Object-Oriented Programming in Python

UM6P — SASE

January 9, 2026

Today's Roadmap

1. The Case Study: An RPG Battle System
2. Phase 1: The Spaghetti (Procedural Nightmare)
3. Phase 2: Structured Procedural (Functions)
4. Phase 3: Classes & Objects
5. Phase 4: Encapsulation
6. Phase 5: Abstraction
7. Phase 6: Inheritance
8. Phase 7: Polymorphism
9. Summary

The Case Study: An RPG Battle System

The Mission

We need to build a simple **Role Playing Game (RPG)** combat system.

Requirements:

- We have a **Hero** and a **Monster**.
- They have names, health (HP), and attack power.
- They take turns attacking each other.
- When HP reaches 0, the character dies.

Try It (1–2 min)

Before we start: if I add **10 monsters**, what do you think breaks first?

- Naming? Copy/paste? Bugs? **All of the above?**

Phase 1: The Spaghetti (Procedural Nightmare)

Attempt 1: Simple Variables

Let's just use variables. Simple, right?

```python
# The Hero
hero_name = "Arthur"
hero_hp = 100
hero_atk = 15

# The Monster
monster_name = "Goblin"
monster_hp = 50
monster_atk = 5

# Combat Logic
print(f"{hero_name} attacks {monster_name}!")
monster_hp = monster_hp - hero_atk
```

The Problem Escalates

What happens when we add a second monster? Or a second hero?

```
monster2_name = "Orc"
monster2_hp = 80
monster2_atk = 12

# Who is attacking whom?
# Did we subtract hp from monster_hp or monster2_hp?
monster2_hp = monster2_hp - hero_atk
```

The Spaghetti Issues

- **Global State:** Data scattered across the file.
- **Naming Collisions:** hero2_hp, hero3_hp...
- **No Shared Behavior:** Every new feature = more copy/paste.

Phase 2: Structured Procedural (Functions)

Attempt 2: Functions + Dictionaries

```python
def create_char(name, hp, atk):
    return {"name": name, "hp": hp, "atk": atk}

def is_alive(char):
    return char["hp"] > 0

def attack(attacker, target):
    damage = attacker["atk"]
    target["hp"] = max(0, target["hp"] - damage)
    print(f"{attacker['name']} hits {target['name']} for {damage}")

hero = create_char("Arthur", 100, 15)
goblin = create_char("Goblin", 50, 5)

attack(hero, goblin)
print(is_alive(goblin))
```

Is this enough?

This is better, but still flawed.

- **Data is passive:** The dictionary is just a bag of data.
- **No Protection:** Any part of the code can do `hero["hp"] = -9999`.
- **Scalability:** A Wizard with `mana` will force us to rewrite functions.

Try It (1–2 min)

If we add `defense` to characters, what functions must change?

Phase 3: Classes & Objects

Introducing the Class

Definition

Class (The Blueprint): A template definition of the methods and variables in a particular kind of object.

Definition

Object (The House): An instance of a class. The actual thing created from the blueprint.

Defining the Class (Consistent Version)

```python
class Character:
    def __init__(self, name, hp, atk, defense=0):
        self.name = name
        self.hp = hp
        self.atk = atk
        self.defense = defense
    def is_alive(self):
        return self.hp > 0
    def take_damage(self, raw_damage):
        dmg = max(0, raw_damage - self.defense)
        self.hp = max(0, self.hp - dmg)
        return dmg
    def attack(self, target):
        dmg = target.take_damage(self.atk)
        print(f"{self.name} attacks {target.name} for {dmg}!")
```

Quick Test

```python
arthur = Character("Arthur", 100, 15, defense=2)
goblin = Character("Goblin", 50, 5, defense=0)

arthur.attack(goblin)
goblin.attack(arthur)

print("Goblin HP:", goblin.hp)
print("Arthur alive?", arthur.is_alive())
```

Phase 4: Encapsulation

The Problem with Public Data

Right now, anyone can do this: `p1.hp = -500`.

We need to protect internal state.

Concept

Double Underscore (__) triggers **name mangling**.

Internally, `__hp` becomes `_Character__hp`. It discourages accidental access.

Note

This is **not** perfect security. It's a strong convention + a safety rail.

Encapsulation (Part 1): Getters/Setters

```python
class Character:
    def __init__(self, name, hp, atk, defense=0):
        self.name = name
        self.__hp = max(0, hp)
        self.__atk = max(0, atk)
        self.__def = max(0, defense)
    def get_hp(self):
        return self.__hp
    def get_atk(self):
        return self.__atk
    def get_defense(self):
        return self.__def
    def set_hp(self, value):
        self.__hp = max(0, value)
    def set_atk(self, value):
        self.__atk = max(0, value)
    def set_defense(self, value):
        self.__def = max(0, value)
    def is_alive(self):
        return self.__hp > 0
```

Encapsulation (Part 2): Safe Behavior

```
class Character:
    # assume __init__ + getters/setters exist

    def take_damage(self, raw_damage):
        dmg = max(0, raw_damage - self.__def)
        self.__hp = max(0, self.__hp - dmg)
        return dmg

    def attack(self, target):
        dmg = target.take_damage(self.__atk)
        print(f"{self.name} attacks {target.name} for {dmg}!")
```

Try It (1–2 min)

Try: `arthur.__hp = 999`. Then check with `arthur.get_hp()`. Did you actually change the real HP?

Using Encapsulation

```python
arthur = Character("Arthur", 100, 15, defense=2)

# Direct access fails
# print(arthur.__hp)  # AttributeError

arthur.set_hp(-500)
print(arthur.get_hp())  # 0
```

Checkpoint

Invariant enforced: HP never goes below 0.

Phase 5: Abstraction

Abstraction creates a simple interface for complex behavior (focus on what is **exposed**).

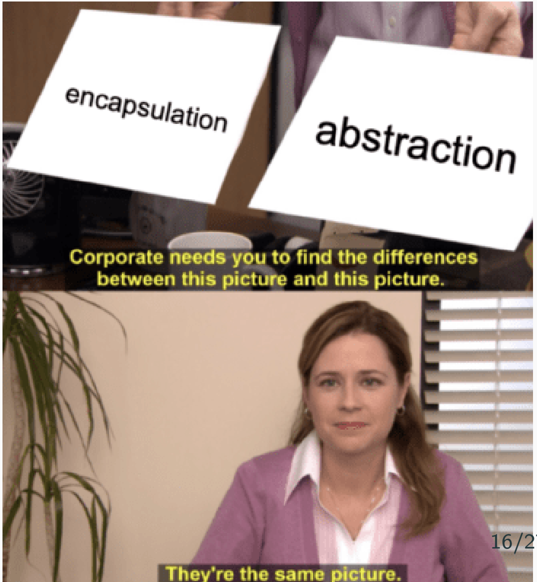
Note

A Common Confusion

They feel similar because both reduce complexity.

Practical memory hook:

- **Encapsulation** = protect state + enforce rules
- **Abstraction** = simplify how others use your code



Personal opinion

Honestly, the difference is kinda tiny. I wouldn't overthink it. "Abstraction" is the more common word, and it basically just means making something easier to use by adding a simple layer on top.

Abstraction in Action (Clean Public API)

```python
class Character:
    # internal detail (private helper)
    def __compute_damage(self, raw_damage):
        return max(0, raw_damage - self.__def)

    # public behavior (simple to use)
    def take_damage(self, raw_damage):
        dmg = self.__compute_damage(raw_damage)
        self.__hp = max(0, self.__hp - dmg)
        return dmg

    def attack(self, target):
        dmg = target.take_damage(self.__atk)
        print(f"{self.name} attacks {target.name} for {dmg}!")
```

Checkpoint

Users call `attack`/`take_damage`; the math stays hidden.

Phase 6: Inheritance

Inheritance

Concept

Inheritance lets a **child** class reuse (inherit) the **properties** and **methods** of a **parent** class.

Why it matters:

- **Code reuse:** write common logic once in the parent.
- **Specialization:** the child adds new features or custom behavior.
- **Clean structure:** `Mage` *is a* `Character`, but with extra abilities.

Try It (1–2 min)

Quick check: what should live in `Character` vs in `Mage`?

Inheritance (Part 1): The Parent Class

```python
class Character:
    def __init__(self, name, hp, atk, defense=0):
        self.name = name
        self.__hp = max(0, hp)
        self.__atk = max(0, atk)
        self.__def = max(0, defense)
    def get_hp(self):
        return self.__hp
    def is_alive(self):
        return self.__hp > 0
    def take_damage(self, raw_damage):
        dmg = max(0, raw_damage - self.__def)
        self.__hp = max(0, self.__hp - dmg)
        return dmg
    def attack(self, target):
        dmg = target.take_damage(self.__atk)
        print(f"{self.name} attacks {target.name} for {dmg}!")
```

Inheritance (Part 2): The Mage Child Class

```python
class Mage(Character):
    def __init__(self, name, hp, atk, defense=0, mana=50):
        super().__init__(name, hp, atk, defense)
        self.__mana = max(0, mana)

    def get_mana(self):
        return self.__mana

    def drink_mana_potion(self, amount):
        self.__mana = max(0, self.__mana + amount)

    def cast_fireball(self, target, cost=15, bonus_damage=20):
        if self.__mana < cost:
            print(f"{self.name} tried to cast Fireball... not enough mana!")
            return
        self.__mana -= cost
        dmg = target.take_damage(bonus_damage)
        print(f"{self.name} casts Fireball on {target.name} for {dmg}!")
```

Live Coding: Mini Test (Mage)

```python
arthur = Character("Arthur", 100, 15, defense=2)
merlin = Mage("Merlin", 70, 6, defense=1, mana=30)
goblin = Character("Goblin", 50, 5)

merlin.cast_fireball(goblin)
merlin.cast_fireball(goblin)
merlin.cast_fireball(goblin)

print("Goblin alive?", goblin.is_alive())
print("Merlin mana:", merlin.get_mana())
```

Phase 7: Polymorphism

Concept

Polymorphism = one interface, many behaviors.

Polymorphism (Part 1): Define the Interface

```python
class Character:
    # (same Character as before)
    def take_turn(self, enemy):
        # Default behavior: just attack
        self.attack(enemy)

class Warrior(Character):
    def take_turn(self, enemy):
        print(f"{self.name} goes for a heavy strike!")
        self.attack(enemy)
```

Checkpoint

Same method name (take_turn) across types = common interface.

Polymorphism (Part 2): Mage Overrides the Same Interface

```python
class Mage(Character):
    # (same Mage as before)
    def take_turn(self, enemy):
        if self.get_mana() >= 15:
            self.cast_fireball(enemy, cost=15, bonus_damage=20)
        else:
            print(f"{self.name} is out of mana, uses staff!")
            self.attack(enemy)
```

Checkpoint

Same interface, different behavior. That's polymorphism.

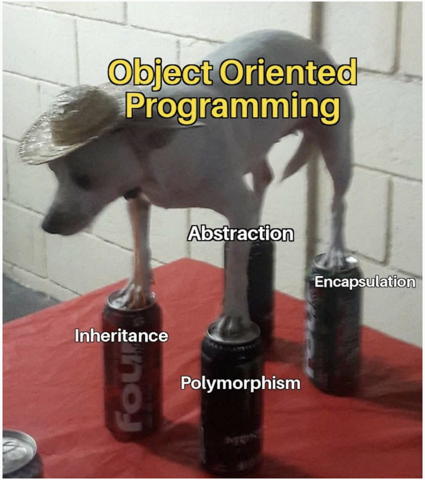
The Polymorphic Loop (Fixed + Runnable)

```
party = [
    Warrior("Conan", 120, 18, defense=3),
    Mage("Merlin", 70, 6, defense=1, mana=30),
]
enemy = Character("Goblin", 60, 7, defense=0)

turn = 1
while enemy.is_alive() and any(p.is_alive() for p in party):
    print(f"\n--- Turn {turn} ---")
    for member in party:
        if member.is_alive() and enemy.is_alive():
            member.take_turn(enemy)
    turn += 1

print("\nEnemy alive?", enemy.is_alive())
```

Summary

Recap

1. **Classes:** Blueprints for data + behavior.
2. **Encapsulation:** Protect state + enforce invariants.
3. **Abstraction:** Hide complexity behind simple methods.
4. **Inheritance:** Extend and reuse code via `super()`.
5. **Polymorphism:** Same interface (`take_turn`), different behaviors.



Thank You!