:

# 🐍 Python Interview Notes – Concurrency, Asyncio, Threading, Multiprocessing

## 🧠 Python Basics Summary

### 🔹 ****1. Python Data Types****

* **int** – Integer numbers: 1, -5
* **float** – Decimal numbers: 3.14, -2.5
* **str** – Strings: "hello", 'world'
* **bool** – Boolean: True, False
* **list** – Ordered, mutable: [1, 2, "a"]
* **tuple** – Ordered, immutable: (1, 2, "a")
* **set** – Unordered, unique items: {1, 2, 3}
* **dict** – Key-value pairs: {"name": "Alice", "age": 25}
* **None** – Null type: None

### 🔹 ****2. Conditional Statements****

python

CopyEdit

x = 10

if x > 5:

print("Greater")

elif x == 5:

print("Equal")

else:

print("Smaller")

### 🔹 ****3. Loops****

**For loop**:

python

CopyEdit

for i in range(5):

print(i)

**While loop**:

python

CopyEdit

count = 0

while count < 5:

print(count)

count += 1

### 🔹 ****4. List Comprehension****

python

CopyEdit

# Square of even numbers from 0 to 9

squares = [x\*\*2 for x in range(10) if x % 2 == 0]

### 🔹 ****5. Scope in Python****

* **Local**: Inside a function
* **Enclosing**: In the outer function (closure context)
* **Global**: Defined at the top level of the script
* **Built-in**: Names pre-defined by Python (e.g., len, sum)

👉 Known as **LEGB Rule** (Local → Enclosing → Global → Built-in)

### 🔹 ****6. Closures****

A closure is when an inner function remembers variables from its enclosing scope even after the outer function has finished executing.

python

CopyEdit

def outer():

x = 5

def inner():

print(x) # x is remembered

return inner

func = outer()

func() # Output: 5

## 🐍 Python Interview Summary Notes

### ✅ 1. Object-Oriented Programming (OOPs) in Python

| Concept | Description |
| --- | --- |
| **Class** | Blueprint for creating objects. |
| **Object** | Instance of a class. |
| **Encapsulation** | Hiding internal state using private variables (\_var, \_\_var). |
| **Abstraction** | Hiding complex logic using simple interfaces. Achieved via classes or ABCs. |
| **Inheritance** | One class inherits properties from another (class B(A):). |
| **Polymorphism** | Same function behaves differently in different classes. |

python

CopyEdit

class Animal:

def speak(self): print("Animal sound")

class Dog(Animal):

def speak(self): print("Bark")

### ⚠️ 2. Exception Handling in Python

| Concept | Syntax & Notes |
| --- | --- |
| Try/Except | Catches exceptions |
| Else | Runs if no exception |
| Finally | Always runs (cleanup etc.) |
| Custom Exception | class MyError(Exception): pass |

python

CopyEdit

try:

result = 10 / 0

except ZeroDivisionError:

print("Cannot divide by zero")

finally:

print("Cleanup")

## 🔄 What is Concurrency?

**Concurrency** means multiple tasks are in progress at the same time (but not necessarily running simultaneously).

Python supports concurrency via:

| Method | Best For | True Parallelism? | GIL Aware? |
| --- | --- | --- | --- |
| threading | I/O-bound tasks | ❌ No | ❌ Yes |
| multiprocessing | CPU-bound tasks | ✅ Yes | ✅ Avoids GIL |
| asyncio | I/O-bound tasks | ❌ No | ✅ Single-threaded |

## 🧵 1. threading – For I/O-bound Tasks

python

CopyEdit

import threading

import time

def send\_email():

time.sleep(2)

print("Email sent!")

def log\_response():

print("Response sent!")

t1 = threading.Thread(target=send\_email)

t1.start()

log\_response()

### ✅ Key Points:

* Threads share memory.
* Great for I/O-bound tasks (e.g., file I/O, APIs).
* Blocked by GIL for CPU tasks.
* Good for **parallel waiting**, not **parallel computing**.

## 💥 2. multiprocessing – For CPU-bound Tasks

python

CopyEdit

from multiprocessing import Process

def square(n):

print(n \* n)

p = Process(target=square, args=(5,))

p.start()

p.join()

### ✅ Key Points:

* Separate memory space, runs on multiple CPU cores.
* Not blocked by GIL.
* Ideal for CPU-heavy tasks (math, image processing).
* Slower communication between processes than threads.

## ⚡ 3. asyncio – For Async I/O

python

CopyEdit

import asyncio

async def say\_hello():

await asyncio.sleep(1)

print("Hello")

asyncio.run(say\_hello())

### ✅ Key Points:

* Uses async and await.
* Cooperative multitasking using an event loop.
* Efficient for **many** I/O-bound tasks (like calling 100 APIs).
* Not parallel, just fast-switching via event loop.

## 🔁 JavaScript-style Event Loop in Python

### Similarities to JS:

* Uses **event loop** to handle tasks.
* Handles I/O operations **without blocking**.
* await releases control back to the event loop.

### Event Loop Flow:

1. Synchronous code runs first.
2. await pauses the coroutine.
3. Event loop runs next task (like a microtask).
4. Completed async task resumes later.

## ⚙️ I/O-bound vs CPU-bound

| Task Type | CPU-bound | I/O-bound |
| --- | --- | --- |
| Definition | Slowed by CPU | Slowed by Input/Output (e.g. network, disk) |
| Uses CPU? | ✅ Yes | ❌ No |
| Examples | Math, image processing | API calls, DB reads, file write |
| Best Tool | multiprocessing | asyncio / threading |

## 🔍 When to Use What?

| Scenario | Tool |
| --- | --- |
| Heavy computation (math, image) | multiprocessing |
| Fetching data from APIs | asyncio |
| Sending email in background | asyncio / threading |
| Reading large file | asyncio |
| Simulating background job | threading |

## 🔄 Quick Analogy Summary:

| Feature | Threading | Asyncio | Multiprocessing |
| --- | --- | --- | --- |
| Type | OS-level threads | Coroutines (green threads) | Independent processes |
| Parallel? | ❌ Not for CPU-heavy | ❌ No (cooperative) | ✅ Yes |
| GIL safe? | ❌ No | ✅ Yes | ✅ Yes |
| Memory Shared? | ✅ Yes | ✅ Yes | ❌ No (isolated) |
| Use Case | I/O-bound | I/O-bound | CPU-bound |

## 🧠 Final Core Difference Between asyncio and threading

| Feature | asyncio | threading |
| --- | --- | --- |
| **Concurrency Model** | **Cooperative multitasking** | **Preemptive multitasking** (OS-controlled) |
| **Underlying Mechanism** | Single-threaded **event loop** | Multiple **OS-level threads** |
| **Switching Control** | You **manually yield** with await | OS **automatically switches** between threads |
| **Runs in Parallel?** | ❌ No true parallelism | ✅ Threads may run in parallel on I/O waits |
| **Memory** | Shared (single thread) | Shared memory across threads |
| **Code Complexity** | Needs async/await syntax everywhere | Can use regular functions |
| **Best Use Case** | High I/O tasks with **many lightweight jobs** | Blocking I/O like file reading or API calls |
| **CPU Usage** | Low CPU load (ideal for 10,000 I/O tasks) | Slightly more CPU usage due to threads |
| **Error Risk** | Fewer race conditions | Prone to **race conditions**, needs locking |

### 🔍 What does this really mean?

* **Asyncio** is like one person doing many small tasks and saying “I'll wait, do something else for now.”
* **Threading** is like having multiple people doing tasks at the same time — the OS manages who works when.

### 🧪 Practical Difference Example:

If you want to **download 1000 URLs**:

* asyncio scales better with fewer resources.
* threading may hit system limits (thread pool limits, memory).

## ✅ Final Verdict

| If you need to: | Use: |
| --- | --- |
| Handle 1000+ network/API requests | asyncio |
| Read/write files or handle 10-100 tasks | threading |
| Avoid async/await syntax | threading |
| Avoid race conditions | asyncio |

## 🧵 Multithreading vs 🔥 Multiprocessing in Python

| Feature | threading (Multithreading) | multiprocessing |
| --- | --- | --- |
| **Concurrency Model** | Multiple threads **within a single process** | Multiple **independent processes** |
| **Parallelism** | ❌ No true parallelism due to the GIL | ✅ True parallelism (each process has its own Python interpreter) |
| **GIL Affected?** | ✅ Yes – only one thread runs Python bytecode at a time | ❌ No – each process has its own GIL |
| **Memory Sharing** | ✅ Yes (shared memory space) | ❌ No (separate memory space) |
| **Communication** | Easy (same memory) but needs **thread locks** | Harder (need Queues, Pipes, or Managers) |
| **Setup Cost** | Low overhead | High overhead (process creation is expensive) |
| **Best For** | I/O-bound tasks | CPU-bound tasks (e.g., computation-heavy) |
| **Risk of Bugs** | High – race conditions, deadlocks | Lower (isolated memory avoids data clashes) |
| **Crash Effect** | One thread crashing may affect others | One process crashing doesn’t affect others |

### 🧠 Global Interpreter Lock (GIL)

* A mutex that allows **only one thread** to execute Python bytecode at a time.
* Makes multithreading inefficient for **CPU-bound** tasks in CPython.
* Multiprocessing avoids the GIL by creating separate memory and interpreter spaces.

### 📌 Example: Same Task with Both

#### 1. Using Threads (I/O-bound)

python

CopyEdit

import threading

import time

def task():

time.sleep(2)

print("Done")

for \_ in range(3):

t = threading.Thread(target=task)

t.start()

#### 2. Using Processes (CPU-bound)

python

CopyEdit

from multiprocessing import Process

import time

def compute():

# heavy CPU task

total = 0

for \_ in range(10\*\*7):

total += 1

print("Done")

for \_ in range(3):

p = Process(target=compute)

p.start()

### 🧪 Summary Decision Table

| Situation | Use This |
| --- | --- |
| API calls, file I/O, network latency | threading |
| Image processing, ML computation | multiprocessing |
| Need shared memory or object states | threading |
| Want true CPU parallelism | multiprocessing |

🔹 **7. Lambda Functions (Anonymous Functions)**

**Syntax**:

python

CopyEdit

lambda arguments: expression

**Examples**:

python

CopyEdit

add = lambda a, b: a + b

print(add(3, 4)) # Output: 7

# With map

nums = [1, 2, 3]

doubled = list(map(lambda x: x \* 2, nums)) # [2, 4, 6]

# With filter

evens = list(filter(lambda x: x % 2 == 0, nums)) # [2]

**Use Cases**: Use in places where you need a quick, short function for map(), filter(), sorted(), etc.

## 🎁 Decorators in Python

### 🔸 What is a Decorator?

A **decorator** is a function that **takes another function as input**, adds some functionality, and **returns a new function** — without modifying the original one.

In simple terms, it **wraps** a function to extend its behavior.

### 🧱 Syntax

python

CopyEdit

def decorator\_func(original\_func):

def wrapper\_func():

print("Before the function runs")

original\_func()

print("After the function runs")

return wrapper\_func

Usage:

python

CopyEdit

@decorator\_func

def say\_hello():

print("Hello!")

say\_hello()

**Output:**

pgsql

CopyEdit

Before the function runs

Hello!

After the function runs

The @decorator\_func is **syntactic sugar** for:

python

CopyEdit

say\_hello = decorator\_func(say\_hello)

### 🧩 Decorator with Arguments

python

CopyEdit

def decorator\_func(func):

def wrapper(\*args, \*\*kwargs):

print(f"Running {func.\_\_name\_\_}")

return func(\*args, \*\*kwargs)

return wrapper

@decorator\_func

def greet(name):

print(f"Hello, {name}!")

greet("Alice")

Output:

sql

CopyEdit

Running greet

Hello, Alice!

### 🔁 Use Cases

* Logging
* Access control (permissions)
* Measuring execution time
* Caching
* Rate limiting in APIs

### ✅ Built-in Decorators

* @staticmethod – Define a static method in a class.
* @classmethod – Define a class method (gets cls instead of self).
* @property – Make a method behave like an attribute.

Example:

python

CopyEdit

class Circle:

def \_\_init\_\_(self, radius):

self.\_radius = radius

@property

def area(self):

return 3.14 \* self.\_radius \*\* 2

c = Circle(5)

print(c.area) # No parentheses!

### 🚀 Bonus: Timer Decorator Example

python

CopyEdit

import time

def timer(func):

def wrapper(\*args, \*\*kwargs):

start = time.time()

result = func(\*args, \*\*kwargs)

end = time.time()

print(f"{func.\_\_name\_\_} ran in {end - start:.2f}s")

return result

return wrapper

@timer

def slow\_function():

time.sleep(2)

print("Done")

slow\_function()

🧠 **Python's "spread-like" behavior (e.g. unpacking with** \* **or** \*\***) creates a** shallow copy — **not a deep copy**.

## 🧯 Examples of "Spread Operator" in Python

### 🔹 For Lists (\* operator)

python

CopyEdit

original = [[1, 2], [3, 4]]

copied = [\*original] # Like [...original] in JS

copied[0][0] = 999

print("Original:", original)

print("Copied:", copied)

**Output:**

lua

CopyEdit

Original: [[999, 2], [3, 4]]

Copied: [[999, 2], [3, 4]]

🧨 **Shallow copy**: Inner lists are still shared.

### 🔸 For Dictionaries (\*\* operator)

python

CopyEdit

original = {"a": {"nested": 1}, "b": 2}

copied = {\*\*original}

copied["a"]["nested"] = 999

print("Original:", original)

print("Copied:", copied)

**Output:**

yaml

CopyEdit

Original: {'a': {'nested': 999}, 'b': 2}

Copied: {'a': {'nested': 999}, 'b': 2}

🧨 Again, shallow copy — the inner dict is still shared.

### ✅ If You Want a Deep Copy?

Use:

python

CopyEdit

import copy

deep = copy.deepcopy(original)

### ✅ Summary

| Operator | Type of Copy | Notes |
| --- | --- | --- |
| \*list | Shallow copy | Copies outer list only |
| \*\*dict | Shallow copy | Copies top-level keys only |
| copy.copy() | Shallow copy | Similar to above |
| copy.deepcopy() | Deep copy | Recursively copies everything |

## 🧯 Examples of "Spread Operator" in Python

### 🔹 For Lists (\* operator)

python

CopyEdit

original = [[1, 2], [3, 4]]

copied = [\*original] # Like [...original] in JS

copied[0][0] = 999

print("Original:", original)

print("Copied:", copied)

**Output:**

lua

CopyEdit

Original: [[999, 2], [3, 4]]

Copied: [[999, 2], [3, 4]]

🧨 **Shallow copy**: Inner lists are still shared.

### 🔸 For Dictionaries (\*\* operator)

python

CopyEdit

original = {"a": {"nested": 1}, "b": 2}

copied = {\*\*original}

copied["a"]["nested"] = 999

print("Original:", original)

print("Copied:", copied)

**Output:**

yaml

CopyEdit

Original: {'a': {'nested': 999}, 'b': 2}

Copied: {'a': {'nested': 999}, 'b': 2}

🧨 Again, shallow copy — the inner dict is still shared.

### ✅ If You Want a Deep Copy?

Use:

python

CopyEdit

import copy

deep = copy.deepcopy(original)

### ✅ Summary

| Operator | Type of Copy | Notes |
| --- | --- | --- |
| \*list | Shallow copy | Copies outer list only |
| \*\*dict | Shallow copy | Copies top-level keys only |
| copy.copy() | Shallow copy | Similar to above |
| copy.deepcopy() | Deep copy | Recursively copies everything |

## ✅ Does a Coroutine Pause Execution?

Yes, **a coroutine** can **pause itself** — but not the entire program.

## 💡 Here’s the Key Distinction

### 🔹 Coroutine (async def + await)

* **Pauses itself only** when it hits an await.
* While it's paused, the **event loop continues running**, doing other work (like running other coroutines).

So:

**Coroutines pause themselves**, not the main thread or event loop.

### 🔸 Asyncio Event Loop

* The **event loop is the scheduler**.
* It runs coroutines.
* When a coroutine awaits something (e.g. sleep, I/O), the event loop suspends it and picks the next coroutine to run.

### 🧠 Analogy

Imagine the **event loop as a manager**, and **coroutines as employees**:

* Each employee (coroutine) says:

“Hey boss, I’m waiting for a file to download — I’ll be back in 2 minutes.”

* The boss (event loop) replies:

“Cool, I’ll go check on other employees in the meantime.”

No one blocks the manager (event loop), and work gets done efficiently.

### 🔁 Recap

| Concept | Who pauses? | Who keeps going? |
| --- | --- | --- |
| Coroutine + await | The coroutine itself | The event loop continues |
| Thread.sleep (bad) | Whole thread | Entire program might block |
| asyncio.sleep | Only that coroutine pauses | Others continue running |

### 📘 Django Models — Core Concepts Cheat Sheet

#### ✅ 1. Model Basics

* A Django model is a Python class that subclasses django.db.models.Model.
* Each model maps to a single table in the database.
* Fields define table columns (CharField, IntegerField, ForeignKey, etc).

class Book(models.Model):

title = models.CharField(max\_length=200)

author = models.ForeignKey('Author', on\_delete=models.CASCADE)

#### 🔄 2. on\_delete Behaviors in ForeignKey

Specifies what happens when the referenced object is deleted:

| Option | Behavior |
| --- | --- |
| CASCADE | Deletes child when parent is deleted. (Most common) |
| PROTECT | Prevents deletion of parent if child exists. Raises ProtectedError. |
| SET\_NULL | Sets the FK to NULL when parent is deleted (requires null=True). |
| SET\_DEFAULT | Sets FK to default value (requires default= on field). |
| SET(...) | Sets FK to a custom value or callable. |
| DO\_NOTHING | No DB action. Risk of referential integrity issues. |

class Book(models.Model):

author = models.ForeignKey('Author', on\_delete=models.SET\_NULL, null=True)

#### 🔁 3. ManyToManyField

* Models many-to-many relationships.
* Django auto-creates an intermediate table.

class Student(models.Model):

name = models.CharField(max\_length=100)

class Course(models.Model):

title = models.CharField(max\_length=100)

students = models.ManyToManyField(Student)

You can also customize the intermediate table using through=.

class Enrollment(models.Model):

student = models.ForeignKey(Student, on\_delete=models.CASCADE)

course = models.ForeignKey(Course, on\_delete=models.CASCADE)

date\_enrolled = models.DateField()

class Course(models.Model):

title = models.CharField(max\_length=100)

students = models.ManyToManyField(Student, through='Enrollment')

#### ⚙️ 4. Model Meta Class

* Meta is an inner class used to configure model behavior (table name, ordering, constraints, etc).

class Book(models.Model):

title = models.CharField(max\_length=100)

class Meta:

db\_table = 'library\_books' # custom table name

ordering = ['title'] # default ordering

verbose\_name = 'Book'

verbose\_name\_plural = 'Books'

**Common Meta options:**

* db\_table: Custom DB table name.
* ordering: Default ordering of querysets.
* verbose\_name, verbose\_name\_plural: Human-readable names.
* unique\_together / constraints: For composite uniqueness.

### ⚠️ 5. The N+1 Query Problem

**Problem**: When querying related models in a loop, Django may hit the database once for each iteration — causing N+1 queries.

**Example (Inefficient):**

books = Book.objects.all() # 1 query

for book in books:

print(book.author.name) # 1 query per book (N queries)

➡️ If you have 100 books, this results in **1 + 100 = 101 queries!**

#### ✅ Solution: Use select\_related() and prefetch\_related()

|  |  |  |
| --- | --- | --- |
| Method | Use when... | Works on... |
| select\_related() | You have a **ForeignKey or OneToOne** | Single-valued relations |
| prefetch\_related() | You have a **ManyToMany or reverse FK** | Multi-valued relations |

**Using select\_related:**

books = Book.objects.select\_related('author')

for book in books:

print(book.author.name) # Only 1 query for all books + authors!

**Using prefetch\_related:**

courses = Course.objects.prefetch\_related('students')

for course in courses:

for student in course.students.all():

print(student.name)

Great — let's dive into your **custom middleware** idea and implement it in full.

### ✅ Goal:

Create a Django middleware that:

* Adds a **custom response header** called Test-Token
* This token includes:
  + An **access token** (placeholder or real)
  + The **user’s name**, if authenticated

### 🧠 Step-by-step logic:

1. If the user is authenticated, retrieve their username.
2. Generate or fetch an access token.
3. Combine both into a custom header value.
4. Attach it to the **response object**.

### 🧪 Full Middleware Code Example

python

CopyEdit

# middleware/custom\_token.py

class CustomTokenMiddleware:

def \_\_init\_\_(self, get\_response):

self.get\_response = get\_response

def \_\_call\_\_(self, request):

# This is run \*before\* the view is called

response = self.get\_response(request)

# This is run \*after\* the view returns a response

user = getattr(request, 'user', None)

if user and user.is\_authenticated:

username = user.username

access\_token = self.generate\_access\_token(user)

custom\_token = f"{access\_token}:{username}"

response['Test-Token'] = custom\_token

else:

response['Test-Token'] = 'anonymous'

return response

def generate\_access\_token(self, user):

# Just for demonstration — in real use, use JWT or a token system

import hashlib

return hashlib.sha256(user.username.encode()).hexdigest()

### 🛠 How to Use It

1. Add it to your MIDDLEWARE list in settings.py:

python

CopyEdit

MIDDLEWARE = [

...

'middleware.custom\_token.CustomTokenMiddleware',

]

## ✅ 1. ****Custom Managers and QuerySets in Django****

### 🔹 What is a Manager?

A **Manager** in Django is an interface through which Django ORM interacts with your database. By default, every model gets a manager named .objects.

python

CopyEdit

class Book(models.Model):

title = models.CharField(max\_length=100)

# Usage:

books = Book.objects.all()

Here, objects is the default manager. You can replace or extend it to return **filtered querysets** by default.

### 🔹 Why create a custom manager?

Let’s say you have a model with a Boolean field like is\_published, and you only want to work with published items most of the time.

#### ✅ Default Manager:

python

CopyEdit

class Article(models.Model):

title = models.CharField(max\_length=100)

is\_published = models.BooleanField(default=False)

# This gives you ALL articles:

Article.objects.all()

But let’s say **you want to get only published articles** by default. Here’s how:

### 🔹 How to define a custom manager

python

CopyEdit

class PublishedManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset().filter(is\_published=True)

Now use it inside your model:

python

CopyEdit

class Article(models.Model):

title = models.CharField(max\_length=100)

is\_published = models.BooleanField(default=False)

objects = models.Manager() # Default manager

published = PublishedManager() # Custom manager

Now:

python

CopyEdit

Article.objects.all() # Returns all articles

Article.published.all() # Returns only published ones!

This is **exactly** how Soft Delete also works — by customizing get\_queryset() to hide deleted rows.

### 🔹 BONUS: Custom QuerySets

You can make your custom **QuerySet** class to add more methods:

python

CopyEdit

class ArticleQuerySet(models.QuerySet):

def published(self):

return self.filter(is\_published=True)

class Article(models.Model):

title = models.CharField(max\_length=100)

is\_published = models.BooleanField(default=False)

objects = ArticleQuerySet.as\_manager()

Then you can do:

python

CopyEdit

Article.objects.published()

## 🔁 2. ****ManyToMany in Django****

### 🧠 What does it mean?

Let’s say:

* A student can enroll in **many** courses.
* A course can have **many** students.

This is a **Many-to-Many** relationship.

### 🔹 Simple way:

python

CopyEdit

class Student(models.Model):

name = models.CharField(max\_length=100)

class Course(models.Model):

title = models.CharField(max\_length=100)

students = models.ManyToManyField(Student)

Now Django creates a hidden **intermediate table** automatically behind the scenes:

sql

CopyEdit

course\_students(course\_id, student\_id)

You can use:

python

CopyEdit

course = Course.objects.get(id=1)

course.students.all() # All students in the course

student = Student.objects.get(id=1)

student.course\_set.all() # All courses this student enrolled in

### 🔹 Customizing the Intermediate Table

Suppose you want to **track more info** about the relationship — e.g., when the student enrolled.

You need to create a model for the intermediate table.

python

CopyEdit

class Enrollment(models.Model):

student = models.ForeignKey(Student, on\_delete=models.CASCADE)

course = models.ForeignKey(Course, on\_delete=models.CASCADE)

date\_enrolled = models.DateField()

Now connect it using through=:

python

CopyEdit

class Course(models.Model):

title = models.CharField(max\_length=100)

students = models.ManyToManyField(Student, through='Enrollment')

Now you **can't** use course.students.add() anymore, because you must create an Enrollment manually:

python

CopyEdit

Enrollment.objects.create(student=student\_obj, course=course\_obj, date\_enrolled="2024-05-01")

## 👤 3. ****Custom User Model in Django (Over Default**** User****)****

### ❗Why override Django's default User model?

The default User has:

* username, first\_name, last\_name, email, password

But maybe:

* You want to use **email instead of username**
* Or add fields like phone\_number, is\_verified, etc.

### 🔹 Option 1: ****Extend with OneToOneField (NOT recommended)****

This is a "profile" model that extends the existing user:

python

CopyEdit

from django.contrib.auth.models import User

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

phone\_number = models.CharField(max\_length=20)

But this **does not change the User model itself**. You’ll always have two models.

### 🔹 Option 2: ****Custom User Model (Best Practice)****

You make your **own User model from scratch**, inheriting from AbstractBaseUser or AbstractUser.

#### 👉 Simple example using AbstractUser (inherits all functionality):

python

CopyEdit

from django.contrib.auth.models import AbstractUser

from django.db import models

class CustomUser(AbstractUser):

phone\_number = models.CharField(max\_length=20, blank=True, null=True)

Now in settings.py, tell Django to use it:

python

CopyEdit

AUTH\_USER\_MODEL = 'yourapp.CustomUser'

Then create and run migrations.

### 🔹 Want to use Email instead of Username?

Use AbstractBaseUser + BaseUserManager:

python

CopyEdit

from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin

class CustomUserManager(BaseUserManager):

def create\_user(self, email, password=None, \*\*extra\_fields):

if not email:

raise ValueError("Email required")

email = self.normalize\_email(email)

user = self.model(email=email, \*\*extra\_fields)

user.set\_password(password)

user.save(using=self.\_db)

return user

def create\_superuser(self, email, password=None, \*\*extra\_fields):

extra\_fields.setdefault("is\_staff", True)

extra\_fields.setdefault("is\_superuser", True)

return self.create\_user(email, password, \*\*extra\_fields)

python

CopyEdit

class CustomUser(AbstractBaseUser, PermissionsMixin):

email = models.EmailField(unique=True)

is\_active = models.BooleanField(default=True)

is\_staff = models.BooleanField(default=False)

objects = CustomUserManager()

USERNAME\_FIELD = 'email'

REQUIRED\_FIELDS = [] # For createsuperuser

## 🧠 Summary

| Concept | Key Idea |
| --- | --- |
| Custom Manager | Let you control .objects to return filtered querysets |
| ManyToMany | Lets both sides of the relation have multiple records |
| Custom User | Best practice for modern apps. Use AbstractUser or AbstractBaseUser |

Perfect. You’re ready to move beyond basics — now it’s time to **solidify your understanding** of **Python internals**, **tricky syntax**, and **interview-level concepts**.

Let’s start with your example and then go into a **list of deep Python questions** (with explanations).

## 🔍 a += 1 vs a = a + 1 in Python

### ✅ Similarity:

Both increment a by 1, and for **immutable types like** int**,** str**, tuple**, they behave the **same way**.

python

CopyEdit

a = 5

a += 1 # a = 6

a = a + 1 # a = 7

### ⚠️ Difference (for ****mutable objects**** like lists):

python

CopyEdit

a = [1, 2]

b = a

a += [3] # modifies in-place

print(a) # [1, 2, 3]

print(b) # [1, 2, 3] ✅ same reference

a = a + [4] # creates a new list

print(a) # [1, 2, 3, 4]

print(b) # [1, 2, 3] ❌ b unchanged

**Why?**

* += calls the **in-place method** (\_\_iadd\_\_)
* a = a + [4] creates a **new list object** using \_\_add\_\_

## 💡 Deep Python Questions That Test True Understanding

Here’s a curated list of questions you should know with real-world dev depth.

### 1. ****Mutable Default Arguments****

python

CopyEdit

def append\_item(item, my\_list=[]):

my\_list.append(item)

return my\_list

print(append\_item(1)) # [1]

print(append\_item(2)) # [1, 2] ❗

**Why does it keep the old list?**

Because default arguments are evaluated **once**, not every time the function is called.

🔧 Fix:

python

CopyEdit

def append\_item(item, my\_list=None):

if my\_list is None:

my\_list = []

### 2. ****is vs ==****

python

CopyEdit

a = 1000

b = 1000

print(a == b) # True

print(a is b) # False ❗

* == checks **value**
* is checks **identity** (memory address)

For small integers (-5 to 256), Python caches them. So:

python

CopyEdit

x = 5

y = 5

print(x is y) # True ✅

### 3. ****List comprehension scope****

python

CopyEdit

x = 5

print([x for x in range(3)])

print(x) # 5 in Python 3 ✅, 2 in Python 2 ❌

List comprehension has its own scope in Python 3.

### 4. ****Generators vs List Comprehension****

python

CopyEdit

a = [x\*x for x in range(1000)]

b = (x\*x for x in range(1000))

* a is a **list in memory**
* b is a **generator** — lazy-loaded, memory-efficient

### 5. ****Why is**** None ****not the same as**** False

python

CopyEdit

a = None

if a:

print("yes")

else:

print("no") # prints "no"

* None is a singleton object
* Only False, 0, '', [], {} and None are falsy

### 6. \*\*Unpacking and \*args, ****kwargs****

python

CopyEdit

def func(a, b, \*args, \*\*kwargs):

pass

func(1, 2, 3, 4, x=5, y=6)

# args = (3, 4)

# kwargs = {'x': 5, 'y': 6}

Unpacking:

python

CopyEdit

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

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

### 7. ****LEGB Rule (Scope)****

Python looks for variables in:

* **L**ocal
* **E**nclosing
* **G**lobal
* **B**uiltins

python

CopyEdit

x = "global"

def outer():

x = "enclosed"

def inner():

x = "local"

print(x)

inner()

outer() # prints local

### 8. ****Decorators****

A function that wraps another function.

python

CopyEdit

def decorator(func):

def wrapper(\*args, \*\*kwargs):

print("Before call")

result = func(\*args, \*\*kwargs)

print("After call")

return result

return wrapper

@decorator

def say\_hello():

print("Hello")

say\_hello()

### 9. ****Class vs Instance Variables****

python

CopyEdit

class Test:

count = 0

def \_\_init\_\_(self):

self.data = []

a = Test()

b = Test()

a.data.append(1)

print(b.data) # []

print(Test.count) # 0

Test.count += 1

print(a.count) # 1

* self.var → instance-specific
* Class.var → shared

### 10. ****Walrus Operator (:=)****

Introduced in Python 3.8

python

CopyEdit

while (n := input("Enter: ")) != "q":

print(n)

Assigns and compares in one line.

## 🔥 Pro Interview Questions That Test Your Python Brain

| Topic | Sample Question |
| --- | --- |
| Interning | Why are a is b True for a = "hello" and b = "hello"? |
| Copying | Difference between copy.copy() and copy.deepcopy()? |
| Iterators | What makes a class iterable? \_\_iter\_\_() vs \_\_next\_\_() |
| Closures | What are closures and how are they used in Python? |
| Dunder Methods | What is \_\_repr\_\_ vs \_\_str\_\_? What is \_\_eq\_\_, \_\_lt\_\_? |
| Data Classes | Difference between regular class and @dataclass? |
| GIL | What is the Global Interpreter Lock and why does it matter? |
| Coroutines | Difference between Generator and Coroutine (async def) |

## ⚡ Next Steps

If you want, I can:

* Quiz you daily with these types of questions
* Give you one concept per day with explanation and code
* Help you build small real-world Python tools to reinforce concepts

Want to start with **one of those**? Or should I give you a full PDF-style doc covering all deep topics?

## ****1. Interning in Python: Why**** a is b ****is True for**** a = "hello"

### ****What is Interning?****

**Interning** means Python **reuses memory** for **some immutable objects** like:

* Small integers (from -5 to 256)
* Short strings (e.g., "hello", "python")
* Common identifiers

This is an **optimization** for speed and memory.

### ****Example:****

python

CopyEdit

a = "hello"

b = "hello"

print(a == b) # True → values are equal

print(a is b) # True → same memory object

### ****But not always!****

python

CopyEdit

a = "hello world, this is Vishnu Kurup"

b = "hello world, this is Vishnu Kurup"

print(a == b) # True

print(a is b) # False ❗️new object

Long or complex strings are usually **not interned**.

### ****Force Interning****

You can **force interning** using:

python

CopyEdit

import sys

a = sys.intern("long string Vishnu Kurup")

b = sys.intern("long string Vishnu Kurup")

print(a is b) # True ✅

### ****Numbers too:****

python

CopyEdit

x = 100

y = 100

print(x is y) # True (cached small int)

x = 1000

y = 1000

print(x is y) # False (new object)

### ****Why should you care?****

* Use == for value checks, **never** is **for equality**
* is is for **identity** (memory), like:

python

CopyEdit

if x is None: # ✅ correct

### ****Interview-style Trick Question****

python

CopyEdit

a = "py" + "thon"

b = "python"

print(a is b) # True ✅ if both constants

a = "py"

b = a + "thon"

print(b is "python") # False ❌ runtime concat

Let me know if this was clear — and then I’ll move to **Part 2:** copy.copy() **vs** copy.deepcopy() next.

Would you like me to create a document or track all of these as we go?

##### You said:

Yeah go on to nxt

###### ChatGPT said:

Awesome — let’s move on to **Part 2: Copying in Python**.

## ****2.**** copy.copy() ****vs**** copy.deepcopy()

### ****Why do we even need this?****

Python variables **point to memory**. If you just do:

python

CopyEdit

a = [1, 2, [3, 4]]

b = a

Then a **and** b **refer to the same object**. Change one — both change.

### ****What is**** copy.copy()****? (Shallow Copy)****

python

CopyEdit

import copy

a = [1, 2, [3, 4]]

b = copy.copy(a)

print(a == b) # True → values match

print(a is b) # False → different object

BUT:

python

CopyEdit

b[2][0] = 999

print(a) # [1, 2, [999, 4]] ❌ Inner list is still shared!

* **Shallow copy** only clones the top-level object.
* **Nested objects are still shared** (pointers copied).

### ****What is**** copy.deepcopy()****? (Deep Copy)****

python

CopyEdit

import copy

a = [1, 2, [3, 4]]

b = copy.deepcopy(a)

b[2][0] = 999

print(a) # [1, 2, [3, 4]] ✅

* **Deep copy recursively copies everything**, even nested stuff.

### ****Analogy****

| Copy Type | What gets copied | What's still shared |
| --- | --- | --- |
| copy.copy() | Outer object | Inner nested objects |
| copy.deepcopy() | Outer + inner (all levels) | Nothing |

### ****Trick Interview Question:****

python

CopyEdit

a = [[1, 2], [3, 4]]

b = a[:]

b[0][0] = 99

print(a) # [[99, 2], [3, 4]] → Why?

**Answer:** a[:] is a shallow copy, a[0] and b[0] point to the same nested list.

Great! Let’s jump into **Part 3: Iterators in Python**

## ****3. Iterators and Iterables:**** \_\_iter\_\_() ****vs**** \_\_next\_\_()

### ****Basic Idea****

* An **iterable** is something you can loop over (for item in x).
* An **iterator** is the actual object that knows how to **yield the next item**.

### ****What is an Iterable?****

If an object has \_\_iter\_\_(), it’s iterable.

python

CopyEdit

x = [1, 2, 3] # list is iterable

it = iter(x) # returns an iterator

print(next(it)) # 1

### ****What is an Iterator?****

An object is an **iterator** if it:

* Has \_\_iter\_\_() that returns self
* Has a \_\_next\_\_() method

python

CopyEdit

class Count:

def \_\_init\_\_(self):

self.num = 0

def \_\_iter\_\_(self): # makes it iterable

return self

def \_\_next\_\_(self): # makes it an iterator

if self.num < 3:

self.num += 1

return self.num

else:

raise StopIteration

c = Count()

for i in c:

print(i)

**Output:**

CopyEdit

1

2

3

### ****So what’s the difference?****

| Method | Purpose |
| --- | --- |
| \_\_iter\_\_() | Returns the iterator object itself |
| \_\_next\_\_() | Returns the next value |

### ****Under the Hood****

python

CopyEdit

for i in some\_iterable:

print(i)

This is what Python really does:

python

CopyEdit

it = iter(some\_iterable)

while True:

try:

item = next(it)

print(item)

except StopIteration:

break

### ****Trick Interview Question****

python

CopyEdit

x = [1, 2, 3]

y = iter(x)

print(next(y)) # 1

print(next(y)) # 2

for i in y:

print(i) # What prints?

**Answer:** Only 3 prints, because y has already advanced past 1 and 2.

## ****PART 1 — LISTS in Python****

### 1. ****What is a List?****

* An ordered, mutable (changeable) collection.
* Defined using square brackets:

python

CopyEdit

nums = [1, 2, 3]

### 2. ****Key Properties****

| Feature | Detail |
| --- | --- |
| Ordered | Preserves insertion order |
| Mutable | You can change values |
| Heterogeneous | Can hold multiple data types |
| Dynamic | Resizable automatically |

### 3. ****Important List Operations****

python

CopyEdit

nums = [1, 2, 3]

nums.append(4) # [1, 2, 3, 4]

nums.insert(1, 10) # [1, 10, 2, 3, 4]

nums.pop() # Removes last item

nums.remove(10) # Removes value 10

nums.sort() # Sorts in-place

nums.reverse() # Reverses in-place

### 4. ****List Copying Pitfall****

python

CopyEdit

a = [1, 2, 3]

b = a

b[0] = 100

print(a) # [100, 2, 3] -- Because they point to same list

To avoid that:

python

CopyEdit

c = a.copy() # or list(a)

### 5. ****List Comprehensions****

python

CopyEdit

squares = [x\*x for x in range(5)] # [0, 1, 4, 9, 16]

evens = [x for x in range(10) if x % 2 == 0]

### 6. ****Mutable Defaults (Deadly Interview Bug)****

python

CopyEdit

def append\_to\_list(value, my\_list=[]):

my\_list.append(value)

return my\_list

print(append\_to\_list(1)) # [1]

print(append\_to\_list(2)) # [1, 2] — Why??

# FIX:

def safe\_append(value, my\_list=None):

if my\_list is None:

my\_list = []

my\_list.append(value)

return my\_list

### 7. ****Aliasing vs Copying****

python

CopyEdit

x = [[1, 2], [3, 4]]

y = x.copy() # Shallow copy

y[0][0] = 100

print(x) # [[100, 2], [3, 4]] — because inner lists are shared

Use:

python

CopyEdit

import copy

z = copy.deepcopy(x)

### 8. ****Tricky Interview Question****

python

CopyEdit

a = [1, 2, 3]

b = a

a += [4]

print(a) # [1, 2, 3, 4]

print(b) # [1, 2, 3, 4]

Now compare with:

python

CopyEdit

a = [1, 2, 3]

b = a

a = a + [4]

print(a) # [1, 2, 3, 4]

print(b) # [1, 2, 3]

**Why?**

* a += [4] mutates the list
* a = a + [4] creates a new list

## ****PART 2 — DICTIONARIES (****dict****)****

### 1. ****What is a Dictionary?****

* An **unordered**, mutable collection of key-value pairs (as of Python 3.6+, dictionaries preserve insertion order).
* Think of it like a real-world dictionary:

python

CopyEdit

person = {'name': 'Alice', 'age': 25}

### 2. ****Core Properties****

| Feature | Value |
| --- | --- |
| Key-Value | You access values via keys |
| Mutable | Values can be changed |
| Keys Unique | Duplicate keys overwrite |
| Fast Access | O(1) time complexity for lookups |

### 3. ****Common Operations****

python

CopyEdit

d = {'a': 1, 'b': 2}

d['c'] = 3 # Add new key-value

d['a'] = 10 # Update existing key

del d['b'] # Delete key

val = d.get('z', 0) # Get with default fallback

keys = d.keys() # dict\_keys(['a', 'c'])

values = d.values()

items = d.items() # Key-value pairs

### 4. ****Dictionary Comprehension****

python

CopyEdit

squares = {x: x\*x for x in range(5)}

# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

### 5. ****Immutability of Keys****

Only **immutable** objects (e.g., strings, numbers, tuples) can be dictionary keys.  
You **can't** use lists or other dictionaries as keys.

python

CopyEdit

d = {[1,2]: "fail"} # ❌ TypeError

### 6. ****Interview Traps****

**Q1: What happens here?**

python

CopyEdit

x = {}

x[0] = "zero"

x[False] = "false"

print(x) # {0: 'false'}

**Why?**  
In Python, False == 0, so the second assignment **overwrites** the first.

**Q2: What happens here?**

python

CopyEdit

x = {[]} # ❌ TypeError: unhashable type: 'list'

Lists are mutable and can't be hashed, so can't be used as keys or set elements.

### 7. ****Mutable Defaults Trap****

python

CopyEdit

from collections import defaultdict

d = defaultdict(list)

d['a'].append(1)

print(d['a']) # [1]

### 8. ****How**** dict ****Works Internally****

* Uses a **hash table** under the hood.
* Keys are **hashed** to find the correct bucket.
* Lookups are O(1) on average.

### 9. setdefault() ****vs**** get()

python

CopyEdit

d = {}

d.setdefault('a', []).append(1)

print(d) # {'a': [1]}

setdefault is like get, but also **sets a default value** if the key is missing.

### 10. ****Merging Dicts****

python

CopyEdit

a = {'x': 1}

b = {'y': 2}

c = {\*\*a, \*\*b} # {'x': 1, 'y': 2}

In Python 3.9+:

python

CopyEdit

c = a | b

Awesome — let’s jump into:

## ✅ ****PART 3 — SETS****

### 1. ****What is a Set?****

* An **unordered** collection of **unique** elements.
* Similar to sets in mathematics.

python

CopyEdit

s = {1, 2, 3}

### 2. ****Key Features****

| Feature | Detail |
| --- | --- |
| Unique elements | Duplicates are auto-removed |
| Mutable | Can add/remove items |
| No indexing | Unordered, no s[0] access |
| Fast lookup | O(1) average time |

python

CopyEdit

s = {1, 2, 3, 2, 1}

print(s) # {1, 2, 3}

### 3. ****Common Operations****

python

CopyEdit

s.add(4) # Add element

s.remove(2) # Remove element (KeyError if missing)

s.discard(5) # Remove, no error if missing

s.pop() # Remove random element

s.clear() # Remove all elements

### 4. ****Set Operations (Like Math)****

python

CopyEdit

a = {1, 2, 3}

b = {3, 4, 5}

a | b # Union: {1, 2, 3, 4, 5}

a & b # Intersection: {3}

a - b # Difference: {1, 2}

a ^ b # Symmetric Diff: {1, 2, 4, 5}

### 5. ****Set Comprehension****

python

CopyEdit

evens = {x for x in range(10) if x % 2 == 0}

### 6. ****Frozen Set****

An **immutable version** of set.

python

CopyEdit

fs = frozenset([1, 2, 3])

# You can't add/remove from fs

## ✅ ****PART 4 — TUPLES****

### 1. ****What is a Tuple?****

* An **immutable**, ordered sequence of elements.

python

CopyEdit

t = (1, 2, 3)

### 2. ****Tuple Features****

| Property | Tuple |
| --- | --- |
| Ordered | Yes |
| Immutable | Yes |
| Allows dupes | Yes |
| Indexing | Yes: t[0] |
| Nesting | Yes |

### 3. ****Packing & Unpacking****

python

CopyEdit

t = 1, 2, 3 # Packing

a, b, c = t # Unpacking

python

CopyEdit

a, \*rest = (1, 2, 3, 4) # a = 1, rest = [2, 3, 4]

### 4. ****Single Element Tuple Trap****

python

CopyEdit

t = (1) # Not a tuple!

t = (1,) # ✅ This is a tuple

### 5. ****Tuple Uses****

* Function returns
* Dictionary keys
* Unpacking in loops or comprehensions
* Data integrity (data shouldn't change)

## ✅ BONUS: ****All Mutable vs Immutable Built-in Data Types in Python****

### ✅ ****Mutable (Can Be Changed)****

| Type | Example |
| --- | --- |
| list | [1, 2, 3] |
| dict | {'a': 1} |
| set | {1, 2, 3} |
| bytearray | bytearray(b'abc') |
| defaultdict | defaultdict(list) |
| deque | collections.deque() |

### ❌ ****Immutable (Cannot Be Changed)****

| Type | Example |
| --- | --- |
| int | 5 |
| float | 3.14 |
| bool | True, False |
| str | 'hello' |
| tuple | (1, 2) |
| frozenset | frozenset([1,2]) |
| bytes | b'abc' |
| complex | 3 + 4j |
| range | range(10) |

⚠️ Note: Immutable means the **object cannot be modified in-place**. But you can reassign a variable to point to a new object.

### 🧠 What is "context" in Django?

In Django, **context** is a Python dictionary that holds key-value pairs, where:

* **keys** are the variable names you want to use in your template
* **values** are the data you want to render

### ✅ Example (Django View + Template):

#### views.py:

python

CopyEdit

from django.shortcuts import render

def dashboard(request):

context = {

'username': 'Vishnu',

'notifications': 5,

}

return render(request, 'dashboard.html', context)

#### dashboard.html:

html

CopyEdit

<h1>Welcome, {{ username }}!</h1>

<p>You have {{ notifications }} new notifications.</p>

In this example:

* context = {'username': 'Vishnu', 'notifications': 5} is passed to the template
* Inside the HTML file, {{ username }} and {{ notifications }} access those values

### 🐍 What about "context" in Python (outside Django)?

* In **plain Python**, "context" can refer to the environment or scope in which a block of code runs.
* Example: Using with statement (context manager):

python

CopyEdit

with open('file.txt', 'r') as f:

content = f.read()

Here, the with statement creates a **context** where the file is open. Once the block is done, the file is closed automatically — this is known as a **context manager**.

### TL;DR

| Term | Meaning |
| --- | --- |
| Django Context | Dictionary sent from view to template |
| Python Context | Scope or environment (e.g., in a with block using context managers) |

Would you like to dive deeper into either of these, like how context processors work in Django?

4o

##### You said:

Url shortener

###### ChatGPT said:

A **URL shortener** is a tool that converts a long URL (like https://example.com/products/category/item?id=12345) into a shorter version (like https://short.ly/abc123). These are useful for sharing, especially on platforms with character limits.

### 🚀 Let's build a ****basic URL shortener**** in Django!

Here’s an overview:

#### 🧱 Models:

python

CopyEdit

# models.py

from django.db import models

import string, random

def generate\_short\_code():

return ''.join(random.choices(string.ascii\_letters + string.digits, k=6))

class URL(models.Model):

original\_url = models.URLField()

short\_code = models.CharField(max\_length=6, unique=True, default=generate\_short\_code)

created\_at = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return f"{self.short\_code} -> {self.original\_url}"

#### 🧠 Views:

python

CopyEdit

# views.py

from django.shortcuts import render, redirect, get\_object\_or\_404

from .models import URL

from django.http import HttpResponse

def home(request):

if request.method == 'POST':

original\_url = request.POST['original\_url']

url\_obj = URL.objects.create(original\_url=original\_url)

return render(request, 'home.html', {'short\_url': request.build\_absolute\_uri('/') + url\_obj.short\_code})

return render(request, 'home.html')

def redirect\_url(request, short\_code):

url\_obj = get\_object\_or\_404(URL, short\_code=short\_code)

return redirect(url\_obj.original\_url)

#### 🌐 URLs:

python

CopyEdit

# urls.py

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'),

path('<str:short\_code>/', views.redirect\_url, name='redirect\_url'),

]

#### 🖼️ Template (home.html):

html

CopyEdit

<form method="POST">

{% csrf\_token %}

<input type="url" name="original\_url" placeholder="Enter URL" required>

<button type="submit">Shorten</button>

</form>

{% if short\_url %}

<p>Shortened URL: <a href="{{ short\_url }}">{{ short\_url }}</a></p>

{% endif %}

### ✅ How it works:

1. User submits a long URL via the form.
2. Django generates a short code (6 characters).
3. The short code + base URL becomes the shortened URL.
4. When someone accesses /abc123, Django redirects to the original URL.

Generators in Python are **special types of iterators** that allow you to **generate values on the fly** using yield instead of storing them in memory all at once. They are useful for **efficient memory usage and lazy evaluation**, especially when working with large data sets or infinite sequences.

### 🔧 How Generators Work

A **generator function** looks like a normal function, but it uses the yield keyword to return values **one at a time**, pausing between each and resuming from where it left off.

#### ✅ Example:

python

CopyEdit

def count\_up\_to(max):

count = 1

while count <= max:

yield count

count += 1

python

CopyEdit

counter = count\_up\_to(3)

print(next(counter)) # 1

print(next(counter)) # 2

print(next(counter)) # 3

# next(counter) would raise StopIteration

### ✅ Key Points:

| Concept | Description |
| --- | --- |
| yield | Used to produce a value and pause the generator. State is saved. |
| Generator Function | Function with at least one yield. |
| Lazy Evaluation | Values are produced only when requested. |
| Memory Efficient | Doesn’t store all values in memory (like lists or tuples). |
| Iterable | Can be used in loops (for, next()). |

### 🔁 Generator vs List

#### List:

python

CopyEdit

[ x \* x for x in range(1000000) ] # Uses a lot of memory

#### Generator:

python

CopyEdit

(x \* x for x in range(1000000)) # Uses very little memory

### 📦 Real-Life Uses of Generators

* Reading large files line-by-line.
* Streaming data from APIs or databases.
* Producing infinite sequences (e.g., Fibonacci, primes).
* Pipelines for data processing.

## ✅ Visual Explanation of Generators

Imagine a generator as a **"pause-and-resume" machine**:

python

CopyEdit

def simple\_gen():

print("Start")

yield 1

print("Middle")

yield 2

print("End")

When you use it:

python

CopyEdit

g = simple\_gen()

next(g)

# Output:

# Start

# => 1

next(g)

# Output:

# Middle

# => 2

next(g)

# Output:

# End

# => StopIteration

Each yield:

* Pauses the function.
* Returns a value.
* Remembers where it left off.

## 🔄 Generator vs Async Generator

| Feature | Generator | Async Generator |
| --- | --- | --- |
| Syntax | def + yield | async def + yield |
| Used with | for, next() | async for, await |
| Best for | CPU-bound, synchronous work | I/O-bound, asynchronous work |

### 🧵 Async Generator Example

python

CopyEdit

import asyncio

async def async\_gen():

for i in range(3):

await asyncio.sleep(1) # simulate I/O

yield i

async def main():

async for num in async\_gen():

print(num)

# To run:

# asyncio.run(main())

## 🧠 Generators vs Coroutines

| Concept | Generator | Coroutine |
| --- | --- | --- |
| Purpose | Generate sequence of values lazily | Pause/resume functions for concurrency |
| Keyword | yield | await |
| Use case | Memory-efficient iteration | Non-blocking async programming |
| Example use | Read big files, streaming | Async HTTP calls, DB I/O |

### 🌐 Coroutine Example

python

CopyEdit

import asyncio

async def greet():

await asyncio.sleep(1)

return "Hello!"

async def main():

result = await greet()

print(result)

# asyncio.run(main())

## 🧠 Summary

* **Generators**: Great for streaming data, memory efficiency.
* **Async Generators**: Like generators, but used with await, for asynchronous data streaming.
* **Coroutines**: Functions for asynchronous tasks, often handling I/O operations efficiently.

## ✅ 1. ****CSRF Token Exempt in Django****

### 🔐 What is CSRF?

**CSRF (Cross-Site Request Forgery)** is a type of attack where a malicious website tricks a user’s browser into making unintended requests to your app **while the user is authenticated**.

Django protects against CSRF attacks by requiring a **CSRF token** on POST, PUT, DELETE requests (i.e., unsafe methods).

### ❌ What is @csrf\_exempt?

It's a **Django decorator** used to **disable CSRF protection** on a particular view.

python

CopyEdit

from django.views.decorators.csrf import csrf\_exempt

from django.http import HttpResponse

@csrf\_exempt

def my\_view(request):

return HttpResponse("CSRF check skipped!")

### 🧨 When to use?

* When you're handling **API requests** from external clients (like mobile apps or third-party integrations).
* When you're using **token-based authentication** (e.g., JWT), which **does not rely on cookies**.

⚠️ **Don't use** @csrf\_exempt **unless you understand the security risk**. It makes your view vulnerable to CSRF if the client uses cookies for authentication.

### ✅ Alternative: Use @csrf\_protect

python

CopyEdit

from django.views.decorators.csrf import csrf\_protect

@csrf\_protect

def safe\_view(request):

...

Or for class-based views:

python

CopyEdit

from django.utils.decorators import method\_decorator

@method\_decorator(csrf\_exempt, name='dispatch')

class MyApiView(View):

...

## 🐒 2. ****Monkey Patching in Python****

### 🧠 What is Monkey Patching?

**Monkey patching** means **modifying or extending code at runtime** without changing the original source code. It's often used to fix or alter behavior temporarily.

### ✅ Example:

python

CopyEdit

class Dog:

def speak(self):

return "Woof"

dog = Dog()

print(dog.speak()) # Woof

# 🐒 Monkey patching the method

def new\_speak():

return "Meow"

Dog.speak = new\_speak

print(dog.speak()) # Meow

We changed how Dog.speak() works **without touching** the original class!

### 🧰 Use Cases

* **Testing**: Replace a method with a fake (mock) during tests.
* **Debugging**: Inject logging or fix a bug without changing external libraries.
* **Dynamic behavior**: Patch third-party libraries at runtime.

### ⚠️ Dangers of Monkey Patching

* Hard to debug.
* Breaks encapsulation.
* Can introduce unexpected bugs, especially in large teams or codebases.

### 🔐 Monkey Patching in Real Life Example (Django):

Imagine you want to patch a Django method for debugging:

python

CopyEdit

from django.db import models

original\_save = models.Model.save

def patched\_save(self, \*args, \*\*kwargs):

print("Saving:", self)

return original\_save(self, \*args, \*\*kwargs)

models.Model.save = patched\_save

Now **every model's save() method** prints a message before saving.

## 🔄 Reversing with ****List Slicing****

python

CopyEdit

a = [10, 20, 30, 40, 50]

reversed\_list = a[::-1]

print(reversed\_list) # [50, 40, 30, 20, 10]

### 📌 Why does [::-1] reverse the list?

* **Syntax:** a[start:stop:step]
* You're saying:
  + Start from the end (default when step is negative)
  + Go until the start (default stop is before index 0)
  + Step by -1 → move backwards.

## 🔁 Reversing with range()

python

CopyEdit

for i in range(5, 0, -1):

print(i)

### 💡 What happens here?

* range(start=5, stop=0, step=-1)
* You get: 5, 4, 3, 2, 1
* The stop value **is not included**, so it stops **before** reaching 0.

If you want to reverse a list with range():

python

CopyEdit

a = [10, 20, 30, 40, 50]

for i in range(len(a) - 1, -1, -1):

print(a[i])

This prints the list in reverse:

CopyEdit

50

40

30

20

10

## 🧠 Summary

| Method | How It Works | Example |
| --- | --- | --- |
| a[::-1] | Slice backwards with step = -1 | a = [1,2,3] → [3,2,1] |
| range(5, 0, -1) | Count backward from 5 to 1 | → 5,4,3,2,1 |
| range(len(a)-1, -1, -1) | Index backward in a list | Loop through list in reverse |

## 🧠 1. Built-in Functions (Global Scope)

These work on many data types.

| Function | Description | Example |
| --- | --- | --- |
| len() | Get length | len([1, 2, 3]) → 3 |
| type() | Get data type | type(5) → <class 'int'> |
| str() | Convert to string | str(123) → '123' |
| int() | Convert to int | int("123") → 123 |
| float() | Convert to float | float("3.14") → 3.14 |
| sum() | Sum of elements | sum([1,2,3]) → 6 |
| max() | Max value | max([1,5,3]) → 5 |
| min() | Min value | min([1,5,3]) → 1 |
| sorted() | Sorts elements (returns new list) | sorted([3,1,2]) → [1,2,3] |
| reversed() | Returns reversed iterator | list(reversed([1,2])) |
| range() | Sequence of numbers | list(range(3)) → [0,1,2] |
| enumerate() | Index + item in loop | for i,v in enumerate(...) |
| zip() | Combine iterables | zip([1,2],[‘a’,’b’]) |
| map() | Apply function | map(str, [1, 2]) → '1','2' |
| filter() | Filter items | filter(lambda x:x>2,[1,3]) |
| any() | True if any True | any([False, True]) |
| all() | True if all True | all([True, True]) |

## 📋 2. List Methods (list.)

python

CopyEdit

a = [10, 20, 30]

| Method | Description | Example |
| --- | --- | --- |
| append(x) | Add at end | a.append(40) |
| extend(b) | Add all from b | a.extend([50, 60]) |
| insert(i,x) | Add at position | a.insert(1, 15) |
| remove(x) | Remove first x | a.remove(20) |
| pop() | Remove & return last | a.pop() → 30 |
| index(x) | Find index of x | a.index(10) → 0 |
| count(x) | Count x | a.count(20) → 1 |
| sort() | Sort in place | a.sort() |
| reverse() | Reverse in place | a.reverse() |
| copy() | Shallow copy | b = a.copy() |
| clear() | Empty the list | a.clear() |

## 🔑 3. Dictionary Methods (dict.)

python

CopyEdit

d = {'a': 1, 'b': 2}

| Method | Description | Example |
| --- | --- | --- |
| get(k) | Get value or None | d.get('a') → 1 |
| keys() | All keys | d.keys() → dict\_keys(...) |
| values() | All values | d.values() |
| items() | Key-value pairs | d.items() |
| update(d2) | Merge d2 into d | d.update({'c':3}) |
| pop(k) | Remove and return k's value | d.pop('b') → 2 |
| popitem() | Remove last item (3.7+) | d.popitem() |
| setdefault(k,v) | Return d[k], or set d[k]=v | d.setdefault('x', 0) |
| clear() | Remove all items | d.clear() |
| copy() | Copy dict | d.copy() |

## 📌 4. Set Methods (set())

python

CopyEdit

s = {1, 2, 3}

| Method | Description |
| --- | --- |
| add(x) | Add x |
| remove(x) | Remove x, error if not found |
| discard(x) | Remove x, no error if missing |
| pop() | Remove random element |
| union(s2) | All elements |
| intersection(s2) | Common elements |
| difference(s2) | In s but not s2 |
| symmetric\_difference(s2) | In one but not both |

## 🧮 5. Tuple (tuple())

* Immutable list.
* Supports indexing, slicing, etc.
* No methods like append or remove.

## ✅ Tips for len() & Similar

* len(x) → Works on list, dict, tuple, str, set, etc.
* type(x) → Use to debug what's inside.
* isinstance(x, list) → Check type before calling list methods.

## 🔍 Visual Summary

python

CopyEdit

# Built-in

len(), max(), min(), sum(), sorted(), any(), all()

# List

append(), insert(), pop(), remove(), sort(), reverse()

# Dict

get(), items(), keys(), values(), update(), pop()

# Set

add(), remove(), union(), intersection(), difference()

# Other

range(), map(), filter(), zip(), enumerate()

## 🔒 1. ****Security Middleware****

| Middleware | Purpose |
| --- | --- |
| SecurityMiddleware | Enables several security features (HSTS, SSL redirect, etc.) |
| XFrameOptionsMiddleware | Prevents clickjacking by setting X-Frame-Options header |
| CsrfViewMiddleware | Protects against Cross-Site Request Forgery |
| CommonMiddleware | Adds support for things like APPEND\_SLASH, trailing slash handling |

## 👤 2. ****Session & Authentication Middleware****

| Middleware | Purpose |
| --- | --- |
| SessionMiddleware | Enables session support (stores sessions in DB or other backends) |
| AuthenticationMiddleware | Associates users with requests using sessions |
| MessageMiddleware | Enables cookie- or session-based flash messages via messages framework |

## 🧠 3. ****Locale & Timezone Middleware****

| Middleware | Purpose |
| --- | --- |
| LocaleMiddleware | Enables language selection using the Accept-Language header |
| TimezoneMiddleware | Automatically activates user-specific time zones |

## 🧹 4. ****Response/Request Processing Middleware****

| Middleware | Purpose |
| --- | --- |
| GZipMiddleware | Compresses responses with GZip for performance |
| ConditionalGetMiddleware | Adds support for HTTP 304 (Not Modified) based on ETag/Last-Modified |
| CommonMiddleware | Adds content headers, redirects with/without slash, etc. |

## 🔌 5. ****Others****

| Middleware | Purpose |
| --- | --- |
| BrokenLinkEmailsMiddleware | Emails you 404 errors if configured (useful in prod) |
| AdminEmailHandler (logging handler) | Sends error logs via email to admins |
| StaticFilesMiddleware (dev only) | Serves static files in development (not in production) |

## 🧱 Example MIDDLEWARE Setting

python

CopyEdit

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

## ✅ JWT Is Not Built-In

JWT (JSON Web Tokens) is handled by third-party middleware, such as:

* djangorestframework-simplejwt
* django-rest-framework-jwt (older)

These are typically added to your DRF authentication classes, not MIDDLEWARE.

### 🧠 1. ****Monkey Patching at the Class Level**** → Affects ****All Instances****

If you patch a method **on the class**, every existing and future instance will use the new method.

python

CopyEdit

class Dog:

def bark(self):

return "woof"

def new\_bark(self):

return "meow"

Dog.bark = new\_bark # 👈 Monkey patch at class level

d1 = Dog()

d2 = Dog()

print(d1.bark()) # 👉 "meow"

print(d2.bark()) # 👉 "meow"

✔️ All instances now use the new method.

### 🐶 2. ****Monkey Patching at the Instance Level**** → Affects ****Only That Instance****

You can patch the method **only on one object**, using types.MethodType.

python

CopyEdit

import types

class Dog:

def bark(self):

return "woof"

def quiet\_bark(self):

return "..."

d1 = Dog()

d2 = Dog()

d1.bark = types.MethodType(quiet\_bark, d1) # 👈 Only d1 is patched

print(d1.bark()) # 👉 "..."

print(d2.bark()) # 👉 "woof"

✔️ Only d1 is affected; d2 remains unchanged.

### 🧪 Summary

| Patching Level | Affects | Example |
| --- | --- | --- |
| Class | All instances | Dog.bark = new\_func |
| Specific instance | Only that object | d1.bark = MethodType(...) |

Exactly — in **Python**, **there are no strict access controls** like private, protected, or final, and Python does **not prevent monkey patching** at runtime.

### 🔓 Python's Philosophy: "We are all consenting adults here."

This means:

* You **can** override methods and attributes — even **standard library functions**.
* You **won’t get an error** like you would in **Java** (e.g. trying to override a final method).
* **Static methods** and **class methods** in Python can still be monkey-patched — Python won’t stop you.

### 🧱 Comparison with Java and JavaScript:

| Feature | Java | JavaScript | Python |
| --- | --- | --- | --- |
| Method access control | Yes (private, final) | No strict enforcement | No strict enforcement |
| Monkey patching | ❌ Usually disallowed | ✅ Yes (for objects) | ✅ Yes (everywhere) |
| Static method safety | Locked (final static) | Can override dynamically | Can override dynamically |
| Runtime changes | ❌ Mostly static | ✅ Yes | ✅ Yes |

### 🔥 Example: Overriding a built-in method in Python

python

CopyEdit

print = lambda \*args: None # 🙃 Disable all prints

print("This won't show") # Does nothing

### 🧪 Here's how it works:

| Modifier Style | Name Format | Meaning in Python | Can Access From Outside? |
| --- | --- | --- | --- |
| **Public** | name | Default — accessible from anywhere | ✅ Yes |
| **Protected** (⚠️) | \_name | Convention only — "internal use" | ✅ Yes (but shouldn't) |
| **Private** (⚠️) | \_\_name | Name mangled to \_ClassName\_\_name | ❌ No (but still possible) |

### 🔓 Example:

python

CopyEdit

class Dog:

def \_\_init\_\_(self):

self.name = "Rocky" # Public

self.\_sound = "woof" # Protected (by convention)

self.\_\_secret = "I bite!" # Private (name mangling)

d = Dog()

print(d.name) # ✅ OK

print(d.\_sound) # ✅ Technically works, but avoid

# print(d.\_\_secret) # ❌ AttributeError

print(d.\_Dog\_\_secret) # ⚠️ HACKY way to access private var

### 😎 So what’s the takeaway?

* Python trusts the developer: “If you want to shoot yourself in the foot, go ahead.”
* These conventions (\_, \_\_) are respected by tools like linters, IDEs, and doc generators — but **Python won't block you**.
* In contrast, **Java** will enforce access control at compile time.

Here’s a **Python cheatsheet of commonly used methods and functions**, organized by **type** and including **global built-ins** like len(), and **object-specific methods** like list.copy() and str.upper().

### ✅ ****Global Built-in Functions (Work with most types)****

These are not methods, but functions available everywhere:

| Function | Description |
| --- | --- |
| len(x) | Returns length of x |
| type(x) | Returns type of x |
| id(x) | Returns memory address of x |
| str(x) | Converts x to string |
| int(x) | Converts x to integer |
| float(x) | Converts x to float |
| list(x) | Converts iterable x to list |
| set(x) | Converts iterable x to set |
| dict() | Creates empty dictionary |
| sorted(x) | Returns a sorted list from iterable |
| reversed(x) | Returns a reverse iterator |
| enumerate(x) | Returns index-value pairs from iterable |
| zip(a, b) | Pairs elements from two iterables |
| map(f, x) | Applies function to each item of iterable |
| filter(f, x) | Filters items using a function |
| any(x) | True if any element is True |
| all(x) | True if all elements are True |
| sum(x) | Returns the sum of elements in x |
| max(x) | Returns max element |
| min(x) | Returns min element |

### 📜 ****String Methods (****str****)****

python

CopyEdit

s = "vishnu"

| Method | Description |
| --- | --- |
| s.upper() | 'VISHNU' |
| s.lower() | 'vishnu' |
| s.capitalize() | 'Vishnu' |
| s.title() | 'Vishnu' |
| s.strip() | Remove surrounding whitespace |
| s.split() | Splits string into list by spaces (or delimiter) |
| s.join(list) | Joins list into a string with s as separator |
| s.replace(a, b) | Replace substring a with b |
| s.find("i") | Returns index of first match, -1 if not found |
| s.isdigit() | Checks if all characters are digits |
| s.isalpha() | Checks if all are alphabets |
| s.isalnum() | Checks if all are alphanumeric |

### 📋 ****List Methods (****list****)****

python

CopyEdit

l = [1, 2, 3]

| Method | Description |
| --- | --- |
| l.append(x) | Adds x to end |
| l.extend([x, y]) | Adds multiple elements |
| l.insert(i, x) | Inserts x at index i |
| l.remove(x) | Removes first occurrence of x |
| l.pop() | Removes and returns last item |
| l.pop(i) | Removes and returns item at index i |
| l.clear() | Removes all elements |
| l.index(x) | Finds index of first occurrence |
| l.count(x) | Number of times x appears |
| l.sort() | Sorts the list in-place |
| l.reverse() | Reverses the list in-place |
| l.copy() | Shallow copy of the list |

### 🔢 ****Set Methods (****set****)****

python

CopyEdit

s = {1, 2, 3}

| Method | Description |
| --- | --- |
| s.add(x) | Adds x to set |
| s.remove(x) | Removes x, error if not present |
| s.discard(x) | Removes x if present |
| s.pop() | Removes arbitrary element |
| s.clear() | Empties the set |
| s.union(t) | Elements in either set |
| s.intersection(t) | Elements in both |
| s.difference(t) | Elements in s but not in t |
| s.symmetric\_difference(t) | Elements in s or t but not both |

### 🧾 ****Dictionary Methods (****dict****)****

python

CopyEdit

d = {'a': 1, 'b': 2}

| Method | Description |
| --- | --- |
| d.get(k) | Returns value for key k |
| d.keys() | All keys |
| d.values() | All values |
| d.items() | All key-value pairs |
| d.update({...}) | Updates/merges dictionary |
| d.pop(k) | Removes and returns value at key k |
| d.popitem() | Removes and returns arbitrary key-value pair |
| d.clear() | Empties dictionary |
| d.copy() | Shallow copy |
| d.setdefault(k, v) | Inserts k with value v if not present |

### 🔍 ****Tuple Methods (****tuple****)****

python

CopyEdit

t = (1, 2, 3)

| Method | Description |
| --- | --- |
| t.count(x) | Count occurrences of x |
| t.index(x) | Index of x |

### 🔧 ****Object Utility / Static Methods****

You can define static or class methods in your custom classes:

python

CopyEdit

class Demo:

@staticmethod

def utility():

print("Static method")

@classmethod

def construct(cls):

return cls()

* @staticmethod: Doesn't access self or cls, like a utility.
* @classmethod: Uses cls, can construct or modify class state.

### ✨ Bonus: Type Conversion & Creation Shortcuts

| Expression | Meaning |
| --- | --- |
| list("abc") | ['a', 'b', 'c'] |
| dict([(1, 'a'), (2, 'b')]) | Convert list of tuples to dict |
| set("hello") | Unique letters: {'h', 'e', 'l', 'o'} |
| tuple([1,2,3]) | Convert list to tuple |

Here's your code:

python

CopyEdit

squares = []

for x in range(5):

squares.append(lambda: x\*\*2)

print(squares[2]())

print(squares[4]())

### ✅ What you expect:

You might think:

* squares[2]() should evaluate 2\*\*2 = 4
* squares[4]() should evaluate 4\*\*2 = 16

But what you **actually get** is:

CopyEdit

16

16

### ❓ Why?

The issue is **late binding** in Python closures.

#### Key Point:

**Lambdas (and functions) capture variables by reference, not by value.**

So, all the lambda: x\*\*2 functions **remember the same** x — and after the loop finishes, x == 4. Therefore, when you later call any of the lambdas, they all use x = 4, so x\*\*2 = 16.

### ✅ How to fix it

You need to **bind the current value of** x **at each iteration**. One way to do this is using a default argument in the lambda:

python

CopyEdit

squares = []

for x in range(5):

squares.append(lambda x=x: x\*\*2)

print(squares[2]()) # 4

print(squares[4]()) # 16

Now each lambda remembers its own x as a default argument — this works because default arguments are evaluated at definition time.

### Summary:

| Issue | Solution |
| --- | --- |
| Late binding of x | Use lambda x=x: x\*\*2 |

### 1. ✅ ****In a**** for ****loop**** – for ****iteration****

python

CopyEdit

for x in [1, 2, 3]:

print(x)

* This means: "For each x **in** the list [1, 2, 3], do something."
* Here, in is used to **loop over items** in a collection.

### 2. ✅ ****For membership checks**** – for ****validation / existence****

python

CopyEdit

if 3 in [1, 2, 3]:

print("Found it!")

* This means: "Is 3 **in** the list?"
* Here, in checks whether a value **exists in** a collection like a list, set, tuple, string, or dictionary keys.

### 🔍 Other Examples

#### Strings:

python

CopyEdit

if "cat" in "catalog":

print("Yes")

#### Sets:

python

CopyEdit

if "apple" in {"apple", "banana"}:

print("Exists")

#### Dictionaries (keys only):

python

CopyEdit

my\_dict = {"a": 1, "b": 2}

if "a" in my\_dict: # checks keys, not values

print("Key exists")

### 🔑 Summary:

| Use Case | Example | Meaning |
| --- | --- | --- |
| **Iteration** | for x in list: | Go through each element |
| **Validation** | if x in list: | Check if x is a member of list |

## 🐍 ****Python / Django: Using a Custom**** manage2.py

### ✅ What We Learned:

* You **can create a new** manage2.py file and use it in place of manage.py.
* It works as long as you **include the same logic**, especially:

python

CopyEdit

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'your\_project.settings')

from django.core.management import execute\_from\_command\_line

execute\_from\_command\_line(sys.argv)

* You can now run commands like:

bash

CopyEdit

python manage2.py runserver

python manage2.py migrate

* React's manage.py is just an **entry point** — as long as the structure is correct, its name doesn’t matter.

## ✅ What Are Migrations in Django?

Migrations in Django are Python files that describe **changes to your models** (i.e., your database schema). These changes are then applied to your **actual database**.

There are 3 key components:

* **Models**: Python classes representing your DB tables.
* **Migration files**: Version-controlled Python scripts for schema changes.
* **Database schema**: The actual table structure in PostgreSQL, MySQL, etc.

## 🔄 Django Migration Flow: Step-by-Step

1. **Change your model** → e.g., add a new field.
2. Run python manage.py makemigrations → creates a **migration file** that describes the change.
3. Run python manage.py migrate → applies that migration **to your database**.

## 🏷️ What Does “Marking a Migration” Mean?

"Marking a migration" means **telling Django that a migration has been applied**, without actually applying it to the database.

This is done using:

css

CopyEdit

python manage.py migrate appname migrationname --fake

For example:

bash

CopyEdit

python manage.py migrate inventory 0004\_auto --fake

This marks the migration as "done" in the migration history table (django\_migrations), **without executing any SQL** in the DB.

## 🧠 Why/When Do You “Fake” a Migration?

* You manually changed the DB schema (bad practice but sometimes necessary).
* You imported a production DB dump that already has the changes.
* You want to align the migration history with the schema **without applying the same changes again** (which would error out).

## 🔍 Under the Hood: How Django Tracks Migrations

Django keeps track of applied migrations using a table in your database:

sql

CopyEdit

SELECT \* FROM django\_migrations;

Each row = one applied migration, e.g., ("myapp", "0003\_add\_vendor\_field")

So when you run migrate, Django checks:

* Which migrations are missing in django\_migrations
* Applies only those, **in order**

## 🧩 Common Migration Mechanisms

| Command | What it Does |
| --- | --- |
| makemigrations | Generates a .py migration file from changes in models.py |
| migrate | Applies the migration to the DB |
| migrate --fake | Marks migration as applied without running SQL |
| migrate appname zero | Unapplies all migrations (resets DB schema of that app) |
| showmigrations | Shows all migrations and whether they’ve been applied |
| sqlmigrate | Shows the raw SQL Django would run for a migration |
| squashmigrations | Combines many migrations into one to clean up history |

## 🛠️ Advanced Case: Fake Initial Migration

Suppose you imported a pre-built DB schema, but no migrations exist yet. You can do:

bash

CopyEdit

python manage.py makemigrations myapp

python manage.py migrate myapp --fake-initial

This tells Django: “If the DB schema already matches this migration, just mark it as done.”

## ⚠️ Mismatch Scenarios & Errors

Here’s what happens when things go wrong:

| Error | Meaning | Fix |
| --- | --- | --- |
| column already exists | Migration is trying to add something already in DB | Use --fake |
| column does not exist | Migration expects a column that’s missing | Undo the migration or adjust the DB manually |
| no such table | Django thinks the table exists due to fake migration | Unfake (reset), or manually add the table |

## 🧾 Summary

* **Marking** = faking a migration as "applied" without executing changes.
* Only do this when you're sure your DB schema matches the migration.
* Always prefer letting Django manage migrations.
* Use --fake and --fake-initial carefully and only when absolutely needed.
* Never manually change the DB unless you're 100% sure — always use model + makemigrations + migrate.

Great question! Here’s a clear answer tailored to your situation:

### Do you need to create/apply migrations again after renaming the database?

\*\*No, you do not need to create or apply migrations again just because you renamed the database.\*\*

#### Why?

- \*\*Migrations are code files\*\* in your Django app (in the `migrations/` folders) that describe changes to your models over time.

- \*\*Applying migrations\*\* (`python manage.py migrate`) updates the actual database schema (tables, columns, etc.) to match your models.

- \*\*Renaming the database\*\* does not change the schema, data, or the migration history inside the database. It only changes the database’s name.

- As long as your `settings.py` now points to the new database name (`acq\_cdf\_new`), Django will continue to use the same tables and migration history as before.

#### What would require new migrations?

- Changing your models (adding/removing fields, tables, etc.).

- Deleting your database and creating a new, empty one (in which case you’d need to re-apply migrations to set up the schema).

---

### How does Django handle migrations? (Summary)

1. \*\*Migration Files:\*\*

- Located in each app’s `migrations/` directory.

- Created with `python manage.py makemigrations`.

- Describe changes to your models.

2. \*\*Applying Migrations:\*\*

- Run with `python manage.py migrate`.

- Django checks the `django\_migrations` table in your database to see which migrations have already been applied.

- Only unapplied migrations are run.

3. \*\*Migration State:\*\*

- The state is tracked in the database, not in the database name.

- As long as the schema and the `django\_migrations` table are intact, Django knows what’s been applied.

---

### What should you do now?

- Make sure your `settings.py` uses the new database name (`acq\_cdf\_new`).

- Run your app as usual.

- Only run `makemigrations`/`migrate` if you change your models or if you set up a new, empty database.

---

\*\*In summary:\*\*

Renaming the database does not require new migrations or re-applying migrations, as long as the schema and migration history are preserved. Just update your settings and continue working!

If you have any issues connecting or see migration errors, let me know!