Phase 1: Inbuilt Data Structures & Functional Programming

Focus: List/Dict Comprehensions, Map, Filter, and Nested JSON.

1. The Global Logistics Tracker

Scenario: A shipping company has a nested JSON representing containers across various ships. Each container has a list of items with weights and destination codes. You need to extract a flat list of all "Fragile" items destined for "LON" (London) that weigh over 10kg.

- **Input:** A nested list of dictionaries (Ships -> Containers -> Items).
- **Output:** List of strings (Item names).
- **Hint:** Use a nested list comprehension or a combination of `filter` and a custom flattening function.

2. E-commerce Dynamic Discounting

Scenario: A retail platform has a dictionary of products: `{'id': {'price': 100, 'category': 'Electronics'}}`. Apply a 15% discount to all 'Electronics' and a 10% discount to 'Fashion', but only if the price is above $50.

- **Input:** Dictionary of dictionaries.
- **Output:** A new dictionary with updated prices using **Dictionary Comprehension**.
- **Hint:** Use conditional logic inside the value expression of the comprehension.

3. Smart Grid Energy Sensor Cleanup

Scenario: IoT sensors send hourly readings. Some readings are `None` (malfunctions). You receive a list of lists representing 24-hour cycles. Calculate the average reading for each day, strictly ignoring `None` values.

- **Input:** `[[22, None, 25], [None, None, 19], [20, 21, 22]]`
- **Output:** `[23.5, 19.0, 21.0]`
- **Hint:** Use `map` combined with a lambda that filters `None` before calculating `sum()/len()`.

4. Genomic Sequence Pattern Matcher

Scenario: You have a list of DNA sequences. You need to identify sequences that contain a specific "TATA" box but also have a GC-content (ratio of G and C to total length) higher than 50%.

- **Input:** List of strings.
- **Output:** Filtered list of strings.
- **Hint:** Use `filter()` with a complex multi-condition lambda or a helper function.

5. Multi-Currency Ledger Reconciliation

Scenario: A fintech app receives a JSON of transactions in various currencies. You have a conversion rate dictionary. Convert all transactions to USD and group them by "Status" (Success/Failed).

- **Input:** `transactions = [{"amt": 100, "curr": "EUR", "status": "Success"}, ...]`
- **Output:** `{"Success": [total_usd], "Failed": [total_usd]}`
- **Hint:** Use `map` to convert values, then a dictionary comprehension to group them.

Phase 2: Objects, Classes, and Modeling

Focus: Encapsulation, Inheritance, and Real-world abstractions.

6. The "Universal Remote" Interface

Scenario: Model a Smart Home system where different devices (Light, AC, TV) have different methods for `turn_on()`. Create a `RemoteControl` class that can trigger any device without knowing its type.

- **Concepts:** Polymorphism and Abstract Base Classes (ABC).
- **Input:** List of device objects.
- **Output:** Status logs of devices being activated.
- **Hint:** Define an abstract `Device` class with an `@abstractmethod`.

7. Banking System: The "No-Negative" Vault

Scenario: Create an `Account` class where the `balance` cannot be accessed directly. Users can only `deposit` or `withdraw`. If a withdrawal exceeds the balance, raise a custom `InSufficientFundsError`.

- **Concepts:** Private variables (`__balance`) and Property Decorators.
- **Hint:** Use `@property` for the getter and ensure the setter prevents negative values.

8. The RPG Character Evolution

Scenario: In a game, a `Warrior` can "Level Up" to a `Paladin`. A `Paladin` inherits all `Warrior` skills but adds a `heal()` ability. However, the `attack()` method must be 20% stronger than the base `Warrior`.

- **Concepts:** Method Overriding and `super()`.
- **Hint:** Call `super().attack() * 1.2` in the child class.

9. Library Management System (Object Modeling)

Scenario: Model a library with `Book`, `Member`, and `Library` classes. A `Library` contains a list of `Books`. A `Member` can borrow a book, which changes the book's `is_available` status and adds it to the `Member`'s borrowed list.

- **Concepts:** Composition and Aggregation.
- **Hint:** The `Library` should have a method `lend_book(isbn, member_id)`.

10. The Automated Restaurant (Advanced Modeling)

Scenario: Create a system where an `Order` consists of multiple `FoodItem` objects. Each `FoodItem` has a `price` and `prep_time`. The `Order` should calculate total price and the longest `prep_time` (since items are cooked in parallel).

- **Concepts:** List of objects as attributes.
- **Hint:** Use `max()` with a key argument on the list of objects.

Phase 3: Nested Data & JSON Handling

Focus: Deeply nested structures and API-style data parsing.

11. Social Media Analytics Aggregator

Scenario: You get a JSON representing a user's posts. Each post has a list of comments, and each comment has a "likes" count. Find the total number of likes across all comments on all posts for a specific user.

- **Input:** Deeply nested JSON.
- **Output:** Integer (Total Likes).
- **Hint:** Use nested for-loops or a double-list comprehension to flatten the likes into a single list before summing.

12. Corporate Hierarchy Search

Scenario: An HR JSON represents a company tree (CEO -> Managers -> Employees). Write a function that finds the "Department" of a specific employee ID by traversing the nested structure.

- **Input:** Recursive Dictionary.
- **Output:** String (Dept Name).
- **Hint:** This requires a recursive function that checks if the ID is in the current level or moves deeper.

13. Weather Station Data Normalizer

Scenario: You have a list of JSON objects from different stations. Some use Celsius, some Fahrenheit. Normalize all to Celsius and filter out readings that are physically impossible (e.g., below Absolute Zero).

- **Input:** `[{"temp": 32, "unit": "F"}, {"temp": 20, "unit": "C"}]`
- **Output:** List of normalized floats.
- **Hint:** Use `map()` with a conversion function that includes error handling.

14. Configuration File Merger

Scenario: You have a `default_config` dict and a `user_config` dict. Merge them so that user settings override defaults, but nested dictionaries (like `theme: {color: blue, font: Arial}`) are merged rather than overwritten.

- **Input:** Two nested dictionaries.
- **Output:** One merged dictionary.
- **Hint:** This is "Deep Merging." Use recursion to check if a key exists in both and is a dictionary.

15. The Flight Itinerary Flattener

Scenario: A travel API returns a nested JSON of "Legs" and "Segments." Extract only the `arrival_time` of the *final* segment of every flight option.

- **Input:** Nested JSON.
- **Output:** List of timestamps.
- **Hint:** Access the last element of the `segments` list using `[-1]`.

Phase 4: The "Complete OOP" Challenge

Focus: Class methods, Static methods, and Complex relationships.

16. The Payment Gateway Adapter

Scenario: Your system uses `Stripe`. Suddenly, you need to add `PayPal`. Both have different method names for charging (`charge_card` vs `make_payment`). Create an Adapter pattern so the main code only calls `process()`.

- **Hint:** Create a base `PaymentProcessor` and wrap the third-party classes.

17. The Database Singleton

Scenario: Ensure that your `DatabaseConnection` class can only ever have **one** instance, no matter how many times it is instantiated.

- **Concepts:** `__new__` method or Singleton Pattern.
- **Hint:** Store the instance in a class-level variable.

18. Vehicle Factory Pattern

Scenario: Create a `VehicleFactory` that takes a string ("Electric", "Gas") and returns an instance of the corresponding class (`Tesla` or `Toyota`).

- **Concepts:** Factory Pattern.
- **Output:** An object of the specific type.

19. Employee Payroll with Mixins

Scenario: Create a `SalaryEmployee` and a `CommissionEmployee`. Use a `TaxMixin` class to provide a `calculate_tax()` method to both without duplicating code in the hierarchy.

- **Concepts:** Multiple Inheritance / Mixins.

20. Smart Home "Scene" Manager

Scenario: An `Action` class stores a function and its arguments. A `Scene` class stores a list of `Action` objects. When `Scene.run()` is called, it executes all actions.

- **Concepts:** Storing methods as variables, `*args`.

Phase 5: Integrated Real-World Scenarios

Combining everything: Classes + Comprehensions + JSON.

21. University Course Registration

Scenario: A `Course` has a `max_capacity`. A `Student` has a `list_of_grades`. A `RegistrationSystem` class handles enrolling students but only if they have a GPA > 3.0 and the course isn't full.

- **Task:** Use a list comprehension to find all students eligible for a specific "Advanced AI" course.

22. Inventory "Low Stock" Emailer

Scenario: You have a JSON of 5000 items. Create a system that uses `filter` to find items where `stock < reorder_level`, then uses `map` to format these into "Alert: [ItemName] is low" strings.

23. The Undo/Redo Text Editor

Scenario: Model a `TextEditor` class that uses a **Stack** (List) to store `Command` objects. Each `Command` has an `execute()` and `undo()` method.

- **Hint:** This covers Command Pattern and OOP state management.

24. Crypto Portfolio Tracker

Scenario: A user has a list of `Asset` objects (BTC, ETH). Fetch the current prices (from a mock JSON dict). Use a property decorator to calculate the `total_value_usd` dynamically.

25. Ride-Sharing Fare Calculator

Scenario: Different ride types (`UberX`, `UberBlack`, `UberPool`) have different base fares and per-mile rates. Use inheritance to model these and a class method to update the "Surge Multiplier" globally.

26. Hospital Triage System

Scenario: Patients are stored in a list of dictionaries with `severity` (1-5) and `arrival_time`. Use `sorted()` with a lambda to order them by severity first, then time.

27. Movie Recommendation Filter

Scenario: Given a JSON of movies with nested genres and IMDB ratings, use a nested list comprehension to find all "Sci-Fi" movies with a rating > 8.5.

28. Automated Email Spammer Detector

Scenario: A `Message` object has a `body` and `sender`. Create a `SpamFilter` class that uses a list of "keywords" and `filter()` to flag messages as `is_spam = True`.

29. File System Simulator

Scenario: Model `File` and `Folder` classes. A `Folder` can contain `Files` or other `Folders`. Implement a `get_size()` method that recursively calculates total size.

30. E-commerce "Cart" with JSON Export

Scenario: A `Cart` class holds `Product` objects. Implement a method `to_json()` that converts the current cart state into a nested JSON structure for an API.