.

Let's start with **Python Core Concepts – Set 1: 25 Deep Interview Questions with Answers & Explanations**. I'll continue in sets of 25 for easier study.

♦ Set 1: Core Python – 25 Questions

1. What are Python's key features that make it a preferred language?

✓ Answer:

- Easy to learn and read (clean syntax)
- Interpreted language (line-by-line execution)
- Dynamically typed (no need to declare variable types)
- Extensive standard library
- Supports multiple programming paradigms (OOP, procedural, functional)
- Large community support

Deep Explanation: Python's simplicity enables rapid development, especially useful in startups and scalable applications. Its dynamic nature allows faster prototyping but needs care for type-safety in large codebases.

2. Explain the difference between ``is`` and ``==`` operators.

✓ Answer:

- ``==`` compares **values**
- ``is`` compares **identity (memory address)**

```
python
```

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

```
a is b # False
```

Deep Explanation: Even though the lists look the same (``==`` is True), they're stored at different memory locations, so ``is`` is False. Use ``is`` only when checking against ``None`` or identity checks.

3. What is the difference between mutable and immutable data types in Python?

✓ **Answer:**

- **Mutable:** can be changed in place (e.g., list, dict, set)
- **Immutable:** cannot be changed (e.g., int, float, str, tuple)

Deep Explanation: Understanding mutability is crucial when working with function arguments or caching. Mutable types can lead to unintended side effects if not handled carefully.

4. What are Python's namespaces and scopes?

✓ **Answer:**

- **Namespace:** a container mapping names to objects
- **Scope:** the visibility/accessibility of variables

Python follows **LEGB** rule:

- Local
- Enclosing
- Global
- Built-in

Deep Explanation: When you reference a variable, Python searches through these scopes in order. Misunderstanding scopes can lead to bugs, especially with closures and lambdas.


5. What is a Python decorator?

✓ **Answer:** A decorator is a function that modifies the behavior of another function without changing its code.

```
python


def decorator(func):
    def wrapper():
        print("Before call")
        func()
        print("After call")
    return wrapper

@decorator
def say_hello():
    print("Hello")
```

 **Deep Explanation:** Useful for logging, access control, memoization. Understanding closures is key to writing decorators. Widely used in Django (e.g., `@login_required`).

6. Explain Python's garbage collection mechanism.


 **Answer:** Python uses **reference counting** and **cyclic garbage collection**.

 **Deep Explanation:** Every object has a reference count. When it drops to zero, it's deallocated. The garbage collector also removes circular references (e.g., in self-referencing objects).


7. What is the difference between `@staticmethod`, `@classmethod`, and a regular method?

 **Answer:**

Decorator	First Arg	Bound to	Use Case
Regular method	<code>self</code>	instance	Access/modify instance state
<code>@classmethod</code>	<code>cls</code>	class	Modify class state
<code>@staticmethod</code>	None	none	Utility functions


 **Deep Explanation:** Choose wisely based on whether the method needs access to class-level or instance-level data. Clean architecture principles recommend limiting unnecessary dependencies.

8. What is a generator? How is it different from a list comprehension?

 **Answer:** Generators use `yield` to produce values lazily, one at a time. They don't store the entire sequence in memory.

```
python


def gen():
    yield 1
    yield 2
```

 **Deep Explanation:** Useful for large data (e.g., reading big files, streaming data). Compared to list comprehensions, they save memory and are more performant in many use cases.

9. How does Python manage memory?

 **Answer:**


- **Private heap space** managed by Python's memory manager
- **Garbage collector** for reclaiming memory
- Built-in `gc` module for manual control

 **Deep Explanation:** Understanding memory management is essential for performance tuning and large-scale application optimization.

10. What are Python's data structures?

✓ Answer:

- **List:** ordered, mutable
- **Tuple:** ordered, immutable
- **Set:** unordered, unique
- **Dict:** key-value pairs

 **Deep Explanation:** Choose data structures based on time/space trade-offs. For example, `set` provides $O(1)$ lookup, ideal for membership testing.


**11. What are *args and kwargs?

✓ Answer:

- `*args`: variable-length positional arguments (tuple)
- `**kwargs`: variable-length keyword arguments (dict)

python

```
def func(*args, **kwargs):
    print(args, kwargs)
```

 **Deep Explanation:** Enables writing flexible APIs. Used extensively in decorators and Django view functions.

12. Difference between shallow copy and deep copy?

✓ Answer:

- **Shallow copy:** copies outer object, not nested objects
- **Deep copy:** copies all levels of nested objects

python


```
import copy
shallow = copy.copy(obj)
deep = copy.deepcopy(obj)
```

13. What is a Python lambda?

✓ Answer: A short anonymous function:

python

```
lambda x: x * 2
```


 **Deep Explanation:** Good for one-liners and used with `map()`, `filter()`, `reduce()`. Avoid for complex logic—hurts readability.

14. Explain Python's `with` statement.


 **Answer:** Used for context management (resource handling):

```
python

with open('file.txt') as f:
    data = f.read()
```


 **Deep Explanation:** Ensures proper acquisition and release of resources, even on exceptions. Implements `__enter__` and `__exit__`.

15. What is monkey patching in Python?


 **Answer:** Modifying or extending code at runtime.

```
python

import some_module
some_module.func = my_custom_func
```

 **Deep Explanation:** Powerful but dangerous. Can break encapsulation. Use with caution in testing/mockings.

16. Explain Python's GIL (Global Interpreter Lock).

 **Answer:** A mutex that allows only one thread to execute Python bytecode at a time.

 **Deep Explanation:** Limits multi-threading performance. For true parallelism, use multiprocessing.


17. Difference between `isinstance()` and `type()`?

 **Answer:**

- `type(obj)` returns exact type
 - `isinstance(obj, Class)` supports inheritance check
-

18. What is duck typing in Python?

 **Answer:** "If it looks like a duck and quacks like a duck, it's a duck."

 **Deep Explanation:** Focus on behavior, not type. Encourages flexible code but requires strong tests.

19. What are metaclasses?

✓ **Answer:** Classes that create classes. Default metaclass: ``type``

📘 **Deep Explanation:** Used for class customization, auto-registering models in Django, enforcing rules at class level.

20. Difference between ``del``, ``remove()``, ``pop()``?

✓ **Answer:**

- ``del``: deletes by index or variable
 - ``remove()``: removes by value
 - ``pop()``: removes by index and returns item
-

21. How to handle exceptions in Python?

✓ **Answer:**

```
python

`try:
    risky_code()
except ValueError as e:
    print(e)
finally:
    cleanup()
`
```

📘 **Deep Explanation:** Exception handling prevents crashing and ensures cleanup. Always use specific exceptions, not bare ``except``.

22. What are magic methods in Python?

✓ **Answer:** Methods with double underscores (e.g., ``__init__``, ``__str__``, ``__len__``) used to customize object behavior.

23. What's the difference between ``str()`` and ``repr()``?

✓ **Answer:**

- ``str()``: readable string for end-users
 - ``repr()``: developer-focused, unambiguous
-

24. What is the ``__slots__`` mechanism?

✓ **Answer:** Restricts dynamic attribute creation, saves memory:

```
python

`class MyClass:
    __slots__ = ['name']
```

25. What is the difference between `==`, `is`, `in`, `not in`?

✓ Answer:

- `==`: value equality
- `is`: identity
- `in`: membership
- `not in`: inverse membership

Would you like me to continue with Set 2 (Advanced Python: List Comprehensions, Decorators, Iterators, etc.), or jump into Django, SQL, or Data Structures next?



Set 2

Perfect! Here's your 🔥 **Set 2: Advanced Python Interview Questions (25 questions with answers and deep explanations)**—specially curated for cracking 10+ LPA Python roles.

♦ **Set 2: Advanced Python (List Comprehensions, Iterators, Decorators, Generators, etc.)**

1. What is the difference between a list comprehension and a generator expression?

✓ Answer:

```
python

`# List comprehension
lst = [x*x for x in range(5)]

# Generator expression
gen = (x*x for x in range(5))
```

📘 Deep Explanation:

- **List comprehension:** returns a full list in memory.
- **Generator expression:** returns an iterator, computes on-the-fly.
- Use **generators** for large datasets to save memory.

2. What is the difference between `yield` and `return`?

✓ Answer:

- ``return`` terminates a function and sends back a value.
- ``yield`` pauses function execution and returns a **generator object**.

python

```
`def count_up():
    yield 1
    yield 2
`
```


 **Deep Explanation:** ``yield`` is used for **lazy evaluation**, perfect for pipelines, file streaming, and infinite sequences.

3. What is an iterator in Python?

 **Answer:** An object with ``__iter__()`` and ``__next__()`` methods.

python

```
`it = iter([1, 2, 3])
print(next(it)) # 1
`
```


 **Deep Explanation:** Iterators are the foundation of Python's for-loops, generators, and comprehensions. Once exhausted, they can't be reused.

4. What is a Python context manager?


 **Answer:** An object that controls a runtime context using ``__enter__()`` and ``__exit__()`` methods.

python

```
`with open('file.txt') as f:
    pass
`
```

 **Deep Explanation:** Ensures proper acquisition/release of resources (files, DB connections). Custom context managers can be built using ``contextlib``.

5. What is a closure in Python?

 **Answer:** A function object that remembers variables from its enclosing scope even if that scope is gone.


python

```
`def outer(x):
    def inner(y):
        return x + y
    return inner

add_5 = outer(5)
`
```




```
print(add_5(3)) # 8
```

 **Deep Explanation:** Closures are used to maintain state across function calls without using global variables or classes.

6. How is a generator different from an iterator?

 **Answer:**

- Generators **automatically** create iterators using ``yield``.
- Iterators must manually implement ``__iter__()`` and ``__next__()``.

 **Deep Explanation:** Generators simplify the creation of iterators and are preferred in real-time streaming or lazy evaluation.

7. What is the difference between ``map()``, ``filter()``, and ``reduce()``?

 **Answer:**

```
python
```

```
`from functools import reduce

map_result = list(map(lambda x: x*2, [1, 2, 3]))
filter_result = list(filter(lambda x: x % 2 == 0, [1, 2, 3]))
reduce_result = reduce(lambda x, y: x + y, [1, 2, 3])
```

 **Deep Explanation:**

- ``map()``: transforms elements
- ``filter()``: selects elements
- ``reduce()``: aggregates elements

Functional tools that support cleaner data pipelines.

8. What is a decorator with arguments?


 **Answer:**

```
python
```

```
`def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
            return wrapper
        return decorator

@repeat(3)
def greet():
```

```
print("Hi")
```

 **Deep Explanation:** Decorators with arguments use **3 levels** of nested functions. Critical for writing customizable behaviors (e.g., retries, logging, access control).

9. What is the difference between `any()` and `all()`?

 **Answer:**

- `any()`: returns True if **any** element is true.
- `all()`: returns True if **all** elements are true.

python

```
any([False, True, False]) # True
all([True, True, True])   # True
```

10. What is the `zip()` function in Python?

 **Answer:** Combines multiple iterables into tuples:

python

```
zip([1, 2], ['a', 'b']) # [(1, 'a'), (2, 'b')]
```

 **Deep Explanation:** Great for iterating in parallel. Can be unzipped using `zip(*zipped_list)`.

11. What are the different ways to reverse a list in Python?

 **Answer:**


- `lst[::-1]`
- `reversed(lst)`
- `lst.reverse()` (in-place)

12. What is the purpose of `enumerate()`?

 **Answer:**

python

```
for index, value in enumerate(['a', 'b', 'c']):
    print(index, value)
```

 **Deep Explanation:** Provides clean looping with index and value without manually maintaining counters.

13. What's the difference between deep and shallow copy for nested structures?

✓ **Answer:**

- **Shallow:** references nested objects
- **Deep:** recursively copies all levels

python


```
`import copy
shallow = copy.copy(nested)
deep = copy.deepcopy(nested)
```

14. What is list unpacking?

✓ **Answer:**

python

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


 **Deep Explanation:** Useful in function arguments and return values. Enhances readability and flexibility.

15. How can you flatten a nested list in Python?

✓ **Answer:**

python

```
`nested = [[1, 2], [3, 4]]
flat = [x for sublist in nested for x in sublist]
```

 **Deep Explanation:** Nested comprehensions are powerful but can be tricky. Consider recursion or `itertools.chain()` for deeper nesting.

16. What are Python's built-in functions for functional programming?


✓ **Answer:**

- ``map()`, `filter()`, `reduce()`, `zip()`, `sorted()`, `enumerate()```
- ``functools.partial`, `operator`` module functions

17. What are comprehensions and which types exist?

✓ Answer:

- List `[x for x in iterable]`
- Dict `{k: v for k, v in iterable}`
- Set `{x for x in iterable}`
- Generator `(x for x in iterable)`

 **Deep Explanation:** Used for concise and readable transformations. Dict and set comprehensions are less known but powerful.

18. How do you sort a list of tuples by the second value?

✓ Answer:

```
python

data = [(1, 3), (2, 2), (3, 1)]
sorted(data, key=lambda x: x[1])
```

19. Explain the `collections` module.

✓ Answer: Provides high-performance alternatives to built-ins:

- `Counter`: count items
 - `defaultdict`: auto-creates missing keys
 - `deque`: fast append/pop from both ends
 - `namedtuple`: lightweight objects
-

20. What is a `namedtuple`?

✓ Answer:

```
python

from collections import namedtuple
Point = namedtuple('Point', 'x y')
p = Point(1, 2)
```

 **Deep Explanation:** Provides immutability and dot-access like a class, but with less overhead.

21. What is a `defaultdict`?

✓ Answer:

```
python
```

```
`from collections import defaultdict
d = defaultdict(int)
d['a'] += 1
`
```

 **Deep Explanation:** Prevents `KeyError`. Useful in frequency counts, grouping items.

22. How does slicing work in Python?

 **Answer:** Syntax: `sequence[start:stop:step]`

```
python
```

```
`s = 'python'
s[1:4] # 'yth'
s[::-1] # reversed
`
```


23. What is unpacking using `*` and `**` in functions?

 **Answer:**

```
python
```

```
`def add(a, b, c): ...
args = (1, 2, 3)
add(*args)

def show(**kwargs): ...
show(name='John')
`
```

 **Deep Explanation:** `*` unpacks sequences; `**` unpacks dicts. Vital for decorators and flexible APIs.

24. How do you merge two dictionaries in Python?

 **Answer:**

```
python
```

```
`# Python 3.9+
merged = dict1 | dict2

# Old way
merged = {**dict1, **dict2}
`
```


25. What is `functools.lru_cache()` used for?

 **Answer:**

```
python

`from functools import lru_cache

@lru_cache(maxsize=100)
def fib(n):
    ...
```

 **Deep Explanation:** Caches results of expensive function calls. Reduces recomputation, especially in recursion-heavy algorithms.

Would you like **Set 3: Object-Oriented Python** next, or jump to Django, SQL, or Data Structures?



Set 3

Awesome! Let's dive into 🔥 **Set 3: Object-Oriented Programming (OOP) in Python** — a crucial topic for high-paying Python roles (10+ LPA). These questions go deep into OOP design, principles, dunder methods, and interview-level scenarios.

♦ Set 3: Object-Oriented Python (25 Questions + Deep Explanations)

1. What is Object-Oriented Programming (OOP) in Python?

✅ **Answer:** OOP is a paradigm where everything is an object. It focuses on:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Python uses classes and objects to implement OOP.

2. What is a class and an object in Python?

✅ **Answer:**

```
python

`class Dog:
    def bark(self):
        print("Woof!")

d = Dog() # Object
d.bark()
```

 **Deep Explanation:**

- **Class:** blueprint for objects.
- **Object:** instance of a class. Contains its own data (state) and methods (behavior).

3. What is the difference between `__init__` and `__new__`?

✓ **Answer:**

- `__new__()` creates the instance.
- `__init__()` initializes it.

python

```
`class A:
    def __new__(cls):
        print("Creating instance")
        return super().__new__(cls)

    def __init__(self):
        print("Initializing instance")
`
```

📘 **Deep Explanation:** `__new__()` is rarely overridden unless creating immutable types (like customizing `int`, `tuple`, etc).

4. What is `self` in Python classes?

✓ **Answer:** `self` refers to the **current object instance** of the class.

📘 **Deep Explanation:** It is used to access instance variables and methods. Not a keyword—just a naming convention.

5. What is inheritance? Types?

✓ **Answer:** Inheritance lets a class (child) reuse features of another class (parent).

Types:

- Single
- Multi-level
- Multiple
- Hierarchical
- Hybrid

python

```
`class Animal:
    def speak(self): print("Sound")

class Dog(Animal):
    def bark(self): print("Woof")
`
```


6. What is method overriding?

✓ **Answer:** Redefining a parent method in the child class.

```
python

`class Parent:
    def greet(self):
        print("Hello")

class Child(Parent):
    def greet(self):
        print("Hi")
`
```

 **Deep Explanation:** Polymorphism in action — behavior is redefined for specialized use cases.

7. What is method overloading in Python?

✓ **Answer:** Python **does not support true method overloading**. Latest definition replaces the old.

But you can emulate it with default or variable arguments:

```
python

`def greet(name=None):
    if name:
        print("Hi", name)
    else:
        print("Hi")
`
```

8. What is encapsulation?

✓ **Answer:** Encapsulation hides internal state using access modifiers:

- `_protected` (convention)
- `__private` (name mangling)


9. What is abstraction?

✓ **Answer:** Abstraction hides complexity using abstract classes or interfaces.

```
python

`from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
`
```

 **Deep Explanation:** Prevents instantiation and enforces method implementation in child classes.

10. What is multiple inheritance?

✓ Answer:

```
python

`class A:
    def greet(self): print("Hello from A")

class B:
    def greet(self): print("Hello from B")

class C(A, B):
    pass
`
```

 **Deep Explanation:** Python uses **MRO (Method Resolution Order)** to resolve ambiguity (`C.__mro__`).


11. What is `super()` in Python?

✓ Answer:

```
python

`class Parent:
    def greet(self): print("Hi")

class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello")
`
```

 **Deep Explanation:** Calls the parent method. Useful in cooperative multiple inheritance to maintain MRO consistency.


12. What is duck typing in Python?

✓ Answer: "If it walks like a duck and quacks like a duck, it's a duck."

```
python

`def quack(thing):
    thing.quack()

class Duck:
    def quack(self): print("Quack!")
`
```


 **Deep Explanation:** Emphasizes behavior, not inheritance—**dynamic typing** at its core.

13. What is a static method?

✓ Answer:

```
python
```

```
`class Math:
    @staticmethod
    def add(x, y): return x + y
```

 **Deep Explanation:** Doesn't take `self` or `cls`. Like regular functions placed inside a class for grouping.

14. What is a class method?

 **Answer:**

```
python
```

```
`class MyClass:
    count = 0

    @classmethod
    def increment(cls):
        cls.count += 1
```

 **Deep Explanation:** Operates on the class, not instances. Useful for factory methods or shared state.

15. Difference between `@staticmethod` and `@classmethod`?

 **Answer:**


Aspect	`@staticmethod`	`@classmethod`
First arg	None	`cls`
Access class?	No	Yes
Use-case	Utility logic	Factory patterns

16. What are dunder methods (magic methods)?

 **Answer:** Special methods surrounded by `__`.

Examples:

- `__init__` — constructor
- `__str__` — string representation
- `__len__`, `__getitem__`, `__add__`

 **Deep Explanation:** They allow classes to emulate built-in types (operator overloading).

17. What is the difference between `is` and `==`?

 **Answer:**

- `==`: checks **value**
- `is`: checks **identity** (same memory)

18. How do you make a class iterable?

✓ **Answer:** Implement `__iter__()` and `__next__()`.

python

```
`class Counter:
    def __init__(self): self.i = 0
    def __iter__(self): return self
    def __next__(self):
        self.i += 1
        if self.i <= 5:
            return self.i
        raise StopIteration
`
```

19. What is composition vs inheritance?

✓ **Answer:**

- **Inheritance:** "is a" relationship
- **Composition:** "has a" relationship

python

```
`class Engine: ...
class Car:
    def __init__(self):
        self.engine = Engine() # composition
`
```

20. What is polymorphism in Python?

✓ **Answer:** Same interface, different behavior.

python

```
`class Cat:
    def speak(self): return "Meow"

class Dog:
    def speak(self): return "Woof"

for animal in (Cat(), Dog()):
    print(animal.speak())
`
```

21. Can Python do operator overloading?

✓ **Answer:** Yes, using dunder methods like `__add__`, `__mul__`, etc.

python

```
`class Vector:
    def __init__(self, x): self.x = x
    def __add__(self, other): return Vector(self.x + other.x)
```

22. How to prevent a class from being inherited?

✓ **Answer:** Use `__init_subclass__` or raise exception in subclassing:

python

```
`class Base:
    def __init_subclass__(cls):
        raise TypeError("No subclassing allowed")
```

23. What is the difference between class variable and instance variable?

✓ **Answer:**


- **Class variable:** shared by all instances.
- **Instance variable:** unique to each object.

24. Can you dynamically add methods to a class?

✓ **Answer:**

python

```
`def greet(self): print("Hello")
MyClass.greet = greet
```


 **Deep Explanation:** Python's dynamic nature allows monkey patching, useful in testing/mockings.

25. What's the role of `__slots__`?

✓ **Answer:**

python


```
`class Point:
    __slots__ = ['x', 'y']
```

 **Deep Explanation:** Limits instance attribute creation, **reduces memory usage**, and boosts speed. Useful in high-performance classes.

Would you like **Set 4: SQL, Django**, or move to **Data Structures and Algorithms** next?



Set 4

Great! Here's  **Set 4: SQL Interview Questions with Deep Explanations** — perfect for 10 LPA Python full-stack or backend roles. SQL is **crucial** for database design, performance, and backend integration.

♦ Set 4: SQL Interview Questions (25 Questions + Deep Explanations)

1. What is SQL?

✅ **Answer:** SQL (Structured Query Language) is used to manage relational databases — for creating, reading, updating, and deleting (CRUD) data.

2. What are the different types of SQL commands?

✅ **Answer:**

- **DDL** – Data Definition Language (e.g., `CREATE`, `ALTER`)
- **DML** – Data Manipulation Language (e.g., `INSERT`, `UPDATE`)
- **DQL** – Data Query Language (`SELECT`)
- **DCL** – Data Control Language (`GRANT`, `REVOKE`)
- **TCL** – Transaction Control Language (`COMMIT`, `ROLLBACK`)

3. Difference between `WHERE` and `HAVING`?

✅ **Answer:**

Clause	Filters on	Used With
<code>WHERE</code>	Rows (before group)	Any query
<code>HAVING</code>	Groups (after agg)	Group By

sql

```
`SELECT` dept, COUNT(*) FROM employees
GROUP BY dept
HAVING COUNT(*) > 5;
```

4. What is a primary key?

✅ **Answer:**

- Uniquely identifies each row.
- Cannot be `NULL`.
- One per table.

sql

```
`CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100)
);
```

5. What is a foreign key?

✓ **Answer:** Links one table to another. Enforces **referential integrity**.

sql

```
`CREATE TABLE orders (
  id INT PRIMARY KEY,
  user_id INT,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

6. What is the difference between `INNER JOIN` and `LEFT JOIN`?

✓ **Answer:**

JOIN	Returns
`INNER`	Matching rows in both tables
`LEFT`	All from left, matched from right

7. What is normalization?

✓ **Answer:** Process of organizing data to:

- Reduce redundancy
- Improve integrity

Forms:

- 1NF: Atomic values
- 2NF: Full functional dependency
- 3NF: No transitive dependency

8. Write a query to fetch 2nd highest salary.

✓ **Answer:**

```
sql
```

```
`SELECT MAX(salary) FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);`
```

9. Difference between `UNION` and `UNION ALL`?

✓ **Answer:**

- `UNION` removes duplicates
- `UNION ALL` keeps all rows

```
sql
```

```
`SELECT name FROM A
UNION
SELECT name FROM B;`
```

10. What are indexes in SQL?

✓ **Answer:** Indexes speed up data retrieval by creating lookup structures.

```
sql
```

```
`CREATE INDEX idx_name ON employees(name);`
```

⚠ **Trade-off:** Speeds up read, slows down write operations.

11. What is a subquery?

✓ **Answer:** A query inside another query.

```
sql
```

```
`SELECT name FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);`
```

12. What is a correlated subquery?

✓ **Answer:** Depends on the outer query.

```
sql
```

```
`SELECT e1.name FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.dept = e2.dept);`
```

13. What is the difference between `DELETE`, `TRUNCATE`, and `DROP`?

✓ Answer:

Command	Deletes Data?	Rollback?	Removes Structure?
`DELETE`	Yes (row-wise)	Yes	No
`TRUNCATE`	Yes (all rows)	No	No
`DROP`	Yes (table)	No	Yes

14. What is a view?

✓ Answer: A virtual table based on SQL query.

```
sql

`CREATE VIEW active_users AS
SELECT * FROM users WHERE status = 'active';
`
```

15. How do you find duplicate records in a table?

✓ Answer:

```
sql

`SELECT name, COUNT(*)
FROM users
GROUP BY name
HAVING COUNT(*) > 1;
`
```

16. Explain ACID properties.

✓ Answer:

- **Atomicity** – All or nothing
- **Consistency** – Maintains integrity
- **Isolation** – Transactions don't interfere
- **Durability** – Changes persist even after crash

17. What are stored procedures?

✓ Answer: Precompiled SQL code for reuse.

```
sql

`CREATE PROCEDURE GetUsers()
BEGIN
SELECT * FROM users;
`
```



```
END;
```

18. What is a transaction?

✓ **Answer:** Logical unit of work with `BEGIN`, `COMMIT`, and `ROLLBACK`.

```
sql

`BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id=1;
UPDATE accounts SET balance = balance + 100 WHERE id=2;
COMMIT;
```

19. What are aggregate functions in SQL?

✓ **Answer:**

Functions like:

- `COUNT()`
- `SUM()`
- `AVG()`
- `MAX()`
- `MIN()`

Used with `GROUP BY`.

20. What does `GROUP BY` do?

✓ **Answer:** Aggregates rows with the same values.

```
sql

`SELECT dept, COUNT(*) FROM employees
GROUP BY dept;
```

21. How to use `CASE` in SQL?

✓ **Answer:**

```
sql

`SELECT name,
CASE
  WHEN score >= 90 THEN 'A'
  WHEN score >= 75 THEN 'B'
  ELSE 'C'
END AS grade
```

```
FROM students;
```

22. What is a CTE (Common Table Expression)?

✓ Answer:

```
sql

WITH TopSales AS (
  SELECT * FROM sales WHERE amount > 1000
)
SELECT * FROM TopSales;
```

Temporary result set for readability and reuse.

23. What is a window function?

✓ Answer:

```
sql

SELECT name, salary,
RANK() OVER (PARTITION BY dept ORDER BY salary DESC)
FROM employees;
```

Provides row-level ranking/aggregation **without grouping rows**.

24. What is the difference between `RANK()`, `DENSE_RANK()` and `ROW_NUMBER()`?

✓ Answer:

Function	Gaps?	Unique?
`RANK()`	Yes	No
`DENSE_RANK()`	No	No
`ROW_NUMBER()`	No	Yes

25. Write a query to get employees with top 3 salaries per department.

✓ Answer:

```
sql

SELECT * FROM (
  SELECT name, dept,
  DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rank
  FROM employees
) ranked
WHERE rank <= 3;
```

Want to continue to **Set 5: Django Interview Questions** or switch to **Data Structures & Algorithms** next?



Set 5

Perfect! Here's 🔥 **Set 5: Django Interview Questions** — crafted for high-paying **Python Full Stack / Backend** roles, especially at the 10 LPA level. This set focuses on Django core concepts, project structure, ORM, middleware, views, and more — all with **deep explanations**.

♦ Set 5: Django Interview Questions (25 Questions + Deep Explanations)

1. What is Django?

✓ **Answer:** Django is a **high-level Python web framework** that enables rapid development and clean, pragmatic design. It follows the **MTV architecture** (Model–Template–View), which is similar to MVC.

2. Explain the MTV architecture.

✓ **Answer:**

Component	Role
Model	Handles database (ORM)
Template	HTML + logic (frontend)
View	Business logic (controller-like)

3. What is Django ORM?

✓ **Answer:** Django ORM (Object-Relational Mapping) allows developers to interact with the database using Python code instead of SQL.

```
python
User.objects.filter(is_active=True)
```

📘 **Deep Explanation:** ORM translates this into SQL queries under the hood and supports multiple databases (PostgreSQL, MySQL, SQLite, etc.).

4. What are Django models?

✓ **Answer:** Python classes that define the structure of your database tables.


```
python
```

```
`class Employee(models.Model):
    name = models.CharField(max_length=100)
    salary = models.DecimalField(max_digits=10, decimal_places=2)
```

5. How to perform migrations in Django?

✓ **Answer:**

- ``python manage.py makemigrations``: create migration file
- ``python manage.py migrate``: apply to DB

 Ensures the database schema reflects your model definitions.

6. What are Django views? Types?

✓ **Answer:** Views control the logic and return HTTP responses.

Types:

- **Function-based views (FBV)**
- **Class-based views (CBV)**

```
python
```

```
`def hello(request):
    return HttpResponse("Hello World")
```

7. What are templates in Django?

✓ **Answer:** Templates define how data is displayed in HTML. They support:

- Filters
- Tags (``{% if %}``, ``{% for %}``)
- Template inheritance

```
html
```

```
`<h1>Welcome, {{ user.username }}</h1>
```

8. What is the purpose of ``settings.py``?

✓ **Answer:** Contains global configurations:

- Installed apps

- Database settings
- Middleware
- Static & media file paths
- Authentication settings

9. What is the purpose of `urls.py`?

✓ **Answer:** Handles routing in Django. Maps URLs to views.

```
python

urlpatterns = [
    path('home/', views.home),
]
```

10. Difference between `project` and `app` in Django?

✓ **Answer:**

- **Project:** The entire Django site (root configuration)
- **App:** A module for a specific functionality (like blog, auth)

 One project can contain many apps.

11. What is `manage.py`?

✓ **Answer:** A CLI utility to interact with your Django project.

Examples:

- `runserver`
- `makemigrations`
- `createsuperuser`

12. What are middlewares in Django?

✓ **Answer:** Middleware is a function that runs before/after views. It handles:

- Authentication
- Sessions
- Security (CSRF)
- Custom request/response processing

13. What are static files in Django?

✓ **Answer:** CSS, JavaScript, and image files.

Configured using:

```
python

STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Use `{% load static %}` in templates.

14. How are sessions handled in Django?

✓ **Answer:** By default, sessions are stored in the database. Django assigns each user a unique session ID.

```
python

request.session['username'] = 'john'
```

 Can also use cache or signed cookies.

15. What is CSRF and how does Django handle it?

✓ **Answer: Cross-Site Request Forgery** is an attack where unauthorized commands are sent from a trusted user.

Django uses a CSRF token:

```
html

{% csrf_token %}
```

Enabled via middleware.

16. How do you create a custom model manager in Django?

✓ **Answer:**

```
python

class ActiveManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(is_active=True)

class Employee(models.Model):
    name = models.CharField(max_length=100)
    is_active = models.BooleanField(default=True)

    objects = models.Manager()           # Default
    active = ActiveManager()             # Custom
```

17. How do you implement authentication in Django?

✓ **Answer:** Use `django.contrib.auth`:

```
python

`from django.contrib.auth import authenticate, login

user = authenticate(request, username='john', password='pass')
if user:
    login(request, user)
```

 Supports login, logout, password hashing, permissions, and groups.

18. How do you create an admin user?

✓ **Answer:**

```
bash

`python manage.py createsuperuser
```

Then access via `/admin/`

19. How can you extend the default `User` model?

✓ **Answer:** Option 1: **OneToOne extension**

```
python

`class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField()
`
```

Option 2: **Custom user model (recommended)**

```
python

`class CustomUser(AbstractUser):
    phone = models.CharField(max_length=15)
`
```

And in `settings.py`:

```
python

`AUTH_USER_MODEL = 'yourapp.CustomUser'
```

20. What is Django Rest Framework (DRF)?

✓ **Answer:** A powerful toolkit for building Web APIs in Django.

- Serializers
- ViewSets
- Routers
- Token/Auth handling

21. What are serializers in DRF?

✓ **Answer:** Convert complex data (e.g., models) to JSON and vice versa.

```
python

`class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = '__all__'
```

22. Difference between `ModelForm` and `Form` in Django?

✓ **Answer:**

Form	Usage
`Form`	Pure HTML form
`ModelForm`	Auto form linked to a model

```
python

`class MyForm(forms.Form):
    name = forms.CharField()

`class MyModelForm(forms.ModelForm):
    class Meta:
        model = MyModel
```

23. How do you connect Django with PostgreSQL?

✓ **Answer:** Install package:

```
bash

`pip install psycopg2
```

In `settings.py`:

```
python

`DATABASES = {
    'default': {
```



```
'ENGINE': 'django.db.backends.postgresql',
'NAME': 'dbname',
'USER': 'dbuser',
'PASSWORD': 'pass',
'HOST': 'localhost',
'PORT': '5432',
}
}
```

24. What is `prefetch_related` vs `select_related`?

✓ Answer:

Method	Use-case	Query Type
`select_related`	ForeignKey, OneToOne	JOIN query
`prefetch_related`	ManyToMany, reverse FK	2 separate queries

25. How to deploy a Django app to production?

✓ Answer: Steps:

- Use `gunicorn` + `nginx`
- Set `DEBUG=False`
- Set allowed hosts
- Configure static/media files
- Use PostgreSQL instead of SQLite
- Secure settings (e.g., secret keys, HTTPS)

🔥 That wraps up Set 5! You now have:

- ✓ Python Core (Set 1–3)
- ✓ SQL (Set 4)
- ✓ Django (Set 5)

Ready for **Set 6: Data Structures and Algorithms (DSA) – Python Coding Level** next?



Set 6

Absolutely! Here's your 🔥 **Set 6: Data Structures and Algorithms (DSA) in Python – Deep Interview Questions with Code & Explanations** — specifically designed to **crack 10 LPA+ Python Full Stack / Backend roles**.

These questions test logic, performance thinking, space-time complexity awareness, and hands-on coding skills.

◆ Set 6: DSA in Python – 25 Interview Questions with Answers & Deep Explanations

1. Reverse a Linked List (Iterative and Recursive)

✓ Iterative:

```
python

def reverse_list(head):
    prev = None
    while head:
        nxt = head.next
        head.next = prev
        prev = head
        head = nxt
    return prev
```

✓ Recursive:

```
python

def reverse_list_recursive(head):
    if not head or not head.next:
        return head
    rest = reverse_list_recursive(head.next)
    head.next.next = head
    head.next = None
    return rest
```

 Asked to check recursion, pointers & logic handling

2. Detect a cycle in a linked list

✓ Answer (Floyd's Tortoise and Hare):

```
python

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    if slow == fast:
        return True
    return False
```

 Checks algorithm understanding, pointer manipulation

3. Merge two sorted linked lists


python

```
`def merge_sorted(l1, l2):
    dummy = curr = ListNode()
    while l1 and l2:
        if l1.val < l2.val:
            curr.next, l1 = l1, l1.next
        else:
            curr.next, l2 = l2, l2.next
        curr = curr.next
    curr.next = l1 or l2
    return dummy.next`
```

4. Find Kth largest element in an array

python

```
`import heapq
def kth_largest(nums, k):
    return heapq.nlargest(k, nums)[-1]`
```

 Uses heap — fast and elegant

5. Validate Balanced Parentheses

python

```
`def is_valid(s):
    stack = []
    mapping = {'(': ')', '[': ']', '{': '}'
    for char in s:
        if char in mapping:
            if not stack or stack.pop() != mapping[char]:
                return False
        else:
            stack.append(char)
    return not stack`
```

6. Implement a Queue using 2 Stacks

python

```
`class MyQueue:
    def __init__(self):
        self.inbox, self.outbox = [], []


    def enqueue(self, x):
        self.inbox.append(x)

    def dequeue(self):
        if not self.outbox:
            while self.inbox:
                self.outbox.append(self.inbox.pop())
        return self.outbox.pop()`
```

7. Find the majority element (appears $> n/2$ times)

python

```
def majority(nums):
    count, candidate = 0, None
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
    return candidate
```

 Boyer-Moore Voting Algorithm – Optimal!

8. Find the first non-repeating character in a string

python

```
from collections import Counter
def first_unique(s):
    count = Counter(s)
    for i, c in enumerate(s):
        if count[c] == 1:
            return i
    return -1
```

9. Merge Intervals

python

```
def merge(intervals):
    intervals.sort(key=lambda x: x[0])
    res = [intervals[0]]
    for start, end in intervals[1:]:
        last = res[-1]
        if start <= last[1]:
            last[1] = max(last[1], end)
        else:
            res.append([start, end])
    return res
```

10. Two Sum

python

```
def two_sum(nums, target):
    hashmap = {}
    for i, n in enumerate(nums):
        diff = target - n
        if diff in hashmap:
            return [hashmap[diff], i]
        hashmap[n] = i
```

11. Longest Substring Without Repeating Characters

python

```
`def length_of_longest_substring(s):
    seen = {}
    l = res = 0
    for r, ch in enumerate(s):
        if ch in seen and seen[ch] >= l:
            l = seen[ch] + 1
        seen[ch] = r
        res = max(res, r - l + 1)
    return res`
```

12. Binary Search

python

```
`def binary_search(arr, target):
    l, r = 0, len(arr)-1
    while l <= r:
        m = (l + r) // 2
        if arr[m] == target:
            return m
        elif arr[m] < target:
            l = m + 1
        else:
            r = m - 1
    return -1`
```

13. Find the missing number in [0, n]

python

```
`def missing_number(nums):
    n = len(nums)
    return n * (n + 1) // 2 - sum(nums)`
```

14. Invert a Binary Tree

python

```
`def invert_tree(root):
    if root:
        root.left, root.right = invert_tree(root.right), invert_tree(root.left)
    return root`
```

15. Level Order Traversal of a Binary Tree

python

```
`from collections import deque
def level_order(root):
    if not root: return []
    q, res = deque([root]), []
    while q:
        level = []
        for _ in range(len(q)):
            node = q.popleft()
            level.append(node.val)
            if node.left: q.append(node.left)
            if node.right: q.append(node.right)
        res.append(level)
    return res`
```

16. Detect Palindrome

python

```
`def is_palindrome(s):
    s = ''.join(c for c in s.lower() if c.isalnum())
    return s == s[::-1]`
```

17. Rotate Array by K Steps

python

```
`def rotate(nums, k):
    k %= len(nums)
    nums[:] = nums[-k:] + nums[:-k]`
```

18. Merge Two Sorted Arrays (in-place)

python

```
`def merge(nums1, m, nums2, n):
    while m > 0 and n > 0:
        if nums1[m-1] > nums2[n-1]:
            nums1[m+n-1] = nums1[m-1]
            m -= 1
        else:
            nums1[m+n-1] = nums2[n-1]
            n -= 1
    nums1[:n] = nums2[:n]`
```

19. Max Subarray Sum (Kadane's Algo)

python

```
`def max_subarray(nums):
    max_sum = curr = nums[0]`
```

```
for n in nums[1:]:
    curr = max(n, curr + n)
    max_sum = max(max_sum, curr)
return max_sum
```

20. Detect Anagram

python

```
def is_anagram(s, t):
    return sorted(s) == sorted(t)
```

21. Implement Stack with min() in O(1)

python

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, x):
        self.stack.append(x)
        m = x if not self.min_stack else min(x, self.min_stack[-1])
        self.min_stack.append(m)

    def pop(self):
        self.stack.pop()
        self.min_stack.pop()

    def get_min(self):
        return self.min_stack[-1]
```

22. Top K Frequent Elements

python

```
from collections import Counter
import heapq

def top_k(nums, k):
    count = Counter(nums)
    return [item for item, freq in count.most_common(k)]
```

23. Product of Array Except Self

python

```
def product_except_self(nums):
    res = [1]*len(nums)
    left = right = 1
    for i in range(len(nums)):
        res[i] *= left
```

```

    left *= nums[i]
    for i in reversed(range(len(nums))):
        res[i] *= right
        right *= nums[i]
    return res

```

24. Spiral Matrix Traversal

python

```

def spiral_order(matrix):
    res = []
    while matrix:
        res += matrix.pop(0)
        matrix = list(zip(*matrix))[::-1]
    return res

```

25. Binary Tree - Check if Symmetric

python

```

def is_symmetric(root):
    def is_mirror(t1, t2):
        if not t1 and not t2: return True
        if not t1 or not t2: return False
        return (t1.val == t2.val and
                is_mirror(t1.left, t2.right) and
                is_mirror(t1.right, t2.left))
    return is_mirror(root, root)

```

✓ You now have:

- Python Core
- SQL
- Django
- DSA Coding Level

Want to go for **Set 7: Advanced Python Libraries (NumPy, Pandas, Matplotlib)** or continue with **Django Advanced Concepts / APIs / System Design**?



Set 7

Awesome! Here's your 🔥 **Set 7: Advanced Python Libraries – 100 Coding Interview Questions with Answers & Deep Explanations** — tailored for **10 LPA+ Data, Backend, and Full Stack** roles.

We'll break it down by library:

- NumPy – 33 Qs

- **Pandas** – 34 Qs
- **Matplotlib (with Seaborn touch)** – 33 Qs

Set 7A: NumPy Interview Questions (33 Qs)

1. How do you create a 3x3 identity matrix in NumPy?

```
python
`import numpy as np
np.identity(3)
`
```

2. Create a 3x3 array with random values

```
python
`np.random.rand(3, 3)
`
```

3. Convert a list to a NumPy array

```
python
`arr = np.array([1, 2, 3])
`
```

4. How to create a NumPy array of all zeros or ones?

```
python
`np.zeros((2, 2))
np.ones((2, 2))
`
```

5. Reshape a 1D array into 2D

```
python
`a = np.arange(9)
a.reshape(3, 3)
`
```

6. Flatten a 2D array

```
python
```

```
`arr.flatten()
```

7. Find max, min, mean, and std of array

```
python
```

```
`np.max(arr), np.min(arr), np.mean(arr), np.std(arr)
```

8. Element-wise addition and multiplication

```
python
```

```
`a + b  
a * b  
`
```

9. Matrix multiplication

```
python
```

```
`np.dot(a, b)
```

10. Get diagonal elements

```
python
```

```
`np.diag(a)
```

11. Boolean indexing (filtering)

```
python
```

```
`a[a > 5]
```

12. Replace all odd numbers with -1

```
python
```

```
`a[a % 2 == 1] = -1
```

13. Find unique elements and their counts

```
python

`np.unique(a, return_counts=True)
```

14. Stack arrays vertically & horizontally

```
python

`np.vstack((a, b))
`np.hstack((a, b))
```

15. Transpose a matrix

```
python

`a.T
```

16. Generate random integers

```
python

`np.random.randint(0, 10, size=(3, 3))
```

17. Compute row-wise sum

```
python

`np.sum(arr, axis=1)
```

18. Normalize a NumPy array

```
python

`a = (a - np.min(a)) / (np.max(a) - np.min(a))
```

19. Create array with linearly spaced values

```
python

`np.linspace(0, 1, 5)
```

20. Find non-zero indices

```
python

`np.nonzero(arr)
```

21. Find index of max element

```
python

`np.argmax(arr)
```

22. Broadcasting in NumPy?

 Add scalar to array: NumPy **broadcasts** scalar to all elements.

```
python

`a + 10
```

23. How to shuffle an array?

```
python

`np.random.shuffle(arr)
```

24. Check NaNs in array

```
python

`np.isnan(arr)
```

25. Replace NaNs with 0

```
python

`np.nan_to_num(arr)
```

26. Clip array values

```
python

`np.clip(arr, 0, 10)
```

27. Save/load a NumPy array

```
python

`np.save('arr.npy', arr)
arr = np.load('arr.npy')
```

28. Difference between `np.array()` and `np.asarray()`?

- ✓ `np.array()` always copies data.
- ✓ `np.asarray()` avoids copying if input is already ndarray.

29. Use `where` to replace

```
python

`np.where(arr < 5, 0, 1)
```

30. Count positive values

```
python

`np.sum(arr > 0)
```

31. Compute cumulative sum

```
python

`np.cumsum(arr)
```

32. Compute pairwise distances

```
python

`from scipy.spatial.distance import pdist, squareform
squareform(pdist(arr, metric='euclidean'))
```

33. NumPy vs List performance?

✓ NumPy is C-backed and vectorized → much faster than pure Python lists in numerical ops.

Set 7B: Pandas Interview Questions (34 Qs)

1. Create a DataFrame from dictionary

```
python

`import pandas as pd

data = {'name': ['A', 'B'], 'score': [90, 85]}
df = pd.DataFrame(data)
```

2. Read CSV file

```
python

`df = pd.read_csv('data.csv')
```

3. Display first 5 rows

```
python

`df.head()
```

4. Info & Describe

```
python

`df.info()
df.describe()
```

5. Filter rows with condition

```
python
```

```
`df[df['score'] > 80]
```

6. Select specific columns

```
python
```

```
`df[['name', 'score']]
```

7. Add new column

```
python
```

```
`df['pass'] = df['score'] > 50
```

8. Drop column

```
python
```

```
`df.drop('pass', axis=1, inplace=True)
```

9. Group by a column

```
python
```

```
`df.groupby('class')['score'].mean()
```

10. Sorting DataFrame

```
python
```

```
`df.sort_values('score', ascending=False)
```

11. Handling missing data

```
python
```

```
`df.fillna(0)  
df.dropna()
```

12. Rename columns

```
python

df.rename(columns={'score': 'marks'}, inplace=True)
```

13. Merge two DataFrames

```
python

pd.merge(df1, df2, on='id')
```

14. Concatenate DataFrames

```
python

pd.concat([df1, df2])
```

15. Pivot Table

```
python

df.pivot_table(index='class', values='score', aggfunc='mean')
```

16. Apply function to column

```
python

df['score'].apply(lambda x: x + 5)
```

17. Check for duplicates

```
python

df.duplicated()
```

18. Drop duplicates


```
python

`df.drop_duplicates(inplace=True)
```

19. Reset Index

```
python

`df.reset_index(drop=True)
```

20. Set a column as index

```
python

`df.set_index('name', inplace=True)
```

21. Create datetime column

```
python

`df['date'] = pd.to_datetime(df['date'])
```

22. Extract year/month/day from date

```
python

`df['date'].dt.year
```

23. Value counts

```
python

`df['city'].value_counts()
```

24. Check column data types

```
python

`df.dtypes
```

25. Query method

```
python  
  
`df.query('score > 80')
```

26. Read Excel file

```
python  
  
`pd.read_excel('data.xlsx')
```

27. Export to CSV

```
python  
  
`df.to_csv('output.csv', index=False)
```

28. Replace values

```
python  
  
`df['city'].replace({'NY': 'New York'})
```

29. Check correlation

```
python  
  
`df.corr()
```

30. Multi-level indexing

```
python  
  
`df.set_index(['country', 'state'])
```

31. Count nulls in each column

```
python

`df.isnull().sum()
```

32. Sample rows randomly

```
python

`df.sample(n=5)
```

33. Assign new columns using assign()

```
python

`df = df.assign(status=lambda x: x.score > 80)
```

34. Detect outliers with IQR

```
python

`Q1 = df['score'].quantile(0.25)
Q3 = df['score'].quantile(0.75)
IQR = Q3 - Q1
outliers = df[(df['score'] < Q1 - 1.5 * IQR) | (df['score'] > Q3 + 1.5 * IQR)]
```



Set 7C: Matplotlib + Seaborn (33 Qs)

1. Import and basic plot

```
python

`import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [4, 5, 1])
plt.show()
```

2. Set title, xlabel, ylabel

```
python

`plt.title("Sales")
plt.xlabel("Month")
```

```
plt.ylabel("Revenue")
```

3. Scatter Plot

```
python\nplt.scatter([1, 2, 3], [4, 5, 6])
```

4. Bar Chart

```
python\nplt.bar(['A', 'B'], [10, 20])
```

5. Histogram

```
python\nplt.hist(data, bins=10)
```

6. Pie Chart

```
python\nplt.pie([10, 20, 30], labels=['A', 'B', 'C'])
```

7. Line Style & Color

```
python\nplt.plot(x, y, color='red', linestyle='--')
```

8. Multiple lines on same plot

```
python\nplt.plot(x1, y1)\nplt.plot(x2, y2)
```

9. Add legend

```
python

plt.legend(['Train', 'Test'])
```

10. Subplots

```
python

fig, ax = plt.subplots(1, 2)
```

11. Save plot

```
python

plt.savefig('plot.png')
```

12. Seaborn import

```
python

import seaborn as sns
```

13. Seaborn pairplot

```
python

sns.pairplot(df)
```

14. Correlation heatmap

```
python

sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
```

15. Boxplot

```
python\n\nsns.boxplot(x='class', y='score', data=df)
```

16. Violin plot

```
python\n\nsns.violinplot(x='class', y='score', data=df)
```

17. Swarmplot

```
python\n\nsns.swarmplot(x='class', y='score', data=df)
```

18. Countplot

```
python\n\nsns.countplot(x='gender', data=df)
```

19. Stripplot

```
python\n\nsns.stripplot(x='class', y='score', data=df)
```

20. KDE Plot

```
python\n\nsns.kdeplot(df['score'])
```

21. Set theme

```
python\n\nsns.set_style('whitegrid')
```

22. Add annotations to bar chart

```
python

`for i, v in enumerate(values):
    plt.text(i, v + 0.5, str(v))
```

23. Custom tick labels

```
python

`plt.xticks([0, 1, 2], ['Jan', 'Feb', 'Mar'])
```

24. Rotate x-axis labels

```
python

`plt.xticks(rotation=45)
```

25. Line markers

```
python

`plt.plot(x, y, marker='o')
```

26. Control figure size

```
python

`plt.figure(figsize=(8, 4))
```

27. Plot distribution with seaborn

```
python

`sns.histplot(df['score'], kde=True)
```

28. Seaborn Implot (regression line)

```
python

`sns.lmplot(x='hours', y='score', data=df)
```

29. Color palettes

```
python

`sns.set_palette("pastel")
```

30. Add grid

```
python

`plt.grid(True)
```

31. Tight layout

```
python

`plt.tight_layout()
```

32. Dual-axis plot

```
python

`fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
```

33. Seaborn FacetGrid

```
python

`g = sns.FacetGrid(df, col='gender')
g.map(sns.histplot, 'score')
```

✅ Want **Set 8: Django Advanced / REST APIs / DRF** next? Or go deeper into **System Design + Async Python + Microservices**?



Set 8

Perfect! Here's your 🔥 **Set 8: Django Advanced + REST Framework (DRF) – 100 Interview Questions with Deep Answers** — focused for **10 LPA+ Backend & Full Stack** roles.

We'll break it into three high-impact segments:

- **Part A: Django Core (35 Qs)**
- **Part B: Django REST Framework (DRF) (35 Qs)**
- **Part C: Django + System Design/Architecture (30 Qs)**

Part A: Django Core (35 Questions)

1. What is Django's request-response lifecycle?

🔴 Flow:

Client sends request → URL dispatcher → View → Model (if needed) → Template (if needed) → Response

2. Difference between `request.GET` vs `request.POST`

- `request.GET`: Query params (visible in URL)
- `request.POST`: Form data (secure for sensitive info)

3. What is `settings.py` and key configurations?

- Defines project config: `DATABASES`, `INSTALLED_APPS`, `MIDDLEWARE`, `STATIC_URL`, etc.

4. What is WSGI and ASGI?

- **WSGI**: Synchronous gateway between web server & Python app
- **ASGI**: Async version → Needed for WebSockets, real-time apps

5. Middleware in Django?

🟡 Hooks to process request/response globally:

```
python
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)
        return response
```

6. What is `manage.py` used for?

CLI to:

- Run server
- Apply migrations
- Create users, etc.

7. Role of `urls.py`

Maps URLs to views using Django's `path()` or `re_path()` functions.

8. Django views: Function vs Class-based

- **Function-based:** Simple, direct
- **Class-based:** Reusable, extendable (`ListView`, `DetailView`, etc.)

9. What is a Django model?

ORM-mapped Python class for a DB table.

```
python

class Product(models.Model):
    name = models.CharField(max_length=100)
```

10. Common field types in models

- `CharField`, `TextField`, `IntegerField`, `BooleanField`, `DateTimeField`, `ForeignKey`, `ManyToManyField`

11. How to query the database using ORM?

```
python

Product.objects.filter(price__gt=100)
```

12. What is `Meta` class in Django model?

Used for model metadata:

```
python
```

```
`class Meta:
    ordering = ['-created_at']
```

13. Difference between `null=True` and `blank=True`

- `null=True`: DB-level
- `blank=True`: Form-level

14. Migrations in Django?

Tracks model changes in `migrations/`.

Use `makemigrations` and `migrate`.

15. One-to-one, One-to-many, Many-to-many in models?

- `OneToOneField`, `ForeignKey`, `ManyToManyField`

16. What is a queryset? Lazy or not?

Lazy: Evaluates only when needed.

```
python
```

```
`qs = Product.objects.all() # Not executed yet
```

17. How to override `save()` in a model?

```
python
```

```
`def save(self, *args, **kwargs):
    self.slug = slugify(self.name)
    super().save(*args, **kwargs)
```

18. Signals in Django?

Used to trigger actions after model events:

```
python
```

```
`from django.db.models.signals import post_save
```

19. What is the admin interface?

Auto-generated CRUD UI for models.

Register models using:

```
python

`admin.site.register(Product)
```

20. How to customize the admin panel?

```
python

`class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'price']
```

21. Static vs Media files?

- `STATICFILES_DIRS`: For JS/CSS
- `MEDIA_ROOT`: For user-uploaded content

22. CSRF protection in Django?

Enabled by default.

Use `{% csrf_token %}` in templates.

23. What is a form in Django?

Python class mapped to HTML form.

```
python

`class ContactForm(forms.Form):
    name = forms.CharField()
```

24. How to use ModelForm?

```
python

`class ProductForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = '__all__'
```

25. Custom validators in Django forms?

```
python

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError("Not even")
    ,
```

26. What is a template context?

Dictionary sent to the template for rendering dynamic content.

27. Template inheritance with `{% extends %}`?

Allows DRY by using base.html:

```
html

{% extends 'base.html' %}
```

28. Template filters?

Used to modify variables:

```
html

{{ name|upper }}
```

29. Custom template filter

```
python

@register.filter
def cut(value, arg):
    return value.replace(arg, '')
    ,
```

30. File upload using Django forms?

Use `FileField` + `enctype="multipart/form-data"` in template.

31. Handling 404/500 errors?

Custom templates:

`404.html`, `500.html` in root template folder.

32. How to cache views in Django?

```
python

@cache_page(60 * 15)
def my_view(request):
    ...
```

33. Django environment settings for dev/prod

Use `os.environ.get()` pattern.

34. Use of `reverse()` and `get_absolute_url()`

- `reverse('product_detail', args=[1])`
- `get_absolute_url()` returns canonical URL for object.

35. Django's security features?

- CSRF protection
- SQL injection protection
- Password hashing
- Clickjacking protection

Part B: Django REST Framework (DRF) (35 Questions)

1. What is Django REST Framework (DRF)?

Toolkit to build Web APIs on top of Django.

2. What are serializers in DRF?

Convert complex types like queryset to JSON.

```
python

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
```

3. Difference between `Serializer` vs `ModelSerializer`

- `Serializer`: Manual fields

- `ModelSerializer`: Auto from model

4. APIView vs GenericAPIView vs ViewSet

- `APIView`: Low-level control
- `GenericAPIView`: Adds querysets, serializers
- `ViewSet`: DRY; automatic routing

5. What are mixins in DRF?

DRY classes to reuse behavior like `.list()`, `.create()`

6. Create a read-only APIView

```
python

class ProductDetail(APIView):
    def get(self, request, pk):
        ...
```

7. Token-based authentication in DRF

```
bash

pip install djangorestframework authtoken
```

Enable: `rest_framework.authtoken`

8. How to use `@api_view` decorator?

```
python

@api_view(['GET'])
def product_list(request):
    ...
```

9. What is throttling in DRF?

Limit API requests.

`DEFAULT_THROTTLE_CLASSES`, `DEFAULT_THROTTLE_RATES`

10. What is pagination in DRF?

```
python
```

```
`REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'PageNumberPagination',
    'PAGE_SIZE': 10,
}
```

11. Custom permissions

```
python
```

```
`class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.user == request.user
```

12. How to override `create()` method in ViewSet?

```
python
```

```
`def create(self, request):
    ...
```

13. Use of `request.data` VS `request.query_params`

- `data`: POST body
- `query_params`: GET query

14. What is `BrowsableAPIRenderer`?

DRF's HTML UI for testing API.

15. Renderers and Parsers in DRF

- Renderers: JSON, Browsable
- Parsers: JSON, FormParser, MultiPartParser

16. Custom exception handling

Override `exception_handler()` in settings.

17. Nested serializers


```
python

`class CategorySerializer(serializers.ModelSerializer):
    products = ProductSerializer(many=True)
```

18. HyperlinkedModelSerializer

Adds URL field:

```
python

`class ProductSerializer(HyperlinkedModelSerializer):
    ...
```

19. ViewSet actions: `@action`

```
python

`@action(detail=True, methods=['post'])
def like(self, request, pk=None):
    ...
```

20. API versioning

Enable in settings:

```
python

`'DEFAULT_VERSIONING_CLASS': 'URLPathVersioning'
```

21. DRF routers

Auto-register routes:

```
python

`router = routers.DefaultRouter()
router.register(r'products', ProductViewSet)
```

22. Filtering in DRF

```
python

`filter_backends = [DjangoFilterBackend]
filterset_fields = ['category']
```

23. Search API

```
python

`filter_backends = [SearchFilter]
search_fields = ['name', 'description']
```

24. Ordering API

```
python

`filter_backends = [OrderingFilter]
ordering_fields = ['price']
```

25. What is `is_valid(raise_exception=True)`?

Validates input and raises `ValidationError` automatically.

26. File uploads in DRF

Use `MultiPartParser` in `parser_classes`.

27. How to restrict HTTP methods in ViewSet?

```
python

`http_method_names = ['get', 'post']
```

28. How to test APIs using DRF test client?

```
python

`self.client.get('/api/products/')
```

29. Explain JWT Authentication

JWT: JSON Web Tokens → stateless auth.

Use `django-rest-framework-simplejwt`

30. How to generate and refresh JWT?

```
http
```

```
`/api/token/` → access + refresh
/api/token/refresh/`
```

31. How to serialize custom object not related to model?

Use base `Serializer` class:

```
python
```

```
`class CustomSerializer(serializers.Serializer):
    name = serializers.CharField()
```

32. Validation inside serializer

```
python
```

```
`def validate_price(self, value):
    if value < 0:
        raise serializers.ValidationError("Negative!")
```

33. How to override queryset in `get_queryset()`?

```
python
```

```
`def get_queryset(self):
    return Product.objects.filter(user=self.request.user)
```

34. Rate limiting DRF APIs globally

```
python
```

```
`DEFAULT_THROTTLE_CLASSES = [...]
DEFAULT_THROTTLE_RATES = {'user': '100/day'}
```

35. Difference: `@api_view` VS `APIView` VS `ViewSet`

Feature	`@api_view`	`APIView`	`ViewSet`
Decorator-based	✓	✗	✗
Class-based	✗	✓	✓
Router support	✗	✗	✓

Part C: Django + System Design & Architecture (30 Qs)

1. Monolith vs Microservices in Django?

- Monolith: One big Django app
 - Microservices: Split into services (auth, products, etc.)
-

2. Async support in Django?

As of 3.1+, Django supports async views:

```
python
` async def view(request): ...
```

3. How to scale Django apps?

- Gunicorn + Nginx
 - Caching (Redis)
 - Celery for tasks
 - DB optimization
-

4. Celery in Django?

For background tasks:

```
python
` @app.task
  def send_email(): ...
```

5. Use cases for signals

- Send welcome email on `post_save(User)`
 - Auto-create profile
-

6. Design rate-limited API in Django

- Use DRF throttling
 - Use Redis for custom rate limit
-

7. Django with Redis?

Used for:

- Caching
 - Celery broker
 - Rate limiting
-

8. Caching strategies?

- Per-view caching
 - Template fragment caching
 - Low-level API + Redis
-

9. How to optimize database queries?

- `.select_related()`: ForeignKeys
 - `.prefetch_related()`: ManyToMany
 - Indexing
-

10. Django + Docker?

Containerize Django app for easy deployment.

11. Django + Nginx + Gunicorn flow

Client → Nginx → Gunicorn → Django → DB

12. Custom User Model?

Use `AbstractBaseUser` or `AbstractUser` early in project.

13. API versioning in large systems?

- `/api/v1/`
 - Deprecated support via DRF
-

14. DRF or GraphQL for APIs?

- DRF: RESTful, standard
 - GraphQL: Flexible querying
-

15. Real-time apps in Django?

Use:

- Django Channels
- Redis
- WebSockets

 **Want Set 9 next? (SQL Advanced + Optimization + Design)**
Or continue to **Set 10: Python + Django Coding Challenges for 10 LPA+ Rounds?**



how many set reamning


Based on the plan we've been following, you have about **two sets remaining**:

1. **Set 9:** SQL Advanced + Optimization + Design
2. **Set 10:** Python + Django Coding Challenges for 10 LPA+ Rounds

Let me know if you'd like to move forward with either one!



Set 9

Great! Here's your  **Set 9: SQL Advanced + Optimization + Design – 100 Interview Questions with Deep Answers** — focused for **10 LPA+ Backend and Full Stack roles**.

We will break it into three core segments:

- **Part A: SQL Advanced Concepts (40 Questions)**
- **Part B: SQL Query Optimization (30 Questions)**
- **Part C: SQL Design & Architecture (30 Questions)**

Part A: SQL Advanced Concepts (40 Questions)

1. What is the difference between `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`?

- **`INNER JOIN`:** Returns only matching rows.
- **`LEFT JOIN`:** Returns all rows from left table, matched with right.
- **`RIGHT JOIN`:** Returns all rows from right table, matched with left.
- **`FULL JOIN`:** Returns all rows when there's a match in either left or right table.

2. What is the purpose of the `GROUP BY` clause?

Groups rows based on column values and allows aggregate functions like `COUNT`, `SUM`, `AVG`, etc., to be applied.

3. Explain `HAVING` vs `WHERE` clause.

- `WHERE`: Filters rows before aggregation.
- `HAVING`: Filters after aggregation.

4. What are window functions?

Window functions perform calculations across a set of table rows related to the current row. Examples: `ROW_NUMBER()`, `RANK()`, `SUM() OVER()`

5. What is the difference between `RANK()` and `DENSE_RANK()`?

- `RANK()`: Leaves gaps in ranking when there are ties.
- `DENSE_RANK()`: No gaps in ranking.

6. What is the `COALESCE()` function?

Returns the first non-null value in a list of arguments.

```
sql
`SELECT COALESCE(NULL, NULL, 'default');
-- Returns 'default'
`
```

7. Explain the `CASE` statement in SQL.

It provides conditional logic in SQL queries.

```
sql
`SELECT CASE
      WHEN age < 18 THEN 'Minor'
      ELSE 'Adult'
    END
FROM people;
`
```

8. What are common aggregate functions in SQL?

- `COUNT()`
- `SUM()`

- `AVG()`
- `MAX()`
- `MIN()`

9. What is a subquery?

A subquery is a query inside another query. It can return a single value, row, or multiple rows.

10. What is a correlated subquery?

A subquery that references columns from the outer query. It's executed once for each row in the outer query.

11. Difference between `UNION` and `UNION ALL`?

- `UNION`: Removes duplicates.
- `UNION ALL`: Includes duplicates.

12. What is `EXCEPT` and `INTERSECT` in SQL?

- `EXCEPT`: Returns rows from the first query that are not in the second query.
- `INTERSECT`: Returns only rows common to both queries.

13. What is the `DISTINCT` keyword?

Removes duplicate rows from the result set.

14. How would you retrieve unique records from a table?

Using the `DISTINCT` keyword.

```
sql
SELECT DISTINCT column_name FROM table;
```

15. What is `NULL` and how does it behave in SQL?

`NULL` represents the absence of a value. It behaves differently than an empty string or `0`.

16. How do you check for `NULL` values in SQL?

Use `IS NULL` or `IS NOT NULL`.


```
sql
```

```
`SELECT * FROM table WHERE column IS NULL;
```

17. What are CTEs (Common Table Expressions)?

A temporary result set that is defined within the execution scope of a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

```
sql
```

```
`WITH cte AS (SELECT * FROM table WHERE condition)
SELECT * FROM cte;
```

18. What is a `SELF JOIN`?

A `SELF JOIN` is when a table is joined with itself, usually to compare rows within the same table.

19. What is `NORMALIZATION`?

Normalization is the process of organizing data to reduce redundancy and dependency. Typically involves splitting data into multiple related tables.

20. What is `DENORMALIZATION`?

Denormalization is the process of adding redundant data or merging tables to improve read performance.

21. What is an index in SQL?

An index is a performance optimization feature that speeds up data retrieval from a table.

22. What are the different types of indexes in SQL?

- **Unique Index:** Ensures no duplicate values in a column.
- **Composite Index:** Index on multiple columns.
- **Full-text Index:** For text searching.

23. What is the difference between `PRIMARY KEY` and `UNIQUE` constraint?

- **`PRIMARY KEY`:** Uniquely identifies each record in a table and cannot be `NULL`.
- **`UNIQUE`:** Ensures values are unique but allows one `NULL`.

24. What is the use of `FOREIGN KEY` in SQL?

Ensures referential integrity by linking two tables.

25. What is a `TRIGGER` in SQL?

A trigger is a stored procedure that automatically executes in response to certain events on a table.

26. What are `ACID` properties in a database?

ACID stands for:

- **Atomicity:** All operations in a transaction are completed or none.
 - **Consistency:** The database transitions from one valid state to another.
 - **Isolation:** Transactions do not interfere with each other.
 - **Durability:** Committed transactions are permanent.
-

27. Explain the `ROLLBACK` command.

`ROLLBACK` undoes changes made by a transaction.

28. What is the difference between `DELETE`, `TRUNCATE`, and `DROP`?

- **DELETE:** Removes rows, can be rolled back.
 - **TRUNCATE:** Removes all rows, cannot be rolled back.
 - **DROP:** Removes the table itself.
-

29. What is the `MERGE` statement in SQL?

The `MERGE` statement combines `INSERT`, `UPDATE`, and `DELETE` operations based on a condition.

30. How would you perform a backup and restore in SQL?

- Backup: `BACKUP DATABASE dbname TO disk = 'backupfile.bak'`
 - Restore: `RESTORE DATABASE dbname FROM disk = 'backupfile.bak'`
-

31. What is a `VIEW` in SQL?

A `VIEW` is a virtual table based on the result of a query. It does not store data but presents a subset of data.

32. Difference between `INNER JOIN` and `OUTER JOIN`?

- **INNER JOIN:** Returns only matching rows.

- **OUTER JOIN:** Returns matching rows and also non-matching rows from one or both tables.
-

33. What are stored procedures in SQL?

Stored procedures are precompiled SQL code that can be executed by the database to perform operations.

34. What is the `EXPLAIN` keyword used for?

Used to display the query execution plan, helping with performance optimization.

35. What is `JOIN` optimization?

- Avoid `SELECT *`.
 - Use indexed columns in joins.
 - Avoid joining large tables unnecessarily.
-

36. Explain how `GROUP_CONCAT` works in SQL.

Concatenates values from multiple rows into a single string.

```
sql
`SELECT GROUP_CONCAT(name) FROM users;
```

37. What is `PARTITION BY` used for in window functions?

It divides the result set into partitions to perform calculations across rows in each partition.

38. What is the difference between `CHAR` and `VARCHAR`?

- `CHAR`: Fixed-length string.
 - `VARCHAR`: Variable-length string.
-

39. How do you optimize a query in SQL?

- Use indexes.
 - Avoid unnecessary joins.
 - Use `EXPLAIN` for query analysis.
 - Cache query results.
-

40. What is `AUTO_INCREMENT` in SQL?

Automatically generates a unique value when a new record is inserted into a column.

Part B: SQL Query Optimization (30 Questions)

1. What is a query execution plan and why is it important?

A query execution plan shows how a database will execute a query. It helps identify bottlenecks.

2. How does indexing improve SQL query performance?

Indexes speed up search operations by allowing the database to quickly locate rows based on column values.

3. What is query profiling?

Query profiling measures how long a query takes to execute and identifies performance bottlenecks.

4. What are the disadvantages of indexing?

- Slows down `INSERT`, `UPDATE`, `DELETE` operations.
 - Requires extra storage.
-

5. How to avoid N+1 query problems?

Use `JOIN` or `IN` to retrieve all data in a single query, instead of executing multiple queries.

6. What is the purpose of `EXPLAIN ANALYZE`?

It provides details about the execution plan and actual runtime of each step.

7. What is query caching and how does it improve performance?

Query caching stores the result of a query in memory, so repeated executions are faster.

8. What is partitioning in SQL?

Partitioning divides large tables into smaller, more manageable pieces, improving query performance.

9. What is denormalization and when should you use it?

Denormalization involves adding redundant data for performance improvements in read-heavy systems.

10. How can `DISTINCT` be costly for performance?

``DISTINCT`` requires sorting and comparing all rows, which can be resource-intensive.

11. What are some common ways to optimize a query in SQL?

- Avoid using ``SELECT *``.
 - Use proper indexes.
 - Reduce joins.
 - Use ``LIMIT`` for pagination.
-

12. How does caching work in SQL query optimization?

Cache results of frequent queries to avoid hitting the database repeatedly for the same data.

13. What is the impact of sorting large datasets in SQL?

Sorting large datasets requires resources and can be slow if not indexed properly.

14. How can subqueries affect performance?

Subqueries can cause repeated scans of tables, which slows down performance. Try to use ``JOIN`` instead.

15. How do ``GROUP BY`` and ``HAVING`` affect performance?

Both require sorting and aggregation, which can be expensive. Minimize their use on large datasets.

16. When should you use temporary tables for optimization?

Use temporary tables to break down complex queries and store intermediate results.

17. What are composite indexes and when should you use them?

Composite indexes are indexes on multiple columns. Use them when queries frequently filter on multiple columns.

18. What is normalization, and when does it hurt performance?

Normalization reduces redundancy, but over-normalization can lead to excessive joins, hurting performance.

19. How can you prevent full table scans in SQL?

Use indexes, optimize your queries, and avoid using non-sargable predicates.

20. What is the impact of ``LIKE`` in query performance?

``LIKE`` queries with wildcards at the beginning (``'%text%`'`) can't use indexes and slow down performance.

21. What is the purpose of ``WITH`` clause in SQL?

The ``WITH`` clause defines common table expressions (CTEs), improving readability and reusability.

22. How do joins affect query performance?

Joins, especially on large tables, can be resource-intensive. Optimize them by ensuring indexes on joined columns.

23. What is the purpose of ``EXPLAIN`` keyword?

It provides an execution plan that helps you analyze how a query will be executed.

24. What is database sharding?

Sharding involves breaking a database into smaller, more manageable pieces (shards) to improve scalability.

25. How does database indexing impact ``INSERT`` operations?

Indexing makes ``INSERT`` operations slower since the index needs to be updated.

26. What are ``OR`` conditions and how do they impact performance?

``OR`` conditions prevent the use of indexes, resulting in full table scans and slower queries.

27. What is SQL tuning?

SQL tuning involves optimizing queries to improve execution time, reduce resource usage, and ensure fast retrieval of data.

28. What is query optimization in databases?

Query optimization involves improving the performance of queries through techniques like indexing, reordering joins, and minimizing resource-heavy operations.

29. How do you identify slow queries in SQL?

Use database logs or the ``EXPLAIN`` plan to identify slow queries and bottlenecks.

30. What is an execution plan?

An execution plan describes the steps the SQL engine will take to retrieve data, helping you optimize your queries.



Part C: SQL Design & Architecture (30 Questions)

1. How do you design a database schema?

Understand the business model, identify entities and their relationships, then create tables and define constraints.

2. What is a normalized database schema?

A schema designed to reduce data redundancy and improve data integrity.

3. What is a denormalized schema and when should you use it?

Denormalization combines tables to improve read performance in OLAP systems.

4. What is a star schema?

A star schema is a type of database schema used in data warehousing where data is organized into facts and dimensions.

5. What is the difference between OLAP and OLTP databases?

- **OLTP:** Handles transactional operations.
 - **OLAP:** Optimized for querying and analyzing large datasets.
-

6. How do you handle database scaling?

- Vertical scaling: Add resources to the server.
 - Horizontal scaling: Add more servers and distribute data.
-

7. What is database replication?

Database replication involves copying data from one database to another for redundancy and load balancing.

8. What is a data warehouse?

A centralized repository used for reporting and analysis of large amounts of historical data.

9. What is a data lake?

A data lake stores large amounts of raw data, typically unstructured or semi-structured, for future processing.

10. What is partitioning in SQL?

Partitioning divides a large table into smaller, more manageable parts.

11. What is horizontal vs vertical partitioning?

- **Horizontal Partitioning:** Splitting rows into different tables.
 - **Vertical Partitioning:** Splitting columns into different tables.
-

12. How would you design a highly available database?

Use replication, failover strategies, and clustering to ensure high availability.

13. How do you ensure data integrity in a relational database?

- Use constraints like `PRIMARY KEY`, `FOREIGN KEY`, and `CHECK`.
 - Normalize tables.
 - Use transactions to maintain ACID properties.
-

14. What is the importance of backups?

Regular backups ensure data recovery in case of corruption, failure, or loss.

15. How would you ensure that database transactions are ACID-compliant?

Use transactions with `BEGIN`, `COMMIT`, and `ROLLBACK` commands.

16. How do you design a database for scalability?

Consider sharding, replication, partitioning, and using caching strategies to optimize for large data volumes.

17. What are CAP and ACID properties in database design?

- **CAP:** Consistency, Availability, Partition tolerance.
 - **ACID:** Atomicity, Consistency, Isolation, Durability.
-

18. What is a relational database?

A relational database stores data in tables with rows and columns, maintaining relationships between tables.

19. What is a non-relational database?

A non-relational database stores unstructured or semi-structured data, typically in formats like JSON, BSON, or XML.

20. What is a schema-less database?

A schema-less database doesn't require predefined schemas for data storage. Examples: NoSQL databases.

21. What is a database cluster?

A database cluster involves multiple servers working together to manage a database system.

22. What is the importance of indexing in database design?

Indexes speed up data retrieval by providing faster lookup times.

23. How do you handle concurrent database transactions?

Use locking mechanisms like row-level locks and isolation levels to manage concurrent access.

24. What is referential integrity in a database?

Ensures that relationships between tables remain consistent by enforcing valid `FOREIGN KEY` constraints.

25. How do you design a database for multi-tenancy?

Store each tenant's data separately while maintaining shared application logic and resources.

26. What are some common database design mistakes?

- Over-normalizing
 - Lack of proper indexing
 - Not considering future scalability
-

27. How do you design a database for reporting?

Design it using denormalized schemas like a star or snowflake schema for faster querying.

28. What is database sharding?

Sharding distributes data across multiple machines to improve performance and scalability.

29. What is OLTP and OLAP database design?

- **OLTP:** Focuses on transactional data.
- **OLAP:** Focuses on analytical data.

30. What is a database foreign key constraint?

A foreign key ensures that values in one table correspond to valid rows in another table.

This set should prepare you thoroughly for SQL questions in your interview. Let me know if you need additional information!



Set 10

Here's **Set 10: Python + Django Coding Challenges for 10 LPA+ Rounds**. This set includes **Python** and **Django** coding challenges designed to help you crack a **10 LPA+** Backend or Full Stack role.

Python Coding Challenges (50 Questions)

1. Find the factorial of a number using recursion.

Answer:

```
python

def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

2. Reverse a string without using built-in functions.

Answer:

```
python

def reverse_string(s):
    reversed_str = ''
    for char in s:
        reversed_str = char + reversed_str
    return reversed_str
```

3. Write a Python program to check if a number is prime.

Answer:

python

```
`def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
`
```

4. Write a Python program to generate Fibonacci series up to n-th term.

Answer:

python

```
`def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
`
```

5. Find the largest number in a list without using `max()`.

Answer:

python

```
`def find_largest(lst):
    largest = lst[0]
    for num in lst:
        if num > largest:
            largest = num
    return largest
`
```

6. Write a program to find the common elements between two lists.

Answer:

python

```
`def common_elements(list1, list2):
    return list(set(list1) & set(list2))
`
```

7. Remove duplicates from a list.

Answer:

```
python
```

```
`def remove_duplicates(lst):  
    return list(set(lst))  
`
```

8. Write a Python program to find the GCD (Greatest Common Divisor) of two numbers.

Answer:

```
python
```

```
`def gcd(a, b):  
    while b:  
        a, b = b, a % b  
    return a  
`
```

9. Check if a string is a palindrome.

Answer:

```
python
```

```
`def is_palindrome(s):  
    return s == s[::-1]  
`
```

10. Find the sum of digits of a number.

Answer:

```
python
```

```
`def sum_of_digits(n):  
    return sum(int(digit) for digit in str(n))  
`
```

11. Write a program to merge two sorted lists.

Answer:

```
python
```

```
`def merge_sorted_lists(list1, list2):  
    return sorted(list1 + list2)  
`
```

12. Write a Python program to count the frequency of elements in a list.

Answer:

```
python

`from collections import Counter
def count_frequency(lst):
    return dict(Counter(lst))
`
```

13. Write a program to find the second largest element in a list.

Answer:

```
python

`def second_largest(lst):
    first, second = float('-inf'), float('-inf')
    for num in lst:
        if num > first:
            second = first
            first = num
        elif first > num > second:
            second = num
    return second
`
```

14. Write a Python program to convert a string to an integer without using `int()`.

Answer:

```
python

`def string_to_int(s):
    result = 0
    for char in s:
        result = result * 10 + (ord(char) - ord('0'))
    return result
`
```

15. Find the intersection of two sets.

Answer:

```
python

`def intersection(set1, set2):
    return set1 & set2
`
```

16. Write a Python program to check if a number is an Armstrong number.

Answer:

```
python
```

```
`def is_armstrong(num):
    n = len(str(num))
    return num == sum(int(digit) ** n for digit in str(num))
`
```

17. Write a Python program to sort a list of tuples based on the second element.

Answer:

```
python
```

```
`def sort_by_second(tuples):
    return sorted(tuples, key=lambda x: x[1])
`
```

18. Write a Python program to calculate the power of a number.

Answer:

```
python
```

```
`def power(base, exponent):
    return base ** exponent
`
```

19. Write a program to generate a random password.

Answer:

```
python
```

```
`import random
import string

def generate_password(length):
    characters = string.ascii_letters + string.digits + string.punctuation
    return ''.join(random.choice(characters) for _ in range(length))
`
```

20. Write a Python program to flatten a list of lists.

Answer:

```
python
```

```
`def flatten(lst):
    return [item for sublist in lst for item in sublist]
`
```

21. Write a program to calculate the square of a number using ``lambda``.

Answer:

```
python

`square = lambda x: x * x
```

22. Write a Python program to find the length of a string without using ``len()``.

Answer:

```
python

`def string_length(s):
    count = 0
    for char in s:
        count += 1
    return count
```

23. Reverse a list without using built-in functions.

Answer:

```
python

`def reverse_list(lst):
    return lst[::-1]
```

24. Count the occurrence of each character in a string.

Answer:

```
python

`from collections import Counter
def count_characters(s):
    return dict(Counter(s))
```

25. Write a Python program to find the LCM of two numbers.

Answer:

```
python

`def lcm(a, b):
    def gcd(x, y):
        while y:
            x, y = y, x % y
```

```

    return x
    return abs(a * b) // gcd(a, b)

```

26. Write a Python program to remove all vowels from a string.

Answer:

```

python

def remove_vowels(s):
    vowels = "aeiouAEIOU"
    return ''.join([char for char in s if char not in vowels])

```

27. Write a Python program to check if two strings are anagrams.

Answer:

```

python

def are_anagrams(str1, str2):
    return sorted(str1) == sorted(str2)

```

28. Write a Python program to find the largest prime number less than a given number.

Answer:

```

python

def largest_prime_less_than(n):
    for i in range(n - 1, 1, -1):
        if is_prime(i):
            return i

```

29. Write a Python program to convert a decimal number to binary.

Answer:

```

python

def decimal_to_binary(n):
    return bin(n).replace("0b", "")

```

30. Write a program to find the number of vowels and consonants in a string.

Answer:

python

```
`def count_vowels_consonants(s):
    vowels = "aeiouAEIOU"
    vowels_count = sum(1 for char in s if char in vowels)
    consonants_count = sum(1 for char in s if char.isalpha() and char not in vowels)
    return vowels_count, consonants_count
`
```

31. Write a Python program to find the missing number in an array containing 1 to n.

Answer:

python

```
`def find_missing_number(arr, n):
    total = n * (n + 1) // 2
    return total - sum(arr)
`
```

32. Write a Python program to implement binary search.

Answer:

python

```
`def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
`
```

33. Write a Python program to find the square root of a number using `math` module.

Answer:

python

```
`import math
def square_root(n):
    return math.sqrt(n)
`
```

34. Write a Python program to check if a given year is a leap year.

Answer:

```
python
```

```
`def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```

35. Write a program to print the first n prime numbers.

Answer:

```
python
```

```
`def first_n_primes(n):
    primes = []
    num = 2
    while len(primes) < n:
        if is_prime(num):
            primes.append(num)
        num += 1
    return primes
```

36. Write a Python program to find the median of a list of numbers.

Answer:

```
python
```

```
`def median(lst):
    sorted_lst = sorted(lst)
    n = len(sorted_lst)
    if n % 2 == 1:
        return sorted_lst[n // 2]
    else:
        return (sorted_lst[n // 2 - 1] + sorted_lst[n // 2]) / 2
```

37. Write a Python program to generate a random integer between a given range.

Answer:

```
python
```

```
`import random
def random_integer(a, b):
    return random.randint(a, b)
```

38. Write a Python program to check if a string contains only digits.

Answer:

```
python

`def is_digit(s):
    return s.isdigit()
`
```

39. Write a Python program to find the common divisors of two numbers.

Answer:

```
python

`def common_divisors(a, b):
    divisors = []
    for i in range(1, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            divisors.append(i)
    return divisors
`
```

40. Write a program to count the number of set bits (1's) in a number.

Answer:

```
python

`def count_set_bits(n):
    return bin(n).count('1')
`
```

41. Write a Python program to find the largest palindrome in a list of strings.

Answer:

```
python

`def largest_palindrome(lst):
    return max([word for word in lst if word == word[::-1]], key=len, default="")
`
```

42. Write a Python program to rotate a list by n positions.

Answer:

```
python

`def rotate_list(lst, n):
    return lst[n:] + lst[:n]
`
```

43. Write a Python program to merge two dictionaries.

Answer:

```
python

`def merge_dicts(dict1, dict2):
    return {**dict1, **dict2}
`
```

44. Write a Python program to convert a list of strings into a single string.

Answer:

```
python

`def list_to_string(lst):
    return ''.join(lst)
`
```

45. Write a Python program to implement a queue using two stacks.

Answer:

```
python

`class Queue:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    def enqueue(self, item):
        self.stack1.append(item)

    def dequeue(self):
        if not self.stack2:
            while self.stack1:
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop() if self.stack2 else None
`
```

46. Write a Python program to implement a stack using two queues.

Answer:

```
python

`from collections import deque

class Stack:
    def __init__(self):
        self.queue1 = deque()
        self.queue2 = deque()

    def push(self, x):
        self.queue1.append(x)

    def pop(self):
        if len(self.queue1) == 0:
            return None
`
```

```
while len(self.queue1) > 1:
    self.queue2.append(self.queue1.popleft())
    popped = self.queue1.popleft()
    self.queue1, self.queue2 = self.queue2, self.queue1
    return popped
```

47. Write a Python program to find all unique pairs in a list that sum up to a target value.

Answer:

python

```
def find_pairs(lst, target):
    seen = set()
    pairs = set()
    for num in lst:
        complement = target - num
        if complement in seen:
            pairs.add((min(num, complement), max(num, complement)))
            seen.add(num)
    return list(pairs)
```

48. Write a Python program to flatten a dictionary.

Answer:

python

```
def flatten_dict(d, parent_key='', sep='_'):
    items = []
    for k, v in d.items():
        new_key = f"{parent_key}{sep}{k}" if parent_key else k
        if isinstance(v, dict):
            items.extend(flatten_dict(v, new_key, sep=sep).items())
        else:
            items.append((new_key, v))
    return dict(items)
```

49. Write a Python program to perform quicksort on a list.

Answer:

python

```
def quicksort(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[0]
    less = [x for x in lst[1:] if x <= pivot]
    greater = [x for x in lst[1:] if x > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)
```

50. Write a Python program to check if a given number is a perfect square.

Answer:

```
python
```

```
`import math
def is_perfect_square(n):
    return math.isqrt(n) ** 2 == n
`
```

These **Python** challenges cover a wide array of problems that can be asked in coding interviews for a **10 LPA+** salary package, which focus on **problem-solving** and **algorithmic thinking**. Let me know if you need **Django** questions as well, or any follow-up details!