CAREERFOUNDRY

Python for Web Developers Learning Journal

Objective

We find that the students who do particularly well in our courses are those who practice metacognition. Metacognition is the art of thinking about thinking; developing a deeper understanding of your own thought processes. With the help of this Learning Journal, you'll broaden your metacognitive knowledge and skills by reflecting on what you learn in this course.

Thanks to this Learning Journal, when you finish the course you'll have a complete and detailed record of your learning journey and progress over time. We really recommend that you take the time to complete this Journal; students do better in CF courses and in the working world as a result!

Directions

First, complete the pre-work section before you start your course. Then, once you've begun learning, take time after each Exercise to return to this Journal and respond to the prompts.

There will be 3 to 5 prompts per Exercise, and we recommend spending about 10 to 15 minutes in total answering them. Don't overthink it—just write whatever comes to mind!

Also make sure that, once you've started filling this document in, you upload it as a deliverable on the platform. This is so that your mentor can also see your Journal and how you're progressing over time. Don't worry though—what you write here won't affect how you're graded for the Exercise tasks. The learning journal is mostly for you and your self-evaluation!

Pre-Work: Before You Start the Course

Reflection questions (to complete before your first mentor call)

1. What experiences have you had with coding and/or programming so far? What other experiences (programming-related or not) have you had that may help you as you progress through this course?

As a matter of fact, I might call myself a career software developer. I started programming back in school on Fortran IV, having first live implementation in the 10th grade. I continued at the university, solving physics problems. In 1990-s I did industrial coding - simpler things on FoxPro, MS Access, core banking systems on Unisys LINC 4. Later, until 2005 I was implementing SAP R/3 and BW, doing coding in ABAP. Naturally, I got a good understanding of system architecture, databases and data processing. Basic knowledge of deployment and operations.

2. What do you know about Python already? What do you want to know?

Before starting the course I knew that Python is a popular programming language, especially for data analytics. One of many.

3. What challenges do you think may come up while you take this course? What will help you face them? Think of specific spaces, people, and times of day of week that might be favorable to your facing challenges and growing. Plan for how to solve challenges that arise.

Find practical application for real-life projects? I do not foresee anything dramatic.

Remember, you can always refer to Exercise 1.4 of the Orientation course if you're not sure whom to reach out to for help and support.

Exercise 1.1: Getting Started with Python

Learning Goals

- Summarize the uses and benefits of Python for web development
- Prepare your developer environment for programming with Python

Reflection Questions

1. In your own words, what is the difference between frontend and backend web development? If you were hired to work on backend programming for a web application, what kinds of operations would you be working on?

Short: backend is mostly busy handling data and business logic on a server, frontend handles UX on the human facing device. As a backend developer I'd be handling database design and data exchange, business logic, API endpoints.

2. Imagine you're working as a full-stack developer in the near future. Your team is asking for your advice on whether to use JavaScript or Python for a project, and you think Python would be the better choice. How would you explain the similarities and differences between the two languages

to your team? Drawing from what you learned in this Exercise, what reasons would you give to convince your team that Python is the better option?
(Hint: refer to the Exercise section "The Benefits of Developing with Python")

First of all, I'd make my choice only based on requirements and environment, not a bare "sympathy". Both JavaScript and Python are high-level, dynamically typed languages, but Python's biggest edge is its readability—the clean, indented syntax makes it way easier to debug and share code across the team. It comes with "out-of-the-box" essentials, meaning it handles complex tasks like web security and database connections right out of the box. Plus, with simple package management via `pip` and a massive community to lean on, we can prototype and deploy much faster without reinventing the wheel.

3. Now that you've had an introduction to Python, write down 3 goals you have for yourself and your learning during this Achievement. You can reflect on the following questions if it helps you. What do you want to learn about Python? What do you want to get out of this Achievement? Where or what do you see yourself working on after you complete this Achievement?

Learning Goals:

1. **Master the Basics: Get comfortable enough with Python syntax and core libraries to write clean code without a cheat sheet.**
2. **Practical Project: Build something functional that actually works.**
3. **Leadership: Learn enough to confidently manage Python projects and the people coding them.**

Exercise 1.2: Data Types in Python

Learning Goals

- Explain variables and data types in Python
- Summarize the use of objects in Python
- Create a data structure for your Recipe app

Reflection Questions

1. Imagine you're having a conversation with a future colleague about whether to use the iPython Shell instead of Python's default shell. What reasons would you give to explain the benefits of using the iPython Shell over the default one?

IPython is like the "pro" version of the shell because it gives you syntax highlighting and tab-completion, which makes coding much faster and helps catch typos instantly. Plus, being able to use the `%run` command to execute scripts and keep the variables active is useful for debugging.

2. Python has a host of different data types that allow you to store and organize information. List 4 examples of data types that Python recognizes, briefly define them, and indicate whether they are scalar or non-scalar.

Data type	**Definition**	**Scalar or Non-Scalar?**
float	Numbers with a fractional part (decimals)	Scalar
string	A sequence of characters wrapped in quotes	Scalar
dictionary	A collection of key-value pairs	Non-Scalar
tuple	An immutable, ordered sequence of elements	Non-Scalar

3. A frequent question at job interviews for Python developers is: what is the difference between lists and tuples in Python? Write down how you would respond.

Mutability: Lists are mutable (can be changed after creation), while tuples are immutable (fixed).
Data Integrity: Tuples are safer for data that shouldn't change (like constants or coordinates); lists are for data that changes over time.
Performance: Tuples are faster and more memory-efficient because their fixed size allows Python to allocate memory once. Lists have extra overhead to handle resizing.
Usage: Tuples can be used as dictionary keys because they are fixed and thus hashable, lists cannot.

4. In the task for this Exercise, you decided what you thought was the most suitable data structure for storing all the information for a recipe. Now, imagine you're creating a language-learning app that helps users memorize vocabulary through flashcards. Users can input vocabulary words, definitions, and their category (noun, verb, etc.) into the flashcards. They can then quiz themselves by flipping through the flashcards. Think about the necessary data types and what would be the most suitable data structure for this language-learning app. Between tuples, lists, and dictionaries, which would you choose? Think about their respective advantages and limitations, and where flexibility might be useful if you were to continue developing the language-learning app beyond vocabulary memorization.

I chose a List of Dictionaries for the whole flashcard deck to allow for easy shuffling and iteration during quizzes. Each flashcard is a Dictionary that uses a List of Strings for definitions, Strings for grammatical categories ("noun", "verb", etc.) and tags ("regional", "slang", etc. - to provide flexibility for words with multiple nuances or remarks. To ensure data integrity, I would use Tuples for categories and tags; these act as "closed collections," preventing typos and ensuring that every card follows a consistent, valid classification system.

Code example:

```
# Fixed categories that don't change
CATEGORIES = ("noun", "verb", "adjective", "adverb", "preposition")
# Validated remarks/tags
VALID_TAGS = ("slang", "regional", "outdated", "formal", "archaic")
```

```
card_1 = {
    "word": "lassie",
    "definitions": ["A girl", "A young woman"],          # List of strings for nuance
    "category": CATEGORIES[0],                          # Single string from tuple
    "tags": [VALID_TAGS[1], VALID_TAGS[0]]              # List of strings (from tuple) for multiple remarks
}
card_2 = {
    'word': 'bairn',
    'definition': ['A child'],                          # List of one string
    'category': CATEGORIES[0],                          # References "noun"
    'tags': [VALID_TAGS[1]]                             # References "regional" inside a list
}
```

Exercise 1.3: Functions and Other Operations in Python

Learning Goals

- Implement conditional statements in Python to determine program flow
- Use loops to reduce time and effort in Python programming
- Write functions to organize Python code

Reflection Questions

1. In this Exercise, you learned how to use **if-elif-else** statements to run different tasks based on conditions that you define. Now practice that skill by writing a script for a simple travel app using an **if-elif-else** statement for the following situation:

    - The script should ask the user where they want to travel.
    - The user's input should be checked for 3 different travel destinations that you define.
    - If the user's input is one of those 3 destinations, the following statement should be printed: "Enjoy your stay in ______!"
    - If the user's input is something other than the defined destinations, the following statement should be printed: "Oops, that destination is not currently available."

Write your script here. *(Hint: remember what you learned about indents!)*

```
# Step 1: Ask the user for their destination
destination = input("Where do you want to travel? ")

# Step 2: Check the input against 3 defined destinations
if destination == "Lisbon":
    print("Enjoy your stay in Lisbon!")
```

```
elif destination == "Paris":
   print("Enjoy your stay in Paris!")
elif destination == "New York":
   print("Enjoy your stay in New York!")
else:
   # Step 3: Handle any other input
   print("Oops, that destination is not currently available.")
```

2. Imagine you're at a job interview for a Python developer role. The interviewer says "Explain logical operators in Python". Draft how you would respond.

Logical operators in Python - specifically `and`, `or`, and `not` - are the tools we use to combine or invert multiple boolean conditions within a single expression.
In a professional setting, we use them to control the flow of an application, such as verifying if a user has both the correct permissions and an active subscription before granting access. They follow short-circuit logic, meaning Python is efficient enough to stop evaluating an expression as soon as the final result is certain.

3. What are functions in Python? When and why are they useful?

Functions are reusable blocks of code designed to perform a specific action. You define them using the `def` keyword, and they only run when you "call" them by their name.

When and why they are useful:

- **DRY Principle: They enforce the "Don't Repeat Yourself" (DRY) principle by allowing you to write logic once and use it many times.**
- **Modularity: They help break complex programs into smaller, manageable blocks.**
- **Readability: Descriptive function names (like `calculate_total()`) make code self-explanatory.**
- **Maintenance: If you need to change how a calculation works, you only have to fix it in one place.**

4. In the section for Exercise 1 in this Learning Journal, you were asked in question 3 to set some goals for yourself while you complete this course. In preparation for your next mentor call, make some notes on how you've progressed towards your goals so far.

I have progressed from basic syntax to implementing the core logic of a functional Recipe App, demonstrating the ability to handle nested data structures and complex conditions. This move into practical application shows I am successfully bridging the gap between learning theory and building working software.

Exercise 1.4: File Handling in Python

Learning Goals

- Use files to store and retrieve data in Python

Reflection Questions

1. Why is file storage important when you're using Python? What would happen if you didn't store local files?

File storage allows for data persistence. Without it, all data (like our recipes) exists only in the computer's memory and is lost right after the Python script stops running. Storing local files ensures that data is still there when we need them again.

2. In this Exercise you learned about the pickling process with the `pickle.dump()` method. What are pickles? In which situations would you choose to use pickles and why?

"Pickles" are the result of serialization - converting a complex Python object (like a nested dictionary or a list of lists) into a byte stream for storage. Developers use pickles when they need to save specific Python-only data structures that text files cannot properly handle, as it preserves the exact data types without manual formatting.

3. In Python, what function do you use to find out which directory you're currently in? What if you wanted to change your current working directory?

To find the current working directory, we use `os.getcwd()`. To change it, we use `os.chdir('/path/to/directory')`. Both commands require importing the `os` module.

4. Imagine you're working on a Python script and are worried there may be an error in a block of code. How would you approach the situation to prevent the entire script from terminating due to an error?

I would wrap the risky code in a `try-except` block. This catches the error (exception) if it occurs, allowing the script to handle it smoothly (e.g., by printing a warning or setting a default value) instead of crashing the program.

5. You're now more than halfway through Achievement 1! Take a moment to reflect on your learning in the course so far. How is it going? What's something you're proud of so far? Is there something you're struggling with? What do you need more practice with? Feel free to use these notes to guide your next mentor call.

How it's going: It's going smoothly. The concepts are pretty clear, and I'm finding the logic manageable without it feeling overly complicated.
Proud of: Mastering the complete `try-except-else-finally` suite for robust error handling and successfully implementing data persistence with `pickle`. I'm also proud of incorporating more modern, "Pythonic" tools like f-strings, relative paths, and the `.get()` method to make my code cleaner and more efficient (thanks to my mentor for the advice to use Set type on certain occasions).

Struggling with / Need practice: Dealing with the nuances of file paths - specifically handling user input errors like accidental quotes or incorrect directory levels. I might focus more on input validation and making my code "bulletproof" against unexpected user behavior.

Exercise 1.5: Object-Oriented Programming in Python

Learning Goals

- Apply object-oriented programming concepts to your Recipe app

Reflection Questions

1. In your own words, what is object-oriented programming? What are the benefits of OOP?

Object-Oriented Programming is a paradigm that organizes software design around data, or objects, rather than functions and logic. An object is defined as a data field that has unique attributes and behavior. Instead of focusing on the logic required to manipulate objects, OOP focuses on the objects themselves.

The Benefits:
- **Modularity: Code is divided into independent units, making it easier to troubleshoot and navigate.**
- **Reusability: Code created for one program can be easily bridged into another through the use of established classes.**
- **Scalability: The structured nature of OOP allows systems to grow in complexity without becoming unmanageable.**

2. What are objects and classes in Python? Come up with a real-world example to illustrate how objects and classes work.

A Class serves as a template or a blueprint for creating objects. It defines the structure and the capabilities that any object created from it will possess. An Object is a specific instance of that class, containing actual data.

Real-World Example: Consider a Car. A car manufacturer has a master template (the Class) that specifies a vehicle must have an engine, four wheels, and a steering wheel. It also defines that the vehicle can "accelerate" or "brake." When the factory actually produces a car - a red sedan with a specific VIN number - that physical vehicle is an Object. Multiple objects (cars) can be created from the same template, but each has its own unique color, speed, and location.

3. In your own words, write brief explanations of the following OOP concepts; 100 to 200 words per method is fine.

Method	Description
Inheritance	Inheritance is a mechanism that allows a new class to derive its properties and characteristics from an existing class. This establishes a hierarchy where a "child" class inherits the attributes and methods of a "parent" class. For example, by defining `class ShoppingList(object):`, the `ShoppingList` class inherits fundamental behaviors from Python's base object template. This allows the new class to utilize built-in logic for identity and comparison without manual coding.
Polymorphism	Polymorphism is the ability of different objects to respond to the same method call in a manner specific to their own data type. It allows a single interface to represent a general class of actions. For example, if multiple different classes have a `__str__` method, calling `print()` on any of them will work perfectly, but each will produce a different text output based on its own internal data. The program doesn't need to know the specific class type to execute the display command.
Operator Overloading	Operator Overloading is the practice of defining custom logic for standard operators - such as `+`, `-`, or `==` - when they are applied to user-defined classes. By default, Python does not know how to mathematically interact with custom objects; operator overloading provides the specific instructions for these interactions. For example, in a `Height` class, using `__sub__` allows the use of the `-` operator to calculate the difference between two people. Similarly, in a `ShoppingList` class, `__add__` can be defined to merge two sets of items using the `+` symbol.

Exercise 1.6: Connecting to Databases in Python

Learning Goals

- Create a MySQL database for your Recipe app

Reflection Questions

1. What are databases and what are the advantages of using them?

A database is a structured collection of data organized so that it can be easily accessed, managed, and updated.

- **Advantages:**
  - **Persistence: Data is stored permanently even after the program closes.**
  - **Scalability: They can handle massive amounts of data much more efficiently than text files or spreadsheets.**
  - **Data Integrity: Constraints (like Primary Keys) prevent duplicate or "garbage" data.**
  - **Concurrent Access: Multiple users or applications can read and write data at the same time without corruption.**

2. List 3 data types that can be used in MySQL and describe them briefly:

Data type	Definition
INT	**Stores whole numbers (integers). Used for counts, IDs, or ages.**
VARCHAR(size)	**A variable-length string. You define a max size, and it stores text efficiently up to the “size” limit.**
DECIMAL(m, d)	**Stores exact numeric values with a fixed number of decimals, where m is the total number of digits and d is the number of digits after decimal point. Ideal for money/prices.**

3. In what situations would SQLite be a better choice than MySQL?

SQLite is a "serverless" database, meaning it is just a single file on your computer. It is better when:

- **Simplicity is key: You don't want to deal with installers, users, or permissions (like we did with the MySQL installer).**
- **Mobile/Desktop Apps: It’s used inside almost every smartphone app because it’s lightweight.**
- **Testing/Development: It’s great for quick prototyping before moving to a heavy-duty server like MySQL.**
- **No Internet/Network: When the database only needs to be accessed by one local application at a time.**

4. Think back to what you learned in the Immersion course. What do you think about the differences between JavaScript and Python as programming languages?

Execution Environment: JavaScript was created for the browser (frontend), while Python is the prime tool for data processing, scripts, and backend logic.
Syntax: Python uses indentation to define blocks of code, making it very readable and "clean", but in my opinion, less syntactically reliable. JavaScript uses curly braces `{}` and semicolons, which can feel more cluttered.
Logic Style: JavaScript is often asynchronous (handling many things at once, like button clicks), whereas Python is generally synchronous and procedural, making the flow easier for beginners to follow.

5. Now that you're nearly at the end of Achievement 1, consider what you know about Python so far. What would you say are the limitations of Python as a programming language?

Speed: Because it is an interpreted language, it is generally slower than "compiled" languages like C++ or Java.
Mobile Development: Python is rarely used for building native mobile apps (iOS/Android).
Global Interpreter Lock (GIL): This makes it difficult for Python to perform true multi-threading on multi-core processors.
Memory Consumption: It can be memory-intensive compared to other languages, which is a drawback for high-performance computing.

Exercise 1.7: Finalizing Your Python Program

Learning Goals

- Interact with a database using an object-relational mapper
- Build your final command-line Recipe application

Reflection Questions

1. What is an Object Relational Mapper and what are the advantages of using one?

An Object Relational Mapper (ORM) is a programming technique used to convert data between object-oriented code (like Python) and relational databases (like MySQL). Instead of writing raw SQL queries, the developer interacts with database tables as Python classes and rows as Python objects.

Advantages:

- **Abstraction: Simplifies database interactions by using familiar Python syntax rather than switching languages to SQL.**
- **Maintainability: Centralizes the data schema in the code, making it easier to update and manage as the project grows.**
- **Security: Automatically mitigates common vulnerabilities like SQL injection.**
- **Database Agnostic: Allows for switching underlying database engines (e.g., from SQLite to MySQL) with minimal code changes.**

2. By this point, you've finished creating your Recipe app. How did it go? What's something in the app that you did well with? If you were to start over, what's something about your app that you would change or improve?

The development of the Recipe app demonstrated the practical integration of Python and SQL. A primary success was the implementation of a robust data model using SQLAlchemy, ensuring that recipe attributes were clearly defined and correctly typed.

If the project was restarted, I would modify the approach to managing ingredients. Rather than storing them as a single string I would use a separate `Ingredients` table with a many-to-many relationship. This would allow for more precise filtering and querying of individual items without relying on string-matching techniques like `.like()`.

3. Imagine you're at a job interview. You're asked what experience you have creating an app using Python. Taking your work for this Achievement as an example, draft how you would respond to this question.

"I recently developed a database-driven Recipe Management application using Python and SQLAlchemy. The project focused on bridging the gap between application logic and persistent storage. I built a custom data model to handle recipe details, including cooking times and ingredients, and implemented a functional command-line interface for data entry and retrieval. One of the technical hurdles I navigated involved configuring a Python environment to communicate with a MySQL server using specific database drivers. This experience solidified my ability to manage complex back-end integrations, perform CRUD (Create, Read, Update, Delete) operations programmatically, and use ORMs to maintain clean, scalable code."

4. You've finished Achievement 1! Before moving on to Achievement 2, take a moment to reflect on your learning in the course so far:
    a. What went well during this Achievement?
    b. What's something you're proud of?
    c. What was the most challenging aspect of this Achievement?
    d. Did this Achievement meet your expectations? Did it give you the confidence to start working with your new Python skills?
    e. What's something you want to keep in mind to help you do your best in Achievement 2?

Successes and Pride: The successful transition from basic Python scripts to a fully integrated database application went well. There is a specific sense of pride in resolving configuration conflicts and environment issues, which are common in real-world development.

Challenges: The most challenging aspect was the initial setup of the database connectors. Troubleshooting installation errors regarding C++-extensions and library paths required a deep dive into how Python interacts with system-level dependencies.

Expectations and Confidence: This achievement met expectations by providing a practical workflow for professional software development. It successfully shifted the focus from theoretical syntax to building functional, data-persistent tools, significantly increasing confidence in backend development.

Goal for Achievement 2: Moving forward, a key focus will be on modularizing code from the start. Planning for scalability and error handling during the initial design phase - rather than as an afterthought - will ensure a smoother development cycle in future projects.

Well done—you've now completed the Learning Journal for Achievement 1. As you'll have seen, a little metacognition can go a long way!

Pre-Work: Before You Start Achievement 2

In the final part of the learning journal for Achievement 1, you were asked if there's anything—on reflection—that you'd keep in mind and do similarly or differently during Achievement 2. Think about these questions again:

- Was your study routine effective during Achievement 1? If not, what will you do differently during Achievement 2?
- Reflect on your learning and project work for Achievement 1. What were you most proud of? How will you repeat or build on this in Achievement 2?
- What difficulties did you encounter in the last Achievement? How did you deal with them? How could this experience prepare you for difficulties in Achievement 2?

Note down your answers and discuss them with your mentor in a call if you like.

Remember that can always refer to Exercise 1.4 of the Orientation course if you're not sure whom to reach out to for help and support.

Exercise 2.1: Getting Started with Django

Learning Goals

- Explain MVT architecture and compare it with MVC
- Summarize Django's benefits and drawbacks
- Install and get started with Django

Reflection Questions

1. Suppose you're a web developer in a company and need to decide if you'll use vanilla (plain) Python for a project, or a framework like Django instead. What are the advantages and drawbacks of each?

Vanilla Python

- **Pros:** Maximum flexibility and minimal overhead; you only include the code you actually need.
- **Cons:** Extremely time-consuming for web development, as you must manually handle security, routing, and database connections from scratch.

Django

- **Pros:** Rapid development through built-in features like an automatic admin interface and robust security defaults.
- **Cons:** Higher learning curve and significant "bloat" if the project is simple, as the framework carries overhead you might not use.

2. In your own words, what is the most significant advantage of Model View Template (MVT) architecture over Model View Controller (MVC) architecture?

The most significant advantage is the separation of concerns regarding the UI. In MVT, Django handles the "Controller" logic itself, allowing the developer to focus on the Template. This means the presentation layer is more decoupled; you can swap out or update the front-end design without needing to rewrite the underlying business logic, as the framework manages the bridge between the two automatically.

3. Now that you've had an introduction to the Django framework, write down three goals you have for yourself and your learning process during this Achievement. You can reflect on the following questions if it helps:
   - What do you want to learn about Django?
     - **Master the ORM: I want to fluently interact with databases using Python code rather than writing raw SQL, ensuring I can build data-driven apps efficiently.**
   - What do you want to get out of this Achievement?
     - **Build a Portfolio-Ready API: My goal is to produce a fully functional backend for a multi-user application that demonstrates I can handle authentication and data security.**
   - Where or what do you see yourself working on after you complete this Achievement?
     - **My goal is to become proficient enough to handle complex backend tasks like authentication flow, schema migrations, and API design in a way that allows me to apply these concepts to any modern development environment I work in later.**

Exercise 2.2: Django Project Set Up

Learning Goals

- Describe the basic structure of a Django project
- Summarize the difference between projects and apps
- Create a Django project and run it locally
- Create a superuser for a Django web application

Reflection Questions

1. Suppose you're in an interview. The interviewer gives you their company's website as an example, asking you to convert the website and its different parts into Django terms. How would you proceed? For this question, you can think about your dream company and look at their website for reference.

The Project: Robinhood Global

The **Project** represents the entire Robinhood ecosystem. It serves as the master container that manages global configurations, such as security protocols for financial data, middleware for user sessions, and the top-level URL routing that directs a user to their portfolio, the markets, or their account settings.

The Apps: Functional Modules

I would deconstruct the platform into several independent Django **Apps**, each responsible for a specific pillar of the business logic:

- **`accounts`**: This app would manage the `User` and `Profile` models. It handles the "Know Your Customer" (KYC) data, bank account linking, and authentication logic.
- **`orders`**: A core app dedicated to the `Trade` model. It would record every transaction, including the stock ticker, price at execution, timestamp, and order type (limit vs. market).
- **`market_data`**: This app would handle the models for `Security` (stocks, crypto, options) and their historical price points, allowing the frontend to render charts.
- **`banking`**: Dedicated to the `Transfer` model, managing the movement of cash between the user's bank and their brokerage trading balance.

Models, Views, and Templates

- **Models**: Each app uses Python classes to define the database structure. For instance, the `Order` model would have a foreign key linking it to a `User`, ensuring that trades are strictly associated with the correct individual.
- **Views**: The logic layer. When a user clicks "Buy," a View would process the request, check in the database if the user's trading balance is sufficient, and then trigger the creation of a new `Order` record.
- **Templates/API**: While the web version uses **Templates** to render the dashboard, the mobile app would interact with Django as a REST API, receiving JSON data (like current stock prices) to update the UI in real-time.

2. In your own words, describe the steps you would take to deploy a basic Django application locally on your system.

Deploying locally is about creating a controlled environment where the code can run exactly as planned. Here are the steps:

1. **Environment Setup**: Create a dedicated virtual environment to isolate the project's dependencies (like Django itself) from the rest of the system.
2. **Installation**: Activate the environment and use `pip` to install Django and any other necessary libraries.

3. **Project Initialization**: Use `django-admin startproject` to generate the core file structure (the management folder and configuration package). Rename root folder to avoid confusion with duplicated folder names.
4. **Database Migration**: Run `python manage.py migrate`. This tells Django to look at its internal models and build the initial SQLite database tables needed for features like user permissions.
5. **User Creation**: Run `python manage.py createsuperuser` to establish an administrative account so I can access the backend immediately.
6. **Launch**: Execute `python manage.py runserver` to start the local development server, making the site accessible via `localhost` in the browser.

3. Do some research about the Django admin site and write down how you'd use it during your web application development.

The Django Admin is one of the framework's most powerful embedded features. It is a pre-built, web-based interface that allows developers to interact with the database without writing any custom HTML or CSS.

How I'd use it during development:

- **Rapid Data Entry**: Instead of writing complex SQL scripts or building "Create" forms early on, I can use the Admin to quickly add test recipes, ingredients, or users to see how they look in the app.
- **Data Validation**: I can verify that my Models are working correctly. If I set a field to be "required," the Admin UI will automatically enforce that, helping me catch logic errors in my data structure.
- **User Management**: It's the easiest way to manage permissions - assigning "staff" status to certain users or resetting passwords during the testing phase.
- **Content Moderation**: As the app grows, the Admin serves as the "back office" where I can review, edit, or delete user-generated content (like recipe comments) efficiently.

Exercise 2.3: Django Models

Learning Goals

- Discuss Django models, the "M" part of Django's MVT architecture
- Create apps and models representing different parts of your web application
- Write and run automated tests

Reflection Questions

1. Do some research on Django models. In your own words, write down how Django models work and what their benefits are.

In Django, a Model is a Python class that acts as the blueprint for a table in the database. Instead of writing SQL queries, you define the data structure in Python, and Django's ORM (Object-Relational Mapper) handles the translation to the database.

How they work:

- Each class represents a **table**.
- Each attribute (variable) in the class represents a **database field (column)**.
- Django uses **migrations** to propagate any changes made in Python code (like adding a field) directly into the database schema.

Benefits:

- **Database Agnostic:** We can switch from SQLite to PostgreSQL without changing Python code.
- **Security:** The ORM automatically protects against common threats like SQL injection.
- **Efficiency:** One can create, retrieve, and update data using clean Python methods like `Recipe.objects.all()` rather than SQL strings.

2. In your own words, explain why it is crucial to write test cases from the beginning of a project. You can take an example project to explain your answer.

Writing test cases from the start - often called **Test-Driven Development (TDD,** s. Full-Stack Development course**)** - ensures that the code behaves as expected as it grows in complexity. It acts as a safety net that catches bugs before they reach production.

Why it is crucial:

- **Prevents Regression:** It ensures that adding a new feature doesn't accidentally break something developed some ago.
- **Documentation:** Tests serve as a "living document" that shows other developers exactly how a piece of code is intended to function.
- **Better Design:** Writing a test first forces the developer to think about the inputs and outputs of the logic before even writing it.

Example: The Recipe App

Say, I write a function to calculate the `difficulty` of a recipe based on the number of ingredients.

1. **Without Tests:** I manually click through the app to see if a 10-ingredient recipe shows "Hard." Later, I change the logic to include cooking time, and accidentally break the ingredient count logic. I don't realize it until some user complains.
2. **With Tests:** I have a test case that specifically checks: `input(12 ingredients) -> output("Hard")`. The moment I change the code, the test will fail immediately in the terminal, telling me exactly where the logic went wrong before I ever commit the code.

Exercise 2.4: Django Views and Templates

Learning Goals

- Summarize the process of creating views, templates, and URLs
- Explain how the "V" and "T" parts of MVT architecture work
- Create a frontend page for your web application

Reflection Questions

1. Do some research on Django views. In your own words, use an example to explain how Django views work.

A Django view is the logic center that connects user requests to the final output. Its primary job is to accept a web request and return a web response.

When a user interacts with the application, the process follows a specific sequence:

1. **Request Capture:** The user navigates to a URL. Django's URL configuration identifies which view function or class is mapped to that specific path.
2. **Logic Processing:** The view executes the necessary Python code. This might involve fetching data from the database (via Models), performing calculations, or checking user permissions.
3. **Template Rendering:** Once the view has the data, it loads a Template. It "injects" the data into the HTML placeholders defined in that template.
4. **Response Delivery:** The view returns an `HttpResponse` object containing the completed HTML, which the browser then renders for the user.

Example: In a recipe app, if a user clicks on "French Toast", the **URL** sends the request to, say, a `recipe_detail` **View**. The view queries the database for the ingredients of "French Toast", combines that data with the `recipe_detail.html` **Template**, and sends the finished page back to the user's screen.

2. Imagine you're working on a Django web development project, and you anticipate that you'll have to reuse lots of code in various parts of the project. In this scenario, will you use Django function-based views or class-based views, and why?

If the goal is heavy code reuse across a large project, I would definitely choose Class-Based Views (CBVs).

While Function-Based Views (FBVs) are great for simple, one-off pages, they quickly become repetitive. If in this project I have ten different pages that all need to list items from a database, an FBV would require me to write the same "fetch and render" logic ten times.

Why CBVs win for reuse:

- **Inheritance:** I can create a "Master View" with the common logic and have every other view "inherit" from it.

- **Mixins:** These are like "plug-ins" for classes. If I need five different views to require a user to be logged in, I just add a `LoginRequiredMixin` to those classes instead of writing `if user.is_authenticated` over and over again.
- **DRY Principle:** It keeps the code "Don't Repeat Yourself" compliant, making the project much easier to maintain as it grows.

3. Read Django's documentation on the Django template language and make some notes on its basics.

The Django Template Language (DTL) is the bridge between Python logic and static HTML. It uses specific tags to tell the computer: "This isn't just text, this is a command."

- **Variables `{{ variable }}`:** These act as placeholders. When the view sends data to the template, the template replaces these double-braces with the actual value of that variable (e.g., `{{ recipe_name }}`).
- **Tags `{% tag %}`:** These handle the logic.
  - **Logic:** `{% if %}` and `{% for %}` allow to show content only under certain conditions or loop through a list of recipes.
  - **Inheritance:** The `{% extends "base.html" %}` and `{% block content %}` tags are the most powerful. They allow you to build a single "skeleton" (like header and footer) and just swap out the middle section for different pages.
- **Filters `{{ variable|filter }}`:** These modify how a variable looks without changing the data. For example, `{{ date|date:"M Y" }}` formats a date, or `{{ name|capfirst }}` capitalizes the first letter.

Exercise 2.5: Django MVT Revisited

Learning Goals

- Add images to the model and display them on the frontend of your application
- Create complex views with access to the model
- Display records with views and templates

Reflection Questions

1. In your own words, explain Django static files and how Django handles them.

Static files are assets that don't change while the application is running, such as CSS, JavaScript, and images. Django handles these through a specific workflow:

1. Storage and Organization

Static assets are stored inside a folder named static/ within each of your apps. For example, CSS for the bookstore would live in books/static/books/style.css. Django is pre-configured to look for these folders automatically.

2. Template Integration

To use these files, load the {% load static %} tag at the top of the template. This allows the {% static 'path/to/file' %} tag to generate the correct URL for the browser to find that file.

3. Collection for Production

In development, runserver automatically serves static files for immediate use. For production, the collectstatic command gathers all assets into a single folder (defined by `STATIC_ROOT` in settings), so a dedicated web server can serve them efficiently.

4. Static vs. Media

Static files are part of the code (CSS, JS, logos), whereas Media files are user-uploaded content, like the pic field in our Book model. Media is handled separately via MEDIA_URL and MEDIA_ROOT settings.

2. Look up the following two Django packages on Django's official documentation and/or other trusted sources. Write a brief description of each.

Package	Description
ListView	**`ListView`** is a generic class-based view designed to represent a list of objects from a specific model. ● **Main Purpose**: It automates the process of fetching all records (or a filtered set) from the database and passing them to a template. ● **Automatic Context**: By default, it passes the list to the template as a variable named `object_list`, although can be renamed using `context_object_name`. ● **Workflow**: It performs the database query, handles pagination if configured, and renders the specified `template_name` with the results.
DetailView	**`DetailView`** is a generic class-based view used to display the data for one specific instance of a model. ● **Main Purpose**: It targets a single object, usually identified by a unique ID or slug passed through the URL. ● **Automatic Context**: By default, it passes the individual object to the template as a variable named `object`, or a custom name like `book`, defined as `context_object_name`. ● **Workflow**: It uses the primary key (`pk`) from the URL to look up the exact record in the database. If the object does not exist, it automatically raises a 404 "Not Found" error.

3. You’re now more than halfway through Achievement 2! Take a moment to reflect on your learning in the course so far. How is it going? What’s something you’re proud of so far? Is there something you’re struggling with? What do you need more practice with? You can use these notes to guide your next mentor call.

What I’m Proud Of

- **Mastering the MVT Pattern:** I’ve successfully grasped how Django connects models, views, and templates to create a functional web application.
- **Writing Robust Tests:** I’ve moved beyond hardcoded IDs in testing to use dynamic methods like `reverse()` and object-specific primary keys, making my test suite more reliable and professional.

What I Struggled With

- **Method Logic and Persistence:** I found it challenging to define the logic for the `calculate_difficulty` class method, specifically deciding whether to save the field within that method or not. It has taken me some time to weigh different options in database tables inter-operation, and ultimately I decided not to save into a record in a different table to maintain better data integrity.

What I Need More Practice With

- **Advanced Class Methods:** I'd like to dive deeper into when and how to use `@classmethod` versus `@staticmethod` or standard model methods for complex calculations.
- **Complex Model Relationships:** While I understand the basics, I want more practice with how different tables should interact without creating "side effects" (like unexpected saves in other tables).
- **URL Namespacing and Reversing:** I want to ensure I thoroughly understand how Django’s namespace resolution works so I can build even larger, multi-app projects without confusion.

The specific decision regarding `calculate_difficulty` class method was discussed in my regular mentor call.

Exercise 2.6: User Authentication in Django

Learning Goals

- Create authentication for your web application
- Use GET and POST methods
- Password protect your web application’s views

Reflection Questions

1. In your own words, write down the importance of incorporating authentication into an application. You can take an example application to explain your answer.

Authentication is essentially the digital gatekeeper that verifies identity before granting access to specific areas of a site. It's the difference between a public park and a private club - everyone can see the front gate, but only members get a key to the clubhouse.

Take an online shop as an example:

- **Protecting Personal and Business Data: While anyone can browse the product catalog, you wouldn't want a customer to see another user's credit card details, or a guest to access the store's internal sales reports.**
- **Accountability and Security: It ensures that every action is linked to a specific identity. If a customer places an order or a manager updates a price, the system knows exactly who is responsible, preventing unauthorized or untraceable changes.**
- **Personalization: It allows the application to provide a tailored experience. For a customer, this means seeing their order history and saved addresses; for a store owner, it means access to administrative tools and performance metrics.**

2. In your own words, explain the steps you should take to create a login for your Django web application.

Here are the essential steps to set up a login system in Django:

- **Define the Views**: Start by creating a view - like the `login_view` in `bookstore/views.py` - that handles both GET requests (to display the form) and POST requests (to process the data). Inside this view, use Django's built-in `AuthenticationForm` to handle the data validation.
- **Handle Authentication Logic**: Once the form is submitted, use the `authenticate` function to check the credentials against the database and the `login` function to actually establish the user session.
- **Map the URLs**: Ensure the application knows where to find the login logic by adding a path to `urls.py` and giving it a name, like `name='login'`, to reference it easily throughout the project.
- **Create the Template**: Build the HTML form (e.g., `login.html`) and ensure it includes the `{% csrf_token %}` tag for security. Also use `{{ form.as_p }}` to let Django automatically render the necessary input fields.
- **Configure Settings**: Finally, tell Django where to send users by default by defining `LOGIN_URL` and `LOGIN_REDIRECT_URL` in the `settings.py` file.

3. Look up the following three Django functions on Django's official documentation and/or other trusted sources and write a brief description of each.

Function	Description
authenticate()	This function verifies user credentials (usually a username and password). It checks the provided data against the database and, if valid, returns a `User` object. If the credentials don't match any record, it returns `None`. It doesn't log the user in, it only confirms that he is who he says he is.
redirect()	A shortcut function that sends the user to a different URL. It's

	commonly used after a successful action - like logging in or submitting a form - to transition the user to a new page (e.g., sending him from the login page to the sales records dashboard).
include()	This function allows referencing other URL configurations (`urls.py`). It's used to keep the project organized by "plugging in" app-specific URLs into the main project. For example, it allows the main `urls.py` to hand off any path starting with `books/` to the `books` app.

Exercise 2.7: Data Analysis and Visualization in Django

Learning Goals

- Work on elements of two-way communication like creating forms and buttons
- Implement search and visualization (reports/charts) features
- Use QuerySet API, DataFrames (with pandas), and plotting libraries (with matplotlib)

Reflection Questions

1. Consider your favorite website/application (you can also take CareerFoundry). Think about the various data that your favorite website/application collects. Write down how analyzing the collected data could help the website/application.

Data Collection and Analysis

Here is how Amazon and Netflix leverage their collected data to drive business value:

- **Amazon**: Beyond basic transaction history, Amazon tracks "micro-behaviors" such as how long your cursor hovers over a product, which items you leave in your cart, and your search patterns across different categories.
  - **Predictive Analytics**: By analyzing this, they use "Anticipatory Shipping" to move inventory to local hubs before a purchase is even made.
  - **Dynamic Pricing**: They adjust prices millions of times a day based on competitor data, inventory levels, and individual user demand to maximize conversion.
  - **Recommendation Engine**: Data analysis powers the "Customers who bought this also bought..." feature, which is responsible for a massive portion of their total revenue by creating a seamless cross-selling environment.
- **Netflix**: Netflix is essentially a data science company that also streams video. They collect data on when you pause, rewind, or fast-forward, and even what time of day you watch specific genres.

- **Content Strategy**: Instead of guessing which shows will be hits, they analyze user preferences to greenlight "Originals" that have a mathematically high probability of success based on existing viewer clusters.
- **Personalized Artwork**: They use A/B testing on thumbnails: if you watch a lot of romance, the thumbnail for a movie might show the lead couple, whereas an action fan might see an explosion from the same film.
- **Network Optimization**: They analyze streaming quality data globally to determine where to place their "Open Connect" servers, ensuring users experience less buffering by predicting localized demand.

2. Read the Django official documentation on QuerySet API. Note down the different ways in which you can evaluate a QuerySet.

Evaluating a QuerySet in Django

A Django QuerySet is "lazy," meaning it doesn't actually hit the database until it is evaluated. According to the documentation, you can evaluate a QuerySet in the following ways:

- **Iteration**: The first time you loop over it (e.g., for sale in Sale.objects.all():), the query runs.
- **Slicing**: Using a step parameter in a slice (e.g., qs[0:10:2]) evaluates the query immediately.
- **Pickling/Caching**: Saving the results to a cache or pickling them for storage forces evaluation.
- **repr()**: Calling repr() on a QuerySet (often done in the terminal or during debugging) evaluates it so the results can be displayed.
- **len()**: Calling len() on a QuerySet runs the query and returns the length of the result list (though count() is more efficient for just the number).
- **list()**: Explicitly converting the QuerySet to a list (e.g., list(qs)) forces evaluation.
- **bool()**: Testing the QuerySet in a boolean context (e.g., if qs:) will run the query.

In Django, filter() and values() are fundamental parts of the QuerySet API, but they behave differently regarding evaluation:

- **filter()**: This method is **lazy**. When you call Sale.objects.filter(book__title='Macbeth'), no database query is executed. It simply returns a new, refined QuerySet. It only hits the database when you actually try to use the data (e.g., in a for loop, by calling len(), or by converting it to a list).
- **values()**: This is also **lazy**. It returns a QuerySet that yields dictionaries instead of model instances. Calling qs.values('price', 'quantity') does not trigger the database immediately. However, because it is often used as an intermediate step to prepare data for a frontend or a DataFrame (as we did with pd.DataFrame(qs.values())), it is frequently evaluated shortly after it is defined.

3. In the Exercise, you converted your QuerySet to DataFrame. Now do some research on the advantages and disadvantages of QuerySet and DataFrame, and explain the ways in which DataFrame is better for data processing.

QuerySet vs. DataFrame: Data Processing

In our exercise, we moved data from a Django QuerySet into a pandas DataFrame. Here is why that is often necessary for data science tasks:

Advantages and Disadvantages

- **QuerySet:**
    - ***Pros*: Highly efficient for CRUD (Create, Read, Update, Delete) operations and database-level filtering. It handles relationships (like ForeignKey) seamlessly.**
    - ***Cons*: Very limited mathematical capabilities. You can do basic aggregations (Sum, Avg), but complex statistical analysis is difficult and slow.**
- **DataFrame:**
    - ***Pros*: Designed for heavy-duty data manipulation. It handles missing data well and allows for complex vectorization (performing operations on entire columns at once).**
    - ***Cons*: It is an "in-memory" tool. If you load 10 million rows from a database into a DataFrame, you might crash your server's RAM.**

Why DataFrame is better for processing:

1. **Vectorized Operations: In Python, loops are slow. Pandas performs operations in C under the hood. For example, calculating a 10% tax on every sale in a QuerySet requires a Python loop; in a DataFrame, it's a single, nearly instant operation: df['total'] * 1.1.**
2. **Complex Transformations: Tasks like "Pivot Tables," rolling averages, or correlation matrices are built-in functions in pandas. Doing these in a Django QuerySet would require incredibly complex SQL or slow Python post-processing.**
3. **Integration: DataFrames are the standard input for almost all Python visualization (Matplotlib, Seaborn) and Machine Learning (Scikit-learn) libraries.**

Exercise 2.8: Deploying a Django Project

Learning Goals

- Enhance user experience and look and feel of your web application using CSS and JS
- Deploy your Django web application on a web server
- Curate project deliverables for your portfolio

Reflection Questions

1. Explain how you can use CSS and JavaScript in your Django web application.

CSS and JavaScript use in a Django application is supported by the Static Files system. Django distinguishes "static" files (CSS, JS, images you create) from "media" files (content uploaded by users).

a. The Setup: settings.py

Before using these files, you must tell Django where they live by defining a STATIC_URL (the web address) and a STATICFILES_DIRS (the physical folder on local computer).

```
# bookstore/settings.py
STATIC_URL = '/static/'
```

`STATICFILES_DIRS = [BASE_DIR / 'static']`

b. The Storage: File Structure

Standard practice is to organize the assets into subdirectories within a root static folder:

- static/css/style.css
- static/js/script.js
- static/images/bookstore.jpg

c. The Implementation: Templates

To actually use these in the HTML template, you follow a three-step process:

1. **Load the tag**: Place {% load static %} at the very top of your template.
2. **Link CSS**: Inside the <head> section of your base.html, use the static template tag to build the URL.
   `<link rel="stylesheet" href="{% static 'css/style.css' %}">`
3. **Link JS**: Usually placed at the bottom of the base.html, right before the closing </body> tag.
   `<script src="{% static 'js/main.js' %}"></script>`

d. Why Use the {% static %} Tag?

Instead of hardcoding a path like /static/css/style.css, the template tag makes the app portable. If you later decide to host your files on a different server (like Amazon S3 or a CDN), you only change the setting in one place, and the {% static %} tag automatically updates every link across your entire site.

2. In your own words, explain the steps you'd need to take to deploy your Django web application.

To deploy your Django application, you essentially move it from a local development environment to a live web server. Here are the most critical steps:

- **Security Check**: Set `DEBUG = False` and generate a fresh, unique `SECRET_KEY` in `settings.py`.
- **Static Management**: Run `python manage.py collectstatic` to gather all CSS/JS files into one root folder for the server to serve.
- **Database Migration**: Ensure your production database is set up and run `python manage.py migrate` to create the tables.
- **Environment Variables**: Move sensitive data (API keys, DB credentials) into an `.env` file instead of hardcoding them.
- **Web Server**: Use a production-grade server like **Gunicorn** or **uWSGI** (Django's built-in server is for development only).
- **Reverse Proxy**: Configure **Nginx** or **Apache** to handle incoming traffic and serve your static/media files.
- **Hosting**: Push your code to a platform like **Heroku**, **AWS**, or **DigitalOcean**.

3. (Optional) Connect with a few Django web developers through LinkedIn or any other network. Ask them for their tips on creating a portfolio to showcase Python programming and Django skills. Think about which tips could help you improve your portfolio.

4. You've now finished Achievement 2 and, with it, the whole course! Take a moment to reflect on your learning:
    a. What went well during this Achievement?
    b. What's something you're proud of?
    c. What was the most challenging aspect of this Achievement?
    d. Did this Achievement meet your expectations? Did it give you the confidence to start working with your new Django skills?

What went well: Successfully implementing a modular design using template inheritance, which allowed for a consistent UI across the Bookstore application.
Proudest moment: Resolving the *NoReverseMatch* and *Static* path errors, proving a deep understanding of how Django maps URLs to views and assets.
Challenges: The most difficult hurdle was the "path puzzle" - specifically configuring `STATICFILES_DIRS` and `MEDIA_ROOT` in `settings.py` to ensure files load correctly from the root directory.
Expectations: This achievement provided the confidence to manage a professional Django project structure, moving beyond simple code to a fully integrated web environment.

Well done - you've now completed the Learning Journal for the whole course.