CHAPTER 1: PYTHON MASTERY

Architect's Guide to Professional Python

An architect doesn't just lay bricks; they design the blueprint. They think about how the foundation supports the entire structure, how systems interact, and how to build for future expansion. This guide frames every Python concept from that perspective.

Chapter 1: Python Mastery (Architect's Edition)

Your goal is to write code that is not just functional, but also **scalable, maintainable, and resilient**. This is the bedrock of your Medical-AI roadmap.

---

1. Advanced OOP: The Blueprint of Your System

The Architect's Mindset: Don't think of classes as just containers for data and functions. Think of them as **blueprints for the core entities in your system**. For NEETPrepGPT, your entities are `User`, `Question`, `Exam`, `Subject`, `ScrapedContent`. For Symptom2Specialist, they are `Patient`, `Symptom`, `FHIRRecord`, `Diagnosis`. OOP allows you to model this complex reality in a clean, logical way.

Core Concepts:

- **Inheritance & Polymorphism:**
    - **Why it Matters:** Avoids code duplication and creates a logical hierarchy. An architect asks, "What are the common properties and behaviors here?"
    - **Your Application (NEETPrepGPT):** You'll be extracting different types of questions. Instead of writing separate logic for each, design a base `Question` class.

```
class Question:
    def __init__(self, text: str, chapter: str):
        self.text = text
        self.chapter = chapter

    def display(self):
        raise NotImplementedError("Subclass must
implement this method!")

class MultipleChoiceQuestion(Question):
    def __init__(self, text: str, chapter: str, options:
list[str], correct_option: int):
```

```
        super().__init__(text, chapter)
        self.options = options
        self.correct_option = correct_option

    def display(self):
        # Logic to display MCQ format
        print(f"Q: {self.text}\nOptions:
{self.options}")

class TrueFalseQuestion(Question):
    def __init__(self, text: str, chapter: str, is_true:
bool):
        super().__init__(text, chapter)
        self.is_true = is_true

    def display(self):
        # Logic to display T/F format
        print(f"Q: {self.text}\n(True/False)")
```

- This polymorphic design means you can have a list of different `Question` objects and just call `question.display()` on each one, letting Python handle which specific display method to use.

- **Composition over Inheritance:**
  - **Why it Matters:** This is a key principle for building flexible systems. Inheritance creates a rigid "is-a" relationship (`MCQ` **is a** `Question`). Composition creates a flexible "has-a" relationship.
  - **Your Application (Symptom2Specialist):** A `Patient` record is not a type of `Doctor`, but it **has a** list of `Symptoms` and **has a** `MedicalHistory`.

```
class MedicalHistory:
    # ... details of past illnesses, allergies etc.

class Patient:
    def __init__(self, name: str, patient_id: str):
        self.name = name
        self.id = patient_id
        self.history = MedicalHistory() # Composition! A
Patient 'has a' MedicalHistory.
        self.symptoms = []
```

  - This makes your system modular. You can swap out the `MedicalHistory` component without breaking the `Patient` class.

2. Pythonic Constructs: Elegant & Efficient Machinery

The Architect's Mindset: These aren't just "features"; they are tools to write code that is more readable, efficient, and less error-prone. They allow you to **separate concerns**, keeping your core logic clean.

Core Concepts:

- **Decorators:**
  - **Why it Matters:** They add functionality (logging, timing, caching, authentication) to your functions without cluttering the function's code. This is essential for building a clean API with FastAPI.
  - **Your Application (NEETPrepGPT):** You'll need to know how long your web scraper or API calls take.

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"'{func.__name__}' executed in {end_time
- start_time:.4f}s")
        return result
    return wrapper

@timing_decorator
def scrape_biology_chapter(url: str):
    # Complex scraping logic here...
    print(f"Scraping from {url}...")
    time.sleep(2) # Simulate work
    return "Scraped Data"

scrape_biology_chapter("http://example.com/biology")
# Output: 'scrape_biology_chapter' executed in 2.0005s
```

  - Imagine adding `@require_authentication` to your FastAPI endpoints. The logic for checking a user's JWT token lives in the decorator, not inside every single API function.

- **Generators:**

- **Why it Matters:** Memory efficiency. An architect must plan for scale. What happens when your scraper finds 50,000 questions? Or when you process a 2GB FHIR data file? Loading it all into a list in memory will crash your application. Generators process one item at a time.
- **Your Application (Data Ingestion):** Refactor any data processing to use `yield`.

```
# Bad: Memory Hog
def find_questions_in_file(filepath):
    questions = []
    with open(filepath, 'r') as f:
        for line in f:
            if "<question>" in line:
                questions.append(line) # Appends ALL
questions to a list
    return questions

# Good: Memory Efficient Generator
def yield_questions_from_file(filepath):
    with open(filepath, 'r') as f:
        for line in f:
            if "<question>" in line:
                yield line # Yields one question at a
time

# You can now process millions of questions with minimal
memory usage
for question in
yield_questions_from_file('huge_question_bank.xml'):
    process(question)
```

---

3. Modern Python: Building for Robustness & Clarity

The Architect's Mindset: Your code is not just for the computer; it's for your future self and potential collaborators. Clarity prevents bugs. Robustness ensures your application doesn't fall over when something unexpected happens.

Core Concepts:

- **Type Hinting:**
  - **Why it Matters:** It's self-documenting code. It makes your function signatures a "contract." It tells anyone (and your IDE) exactly what kind of

data to pass in and what to expect back. This is non-negotiable for building reliable AI systems where data integrity is paramount.

- **Your Application (Symptom2Specialist):**

```
from typing import List, Dict

# This is crystal clear. It takes a list of symptoms
# and returns a dictionary of potential diagnoses with
probabilities.
def predict_specialist(symptoms: List[str]) -> Dict[str,
float]:
    # BioBERT model inference logic here...
    predictions = {"Cardiologist": 0.75, "Neurologist":
0.15}
    return predictions
```

- **Comprehensive Exception Handling:**
  - **Why it Matters:** An architect plans for failure. What if the website you're scraping is down? What if a database connection fails? What if the OpenAI API returns an error? Your application must handle this gracefully.
  - **Your Application (General):** Create your own custom exceptions to make error handling more specific and meaningful.

```
# Define custom exceptions
class ScraperRateLimitError(Exception):
    """Raised when the scraper is blocked."""
    pass

class InvalidQuestionFormatError(Exception):
    """Raised when scraped data doesn't match expected
format."""
    pass

# Use them in your code
def get_questions_from_source(source_url: str):
    try:
        response = requests.get(source_url)
        if response.status_code == 429: # Too Many
Requests
            raise ScraperRateLimitError(f"Rate limited
by {source_url}")
        response.raise_for_status() # Raise HTTPError
for other bad responses
```

```
        # ... parsing logic that might fail
        if not is_valid(response.text):
            raise InvalidQuestionFormatError("Content
format is invalid.")
    except requests.exceptions.RequestException as e:
        print(f"Network error: {e}")
    except ScraperRateLimitError as e:
        print(f"Architectural Problem: {e}. Need to back
off and retry.")
    except InvalidQuestionFormatError as e:
        print(f"Data Integrity Problem: {e}. Skipping
this source.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}") #
Generic fallback
```

🛠️ Mini-Project Blueprint: The "StudyNotes" CLI Tool

This project will be your sandbox for forging these architectural concepts into practical skills.

The Vision: A command-line tool to manage study notes, where each note can be tagged by subject and chapter, making it a micro-version of a content management system.

1. Model the Domain (OOP):

- Create a base `Note` class with properties like `content`, `timestamp`.
- Create inherited classes like `TextNote(Note)` and `MCQNote(Note)`. The `MCQNote` will have additional properties like `options` and `answer`.
- Create a `Notebook` class that **holds a list of notes** (Composition!). It will have methods like `add_note()`, `find_notes_by_tag()`.

2. Implement Features (Decorators & Generators):

- Create a `@log_action` decorator. Apply it to `add_note()` and `delete_note()` so that every time a note is added or removed, a message like `"[ACTION]: Added new TextNote"` is printed.
- Implement the `find_notes_by_tag()` method as a **generator**. It should `yield` each matching note one by one, not return a big list.

3. Ensure Robustness (Modern Python):

- Add **type hints** to all your class methods and functions.
- Create and use a custom exception, `NoteNotFoundError`, which you `raise` in your `Notebook` class if a user tries to access a note that doesn't exist.

Starter Skeleton:

```
# notes_manager.py
from datetime import datetime
from typing import List, Generator

# --- Custom Exceptions ---
class NoteNotFoundError(Exception):
    pass

# --- Decorators ---
def log_action(func):
    def wrapper(*args, **kwargs):
        print(f"[ACTION]: Running '{func.__name__}'...")
        result = func(*args, **kwargs)
        print(f"[ACTION]: Finished '{func.__name__}'.")
        return result
    return wrapper

# --- OOP Models ---
class Note:
    def __init__(self, content: str, tags: List[str]):
        self.content = content
        self.tags = tags
        self.created_at = datetime.now()

    def __repr__(self):
        return f"Note(content='{self.content[:20]}...', tags=
{self.tags})"

class Notebook:
    def __init__(self):
        self._notes: List[Note] = []

    @log_action
    def add_note(self, note: Note):
        self._notes.append(note)

    def find_notes_by_tag(self, tag: str) -> Generator[Note, None,
None]:
        print(f"\nSearching for notes tagged with '{tag}':")
        for note in self._notes:
            if tag in note.tags:
                yield note

# --- Main Application Logic ---
if __name__ == "__main__":
```

```
    my_notebook = Notebook()

    note1 = Note("Mitochondria is the powerhouse of the cell.",
["biology", "cell"])
    note2 = Note("Python generators use the `yield` keyword.",
["python", "programming"])

    my_notebook.add_note(note1)
    my_notebook.add_note(note2)

    for found_note in my_notebook.find_notes_by_tag("biology"):
        print(f"  - Found: {found_note}")
```

By completing this, you won't just have learned these concepts; you'll have **architected a small, robust system** from the ground up. This is the exact thinking process you will apply, scaled up, to build NEETPrepGPT. Good luck.

CHAPTER 2: THE DATA SCIENCE STACK

Chapter 2: The Data Science Stack: Architect's Field Notes

Your thinking is correct: this isn't just about learning libraries. This is about building the **sensory and analytical nervous system** for your applications. [cite_start]Your AI is only as good as the data it's trained on, and this stack is your toolkit for making that data pristine, insightful, and actionable[cite: 36]. As the architect of NEETPrepGPT and future systems, you're not just writing code; you are designing the data pipeline that fuels the entire business logic.

🧠 Part 1: NumPy — The High-Performance Computing Bedrock

Architect's Mindset: Don't think of NumPy as just a "math library." Think of it as the **low-level, high-performance foundation** upon which every intelligent data operation is built. When you're dealing with millions of data points—like user scores in NEETPrepGPT or symptom vectors in your Symptom2Specialist bot—standard Python lists are unacceptably slow and memory-hungry. An architect chooses NumPy for its raw speed and efficiency. It’s the concrete and steel rebar of your data structure.

- **Core Concepts:**
    - **The `ndarray`:** The heart of NumPy. It's a contiguous block of memory, which means operations are lightning-fast. This is your primary data object.

    - **Vectorization:** This is the *architectural pattern* that makes NumPy shine. Instead of writing slow `for` loops, you apply operations to entire arrays at once. This is a non-negotiable for scalable systems.
    - **Broadcasting:** A powerful feature that allows NumPy to perform operations on arrays of different shapes, making your code cleaner and more intuitive.

- **Connecting to Your Roadmap:**

    - **NEETPrepGPT:** When you analyze the performance of 10,000 students on 100,000 MCQs, you'll use NumPy arrays to instantly calculate average scores, standard deviations, and percentile ranks. This is impossible to do efficiently with standard Python.
    - **Symptom2Specialist (Phase 2):** The BioBERT model you plan to use works with numerical representations of text called embeddings. These embeddings are massive NumPy arrays. All the math for comparing symptom similarity will happen here.

- ☑ **Architect's Checklist (Your Todos):**

    1. [cite_start]**Source & Validate Initial Telemetry:** "Acquire a Dataset" means sourcing the raw data that will become your system's lifeblood[cite: 161]. Find a CSV of NEET questions or user performance data and load it. Your first job is to understand its dimensions (`.shape`) and data types (`.dtype`).
    2. [cite_start]**Establish Baseline Performance Metrics:** "Analyze with NumPy" means performing initial, high-speed statistical calculations to understand the landscape[cite: 161]. Use NumPy to compute the mean, median, and standard deviation for any numerical columns (e.g., question difficulty, time taken to answer). This is your first health check on the data.

🧠 Part 2: Pandas — The Blueprint for Data Structure & Integrity

Architect's Mindset: If NumPy is the raw material, Pandas is the **architectural blueprint**. It takes raw, messy, unstructured data and imposes a clean, labeled, and relational structure—the **DataFrame**. As an architect, your primary job here is **Data Integrity**. Raw data from web scraping or user input is always dirty. Pandas is your tool for cleaning, transforming, and preparing it so the rest of your system can trust it. A bug caused by a missing value (`NaN`) that you failed to handle is an architectural failure.

- **Core Concepts:**

    - **DataFrame & Series:** The two primary data structures. A DataFrame is your main blueprint—a 2D table with labeled rows and columns. A Series is a single column.

- **Data Cleaning:** Your most critical job. This involves handling missing data (`.fillna()`, `.dropna()`), correcting data types, and removing duplicates.
- **Filtering & Querying:** Selecting subsets of data based on conditions. This is fundamental for asking questions.
- **Grouping & Aggregation (`.groupby()`):** The powerhouse of analysis. This allows you to group data by categories (e.g., by subject, by topic, by user) and then run calculations on those groups.

- **Connecting to Your Roadmap:**
  - [cite_start]**NEETPrepGPT:** You will scrape MCQs from various sources[cite: 73, 83]. This data will be a mess. You'll use Pandas to create a master DataFrame, clean the text, standardize the columns (`'question_text'`, `'option_a'`, `'correct_answer'`), and filter out irrelevant content before it ever reaches your vector database. You'll also use `.groupby('user_id')` to analyze individual student performance.
  - **Future Corporate Projects:** When a stakeholder asks, "Which user demographic is most engaged with our platform?" you won't write a complex SQL query. You'll use Pandas to load the data, group by demographic columns, and aggregate engagement metrics in a few lines of code. This is how you deliver business intelligence fast.

- ☑ **Architect's Checklist (Your Todos):**
  1. [cite_start]**Engineer the Core DataFrames:** "Manipulate with Pandas" means creating the canonical, clean DataFrames that will serve your entire application[cite: 161]. Load your dataset into a DataFrame. Immediately handle missing values, check data types, and create a "golden source" of clean data.
  2. **Extract Actionable Intelligence:** Use `.groupby()` to answer at least one meaningful question from your data. For example: "What is the average difficulty rating for questions in 'Biology' versus 'Physics'?" This demonstrates you can move from raw data to insight.

🧠 Part 3: Matplotlib & Seaborn — The Executive Communication Layer

Architect's Mindset: Data has no value if it can't be understood by humans. Matplotlib and Seaborn are not "charting libraries"; they are your **communication layer**. As an architect, you use them to tell a story, prove a hypothesis, or provide a diagnostic view of your system's health. A good plot can reveal a critical bug or a massive business opportunity that raw numbers would hide.

- **Core Concepts:**

    - **Matplotlib:** The foundational library. It's powerful and highly customizable but can be complex. Think of it as the engine.
    - **Seaborn:** Built on top of Matplotlib. It provides a beautiful, high-level interface for creating common statistical plots. Think of it as the sleek car body built on the powerful engine. For 90% of your needs, start with Seaborn.

- **Connecting to Your Roadmap:**

    - [cite_start]**NEETPrepGPT:** During your pilot phase, you'll need to report on Key Performance Indicators (KPIs) like user retention and engagement[cite: 157, 159, 160]. [cite_start]You will use Seaborn to create histograms of MCQ accuracy rates [cite: 161] [cite_start]and line charts showing user activity over the 4-week trial[cite: 156]. These charts are your proof of success to yourself and any future stakeholders.
    - **Future Corporate Projects:** This is perhaps the most crucial skill for career growth. You will be expected to build dashboards for business leaders. Being able to quickly generate a bar chart of "Revenue by Customer Segment" or a scatter plot of "Ad Spend vs. User Signups" is how you demonstrate the value of your technical work in the language the business understands: visuals.

- ☑ **Architect's Checklist (Your Todos):**

    1. [cite_start]**Build the Initial Diagnostic Dashboard:** "Visualize Results" means creating the first visual report on your data's health and patterns[cite: 161]. Use Matplotlib or Seaborn to create:
        - A **histogram** to understand the distribution of a numerical column (e.g., how many questions have a difficulty score of 1 vs. 5?).
        - A **bar chart** to compare categories (e.g., the number of questions available for each subject). [cite_start]This is your first step towards building the admin dashboard for your MVP[cite: 12].

CHAPTER 3: ASYNCHRONOUS PROGRAMMING WITH ASYNCIO

Architect's Guide to Asynchronous Python

A traditional program is like a person with a single-track mind, doing one task at a time. An asynchronous system is like a master chef in a busy kitchen, starting multiple dishes at once. They put the potatoes on to boil (a slow I/O task), and while they're waiting, they start chopping vegetables (another task). They don't just stand there staring at the pot. This is the essence of building a backend that can handle thousands of users without breaking a sweat.

Chapter 3: Asynchronous Programming with `asyncio`

Your goal here is to understand how to manage and execute tasks that are **I/O-bound**—tasks where your program is just waiting for something external, like an API response, a database query, or a file to be written.

---

1. `async`/`await`: The Language of Concurrency

The Architect's Mindset: `async` and `await` are not just keywords; they are signals to the system's "event loop." When you `await` a function, you are telling the architect (the event loop), "This task is going to take a while. You can go do other useful work and come back to me when it's done." This prevents the entire application from freezing while waiting for one slow operation.

Core Concepts:

- `async def`: This defines a **coroutine**, a special function that can be paused and resumed. It's the "dish" the chef can start.
- `await`: This is the "pause" button. It can only be used inside an `async def` function. It tells the event loop to pause the current coroutine and work on something else until the awaited task (e.g., an API call) completes.

Your Application (NEETPrepGPT Backend): When a user requests to generate an exam, your backend might need to:

1. Authenticate the user (database query).
2. Fetch user preferences (another database query).
3. Call the OpenAI API to generate questions.
4. Log the event to a file.

A synchronous server would do these one by one, making the user wait. An async server can **initiate all these I/O-bound tasks at nearly the same time.**

---

2. The Event Loop: The Master Conductor

The Architect's Mindset: The event loop is the heart of any `asyncio` application. It's the conductor of your orchestra. It keeps track of all the tasks (coroutines) that are running, paused, or finished. It's the "chef" who knows which dish needs attention at any given moment. You don't usually interact with it directly, but understanding its role is crucial for debugging and architectural design.

The loop constantly asks: "Is anyone done waiting? No? Okay, is there any other work I can do? Yes? Let's do that."

---

3. Handling Concurrent I/O: The Real Power

The Architect's Mindset: This is where the magic happens. An architect's job is to design a system that maximizes resource utilization. `asyncio` allows your CPU to do meaningful work instead of sitting idle while waiting for the network or disk.

Core Concepts:

- `asyncio.gather()`: This is your tool for running multiple coroutines concurrently. You give it a list of tasks, and it runs them all, returning the results only when all are complete. It's like telling the chef, "Start these three dishes, and let me know when all of them are ready to be served."

Your Application (Symptom2Specialist): When a user inputs symptoms, you might need to query multiple external medical APIs or databases simultaneously to gather information. `asyncio.gather()` is perfect for this.

🛠️ Mini-Project Blueprint: The "Concurrent API Fetcher"

This project will give you hands-on experience with the core `asyncio` workflow, simulating a real-world scenario where your backend needs to talk to multiple external services at once.

The Vision: A script that fetches data from two different public APIs simultaneously and, at the same time, performs a simulated slow database write, demonstrating that none of the tasks block the others.

1. Set Up Your Tools:

- You'll need an HTTP library that supports `asyncio`. `aiohttp` is the standard choice.

```
pip install aiohttp
```

2. Write Your Coroutines (`async def`):

- **API Fetcher:** Create a coroutine `fetch_api_data(url)` that takes a URL, uses `aiohttp` to get the data, and returns the JSON response.
- **Simulated I/O:** Create a coroutine `simulate_db_write(data)` that prints a message, waits for a few seconds using `await asyncio.sleep(seconds)`, and then prints a completion message. This mimics a slow, non-blocking database operation.

3. Run Concurrently (`asyncio.gather()`):

- In your main `async` function, create tasks for fetching data from two different APIs (e.g., a Pokémon API and a public facts API).
- Create a task for your simulated database write.
- Use `asyncio.gather()` to run all these tasks concurrently.
- Print the results as they come in. Notice how the faster API might return before the slower one, and the "database write" happens in the background.

Starter Code:

```
# concurrent_fetcher.py
import asyncio
import aiohttp
import time

# --- Coroutines ---

async def fetch_api_data(session: aiohttp.ClientSession, url: str)
-> dict:
    """Asynchronously fetches data from a URL and returns JSON."""
    print(f"Starting fetch from {url}...")
    async with session.get(url) as response:
        # await tells the event loop to pause here until the
network responds
        data = await response.json()
        print(f"Finished fetch from {url}.")
        return data

async def simulate_db_write(log_entry: str) -> None:
    """Simulates a slow, non-blocking database write."""
    print(f"Starting simulated DB write for: '{log_entry}'...")
    # await tells the event loop to pause here, allowing other
tasks to run
    await asyncio.sleep(2) # Simulate a 2-second write operation
    print(f"Finished simulated DB write for: '{log_entry}'.")

# --- Main Conductor ---

async def main():
    """The main entry point for our concurrent script."""
    start_time = time.time()

    # Use a single session for all requests for efficiency
    async with aiohttp.ClientSession() as session:
        # Create a list of tasks to run concurrently
        # These are the "dishes" we are telling the "chef" to
start
        tasks = [
```

```
            fetch_api_data(session,
"https://pokeapi.co/api/v2/pokemon/1"), # Fast API
            fetch_api_data(session, "https://official-joke-
api.appspot.com/random_joke"), # Another fast API
            simulate_db_write("User login event")
        ]

        # asyncio.gather() runs all tasks in the list concurrently
        # It waits until ALL of them are complete
        results = await asyncio.gather(*tasks)

    # Process results after they are all available
    pokemon_data, joke_data, _ = results
    print("\n--- All tasks completed! ---")
    print(f"Pokemon fetched: {pokemon_data['name']}")
    print(f"Joke fetched: {joke_data['setup']} -
{joke_data['punchline']}")

    end_time = time.time()
    print(f"\nTotal execution time: {end_time - start_time:.2f}
seconds.")
    print("Notice this is close to the longest single task (2s),
not the sum of all tasks!")


# To run an async main function, you use asyncio.run()
if __name__ == "__main__":
    asyncio.run(main())
```

When you run this, you will see the output interleaved. The fetches will start, the DB write will start, and then things will finish as they become ready. The total time will be just over 2 seconds (the length of the longest task), not 2 seconds + the time for both API calls. This is the **architectural advantage of concurrency.** You've just built a micro-backend that handles multiple operations efficiently, the exact principle you need for a scalable Medical-AI platform.

CHAPTER 4: PROFESSIONAL HABITS AND TOOLING

Your goal is to build a development *process* that is as robust as the code itself. This process provides stability (knowing your code works), a safety net (the ability to undo mistakes), and a collaboration framework, all of which are non-negotiable for building projects for companies.

---

1. `venv`: Your Project's Isolated Cleanroom

The Architect's Mindset: A project's dependencies are a critical part of its architecture. You cannot allow dependencies from one project to "leak" into another. This leads to the infamous "it works on my machine" problem, a nightmare for deployment and collaboration. A virtual environment ensures your project is a self-contained, reproducible unit.

- **Why it Matters:** When you deploy your **NEETPrepGPT** FastAPI backend, the server environment must *exactly* match your development environment. `venv` and a `requirements.txt` file guarantee this. It ensures that the version of `SQLAlchemy` or `OpenAI` you developed with is the same one running in production.
- **The Workflow:**
    1. **Create:** `python3 -m venv venv` (Creates a `venv` folder in your project)
    2. **Activate:** `source venv/bin/activate` (On Windows: `venv\Scripts\activate`) Your shell prompt will change, indicating you're inside the "cleanroom."
    3. **Install:** `pip install pytest fastapi sqlalchemy` (Install packages *inside* the environment)
    4. **Freeze:** `pip freeze > requirements.txt` (Creates a list of the exact versions of all dependencies. This file is committed to Git.)

---

2. Git & GitHub: The Blueprint's History and Review Hall

The Architect's Mindset: Git is not just a backup system. It is the **authoritative history of every decision made in your project's life**. An architect uses this history to understand *why* a change was made months or years ago. GitHub is the professional forum where these architectural changes are proposed, reviewed, and approved before being merged into the final blueprint.

Core Concepts:

- **Atomic Commits:** Each commit should be a single, logical change. A commit message like `"Feat: Add user authentication endpoint"` is an invaluable historical record. A message like `"minor changes"` is useless.
- **Branching Workflow:** A disciplined branching strategy prevents chaos. For your projects, a simple and powerful model is:
    - `main`: This branch represents your stable, production-ready code. You **never** commit directly to it.
    - `develop`: This is your main integration branch. All completed features are merged here. It represents the "next release."

    - `feature/<feature-name>`: All new work happens here. For example, `feature/add-mcq-parser` or `feature/symptom-api-integration`. It's an isolated sandbox where you can work without destabilizing the `develop` branch.

- **Pull Requests (PRs): The Architectural Review:** A PR is a formal request to merge your feature branch into `develop`. It's the most critical collaboration tool. It tells your team (or your future self): "I've completed this feature, here is the code, please review it before it becomes part of the main project." This is where you enforce quality and share knowledge.

---

3. `pytest`: The Automated Quality Assurance Engine

The Architect's Mindset: An architect must build for change. You will constantly be adding features or fixing bugs in your projects. How can you be sure a new feature in your **Symptom2Specialist** bot didn't break the existing diagnosis logic? You can't manually test every single possibility. **Automated tests are your safety net.** They run in seconds and give you the confidence to refactor and expand your system without fear.

- **Why it Matters:** A test suite is a living, executable specification of how your code is supposed to behave. It protects against regressions (re-introducing old bugs).
- **Unit Tests:** These test the smallest "units" of your code—a single function or a method on a class—in isolation.
- **Test Coverage (`--cov`):** This metric tells you what percentage of your code is executed by your tests. A target of >80% is a strong professional standard. It's not about hitting a number; it's about having confidence that the critical paths in your logic are verified and protected.

🛠️ Mini-Project Action Plan: Architecting the "StudyNotes" Workflow

Let's apply this professional discipline to the `StudyNotes` CLI tool from Chapter 1.

☑ **Your Definition of Done:** The project is on GitHub, has a clean commit history, a PR was used to merge a feature, and it has >80% test coverage.

1. Setup the Environment & Repository:

```
# In your project directory
python3 -m venv venv
source venv/bin/activate
pip install pytest pytest-cov
pip freeze > requirements.txt
```

```
git init
git add .
git commit -m "Initial commit: Setup project structure and venv"
# Go to GitHub, create a new repository, and follow the
instructions to push
```

2. Implement the Git Workflow:

```
# Create the develop branch from main
git checkout -b develop

# Now, let's add a new feature (e.g., searching notes)
git checkout -b feature/search-notes

# 1. Write the code for the search feature in your .py file.
# 2. Write the tests for the search feature in your tests/ folder.
# 3. Make small, logical commits as you work.
git add .
git commit -m "Feat: Implement search_notes method in Notebook"
git add .
git commit -m "Test: Add unit tests for search_notes
functionality"

# 4. Once the feature is complete and tested, push it to GitHub
git push origin feature/search-notes

# 5. Go to GitHub and create a Pull Request to merge
`feature/search-notes` into `develop`.
#    Review the code, then merge it.

# 6. Update your local develop branch
git checkout develop
git pull origin develop
```

Repeat this process for every new feature until you have a rich, professional commit history.

3. Write and Run Tests with `pytest`: Create a `tests/` directory. Inside, create `test_notebook.py`.

`tests/test_notebook.py` **Example:**

```
import pytest
from notes_manager import Note, Notebook, NoteNotFoundError #
```

```
Assuming your file is notes_manager.py

def test_add_note():
    """Tests that a note is successfully added to the notebook."""
    notebook = Notebook()
    note = Note("Test content", ["testing"])
    notebook.add_note(note)
    # Use the generator to find the note and confirm it's there
    found_notes = list(notebook.find_notes_by_tag("testing"))
    assert len(found_notes) == 1
    assert found_notes[0].content == "Test content"

def test_find_notes_by_tag_returns_generator():
    """Ensures the find method correctly filters by tag and
returns a generator."""
    notebook = Notebook()
    note1 = Note("Biology note", ["biology"])
    note2 = Note("Python note", ["python"])
    notebook.add_note(note1)
    notebook.add_note(note2)

    # Test finding a tag
    biology_notes = list(notebook.find_notes_by_tag("biology"))
    assert len(biology_notes) == 1
    assert biology_notes[0].content == "Biology note"

    # Test finding a non-existent tag
    math_notes = list(notebook.find_notes_by_tag("math"))
    assert len(math_notes) == 0

# Add more tests for edge cases, errors, etc.
```

Run your tests and check coverage:

```
pytest --cov=notes_manager
```

This will run all your tests and generate a report showing your coverage percentage. Analyze the report to see which lines of code aren't being tested and write new tests to cover them until you exceed 80%.