Advanced Python OOP: The Master Workbook

An intensive guide designed for learners transitioning from intermediate to advanced Python, focusing on conceptual depth, architectural correctness, and the internal mechanics of the Python object model.

1. OOP Foundations & the Four Pillars

🟢 Exercise 1: Fundamental State (Simple)

Concept: Instance initialization and state isolation. Complete the code to ensure each `Drone` has its own unique ID and battery level.

```
class Drone:
    def __init__(self, drone_id: str):
        self.drone_id = ???
        self.battery = 100

d1 = Drone("A1")
d2 = Drone("B2")
# Expected: d1 and d2 have different drone_id values.
```

🟡 Exercise 2: The Polymorphic Interface (Medium)

Concept: Shared behavior across different types without inheritance. Implement a `process_payment` function that works for any object that has a `.charge(amount)` method (Duck Typing).

```
class CreditCard:
    def charge(self, amount): print(f"CC charged ${amount}")

class Crypto:
    def charge(self, amount): print(f"Wallet deducted ${amount}")

def process_payment(method, amount):
    # ???
    pass

process_payment(CreditCard(), 100)
process_payment(Crypto(), 50)
```

🔴 Exercise 3: Protected Internal Logic (Hard)

Concept: Encapsulation and internal state validation. Create a `BankAccount` where the balance can only be modified through `deposit` and `withdraw` methods. The balance attribute should be "protected" (using `_`).

- If `withdraw` exceeds balance, raise `ValueError`.
- `deposit` must only accept positive numbers.

2. Inheritance & MRO (Method Resolution Order)

🟢 Exercise 4: Basic Extension (Simple)

Extend the `Animal` class into a `Dog` class. Override `speak` to return "Woof".

```
class Animal:
    def speak(self): return "Generic sound"

class Dog(Animal):
    # ???
```

🟡 Exercise 5: The `super()` Mechanic (Medium)

Concept: Cooperative initialization. Ensure that `SmartPhone` correctly initializes both its `Phone` and `Camera` components using `super()`.

```
class Phone:
    def __init__(self, number):
        self.number = number

class Camera:
    def __init__(self, resolution):
        self.resolution = resolution

class SmartPhone(Phone, Camera):
    def __init__(self, number, resolution, brand):
        # ??? - Use super() or direct calls? (Hint: Consider MRO)
        self.brand = brand
```

🔴 Exercise 6: The Diamond Problem (Hard)

Concept: Understanding `__mro__`. Identify the output of `d.where_am_i()` and explain why based on the linearization of classes.

```
class A:
    def where_am_i(self): return "A"

class B(A):
    def where_am_i(self): return "B"

class C(A):
    def where_am_i(self): return "C"

class D(B, C):
    pass
```

```
d = D()
print(d.where_am_i()) # Expected?
print(D.__mro__)
```

3. Encapsulation, Properties & Attribute Control

🟢 Exercise 7: Basic @property (Simple)

Convert a `Circle` class with `self.radius` so that it has a read-only `diameter` property.

🟡 Exercise 8: Managed Attributes (Medium)

Concept: `@property.setter`. Create a `Temperature` class where the `celsius` attribute is private. Provide a property that allows setting the temperature, but raises a `ValueError` if the value is below -273.15.

🔴 Exercise 9: Property Deleter & Side Effects (Hard)

Concept: `@property.deleter`. Create a `Session` class with a `token` attribute. When the token is deleted (`del session.token`), it must log "Security Alert: Token deleted" and set an internal `_is_active` flag to `False`.

4. Polymorphism & Method Overriding

🟢 Exercise 10: Operator Overloading (Simple)

Implement `__add__` for a `Point(x, y)` class so that `Point(1, 2) + Point(3, 4)` returns `Point(4, 6)`.

🟡 Exercise 11: Rich Comparisons (Medium)

Concept: `__eq__` and `__lt__`. Implement a `Book` class where books are compared based on their `page_count`.

🔴 Exercise 12: Emulating Containers (Hard)

Concept: `__getitem__`, `__len__`, and `__iter__`. Create a `DataBuffer` class that wraps a list but only allows access to even indices via indexing (`buffer[0]` gets index 0, `buffer[1]` gets index 2).

5. Methods: Instance, Class, and Static

🟢 Exercise 13: Choosing Types (Simple)

Identify if the following methods should be `@classmethod`, `@staticmethod`, or instance methods:

1. A method to create a `User` from a JSON string.
2. A method to format a string "Name: {name}".

3. A method to change a user's password.

🟡 Exercise 14: Alternative Constructors (Medium)

Concept: `@classmethod`. Implement a `Date` class that has a `__init__(year, month, day)` and a class method `from_string(date_str)` (e.g., "2023-10-25").

🔴 Exercise 15: Tracking Class State (Hard)

Concept: `@classmethod` for factory logic. Create a `Widget` class that tracks the total number of widgets created. Implement a `@classmethod` called `get_total_count` and ensure the count increments in `__init__`.

6. Abstract Base Classes (ABC) & Protocols

🟢 Exercise 16: Basic ABC (Simple)

Define an ABC `Shape` with an `@abstractmethod` called `area`. Try to instantiate `Shape` and record the error.

🟡 Exercise 17: Abstract Properties (Medium)

Define an ABC `DatabaseConnector` that requires subclasses to implement a property `connection_string`.

🔴 Exercise 18: Structural Subtyping with Protocol (Hard)

Concept: `typing.Protocol`. Define a `TemplateRenderable` Protocol that requires a `render() -> str` method. Write a function `show(obj: TemplateRenderable)` and demonstrate it working with a class that *does not* inherit from the Protocol.

7. Dispatching & Dynamic Behavior

🟢 Exercise 19: Single Dispatch Intro (Simple)

Use `functools.singledispatch` to create a function `format_data` that behaves differently for `int` and `str`.

🟡 Exercise 20: Method Dispatch (Medium)

Concept: `singledispatchmethod`. Create a `Serializer` class with a `serialize` method that uses `@singledispatchmethod` to handle `dict` and `list` differently.

🔴 Exercise 21: Custom Type Dispatch (Hard)

Extend a dispatched function to handle a custom class `User` without modifying the original function definition (using `.register()`).

8. Object Creation: new & Metaclasses

🟢 Exercise 22: The new Signature (Simple)

Write a class where `__new__` prints "Allocating memory" and `__init__` prints "Initializing object".

🟡 Exercise 23: Singleton Pattern (Medium)

Concept: Using `__new__` for instance control. Implement a `Database` class that only ever allows one instance to exist.

🔴 Exercise 24: Basic Metaclass (Hard)

Concept: `type` inheritance. Create a metaclass `UppercaseMeta` that automatically converts all attribute names of a class to uppercase during creation.

9. Callable Objects & Advanced Patterns

🟢 Exercise 25: The call Method (Simple)

Create a `Multiplier(factor)` class that, when called with a number, returns `number * factor`.

🟡 Exercise 26: Stateful Callables (Medium)

Create a `AverageCalculator` class that keeps track of all numbers passed to it via `__call__` and returns the current running average.

🔴 Exercise 27: Decorator Classes (Hard)

Concept: Using a class as a function decorator. Create a `Logger` class that wraps a function and prints the arguments every time the function is called.

10. Paragraphic / Scenario-Based Hard Exercises

Rules: No starter code. Design the architecture yourself based on requirements.

🧠 Exercise 28: The Plugin System

Scenario: You are building a media player. **Requirements:**

1. Create an ABC `MediaPlugin` requiring `play()` and `Youtube()`.
2. Implement `AudioPlugin` and `VideoPlugin`.
3. Create a `PluginRegistry` that uses a metaclass to automatically register any subclass of `MediaPlugin` into a list when the class is defined. **Expected Outcome:** `PluginRegistry.plugins` should contain the classes `AudioPlugin` and `VideoPlugin` without manually adding them.

🧠 Exercise 29: The Smart Home Conflict

Scenario: A device can be both a `Heater` and a `Cooler`. **Requirements:**

1. Both `Heater` and `Cooler` inherit from `Device`.

2. `Device` has a `work()` method.
3. `Heater.work()` returns "Heating".
4. `Cooler.work()` returns "Cooling".
5. Create a `HVAC` class inheriting from both. Ensure `HVAC().work()` uses `super()` to call the *first* parent but also provides a way to trigger the *second* parent's method via a specific `mode` attribute.

🧠 Exercise 30: Validation Framework

Scenario: You need an attribute validation system using descriptors or properties.
Requirements:

1. Create a `Person` class with `age` and `name`.
2. `age` must be between 0 and 120.
3. `name` must be a non-empty string.
4. If invalid data is assigned, raise a custom `ValidationError`.
5. Implement this using `__setattr__` for one and `@property` for the other to compare the two approaches.

🧠 Exercise 31: The Immutable Record

Scenario: Create a data structure that acts like a dictionary but is strictly immutable.
Requirements:

1. Use `__slots__` to prevent adding new attributes.
2. Override `__setattr__` to prevent changing existing attributes after `__init__`.
3. The class should support "evolution" (a `clone_with(key=value)` method that returns a *new* instance with one change).

🧠 Exercise 32: The Undoable State

Scenario: A class that tracks its own history. **Requirements:**

1. Create a `Document` class with a `content` string.
2. Every time `content` is changed, the old version is stored in a private list.
3. Implement an `undo()` method that reverts the content to the previous state.
4. Use `@property` to manage `content`.

🧠 Exercise 33: Multi-Format Exporter (Dispatch)

Scenario: A system that exports `Report` objects to different formats. **Requirements:**

1. Define a `Report` class.
2. Define a `JSONExporter` and `HTMLExporter` class.
3. Use `singledispatchmethod` inside an `ExportManager` so that passing a `JSONExporter` instance to `manager.export(report, exporter)` calls the JSON logic, and an

`HTMLExporter` calls the HTML logic.

🧠 Exercise 34: Structural Banking (Protocols)

Scenario: A bank system that processes "Account-like" objects. **Requirements:**

1. Define a `Transferable` Protocol with a `balance: float` and `deduct(amount)`.
2. Create a `LegacyAccount` that does *not* inherit from the Protocol but has the required members.
3. Create a function `execute_transfer(source: Transferable, amount: float)` that is type-checked (use `mypy` style thinking).
4. Prove that Python accepts `LegacyAccount` at runtime due to structural subtyping.

🧠 Exercise 35: The API Rate Limiter

Scenario: Use `__call__` to create an object that limits how often a function can be run. **Requirements:**

1. The class `RateLimiter` takes `max_calls` and `period`.
2. It wraps a function.
3. If called more than `max_calls` within the `period`, it raises `RateLimitExceeded`.

🧠 Exercise 36: Dynamic Attribute Mapping

Scenario: A class that wraps a JSON dictionary and allows attribute access (e.g., `data.name` instead of `data['name']`). **Requirements:**

1. Use `__getattr__`.
2. Handle nested dictionaries (e.g., `data.user.id`).
3. If the key doesn't exist, return `None` instead of raising an error.

🧠 Exercise 37: The Dependency Injector (Metaclasses)

Scenario: Automatically inject dependencies into a class. **Requirements:**

1. Create a metaclass `DIMeta`.
2. If a class has a class-level attribute `dependencies = ['db', 'config']`, the metaclass should automatically replace those strings with actual mock objects in the class's `__init__`.

Solutions & Conceptual Deep Dive

🏗️ Foundations & Pillars (Ex 1-3)

Key Insight: `_` is a convention for "protected", while `__` triggers **name mangling**.

```
# Ex 3 Solution
class BankAccount:
    def __init__(self, initial):
```

```
        self._balance = initial
    @property
    def balance(self): return self._balance
    def deposit(self, amt):
        if amt <= 0: raise ValueError("Must be positive")
        self._balance += amt
```

🧬 Inheritance & MRO (Ex 4-6)

Key Insight: Python uses C3 Linearization. `super()` doesn't call the "parent", it calls the *next* class in the MRO.

```
# Ex 6 Explanation
# MRO for D: [D, B, C, A, object]
# d.where_am_i() returns "B" because B is first in MRO after D.
```

🔐 Encapsulation & Properties (Ex 7-9)

Common Mistake: Forgetting that `@property` makes an attribute read-only unless a `.setter` is defined.

```
# Ex 8 Solution
class Temperature:
    def __init__(self, c): self.celsius = c
    @property
    def celsius(self): return self._c
    @celsius.setter
    def celsius(self, val):
        if val < -273.15: raise ValueError("Absolute zero")
        self._c = val
```

🎭 Protocols & ABCs (Ex 16-18)

Key Difference: `ABC` is **Nominal** (you must inherit). `Protocol` is **Structural** (if it looks like a duck, it is a duck).

```
# Ex 18 Solution
from typing import Protocol

class TemplateRenderable(Protocol):
    def render(self) -> str: ...

class MyPage: # No inheritance!
    def render(self): return "<html></html>"

def show(obj: TemplateRenderable):
    print(obj.render())
```

```
show(MyPage()) # Valid!
```

🛠 Metaprogramming (Ex 22-24)

Key Insight: `__new__` is the actual constructor (allocator), `__init__` is the initializer.

```
# Ex 23 Singleton
class Database:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance
```

🧠 Scenario-Based Solutions Highlights

Ex 28 (Plugin System):

```
class PluginRegistry(type):
    plugins = []
    def __init__(cls, name, bases, dct):
        if name != "MediaPlugin":
            PluginRegistry.plugins.append(cls)
        super().__init__(name, bases, dct)

class MediaPlugin(metaclass=PluginRegistry):
    pass

class AudioPlugin(MediaPlugin): pass # Automatically added to registry
```

Ex 31 (Immutable Record):

```
class Record:
    __slots__ = ('name', 'id')
    def __init__(self, name, id):
        object.__setattr__(self, 'name', name)
        object.__setattr__(self, 'id', id)
    def __setattr__(self, key, value):
        raise AttributeError("Immutable")
```

🛑 Debugging & Common Errors

1. `TypeError: Can't instantiate abstract class...`: You forgot to implement one of the `@abstractmethod`s.
2. **Infinite Recursion in Properties**: Calling `self.attr` inside `def attr(self):`. Use `self._attr` (the internal name) instead.