Day 12: OOP Mechanics (Pseudo Code)

Hint 12.1: The Constructor

Task: `__init__`.
Pseudo Code Logic:

```
CLASS User:
    FUNCTION __init__(self, username):
        # 'self' is the empty object just created
        self.name = username
        self.active = True

# Usage:
user_obj = ALLOCATE_MEMORY(User) # Step 1: Create blank
User.__init__(user_obj, "Alice") # Step 2: Fill it
```

Detailed Explanation: The constructor is the factory worker that sets up the empty shell. When you call `User()`, Python first asks the Operating System for a block of memory (the Object). Then, it immediately passes that memory block (as `self`) to `__init__` to set initial values. Without this, the object would be an empty box.

Hint 12.2: The Self Reference

Task: Why `self`?
Pseudo Code Logic:

```
CLASS Car:
    speed = 0
    FUNCTION drive(self):
        self.speed = 100

# Behind the scenes translation:
my_car = Car()
Car.drive(my_car) # Explicitly passing the object
```

Detailed Explanation: Functions inside a class are just standard functions. They don't magically know *which* car they are driving. `self` is the "Instance Pointer." It tells the function which specific block of memory (Car A vs Car B) to modify. If you omit `self`, the function has no target to act upon.

Hint 12.3: The String Representation

Task: `__str__` vs `__repr__`.
Pseudo Code Logic:

```
CLASS Book:
    FUNCTION __str__(self):
        RETURN "Readable Title for Users"

    FUNCTION __repr__(self):
        RETURN "Book(id=101, title='Code')" # Code for Devs

print(book)  # Calls book.__str__()
list_of_books # Calls book.__repr__() inside the list display
```

Detailed Explanation: Objects in memory look like `<__main__.Book object at 0x7f...>`. This is useless to humans. `__str__` is the "UI Layer" (what the user sees). `__repr__` is the "Debug Layer" (what the programmer sees). Overriding these gives your objects a voice.

Hint 12.4: Private Variables

Task: `__variable`.
Pseudo Code Logic:

```
CLASS Account:
    FUNCTION __init__(self):
        self.__balance = 1000  # Mangled Name

# Runtime Memory Map:
# self._Account__balance = 1000
# self.balance = ERROR (Does not exist)
```

Detailed Explanation: True privacy doesn't exist in Python (unlike Java/C++). Instead, Python uses "Name Mangling." If you prefix a variable with double underscores, the interpreter rewrites the variable name in the symbol table, adding the class name as a prefix. This prevents accidental overwrites by children classes but is not secure against hackers.

Hint 12.5: The Property Decorator

Task: Getters and Setters.
Pseudo Code Logic:

```
CLASS Thermometer:
```

```
    @property
    FUNCTION fahrenheit(self):
        RETURN (self.celsius * 1.8) + 32

# Usage
temp = t.fahrenheit  # Looks like variable access
# Actually runs: t.fahrenheit()
```

Detailed Explanation: This implements the "Uniform Access Principle." It allows you to calculate a value dynamically (like conversion) but expose it as if it were a static attribute. This lets you refactor code (changing storage from F to C) without breaking external scripts that read the property.

Hint 12.6: Class vs Instance Variables

Task: Shared vs Unique Data.
Pseudo Code Logic:

```
CLASS Dog:
    species = "Canine" # STATIC (Stored on Class Object)

    FUNCTION __init__(self, name):
        self.name = name # DYNAMIC (Stored on Instance Object
           )

# Memory Optimization:
# "Canine" exists exactly ONCE in RAM.
# "Name" exists N times (once per dog).
```

Detailed Explanation: Class attributes are "Global" to that class type. They are useful for constants (like `PI` or `DEFAULT_CONFIG`). Instance attributes are unique to the specific object. Confusing the two causes bugs where changing one dog's species changes *every* dog's species.

Hint 12.7: Inheritance

Task: `Child(Parent)`.
Pseudo Code Logic:

```
CLASS Animal:
    DEF eat(): ...

CLASS Cat(Animal):
    DEF meow(): ...
```

```
# Lookup Strategy (MRO):
cat.eat()
1. Check Class Cat for 'eat'. Not found.
2. Check Class Animal for 'eat'. Found!
3. Execute Animal.eat(cat)
```

Detailed Explanation: Inheritance is a "Search Path." When you call a method, Python walks up the family tree (Method Resolution Order) until it finds a match. This promotes Code Reuse (DRY Principle) because you define common logic (like `eat`) once in the parent.

Hint 12.8: The Super Proxy

Task: `super()`.
Pseudo Code Logic:

```
CLASS Manager(Employee):
    FUNCTION __init__(self, name, department):
        # Run standard Employee setup first
        super().__init__(name)

        # Then add Manager specific stuff
        self.department = department
```

Detailed Explanation: When you define `__init__` in the child, you **overwrite** the parent's init. The parent's setup code never runs, leaving the object half-built. `super()` creates a temporary delegate object that forces the parent's method to run, ensuring the foundation is laid before you add the walls.

Hint 12.9: Operator Overloading

Task: `obj1 + obj2`.
Pseudo Code Logic:

```
CLASS Vector:
    FUNCTION __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        RETURN Vector(new_x, new_y)

# v1 + v2 becomes Vector.__add__(v1, v2)
```

Detailed Explanation: Operators like `+`, `-`, `==` are just shortcuts for calling

"Dunder Methods" (`__add__`). By implementing these, your custom objects can behave like native Python types (Polymorphism). This makes mathematical code much cleaner and more intuitive.

Hint 12.10: Equality

Task: `User(1) == User(1)`.
Pseudo Code Logic:

```
FUNCTION __eq__(self, other):
    # Default behavior:
    # RETURN address(self) == address(other)

    # Custom behavior (Value Equality):
    RETURN self.id == other.id
```

Detailed Explanation: By default, two objects are only equal if they are literally the *same* object in RAM. For data science, we usually care if they represent the same *data*. Overriding `__eq__` changes the definition of equality from "Identity" to "Equivalence."