Object-Oriented Programming in Python

Introduction to Software Systems

ISS TAs

February 25, 2026

The Functional Approach

Functional Programming (FP)

A paradigm where programs are constructed using functions. It is **stateless**, meaning functions do not modify external data.

```python
# --- STATE (Data) ---
Hero1_name = "P1"; Hero1_hp = 100
Hero2_name = "P2"; Hero2_hp = 80

# --- FUNCTIONS (Logic) ---
def apply_damage(hp, damage):
    return hp - damage

# --- EXECUTION ---
# You must manually track and update every single variable.
Hero1_hp = apply_damage(Hero1_hp, 20)
Hero2_hp = apply_damage(Hero2_hp, 35)

print(f"{Hero1_name} HP: {Hero1_hp}")
```

The Fix: Object-Oriented Programming

However, managing multiple characters with health, power, and levels manually is difficult and error-prone.

The Paradigm Shift

OOP groups related variables (**attributes**) and functions (**methods**) into a single unit called a **class**.

```python
# --- THE BLUEPRINT (Class) ---
class Hero:
    def __init__(self, name, hp):
        self.name = name  # State is stored INSIDE the object
        self.hp = hp

    def take_damage(self, damage):
        self.hp -= damage

# --- THE EXECUTION (Instances) ---
Hero1 = Hero("Arthur", 100)
Hero2 = Hero("Morgana", 80)

Hero1.take_damage(20)
Hero2.take_damage(35)
print(f"{Hero1.name} HP: {Hero1.hp}")
```

Core OOP Definitions

- **Class:** It is a **blueprint**. It defines what a new object of this type should look like.
  Ex: "Hero" class
- **Object:** A specific **instance** of a class, it is an entity created in memory using the class blueprint.
  Ex: "Hero1", "Hero2"
- **Attributes:** Data points that store the **state** / characteristics of each object.
  Ex: "damage", "health"
- **Methods:** Functions defined within a class that dictate the **behaviours** of an object.
  Ex: "jump", "takeDamage", "heal"

Defining a Class

To define a class, we use the **class** keyword followed by the **classname** (capitalized by convention).

We can then define class-level **attributes** and functions within the class. Functions defined within a class are called **methods**. Methods can be defined by using the **def** keyword.

```python
# 1. Define the Blueprint (Class)
class Warrior:
    # Attributes
    health = 100
    stamina = 50

    # Defining a behavior (Method)
    def attack():
        print("Swinging the sword!")
```

Creating and Using Objects

To create an object of a class, we use the following syntax:

```
object1 = Class()
```

We can use the **dot operator** to access class attributes and methods from an object.

```
# Create Instances (Objects)
warrior1 = Warrior()
warrior2 = Warrior()
print(warrior1.health) # Output: 100

# Modify Unique Data (State)
# Changing one object does not affect the other
warrior1.health = 80
warrior2.health = 50

# Access Data (Variables)
print(warrior1.health) # Output: 80
print(warrior2.health) # Output: 50

# Use Methods
Warrior.attack() # Output: Swinging the sword!
```

The "Self" and Initializing Objects

To make objects dynamic, we use two essential tools: **self** and **__init__**.

- **self (The Pointer):** A keyword representing the **specific instance** currently being used. It tells the computer: "Use *this* specific object's data."
- **__init__ (The Constructor):** A special method that runs **automatically** the moment an object is created to set its starting values.

Syntax Guide

- **Defining __init__:** `def __init__(self, param1, param2):`
- **Assigning Data:** `self.variable_name = param1`
- **Instantiating with Data:** `my_object = ClassName("Value1", "Value2")`

Example: self, __init__

```
class Hero:
    def __init__(self, name, health = 80):
        # Self Variables (Instance Attributes)
        # These are unique to EACH hero
        self.name = name
        self.health = health

    def report(self):
        # self.name retrieves the specific hero's name
        print(f"Hero: {self.name} | HP: {self.health}")

# Creating instances with unique data
h1 = Hero("Aragorn", 100)
h2 = Hero("Boromir")

h1.report()  # Output: Hero: Aragorn | HP: 100
h2.report()  # Output: Hero: Boromir | HP: 80
```

Class vs. Instance Attributes

There are two kind of attributes one can use in classes.

1. **Class Attributes**
   These are defined inside the class, outside of methods.
   - **Scope:** Shared by *every* object created from that class.
   - **Use Case:** Constants or settings (e.g., gravity constant or game version).
   - **Syntax:** Accessed via `ClassName.attribute`.
2. **Instance Attribute**
   These are defined inside the class within methods.
   - **Scope:** Unique to *each specific* object.
   - **Use Case:** Individual stats (e.g., health, name, or coordinates).
   - **Syntax:** Accessed via `self.attribute`.

While both are variables within a class, they serve very different purposes. One is a **shared value**, and the other is **personal data**.

Example: Class vs Instance Attributes

```python
class Hero:
    # A Non-Self Variable (Class Attribute)
    # This is shared by EVERY hero
    game_version = "v1.2"

    def __init__(self, name, health = 80):
        # Self Variables (Instance Attributes)
        # These are unique to EACH hero
        self.name = name
        self.health = health

    def report(self):
        # self.name retrieves the specific hero's name
        # Hero.game_version retrieves the shared version for everyone
        print(f"[{Hero.game_version}] Hero: {self.name} | HP: {self.health}")

# Creating instances with unique data
h1 = Hero("Aragorn", 100)
h2 = Hero("Boromir")

h1.report()  # Output: [v1.2] Hero: Aragorn | HP: 100
h2.report()  # Output: [v1.2] Hero: Boromir | HP: 80
```

Activity 1: Build a Calculator

- Create a **Calculator** class.
- Add methods for: `add`, `subtract`, `multiply`, `divide`, `power`, and `modulo`.
- Instantiate your class and print the results for each method using **5** and **3** as your inputs.

Solution: Calculator

```python
class Calculator:
    def add(self, a, b):
        return a + b
    def subtract(self, a, b):
        return a - b
    def multiply(self, a, b):
        return a * b
    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
    def power(self, a, b):
        return a ** b
    def modulo(self, a, b):
        if b == 0:
            raise ValueError("Cannot modulo by zero")
        return a % b

calc = Calculator() #calc is short for Calculator btw
print(calc.add(5, 3))
print(calc.subtract(5, 3))
print(calc.multiply(5, 3))
print(calc.divide(5, 3))
print(calc.power(5, 3))
print(calc.modulo(5, 3))
```

Activity 2: Car Sales Tags

- Create a **Car** class where each instance has a **brand**, **model**, and **color**.
- Add a `display_info()` method that prints out a formatted "Sales Tag" for the car.
- **The Task:** Create three different car objects:
    1. A Red Tesla
    2. A Blue Ford
    3. A Silver Porsche
- Print the Sales Tag for each one.

Solution: Car Sales Tags

```python
class Car:
    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    def display_info(self):
        print(f"--- SALES TAG ---")
        print(f"Brand: {self.brand}")
        print(f"Model: {self.model}")
        print(f"Color: {self.color}\n")

# Creating the specific objects
car1 = Car("Tesla", "Model 3", "Red")
car2 = Car("Ford", "Mustang", "Blue")
car3 = Car("Porsche", "911", "Silver")
```

Activity 3: The Road Trip

- Let's upgrade our **Car** class to handle a journey.
- **New Attributes (these should be instance attributes):**
    - `fuel`: Default value is 100.
    - `distance_traveled`: Defualt value is 0.
- Create a `drive(distance)` method:
    - Every 1 unit of distance driven decreases `fuel` by 1 and increases `distance_traveled` by 1. Add logic so the car cannot drive if the `fuel` is empty.
- Test your `drive` method using `car1` and `car2` objects creaetd earlier.

Solution: The "Road Trip"

```
class Car:
    def __init__(self, brand, model, color, fuel = 100, distance_traveled = 0):
        self.brand = brand
        self.model = model
        self.color = color
        self.fuel = fuel
        self.distance_traveled = distance_traveled

    def display_info(self):
        print(f"--- SALES TAG ---")
        print(f"Brand: {self.brand}")
        print(f"Model: {self.model}")
        print(f"Color: {self.color}\n")

    def drive(self, distance):
        fuel_consumed = distance
        if self.fuel >= fuel_consumed:
            self.fuel -= fuel_consumed
            self.distance_traveled += distance
            print(f"Driving {distance} km. Fuel left: {self.fuel:.2f} units.")
        else:
            print("Not enough fuel to drive this distance.")

car1 = Car("Tesla", "Model 3", "Red")
car2 = Car("Ford", "Mustang", "Blue")
car3 = Car("Porsche", "911", "Silver")

car1.display_info()
car2.display_info()
car3.display_info()

car1.drive(50)
car1.drive(130)
```