Python Full-Stack Internship

15 Days · Django · MySQL · DRF · LangChain · Railway

Python 3.12 | Django 5 | DRF | MySQL | JWT | LangChain | Railway

---

Core2Cloud • core2cloud.in

Practical internship — 1 hour/day — real projects — industry workflow

Contents

DAY 1 · 1 HOUR

Python Setup & Git

Install Python 3, VS Code, venv, and Git — your entire development environment in one session.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Imagine 5 developers all editing the same `app.py` on WhatsApp. Priya adds login. Ravi adds payment. Ravi sends his file — Priya overwrites it. **Ravi's payment feature is gone forever.**

This happened at a Chennai startup in 2022 — they lost 3 weeks of work. **Git is the solution.** And Python's virtual environment (`venv`) means "it works on my machine" stops being an excuse forever.

⚙ What You Install Today (5–25 min)

💡 **THE 4 TOOLS**

Python 3.12 — The language. Run `python --version` to check.
VS Code — Editor. Add Python + Pylance extensions.
pip — Python's package manager. Like npm for JavaScript, Maven for Java.
venv — Virtual environment. Each project gets its own isolated Python world. No version conflicts.

💻 Live Demo — First Python Project (25–45 min)

```
# Step 1: Create project folder
mkdir my-first-project
cd my-first-project

# Step 2: Create virtual environment (isolated Python for this project)
python -m venv venv

# Step 3: Activate it (Windows)
venv\Scripts\activate

# Step 4: Install a package (requests - for calling APIs)
pip install requests

# Step 5: Save your dependencies
pip freeze > requirements.txt   # like package.json for Python

# Step 6: Git setup
git init
git add .
git commit -m "first python project"
```

🐍 Your First Python Program — Hello, Real World

```
# hello.py - runs top to bottom, no main() needed

name = "Priya"
age  = 21

# f-string: embed variables directly in the string
print(f"Hello, {name}! You are {age} years old.")

# Function - def, not public void
def greet(name, age=18):  # age has a default value
    return f"Hi {name}, you are {age}"

print(greet("Ravi", 22))
print(greet("Anita"))  # uses default age=18
```

🎯 You Code It (45–55 min)

1. Create folder `my-portfolio`, create and activate a `venv`
2. Create `main.py` — print your name, city, and 3 goals for this internship using f-strings
3. Add a function `introduce(name, role)` that returns a formatted introduction string
4. Run `git init`, then `git add .` then `git commit -m "first commit"`
5. Create a GitHub repo and push it — `git push -u origin main`
6. **Show the instructor:** Your GitHub repo URL

📄 Day 1 Cheat Sheet — Python Setup & Git

VIRTUAL ENVIRONMENT

python -m venv venv	Create venv
venv\Scripts\activate	Activate (Windows)
source venv/bin/activate	Activate (Mac/Linux)
pip install package	Install package
pip freeze > requirements.txt	Save deps
pip install -r requirements.txt	Restore deps

GIT DAILY WORKFLOW

git pull	Get latest
git add .	Stage all
git commit -m "msg"	Save snapshot
git push	Upload
git status	What changed?
git log --oneline	History

PYTHON BASICS

name = "Priya"	String variable
age = 21	Integer
f"Hello {name}"	f-string
def greet(name):	Function
return value	Return
print(x)	Output

.GITIGNORE FOR PYTHON

venv/	Virtual env
__pycache__/	Compiled files
*.pyc	Bytecode
.env	Secret keys
.DS_Store	Mac metadata
*.sqlite3	Local DB file

DAY 2 · 1 HOUR

Data Structures

list vs dict vs two dicts — the contacts app that makes time & space trade-offs unforgettable.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK — YOUR CONTACTS APP (0–5 MIN)

Your phone has **500 contacts**. You type **"P"** and in 0.2 seconds you see Priya, Pooja, Praveen. It doesn't read all 500 names one by one — that would take 500 steps.

Now WhatsApp gets a call from **+91 98765 43210**. It shows whose number this is before you pick up. To do that instantly, it keeps a **second copy** of contacts mapped by phone number — same data stored twice, 2× the memory, but the answer is instant.

Every data structure trades time for space or space for time. Pick wrong and a 1ms operation becomes a 10-second freeze.

🕒 The Two Costs (5–25 min)

⚡ Time — How many steps?

O(1) = 1 step always (dict lookup)

O(log n) = 10 steps for 1000 items (binary search)

O(n) = 1000 steps for 1000 items (list scan)

💾 Space — How much memory?

O(1) = no extra (list — just the array)

O(n) = extra proportional to items (dict overhead)

O(2n) = double your data (two dicts, bidirectional)

📞 The Contacts App — 4 Structures, 1 Problem

🗓️ OPTION 1 — LIST (NAIVE)

Store contacts as a list of tuples. To find "Priya": check every item from index 0.

Search: O(n) — 500 checks worst case Space: O(n) — lean, no overhead

⚠ For 5 million WhatsApp contacts, this is 5 million checks per search. The app freezes.

OPTION 2 — SORTED LIST + BISECT (SMART)

Sort contacts A–Z, use Python's `bisect` for binary search. Found "Priya" in **9 steps** from 500 contacts.

Search: O(log n) — 9 checks for 500 Space: O(n) — same list, just sorted

⚠ Adding a new contact? Must re-sort or shift elements. Good for read-heavy, bad if contacts change often.

OPTION 3 — DICT (THE RIGHT TOOL)

Python `dict` maps name → phone. Finding "Priya": hash("Priya"), go directly to that slot. No scanning.

Search: O(1) — always 1 step Space: O(n) — ~1.5x more than a plain list

Memory: list vs dict for 500 contacts

list (500 contacts)

500 contacts (raw)

dict (500 contacts) — hash table overhead

500 contacts

hash table overhead — pays for O(1)

⚠ OPTION 4 — TWO DICTS (BIDIRECTIONAL — WHATSAPP CALLER ID)

Need both: name→number AND number→name. Store two dicts — one forward, one reverse.

Search by name: O(1) Search by number: O(1) Space: O(2n) — data stored TWICE

name→phone dict (500)

phone→name dict (same 500, again)

2x overhead

Double memory for instant two-way lookup. WhatsApp makes this trade because caller ID speed matters more than RAM.

💻 Live Coding — ContactBook in Python (25–45 min)

```
# contact_book.py

# ── Option 1: list ──────────────────────────────────────────────
contact_list = []

def add_to_list(name, phone):
    contact_list.append((name, phone))

def find_in_list(name):
    steps = 0
    for contact in contact_list:   # O(n) — checks every contact
        steps += 1
        if contact[0] == name:
            print(f"  list: found in {steps} steps")
            return contact[1]
    return None

# ── Option 3: dict (one-way) ────────────────────────────────────
name_to_phone = {}                       # empty dict

def add_to_dict(name, phone):
    name_to_phone[name] = phone         # O(1) insert

def find_in_dict(name):
    print("  dict: found in 1 step O(1)")
    return name_to_phone.get(name)     # O(1) — always 1 step

# ── Option 4: two dicts (bidirectional) ─────────────────────────
phone_to_name = {}                       # EXTRA memory!

def add_bidirectional(name, phone):
    name_to_phone[name]  = phone        # forward
    phone_to_name[phone] = name         # reverse ── costs 2x space

def whose_number(phone):
    return phone_to_name.get(phone)    # O(1) ── paid with space


# ── Demo ────────────────────────────────────────────────────────
for i in range(1, 500):
    add_to_list(f"Contact{i}", f"900000{i}")
    add_to_dict(f"Contact{i}", f"900000{i}")
```

```
add_to_list("Priya", "9876543210")     # last entry — worst case
add_bidirectional("Priya", "9876543210")

print("Searching for Priya ... ")
find_in_list("Priya")     # 500 steps!
find_in_dict("Priya")     # 1 step

print(f"\nIncoming call: 9876543210")
print(f"Caller: {whose_number('9876543210')}")  # Priya
```

📄 When to Use Each Structure

Structure	Search	Insert	Space	Use it when...	Real Example
list	O(n)	O(1)	O(n)	Ordered data, loop through, rarely search by value	**Zomato order history** — shown newest first, you scroll through in order, never jump to "order #347"
dict	O(1)	O(1)	O(n)+overhead	Instant lookup by key (name, ID, token)	**Contacts app** — type "Priya", instant result. Also: every login session (token → userId), every product page cache
Two dicts	O(1) both	O(1)	O(2n)	Instant lookup in both directions	**WhatsApp caller ID** — name↔number. Same data twice, O(1) both ways. Double memory, zero latency
set	O(1)	O(1)	O(n)+overhead	Unique items, fast membership check	**Instagram following** — "Is Priya following Ravi?" O(1) check in a set of 1 billion users
deque (Stack)	O(n)	O(1)	O(n)	Last-in-first-out: undo, browser back	**VS Code Ctrl+Z** — every keystroke pushed onto a stack, Ctrl+Z pops the last one
deque (Queue)	O(n)	O(1)	O(n)	First-in-first-out: jobs in arrival order	**Swiggy restaurant queue** — first order placed = first cooked. Also: OTP SMS delivery jobs

🎯 You Code It (45–55 min)

1. Copy `contact_book.py` into VS Code and run it — see "500 steps" vs "1 step" yourself

2. Add function `delete_contact(name)` that removes from *both* dicts — get the phone first, then delete both keys
3. Add function `rename_contact(old, new)` — update both dicts correctly
4. **Space challenge:** Add `recent_searches = []` and update `find_in_dict()` to save each search — keep max 10, remove oldest when over
5. **Think out loud:** If RAM was expensive (embedded device), which structure would you drop? Tell the instructor your reasoning

📄 Day 2 Cheat Sheet — Python Data Structures

LIST

contacts = []	Create empty
contacts.append(x)	Add end — O(1)
contacts[0]	By index — O(1)
x in contacts	Search — O(n)
contacts.remove(x)	Remove — O(n)
sorted(contacts)	Sort — O(n log n)

DICT

d = {}	Create empty
d["key"] = val	Insert — O(1)
d["key"]	Lookup — O(1)
d.get("key", default)	Safe get
"key" in d	Check — O(1)
del d["key"]	Delete — O(1)

SET & DEQUE

s = set()	Empty set
s.add(x)	Add — O(1)
x in s	Check — O(1)
from collections import deque	Import
dq.append(x) / dq.pop()	Stack (LIFO)
dq.append(x) / dq.popleft()	Queue (FIFO)

SPACE–TIME RULE

Fast search by key?	dict — costs memory
Both-way lookup?	Two dicts — costs 2x memory
Memory tight?	sorted list + bisect
Unique items?	set — O(1) check
Undo / back?	deque as stack
Job queue?	deque as queue

DAY 3 · 1 HOUR

Python Fundamentals

Functions, loops, f-strings — build an ATM simulation that runs real bank logic.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Every Android app backend at Swiggy, every data pipeline at HDFC Bank, every automation script at TCS — Python.

Python is the #1 language for AI, data, and backend. Today you write a real ATM simulation with balance checks, deposits, and withdrawals — the same logic that runs on every bank's server.

🧉 Python vs Every Other Language (5–25 min)

💡 WHY PYTHON IS DIFFERENT

No semicolons — indentation defines blocks (tabs = structure).
No types needed — `x = 5` not `int x = 5;`
No main() — code at module level runs top-to-bottom.
No return type — `def add(a, b)` not `public int add(int a, int b)`

💻 Live Coding — ATM Simulation (25–45 min)

```
# atm.py

def create_account(owner, balance=0):
    # Python uses dict as a simple data holder
    return {"owner": owner, "balance": balance, "transactions": []}

def deposit(account, amount):
    if amount ≤ 0:
        print("Invalid amount")
        return
    account["balance"] += amount
    account["transactions"].append(("deposit", amount))
    print(f"Deposited ₹{amount}. Balance: ₹{account['balance']}")

def withdraw(account, amount):
    if amount > account["balance"]:
        print("Insufficient funds!")
    else:
        account["balance"] -= amount
        account["transactions"].append(("withdraw", amount))
        print(f"Withdrawn ₹{amount}. Remaining: ₹{account['balance']}")

def show_history(account):
    print(f"\n--- {account['owner']}'s History ---")
    for txn_type, amt in account["transactions"]:
        symbol = "+" if txn_type == "deposit" else "-"
        print(f"  {symbol}₹{amt}")


# Usage
acc = create_account("Priya", 10000)
deposit(acc, 5000)
withdraw(acc, 3000)
withdraw(acc, 20000)   # "Insufficient funds!"
show_history(acc)
```

🔄 Processing Transactions with a Loop

```
transactions = [500, -200, 1000, -50, -800]
balance = 5000

for t in transactions:
    if t > 0:
        balance += t
        print(f"  Deposit ₹{t}  → Balance: ₹{balance}")
    else:
        balance -= abs(t)
        print(f"  Withdraw ₹{abs(t)} → Balance: ₹{balance}")

print(f"Final: ₹{balance}")  # ₹4450
```

🎯 You Code It (45–55 min)

1. Create a `grade_calculator.py` with function `get_grade(marks)`
2. Logic: marks≥90="A", ≥75="B", ≥60="C", ≥50="D", else="F"
3. Add function `process_class(marks_list)` — print grade for each student
4. Add function `class_average(marks_list)` — return average (use `sum()` and `len()`)
5. Test: `marks = [92, 45, 78, 63, 88, 31, 95]`
6. **Show:** All grades + class average printed

📄 Day 3 Cheat Sheet — Python Fundamentals

VARIABLES & TYPES

name = "Priya"	String
age = 21	Integer
gpa = 3.9	Float
active = True	Boolean
type(x)	Check type
str(42)	int to string

FUNCTIONS

def greet(name):	Define
def add(a, b=0):	Default param
return value	Return
greet("Priya")	Call
*args	Variable positional args
**kwargs	Variable keyword args

CONTROL FLOW

USEFUL BUILT-INS

if x > 0:	Condition
elif x == 0:	Else-if
else:	Default
for x in items:	For-each
for i in range(10):	Count loop
while condition:	While loop

len(x)	Length
sum(list)	Sum all
max(list)	Largest
min(list)	Smallest
sorted(list)	Sorted copy
enumerate(list)	Index + value pairs

DAY 4 · 1 HOUR

OOP in Python

Classes, inheritance, dunder methods — why updating one class updates an entire app.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Why does updating Instagram not require reinstalling your phone's OS? Why does a junior developer at Swiggy only need to change one `Restaurant` class to update how all restaurants behave across the app?

Object-Oriented Programming. Everything is a blueprint (class). One change in the blueprint updates every object built from it. This is how Python backends at Swiggy, Razorpay, and Urban Company are structured.

🌍 The 4 Pillars — Python Style (5–25 min)

💡 ANALOGIES

Encapsulation — A capsule hides the medicine. You swallow it, not mix chemicals. Private attributes + public methods.
Inheritance — A SavingsAccount IS a BankAccount. Gets everything from parent, adds its own rules.
Polymorphism — `area()` on a Circle vs Rectangle — same name, different result.
Abstraction — You drive a car without knowing the engine. Abstract base class hides complexity.

💻 Live Coding — Bank Account Hierarchy (25–45 min)

```python
class BankAccount:
    # __init__ is Python's constructor
    def __init__(self, owner, balance=0):
        self.owner   = owner
        self._balance = balance      # _ prefix = "treat as private"
        self._transactions = []

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            self._transactions.append(amount)

    def withdraw(self, amount):
        if amount > self._balance:
            print("Insufficient funds!")
        else:
            self._balance -= amount

    def get_balance(self):
        return self._balance

    def __str__(self):              # what print() shows — dunder method
        return f"{self.owner}: ₹{self._balance}"


class SavingsAccount(BankAccount):  # inherits BankAccount
    MIN_BALANCE = 500

    def withdraw(self, amount):          # Polymorphism: override parent method
        if self._balance - amount < self.MIN_BALANCE:
            print("Cannot withdraw — min balance ₹500 required")
        else:
            super().withdraw(amount)    # call parent's method


class LoanAccount(BankAccount):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)
        self.rate = rate

    def calculate_interest(self, months):
        return self._balance * self.rate * months / 1200
```

```
# Demo
acc  = BankAccount("Priya", 10000)
sav  = SavingsAccount("Ravi", 10000)
loan = LoanAccount("Anita", 50000, 12)   # 12% p.a.

sav.withdraw(9600)            # "Cannot withdraw — min balance ₹500 required"
sav.withdraw(5000)            # OK
print(sav)                      # "Ravi: ₹5000"
print(loan.calculate_interest(6))# ₹3000
```

🎯 You Code It (45–55 min)

1. Create an `Employee` class: `name`, `salary`; method `work()` prints "{name} is working"
2. Create `Manager(Employee)` — add `team_size`, override `work()` to print "{name} is managing {team_size} people"
3. Create `Contractor(Employee)` — add `hours_worked`, add method `calculate_pay()` returning hours × ₹500
4. Add `__str__` to `Employee` that shows name + salary
5. Create one of each, call `work()` on each — see polymorphism: same method, different output
6. **Show:** Print all three objects + all three `work()` outputs

📄 Day 4 Cheat Sheet — OOP in Python

CLASS BASICS

class Car:	Define class
def __init__(self, x):	Constructor
self.speed = x	Instance attribute
car = Car(100)	Create object
car.speed	Access attribute
_name	Convention: private

INHERITANCE

class Dog(Animal):	Inherit
super().__init__(x)	Call parent constructor
super().method()	Call parent method
override method	Just redefine it
isinstance(obj, Cls)	Check type

DUNDER METHODS

CLASS VS INSTANCE

__init__	Constructor
__str__	What print() shows
__repr__	Developer string
__len__	len(obj)
__eq__	obj1 == obj2
__lt__	obj1 < obj2

class X: count = 0	Class attribute (shared)
self.name = x	Instance attribute (unique)
@classmethod	Method on the class itself
@staticmethod	Utility, no self/cls
@property	Getter with dot syntax

DAY 5 · 1 HOUR

Advanced Python

Comprehensions, generators, decorators — write in 1 line what takes 10 in other languages.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

A Netflix engineer filters 10,000 movies to your 20 recommendations in one Python line. A Swiggy data scientist processes 1 million orders in a pipeline that reads like a sentence.

List comprehensions, generators, decorators. The features that let Python code be shorter, faster, and more readable than Java. These are what interviewers look for to tell a Python beginner from a Python developer.

▶ 3 Power Features (5–25 min)

💡 MENTAL MODEL

List Comprehension — A for-loop that returns a new list, written in one line. Replace 5 lines with 1.
Generator — Like a list comprehension but lazy — produces values one at a time. Use it for huge datasets that don't fit in RAM.
Decorator — A function that wraps another function. Add logging, timing, or auth without touching the original code.

💻 Live Coding — Student Data Pipeline (25–45 min)

```
students = [
    {"name": "Priya", "marks": 92, "dept": "CS"},
    {"name": "Ravi",  "marks": 45, "dept": "IT"},
    {"name": "Anita", "marks": 78, "dept": "CS"},
    {"name": "Kumar", "marks": 33, "dept": "EC"},
    {"name": "Meena", "marks": 88, "dept": "CS"},
]

# ── List Comprehension ──────────────────────────────────────
# Normal loop (5 lines):
passed = []
for s in students:
    if s["marks"] ≥ 60:
        passed.append(s["name"])

# Comprehension (1 line — same result):
passed = [s["name"] for s in students if s["marks"] ≥ 60]
print(passed)  # ['Priya', 'Anita', 'Meena']

# CS students, sorted by marks, top 3 names
top_cs = [s["name"] for s in
          sorted(students, key=lambda s: s["marks"], reverse=True)
          if s["dept"] == "CS" and s["marks"] ≥ 60][:3]
print(top_cs)  # ['Priya', 'Meena', 'Anita']

# ── Generator ───────────────────────────────────────────────
# List: loads ALL marks into memory at once
all_marks = [s["marks"] for s in students]

# Generator: produces one mark at a time — use for 1 million records
marks_gen = (s["marks"] for s in students)   # () not []
avg = sum(marks_gen) / len(students)
print(f"Average: {avg}")

# ── Decorator ───────────────────────────────────────────────
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start  = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time()-start:.4f}s")
```

```
        return result
    return wrapper

@timer                          # apply decorator — no code change inside
process_data!
def process_data(students):
    return [s["name"] for s in students if s["marks"] ≥ 60]

process_data(students)   # prints: "process_data took 0.0001s"
```

🎯 You Code It (45–55 min)

1. Create a `products` list of 8 dicts: name, price, in_stock (bool), category (Electronics/Clothing)
2. Use a list comprehension: get names of all in-stock Electronics, sorted by price ascending
3. Use a generator + `sum()`: total value of all in-stock products
4. Write a `@logger` decorator that prints "Calling {function_name}" before every function call
5. Apply `@logger` to your filter function — call it and see the log
6. **Show:** Names list + total value + decorator log output

📄 Day 5 Cheat Sheet — Advanced Python

LIST COMPREHENSION

[x for x in items]	Simple copy
[x for x in items if cond]	Filter
[f(x) for x in items]	Transform
{k:v for k,v in d.items()}	Dict comprehension
{x for x in items}	Set comprehension

LAMBDA & SORTED

lambda x: x*2	Anonymous function
sorted(lst, key=lambda x: x[1])	Sort by field
sorted(lst, reverse=True)	Descending
min/max(lst, key=lambda x: x.gpa)	Min/max by field
filter(lambda x: x>0, lst)	Filter lazy

GENERATOR

(x for x in items)	Generator expression
def gen(): yield x	Generator function
next(gen)	Get next value
list(gen)	Convert to list
sum(x for x in items)	Sum without list

DECORATOR PATTERN

def dec(func):	Outer function
def wrapper(*a,**k):	Wrapper with passthrough
return func(*a,**k)	Call original
return wrapper	Return wrapper
@dec	Apply decorator
@functools.wraps(func)	Preserve name/docs

DAY 6 · 1 HOUR

Database & SQL

MySQL, CREATE TABLE, JOINs — give your app a memory that survives restarts and scales to millions.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Zomato stores 50 million orders. When you press "Reorder" it finds your last order in under 100ms. Paytm processes 10 million payments daily — every one is a SQL query.

Without a database, your Python app loses all data the moment you close the terminal. Today you give your app a permanent memory — one that survives restarts, scales to millions of rows, and can answer complex questions in milliseconds.

📊 Database = Excel with Superpowers (5–25 min)

💡 ANALOGY

A **Table** is like an Excel sheet. Each **row** is one record. A **PRIMARY KEY** is like a unique Aadhaar number — no two rows can share it. But unlike Excel, you query millions of rows in milliseconds.

💻 Live Coding — Student Database (25–45 min)

```sql
-- Create the table
CREATE TABLE students (
    id    INT          PRIMARY KEY AUTO_INCREMENT,
    name  VARCHAR(100) NOT NULL,
    email VARCHAR(150) UNIQUE,
    dept  VARCHAR(50),
    gpa   DECIMAL(3,2) DEFAULT 0.0
);

-- Insert
INSERT INTO students (name, email, dept, gpa)
VALUES ("Priya", "priya@mail.com", "CS", 3.9),
       ("Ravi",  "ravi@mail.com",  "IT", 3.2),
       ("Anita", "anita@mail.com", "CS", 3.7);

-- Read (most common)
SELECT * FROM students WHERE dept = "CS" AND gpa > 3.5;
SELECT name, gpa FROM students ORDER BY gpa DESC LIMIT 3;

-- Aggregate
SELECT dept, COUNT(*) AS total, AVG(gpa) AS avg_gpa
FROM students
GROUP BY dept
ORDER BY avg_gpa DESC;
```

Python + MySQL with mysql-connector

```python
import mysql.connector

conn = mysql.connector.connect(
    host="localhost", user="root", password="yourpassword", database="internship"
)
cursor = conn.cursor(dictionary=True)   # returns dicts not tuples

# Insert a student
cursor.execute(
    "INSERT INTO students (name, email, dept, gpa) VALUES (%s, %s, %s, %s)",
    ("Meena", "meena@mail.com", "CS", 3.8)
)
conn.commit()

# Fetch all CS students
cursor.execute("SELECT * FROM students WHERE dept = %s", ("CS",))
students = cursor.fetchall()
for s in students:
    print(f"{s['name']} - GPA: {s['gpa']}")

cursor.close()
conn.close()
```

🎯 You Code It (45–55 min)

1. Create the `students` table in MySQL Workbench
2. Insert at least 8 students from different departments (CS, IT, EC, ME)
3. Query 1: All students with GPA > 3.5, ordered by GPA descending
4. Query 2: Count students per department
5. Query 3: Department with highest average GPA
6. Query 4: Students whose name contains the letter "a" (use LIKE '%a%')
7. **Bonus:** Write the same queries using `mysql-connector` in Python

Day 6 Cheat Sheet — SQL + Python

DDL **DML**

CREATE TABLE t (...)	New table
PRIMARY KEY AUTO_INCREMENT	Auto ID
NOT NULL	Required
UNIQUE	No duplicates
DEFAULT value	If not provided

INSERT INTO t VALUES (v)	Add row
SELECT * FROM t WHERE c	Read
UPDATE t SET col=v WHERE c	Edit
DELETE FROM t WHERE c	Remove
LIKE '%text%'	Pattern match

SELECT CLAUSES

ORDER BY col DESC	Sort
LIMIT 10	First 10 rows
GROUP BY col	Group rows
HAVING COUNT(*) > 2	Filter groups
COUNT(*) AVG() MAX()	Aggregates

PYTHON MYSQL-CONNECTOR

mysql.connector.connect(...)	Connect
conn.cursor(dictionary=True)	Dict cursor
cursor.execute(sql, (val,))	Run query
cursor.fetchall()	All rows
conn.commit()	Save changes
cursor.close()	Close cursor

DAY 7 · 1 HOUR

Django Basics

Models, Views, URLs — Instagram & Pinterest were built on exactly this framework.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Before Django: building a Python web app meant manually parsing HTTP headers, managing database connections, and writing your own session system — weeks of work just to show a webpage.

After Django: **3 commands and your app is running**. Instagram was built on Django. Disqus serves 500 million users on Django. Pinterest, Mozilla, Bitbucket — all Django. Today you build your first real web app.

□ Django's MVT Architecture (5–25 min)

💡 **ANALOGY — DJANGO IS A RESTAURANT**

Model — The kitchen (stores and manages data in the database).
View — The waiter (receives request, fetches data, returns response).
Template — The menu/plate (how data is presented to the user).
URL — The front door (which URL goes to which view).

💻 Live Coding — First Django App (25–45 min)

```
# Step 1: Create project
pip install django
django-admin startproject studentproject .
python manage.py startapp students

# Step 2: models.py — define your data
from django.db import models

class Student(models.Model):
    name  = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    dept  = models.CharField(max_length=50)
    gpa   = models.DecimalField(max_digits=3, decimal_places=2, default=0.0)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.name} ({self.dept})"

    class Meta:
        ordering = ["-gpa"]   # default sort: highest GPA first
```

```
# Step 3: views.py — handle requests
from django.shortcuts import render
from .models import Student

def student_list(request):
    students = Student.objects.all()          # Django ORM: SELECT * FROM students
    return render(request, "students/list.html", {"students": students})

def student_detail(request, pk):
    student = Student.objects.get(pk=pk)      # SELECT WHERE id=pk
    return render(request, "students/detail.html", {"student": student})
```

```
# Step 4: urls.py
from django.urls import path
from . import views

urlpatterns = [
    path("students/",       views.student_list,   name="student-list"),
    path("students//", views.student_detail, name="student-detail"),
]

# Step 5: Run migrations and start server
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver            # http://127.0.0.1:8000
```

🎯 You Code It (45–55 min)

1. Create a new Django project: `django-admin startproject myapp .`
2. Create app: `python manage.py startapp products`
3. Define a `Product` model: name, price (DecimalField), category, in_stock (BooleanField)
4. Add to `INSTALLED_APPS` in settings.py, run `makemigrations` + `migrate`
5. Register `Product` in `admin.py`, create a superuser, go to `/admin`
6. Add 5 products via the Django admin panel
7. **Show:** Admin panel with 5 products visible

📄 Day 7 Cheat Sheet — Django Basics

DJANGO COMMANDS

django-admin startproject name .	New project
python manage.py startapp name	New app
python manage.py makemigrations	Detect changes
python manage.py migrate	Apply to DB
python manage.py createsuperuser	Admin user
python manage.py runserver	Start server

MODEL FIELDS

CharField(max_length=100)	Short text
TextField()	Long text
IntegerField()	Whole number
DecimalField(max_digits,decimal_places)	Price
BooleanField(default=True)	True/False
DateTimeField(auto_now_add=True)	Timestamp

URL PATTERNS

SETTINGS.PY ESSENTIALS

path("url/", view, name="")	Simple URL
path("url/<int:pk>/", view)	Integer param
path("url/<str:slug>/", view)	String param
include("app.urls")	Include app URLs
reverse("name")	URL from name

INSTALLED_APPS	Add your app here
DATABASES	DB connection
STATIC_URL	CSS/JS location
TEMPLATES	HTML template config
SECRET_KEY	Never commit this!
DEBUG = False	In production

DAY 8 · 1 HOUR

Django ORM

Python code that writes SQL for you — filter, annotate, aggregate without a single SQL string.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Writing raw SQL for every database operation is like assembling IKEA furniture without the instructions — possible but painful.

Django's ORM translates Python into SQL automatically. `Student.objects.filter(dept="CS", gpa__gt=3.5)` — that *is* your SQL query. No string building, no injection risk, no connection management. Used by every Django company in India.

🔎 ORM = Object-Relational Mapping (5–25 min)

💡 **ANALOGY**

The ORM is a translator. You speak Python, it speaks SQL. `Student.objects.all()` → `SELECT * FROM students`. You never need to write SQL directly — but understanding what SQL it generates helps you write faster queries.

💻 Live Coding — ORM Query Toolkit (25–45 min)

```python
from students.models import Student

# ── Basic CRUD ──────────────────────────────────────────────────

# CREATE
s = Student.objects.create(name="Priya", email="p@mail.com", dept="CS", gpa=3.9)

# READ
all_students = Student.objects.all()              # SELECT *
cs_students  = Student.objects.filter(dept="CS")    # WHERE dept='CS'
one          = Student.objects.get(pk=1)          # WHERE id=1 (raises if not found)
first        = Student.objects.first()            # LIMIT 1

# UPDATE
Student.objects.filter(name="Priya").update(gpa=3.95)

# DELETE
Student.objects.filter(gpa__lt=2.0).delete()

# ── Advanced Filters ────────────────────────────────────────────
# __ (double underscore) = "lookup" — maps to SQL operators
passing      = Student.objects.filter(gpa__gte=3.5)        # gpa ≥ 3.5
cs_pass      = Student.objects.filter(dept="CS", gpa__gt=3.5) # AND
with_a       = Student.objects.filter(name__icontains="a")    # LIKE '%a%' case-
insensitive
top3         = Student.objects.order_by("-gpa")[:3]           # ORDER BY gpa DESC
LIMIT 3

# ── Aggregation ─────────────────────────────────────────────────
from django.db.models import Count, Avg, Max

stats = Student.objects.aggregate(
    total=Count("id"),
    avg_gpa=Avg("gpa"),
    max_gpa=Max("gpa")
)
print(stats)  # {'total': 8, 'avg_gpa': 3.5, 'max_gpa': 3.9}

# Per-department stats
dept_stats = (Student.objects
    .values("dept")
    .annotate(count=Count("id"), avg=Avg("gpa"))
```

```
    .order_by("-avg"))
for d in dept_stats:
    print(f"{d['dept']}: {d['count']} students, avg GPA {d['avg']:.2f}")
```

🎯 You Code It (45–55 min)

1. Open your Django project from Day 7 in the shell: `python manage.py shell`
2. Create 8 students covering CS, IT, EC, ME departments with varied GPAs
3. Query: All CS students with GPA > 3.5, ordered by GPA descending
4. Query: Count of students per department using `values()` + `annotate()`
5. Query: Update all students with GPA below 2.0 to have `gpa = 2.0` (floor correction)
6. Query: Get the student with the highest GPA in each department
7. **Show:** Run all queries in shell, show instructor the output

📄 Day 8 Cheat Sheet — Django ORM

QUERYSET METHODS

.all()	All rows
.filter(**kwargs)	WHERE condition
.exclude(**kwargs)	WHERE NOT
.get(**kwargs)	One row (raises if 0 or >1)
.first() .last()	First / last row
.count()	COUNT(*)

LOOKUP OPERATORS

field__exact=v	= v (default)
field__gt=v	> v
field__gte=v	>= v
field__lt=v	< v
field__icontains=v	LIKE %v% (case-insensitive)
field__in=[a,b,c]	IN (a,b,c)

CREATE / UPDATE / DELETE

Model.objects.create(**data)	INSERT + save
obj.save()	INSERT or UPDATE
qs.update(field=val)	UPDATE all in qs
obj.delete()	DELETE one
qs.delete()	DELETE all in qs
get_or_create(**kwargs)	Find or create

AGGREGATION

from django.db.models import Count,Avg	Import
.aggregate(total=Count("id"))	Single stat
.values("dept")	GROUP BY dept
.annotate(c=Count("id"))	Add agg to each group
.order_by("-avg")	Sort groups
.select_related("fk")	JOIN in one query

DAY 9 · 1 HOUR

Django REST Framework

Serializers, ViewSets, Routers — full CRUD JSON API for any model in minutes.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Your Django app returns HTML pages. But a React frontend, a mobile app, and a third-party dashboard all need **JSON data**, not HTML.

Django REST Framework turns your Django models into a full JSON API in minutes. Razorpay, Ola, and Urban Company use DRF-style APIs to power their mobile apps. Today you build an API that any frontend — web, mobile, or Postman — can talk to.

🗺 API = a waiter for data (5–25 min)

💡 REST ANALOGY

GET /students/ — "Give me the menu" (read all)
POST /students/ — "I want to order" (create new)
GET /students/1/ — "What's in order 1?" (read one)
PUT /students/1/ — "Change my order" (full update)
DELETE /students/1/ — "Cancel my order" (delete)

💻 Live Coding — Student REST API (25–45 min)

```
# Install: pip install djangorestframework
# Add 'rest_framework' to INSTALLED_APPS

# serializers.py — converts Model ⇔ JSON
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model  = Student
        fields = "__all__"   # expose every field
```

```python
# views.py — ViewSet = full CRUD in one class
from rest_framework import viewsets, filters
from .models import Student
from .serializers import StudentSerializer

class StudentViewSet(viewsets.ModelViewSet):
    queryset         = Student.objects.all()
    serializer_class = StudentSerializer

    # Filtering: /api/students/?dept=CS&gpa__gte=3.5
    filter_backends  = [filters.SearchFilter, filters.OrderingFilter]
    search_fields    = ["name", "email"]
    ordering_fields  = ["gpa", "name"]
    ordering         = ["-gpa"]
```

```python
# urls.py — Router auto-generates all URLs
from rest_framework.routers import DefaultRouter
from .views import StudentViewSet

router = DefaultRouter()
router.register(r"students", StudentViewSet)

urlpatterns = router.urls
# Gives you automatically:
# GET  /students/          ← list all
# POST /students/          ← create
# GET  /students/{id}/     ← retrieve one
# PUT  /students/{id}/     ← full update
# PATCH /students/{id}/    ← partial update
# DELETE /students/{id}/   ← destroy
```

> 📋 **TEST YOUR API INSTANTLY**
>
> Visit `http://127.0.0.1:8000/api/students/` in your browser — DRF shows a beautiful Browsable API. No Postman needed at first. Hit "POST" to create a student, "DELETE" to remove one — right in the browser.

🎯 You Code It (45–55 min)

1. Install DRF: `pip install djangorestframework`, add to `INSTALLED_APPS`
2. Create `ProductSerializer` and `ProductViewSet` for your Product model from Day 7
3. Register with Router, wire up `urls.py`, run server
4. Use the Browsable API to: create 3 products, update one price, delete one product
5. **Bonus:** Add `search_fields = ["name"]` and test `?search=laptop`

6. **Show:** `GET /api/products/` returns JSON with your 2 remaining products

📄 Day 9 Cheat Sheet — Django REST Framework

SERIALIZER

serializers.ModelSerializer	Auto from Model
fields = "__all__"	All fields
fields = ["id","name"]	Specific fields
read_only_fields = ["id"]	Can't be set via API
serializer.is_valid()	Validate input
serializer.save()	Create or update

VIEWSET

viewsets.ModelViewSet	Full CRUD
viewsets.ReadOnlyModelViewSet	GET only
queryset = Model.objects.all()	Base queryset
serializer_class = MySerializer	Use this serializer
permission_classes = [IsAuthenticated]	Auth required

ROUTER URLS

GET /items/	list
POST /items/	create
GET /items/{id}/	retrieve
PUT /items/{id}/	update
PATCH /items/{id}/	partial_update
DELETE /items/{id}/	destroy

FILTERING

filters.SearchFilter	?search=term
filters.OrderingFilter	?ordering=-price
django-filter package	?price__gte=100
pagination_class	Add paging
get_queryset()	Custom filter logic

DAY 10 · 1 HOUR

Frontend — HTML + JS + API

fetch() calls your Django API — build a live dashboard without page reloads.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Your DRF API returns JSON. But no one wants to use `curl`. Users want buttons, tables, forms.

Today you build an HTML + JavaScript frontend that calls your Django API and shows real data. This is exactly how Swiggy's web frontend talks to their Python backend — `fetch()` hits an API endpoint, the response renders in the browser. No page reloads. Full-stack.

🌐 Browser ↔ API in 3 Steps (5–25 min)

💡 **THE PATTERN**

Step 1: User loads the HTML page — just static HTML, no data yet.
Step 2: JavaScript runs `fetch("/api/students/")` — calls your Django API.
Step 3: API returns JSON — JavaScript puts it in the page without reloading.

💻 Live Coding — Student Dashboard (25–45 min)

```
<!— index.html —>
<!DOCTYPE html>
<html>
<head>
  <title>Student Dashboard</title>
  <style>
    body  { font-family: sans-serif; max-width: 800px; margin: 2rem auto; padding: 0
1rem; }
    table { width: 100%; border-collapse: collapse; margin-top: 1rem; }
    th, td { padding: .6rem 1rem; border: 1px solid #ddd; text-align: left; }
    th    { background: #2563eb; color: white; }
    tr:nth-child(even) { background: #f8f9fa; }
    .btn  { background:#2563eb; color:#fff; border:none; padding:.5rem 1rem; border-
radius:6px; cursor:pointer; }
    .form-row { display:flex; gap:.5rem; margin:.5rem 0; }
    .form-row input { flex:1; padding:.4rem; border:1px solid #ccc; border-radius:4px;
}
  </style>
</head>
<body>
  <h1>🌀 Student Dashboard</h1>

  <div class="form-row">
    <input id="name"  placeholder="Name">
    <input id="email" placeholder="Email">
    <input id="dept"  placeholder="Dept">
    <input id="gpa"   placeholder="GPA" type="number" step=".1">
    <button class="btn" onclick="addStudent()">Add</button>
  </div>

  <table>
    <thead><tr><th>Name</th><th>Email</th><th>Dept</th><th>GPA</th><th>Action</th>
</tr></thead>
    <tbody id="table-body"></tbody>
  </table>

  <script>
    const API = "http://127.0.0.1:8000/api/students/";

    async function loadStudents() {
      const res  = await fetch(API);
      const data = await res.json();
```

```
      const tbody = document.getElementById("table-body");
      tbody.innerHTML = data.map(s => `
        <tr>
          <td>${s.name}</td>
          <td>${s.email}</td>
          <td>${s.dept}</td>
          <td>${s.gpa}</td>
          <td><button class="btn" onclick="deleteStudent(${s.id})">Delete</button>
</td>
        </tr>`).join("");
    }

    async function addStudent() {
      const body = {
        name:  document.getElementById("name").value,
        email: document.getElementById("email").value,
        dept:  document.getElementById("dept").value,
        gpa:   document.getElementById("gpa").value,
      };
      await fetch(API, {
        method:  "POST",
        headers: { "Content-Type": "application/json" },
        body:    JSON.stringify(body)
      });
      loadStudents();  // refresh table
    }

    async function deleteStudent(id) {
      await fetch(`${API}${id}/`, { method: "DELETE" });
      loadStudents();
    }

    loadStudents();  // load on page open
  </script>
</body>
</html>
```

🎯 You Code It (45–55 min)

1. Create the HTML file above, open it in browser — is the table loading from your API?
2. Add an "Edit" button that shows a prompt, takes a new GPA, and sends PATCH request
3. Add a search input + filter button: on click, fetch `?search=term` and refresh table
4. Add CSS: highlight rows where GPA < 2.5 in red background
5. **Bonus:** Add department filter dropdown — selecting "CS" fetches only CS students
6. **Show:** Add a student, delete one, search by name — all without page reload

Day 10 Cheat Sheet — JS + Fetch API

FETCH() PATTERNS

fetch(url)	GET request
await fetch(url)	Wait for response
await res.json()	Parse JSON body
res.status === 200	Check success
res.ok	True if 2xx status

POST / PUT / DELETE

method: "POST"	Create
method: "PUT"	Full update
method: "PATCH"	Partial update
method: "DELETE"	Delete
headers: {Content-Type: application/json}	Tell server it's JSON

DOM MANIPULATION

document.getElementById("id")	Get element
el.innerHTML = html	Set HTML content
el.value	Input value
data.map(x => `html`).join("")	Array to HTML
onclick="fn(id)"	Click handler

ASYNC / AWAIT

async function fn() {}	Declare async
await promise	Wait for result
try { } catch(e) { }	Handle errors
Promise.all([p1, p2])	Run in parallel

DAY 11 · 1 HOUR

Full-Stack Integration

CORS, JWT auth, login flow — wire frontend and backend together end-to-end.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

You have a Django API. You have an HTML frontend. But when you open the HTML file and it calls the API — **CORS Error**. The browser refuses to let a page at `file://` or `localhost:3000` talk to `localhost:8000`.

This exact error blocked the Flipkart web team for a day. Today you wire the full stack end-to-end: CORS fixed, login flow, authenticated API calls, live data on screen.

🔒 What is CORS? (5–25 min)

⚠ CORS IN 30 SECONDS

Browsers enforce a rule: a script on `domain-a.com` cannot fetch data from `domain-b.com` unless domain-b explicitly says "I allow this." This security rule is called CORS — Cross-Origin Resource Sharing.

Fix: install `django-cors-headers` and add `localhost:3000` to the allowlist.

💻 Live Coding — Full-Stack Wiring (25–45 min)

```
# 1. Fix CORS
pip install django-cors-headers

# settings.py
INSTALLED_APPS = [
    ...
    "corsheaders",
    ...
]
MIDDLEWARE = [
    "corsheaders.middleware.CorsMiddleware",   # MUST be first
    "django.middleware.common.CommonMiddleware",
    ...
]
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
    "http://127.0.0.1:5500",   # VS Code Live Server
]
```

```
# 2. Add JWT authentication
pip install djangorestframework-simplejwt

# settings.py
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ],
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticatedOrReadOnly",
    ],
}

# urls.py
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns += [
    path("api/token/",         TokenObtainPairView.as_view()),
    path("api/token/refresh/", TokenRefreshView.as_view()),
]
```

```
// Frontend: login + authenticated request
async function login(username, password) {
    const res = await fetch("/api/token/", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ username, password })
    });
    const { access, refresh } = await res.json();
    localStorage.setItem("access", access);     // save token
}

async function fetchProtected() {
    const token = localStorage.getItem("access");
    const res = await fetch("/api/students/", {
        headers: { "Authorization": `Bearer ${token}` }
    });
    return res.json();
}
```

🎯 You Code It (45–55 min)

1. Install `django-cors-headers`, configure, restart server — confirm no more CORS error
2. Install `djangorestframework-simplejwt`, add token URLs
3. Update your frontend: add a login form that calls `/api/token/` and stores the JWT
4. Update your `addStudent()` to send the Authorization header
5. Test: log in as superuser, add a student — confirm it saves to the Django admin
6. **Show:** Full flow — login in browser, add student, see it in /admin

📄 Day 11 Cheat Sheet — Full-Stack Wiring

CORS SETUP

pip install django-cors-headers	Install
add to INSTALLED_APPS	Register
CorsMiddleware must be first	Critical order
CORS_ALLOWED_ORIGINS = [...]	Allowlist
CORS_ALLOW_ALL_ORIGINS = True	Dev only!

JWT AUTH

POST /api/token/	Get access + refresh
POST /api/token/refresh/	Get new access token
Authorization: Bearer <token>	Header format
localStorage.setItem(key, val)	Store token
localStorage.getItem(key)	Read token

DRF PERMISSIONS

HTTP STATUS CODES

IsAuthenticated	Must log in
IsAuthenticatedOrReadOnly	Read: public, Write: auth
IsAdminUser	Django staff only
AllowAny	No restriction
permission_classes = [...]	Per-view override

200 OK	Success
201 Created	POST success
204 No Content	DELETE success
400 Bad Request	Invalid data
401 Unauthorized	Not logged in
404 Not Found	ID doesn't exist

DAY 12 · 1 HOUR

Capstone: Library Tracker

No instructions. Build a full-stack Library Book Tracker from a requirements doc.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

This is interview day. Build a full product from scratch — just like your first week at a company.

You get a requirements doc. No step-by-step instructions. You decide the models, the API design, and the UI. The instructor is the "client". This is what they test at Amazon, Flipkart, and every Indian startup: **can you turn requirements into a working system?**

📝 Requirements: Library Book Tracker (5–25 min)

💡 WHAT TO BUILD

Models: Book (title, author, isbn, available_copies), Member (name, email, membership_id), Borrow (book FK, member FK, borrowed_on, due_date, returned)
API: CRUD for books and members. POST `/api/borrows/` to borrow a book (decrements available_copies). POST `/api/returns/` to return (increments).
Frontend: Book list with available count. Borrow form. Member dashboard showing active borrows.

📊 Architecture First (25–45 min)

```python
# models.py — design before you code
class Book(models.Model):
    title            = models.CharField(max_length=200)
    author           = models.CharField(max_length=100)
    isbn             = models.CharField(max_length=13, unique=True)
    available_copies = models.IntegerField(default=1)

class Member(models.Model):
    name          = models.CharField(max_length=100)
    email         = models.EmailField(unique=True)
    membership_id = models.CharField(max_length=20, unique=True)

class Borrow(models.Model):
    book        = models.ForeignKey(Book,   on_delete=models.CASCADE)
    member      = models.ForeignKey(Member, on_delete=models.CASCADE)
    borrowed_on = models.DateField(auto_now_add=True)
    due_date    = models.DateField()
    returned    = models.BooleanField(default=False)
```

```python
# Custom ViewSet action for borrowing
from rest_framework.decorators import action
from rest_framework.response import Response
from django.utils import timezone
from datetime import timedelta

class BorrowViewSet(viewsets.ModelViewSet):
    queryset         = Borrow.objects.all()
    serializer_class = BorrowSerializer

    @action(detail=False, methods=["post"])
    def borrow_book(self, request):
        book = Book.objects.get(pk=request.data["book_id"])
        if book.available_copies < 1:
            return Response({"error": "No copies available"}, status=400)
        book.available_copies -= 1
        book.save()
        borrow = Borrow.objects.create(
            book=book,
            member_id=request.data["member_id"],
            due_date=timezone.now().date() + timedelta(days=14)
        )
        return Response(BorrowSerializer(borrow).data, status=201)
```

🎯 You Code It — Build It Solo (45–55 min)

1. Set up project, create models, run migrations
2. Create serializers for Book, Member, Borrow
3. Wire up CRUD ViewSets + the custom `borrow_book` action
4. Add the return action: marks returned=True, increments available_copies
5. Build a simple HTML page: show book list with available count
6. Test full flow: add book, add member, borrow, return — watch available_copies change
7. **Show:** Running demo end-to-end. Explain every model decision.

📄 Day 12 Cheat Sheet — ForeignKey & Custom Actions

FOREIGNKEY **CUSTOM ACTIONS**

ForeignKey(Model, on_delete=CASCADE)	Required FK
on_delete=SET_NULL, null=True	Optional FK
obj.related_model	Access related object
book.borrow_set.all()	Reverse relation
select_related("book")	JOIN to avoid N+1

@action(detail=False, methods=["post"])	List action
@action(detail=True, methods=["post"])	Detail action
return Response(data, status=201)	Return JSON
from rest_framework import status	Status constants

DATE / TIME

from django.utils import timezone	Import
timezone.now()	Current datetime
timezone.now().date()	Today's date
from datetime import timedelta	Import timedelta
date + timedelta(days=14)	2 weeks from now

VALIDATION PATTERN

serializer.is_valid(raise_exception=True)	Auto 400 on fail
serializer.validated_data	Clean data
raise serializers.ValidationError	Custom error
def validate_field(self, value):	Field-level validator

DAY 13 · 1 HOUR

Deployment with Railway

Push to GitHub, deploy on Railway — your Python app live on the internet in 5 minutes.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Your app works on localhost. But localhost is invisible to the world. Deploying means your app runs 24/7 on a server in the cloud, reachable by anyone on the planet.

Railway deploys Python apps in 5 minutes from a GitHub repo. No Docker, no Kubernetes, no AWS configuration. The same platform used by thousands of Indian startups to get their Python backend live the same day.

📦 3 Things a Server Needs (5–25 min)

💡 PRE-DEPLOYMENT CHECKLIST

requirements.txt — Railway reads this to install your packages. Run `pip freeze > requirements.txt`
Procfile — tells Railway how to start your app: `web: gunicorn myproject.wsgi`
Environment Variables — SECRET_KEY, DATABASE_URL, DEBUG=False. Never commit secrets.

💻 Live Demo — Deploy in 5 Steps (25–45 min)

```
# Step 1: Production-ready settings
pip install gunicorn dj-database-url django-environ whitenoise

# settings.py changes for production
import environ
env = environ.Env()
environ.Env.read_env()

DEBUG      = env.bool("DEBUG", default=False)
SECRET_KEY = env("SECRET_KEY")
ALLOWED_HOSTS = [".railway.app", "127.0.0.1"]

# Static files (whitenoise serves CSS/JS without a CDN)
MIDDLEWARE = [ ...  "whitenoise.middleware.WhiteNoiseMiddleware"  ... ]
STATIC_ROOT = BASE_DIR / "staticfiles"

# Database from environment variable
DATABASES = {
    "default": env.db()           # reads DATABASE_URL env var
}
```

```
# Step 2: Procfile (no extension) at project root
web: gunicorn myproject.wsgi --log-file -

# Step 3: runtime.txt (tell Railway which Python version)
python-3.12.0

# Step 4: Collect static files
python manage.py collectstatic --no-input

# Step 5: Push to GitHub, then Railway
git add .
git commit -m "deploy: production settings"
git push origin main

# Railway: connect GitHub repo → New Service → Add env vars:
# SECRET_KEY=your-long-secret-key-here
# DEBUG=False
# Railway auto-provides DATABASE_URL for PostgreSQL
```

> 🚀 **AFTER DEPLOY**
>
> Run migrations on production: in Railway dashboard, open Shell tab and run `python manage.py migrate`. Then `python manage.py createsuperuser`. Now your app is live, has a database, and has an admin user.

🎯 You Code It (45–55 min)

1. Update your Django project: install gunicorn, whitenoise, django-environ
2. Create `Procfile` and `runtime.txt` in project root
3. Update settings.py for production (DEBUG=False, SECRET_KEY from env, whitenoise)
4. Run `pip freeze > requirements.txt`
5. Push to GitHub, create Railway account, connect repo
6. Add environment variables in Railway dashboard, deploy
7. **Show:** Your app running on a `.railway.app` URL — share the link

📄 Day 13 Cheat Sheet — Django Deployment

KEY FILES

requirements.txt	pip freeze > requirements.txt
Procfile	web: gunicorn app.wsgi
runtime.txt	python-3.12.0
.env	Local env vars (never commit)
.gitignore	Add .env, __pycache__, venv

PRODUCTION SETTINGS

DEBUG = False	Never True in prod
ALLOWED_HOSTS = [domain]	Whitelist your domain
SECRET_KEY from env	Never hardcode
collectstatic	Bundle static files
HTTPS only	Railway handles SSL

RAILWAY COMMANDS

railway login	Authenticate CLI
railway init	Link project
railway up	Deploy current code
railway run python manage.py migrate	Run on prod
railway logs	Live logs

ENVIRONMENT VARIABLES

SECRET_KEY	Django secret key
DEBUG	False
DATABASE_URL	Auto from Railway PostgreSQL
ALLOWED_HOSTS	.railway.app
CORS_ALLOWED_ORIGINS	Your frontend URL

DAY 14 · 1 HOUR

LLM with Groq API

Add an AI doubt assistant to your Django app — Groq runs LLaMA 3 at 800 tokens/sec, free.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Groq runs LLaMA 3 at **800 tokens per second** — that's faster than reading. For free.

Every startup building AI features in India right now is using Groq for speed and cost. Today you add an AI feature to your Django app in **12 lines of Python** — a student doubt assistant that answers programming questions instantly.

🌐 LLM = Function Call that Returns Text (5–25 min)

💡 DEMYSTIFIED

An LLM API is just a POST request. You send: `{"model": "llama3", "messages": [{"role": "user", "content": "Explain recursion"}]}` . You get back: `{"content": "Recursion is a function calling itself..."}` . That's it. No magic — just HTTP.

💻 Live Coding — AI Doubt Assistant (25–45 min)

```
# pip install groq
from groq import Groq

client = Groq(api_key="your-groq-api-key")

def ask_ai(question):
    response = client.chat.completions.create(
        model="llama-3.3-70b-versatile",
        messages=[
            {"role": "system", "content": "You are a Python tutor helping beginners.
Be concise and use code examples."},
            {"role": "user",   "content": question}
        ]
    )
    return response.choices[0].message.content

# Test it
answer = ask_ai("What is the difference between a list and a tuple in Python?")
print(answer)
```

```python
# Django view: POST /api/ask/ → returns AI answer
from rest_framework.views import APIView
from rest_framework.response import Response
from groq import Groq

client = Groq(api_key=settings.GROQ_API_KEY)

class AskAIView(APIView):
    def post(self, request):
        question = request.data.get("question", "")
        if not question:
            return Response({"error": "question required"}, status=400)

        response = client.chat.completions.create(
            model="llama-3.3-70b-versatile",
            messages=[
                {"role": "system", "content": "You are a Python tutor. Be concise."},
                {"role": "user",   "content": question}
            ]
        )
        answer = response.choices[0].message.content
        return Response({"answer": answer})
```

🎯 You Code It (45–55 min)

1. Create a free Groq account at console.groq.com and get an API key
2. Install: `pip install groq`
3. Test the `ask_ai()` function with 3 Python questions from your notebook
4. Add `AskAIView` to your Django project, wire up `POST /api/ask/`
5. Build a simple HTML chat interface: text input + submit → fetch `/api/ask/` → show response
6. **Bonus:** Add conversation history — store previous Q&A in a list and pass all messages to the API
7. **Show:** Working chat interface answering Python questions using Groq

📄 Day 14 Cheat Sheet — Groq LLM API

GROQ SETUP

pip install groq	Install
from groq import Groq	Import
Groq(api_key="...")	Create client
store key in .env	Never hardcode
console.groq.com	Free API key

CHAT COMPLETION

client.chat.completions.create()	Main call
model="llama-3.3-70b-versatile"	Fastest free model
messages=[{role, content}]	Conversation array
role: "system"	Set AI persona
role: "user"	User message
role: "assistant"	AI's prior reply

RESPONSE

response.choices[0]	First choice
.message.content	The text reply
response.usage.total_tokens	Tokens used
max_tokens=500	Limit response length
temperature=0.7	Creativity (0=deterministic)

PROMPT ENGINEERING

system prompt	Sets AI's role and style
"Be concise"	Shorter answers
"Use code examples"	More code in response
"Explain to a beginner"	Simpler language
conversation history	Pass all prior messages

DAY 15 · 1 HOUR

Industry Simulation

Feature branches, pull requests, pytest — your first day at a real company, simulated.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Your first day at a company: senior hands you a codebase, a Jira ticket, and says "make a PR by end of day." No guidance. No step-by-step.

Today that's exactly what happens. You get a real git workflow — feature branches, pull requests, code review. You write pytest tests. You fix a deliberately broken feature. This is the gap between "I know Python" and "I'm ready to work at a company."

🌳 Git-Flow + pytest in 1 Hour (5–25 min)

💡 INDUSTRY GIT WORKFLOW

Never commit to main. Create a feature branch — `git checkout -b feature/add-borrow-limit`. Write code. Write tests. Push branch. Open PR. Get reviewed. Merge.

This is how every company from 5-person startups to Google works. Get used to it today.

💻 Live Coding — Testing with pytest (25–45 min)

```
# pip install pytest pytest-django

# conftest.py (tells pytest about Django settings)
import pytest
from django.test import RequestFactory

@pytest.fixture
def api_client():
    from rest_framework.test import APIClient
    return APIClient()

@pytest.fixture
def sample_student(db):
    from students.models import Student
    return Student.objects.create(
        name="Priya", email="p@test.com", dept="CS", gpa=3.9
    )
```

```python
# tests/test_students.py
import pytest

@pytest.mark.django_db
def test_student_list(api_client, sample_student):
    res = api_client.get("/api/students/")
    assert res.status_code == 200
    assert len(res.json()) == 1

@pytest.mark.django_db
def test_create_student(api_client):
    data = {"name":"Ravi","email":"r@test.com","dept":"IT","gpa":3.2}
    res = api_client.post("/api/students/", data, format="json")
    assert res.status_code == 201
    assert res.json()["name"] == "Ravi"

@pytest.mark.django_db
def test_gpa_validation(api_client):
    # GPA can't be negative
    data = {"name":"Bad","email":"b@test.com","dept":"CS","gpa":-1.0}
    res = api_client.post("/api/students/", data, format="json")
    assert res.status_code == 400    # should reject

# Run: pytest -v
# ✓ test_student_list PASSED
# ✓ test_create_student PASSED
# ✓ test_gpa_validation PASSED
```

🍴 Git Workflow — Feature Branch → PR

```
# Industry standard: never commit directly to main

git checkout -b feature/add-borrow-limit

# ... write code and tests ...

git add .
git commit -m "feat: add borrow limit of 3 books per member"

git push origin feature/add-borrow-limit

# GitHub: Create Pull Request
# PR title: "feat: add borrow limit"
# Description: What changed, why, how to test
# Request review from a classmate

# After approval: merge to main
git checkout main
git pull
git branch -d feature/add-borrow-limit
```

🎯 Industry Simulation — Your Final Challenge (45–55 min)

1. Create branch: `git checkout -b feature/search-enhancement`
2. Add a custom search endpoint: `GET /api/students/search/?q=term` that searches name AND email
3. Write 3 pytest tests for this endpoint: empty query, matching results, no results
4. Run `pytest -v` — all 3 must pass
5. Commit with proper message: `feat: add multi-field student search endpoint`
6. Push branch, open a PR on GitHub, assign a classmate as reviewer
7. **Show:** PR open, green tests, reviewer assigned. You are ready for Day 1 at a company.

📄 Day 15 Cheat Sheet — pytest + Git-Flow

PYTEST

DRF TEST CLIENT

pip install pytest pytest-django	Install
def test_name(...):	Test function
assert x == y	Assert equality
@pytest.mark.django_db	Enable DB access
@pytest.fixture	Reusable setup
pytest -v	Run verbose

APIClient()	Test HTTP client
client.get("/url/")	GET request
client.post("/url/", data, format="json")	POST JSON
client.force_authenticate(user)	Login in test
res.status_code	HTTP status
res.json()	Response body

GIT FEATURE BRANCH

git checkout -b feature/name	Create + switch
git branch	List branches
git push origin feature/name	Push branch
git checkout main	Switch to main
git merge feature/name	Merge
git branch -d feature/name	Delete local

COMMIT CONVENTION

feat: add search	New feature
fix: handle empty query	Bug fix
test: add search tests	Tests only
docs: update README	Documentation
refactor: simplify filter	Code cleanup
chore: update deps	Maintenance

Python Full-Stack — Real-World Projects

15 production-grade projects you can build with your Django + DRF + MySQL + Railway skills.

💡 **HOW TO USE THIS LIST**

Each project adds something new. Start Beginner → level up. Put every project on GitHub with a live Railway URL in the README. Product companies like Freshworks, Chargebee, and Zoho actively hire Django developers with public portfolios.

▲ BEGINNER — BUILD CONFIDENCE (1–2 WEEKS EACH)

PROJECT 01

Personal Blog Platform

Write and publish posts with rich text, tag-based categories, comments with moderation. Django admin as CMS. Deployed on Railway.

Django | MySQL | Django ORM | REST API | + Rich text editor

BEGINNER 🕐 1 week

PROJECT 02

URL Shortener

Enter a long URL, get a short code. Track click count per link, top-10 most-clicked dashboard, link expiry date.

Django | MySQL | Django ORM | REST API | + Redis caching

BEGINNER 🕐 1 week

PROJECT 03

Recipe Sharing Platform

Users post recipes with ingredients and steps. Search by ingredient/tag. Bookmark favourites. Star rating with average display.

Django | DRF | MySQL | JWT | + Image upload

BEGINNER — 1–2 weeks

▶ INTERMEDIATE — INDUSTRY-READY (2–3 WEEKS EACH)

PROJECT 04

Expense Tracker with Charts

Log daily expenses by category. Monthly budget limits with over-budget alerts. Chart.js dashboard showing spend breakdown. Export to CSV.

Django | DRF | MySQL | JWT | + Chart.js frontend

INTERMEDIATE — 2 weeks

PROJECT 05

Kanban Task Manager API

Boards, lists, cards (like Trello). Drag-and-drop order via PATCH. Due dates, assignees, labels. DRF + minimal React frontend.

Django | DRF | MySQL | JWT | REST API | + React drag-and-drop

INTERMEDIATE — 2–3 weeks

PROJECT 06

Student Attendance System

Teacher marks attendance per class. QR-code per session students scan. Daily/monthly report per student. SMS alert when <75%.

Django | DRF | MySQL | JWT | + QR code + Twilio SMS

INTERMEDIATE — 2–3 weeks

PROJECT 07

E-learning Platform

Courses with video lessons, progress tracking per student, quiz after each module, auto-generate PDF certificate on completion.

Django | DRF | MySQL | JWT | + PDF certificate gen

INTERMEDIATE — 3 weeks

PROJECT 08

Cricket Score Tracker

Pull live scores from a public API, store match history, player stats, career averages, head-to-head comparisons.

Django DRF MySQL REST API

+ External API polling

INTERMEDIATE 2 weeks

PROJECT 09

Alumni Network Platform

College alumni register, share job opportunities, announce events, endorse skills. Admin approves new signups. Email digest weekly.

Django DRF MySQL JWT

+ Background tasks (Celery)

INTERMEDIATE 3 weeks

★ ADVANCED — PORTFOLIO SHOWCASERS (3–5 WEEKS EACH)

PROJECT 10

Real Estate Listing Portal

Agents post properties with photos, geo-filter by city/price/BHK, saved search alerts, enquiry management with status pipeline.

Django DRF MySQL JWT REST API

+ Google Maps API

ADVANCED 3–4 weeks

PROJECT 11

Multi-vendor Marketplace

Multiple sellers, product listings, buyer cart + checkout, commission calculation, seller payout reports, dispute flow.

Django DRF MySQL JWT

+ Razorpay split payments

ADVANCED 4 weeks

PROJECT 12

Freelance Project Board

Post gigs, freelancers bid, client selects and funds escrow, milestones, release payment on approval, review system.

Django DRF MySQL JWT

+ Escrow payment logic

ADVANCED 4 weeks

PROJECT 13

Online Polling & Survey System

Create multi-question polls with share links, real-time result updates via SSE, analytics by respondent demographics.

Django DRF MySQL JWT

+ Server-Sent Events

ADVANCED 3 weeks

PROJECT 14

News Aggregator & Personaliser

Fetch from 5 news APIs, classify by topic with ML, user preferences save topics/sources, daily email digest at chosen time.

Django | DRF | MySQL | DRF | JWT | + ML classification + Celery

ADVANCED

🕐 3–4 weeks

PROJECT 15

Inventory & POS System

Products, stock levels, low-stock alerts, sales transactions, barcode scan support, daily sales report, supplier reorder flow.

Django | DRF | MySQL | JWT | REST API | + Barcode scanning

ADVANCED

🕐 4–5 weeks

🏆 Skills You Practice Across All Projects

DJANGO + DRF All projects	**JWT AUTHENTICATION** Projects 3–15	**MYSQL + DJANGO ORM** All projects
REST API DESIGN All projects	**RAILWAY DEPLOYMENT** All projects	**FILE/IMAGE UPLOAD** Projects 3,7,10
BACKGROUND TASKS Projects 9,14	**EXTERNAL APIS** Projects 8,10,14	