🐍 Object-Oriented Programming (OOP) in Python

🔒 Encapsulation

📌 What is Encapsulation?

Encapsulation is one of the fundamental principles of OOP.
It means **binding data (variables) and methods (functions) into a single unit (class)** and restricting direct access to some of the object's components.

👉 In Python, encapsulation is achieved using **access modifiers**:

1. **Public (`name`)** – Accessible everywhere.
2. **Protected (`_name`)** – A convention: should not be accessed outside the class (but still possible).
3. **Private (`__name`)** – Strictly hidden from outside the class (name mangling applies).

🎯 Why Encapsulation?

- ✅ Protects data from accidental modification.
- ✅ Increases code security.
- ✅ Controls how the data is accessed or modified.
- ✅ Makes classes easier to maintain.

```
class Student:
    def __init__(self, name, age):
        self.name = name     # public
        self.age = age       # public

s = Student("Rahul", 21)
print(s.name)   # ✅ Accessible
print(s.age)    # ✅ Accessible
```

```
class Student:
    def __init__(self, name, age):
        self.__name = name    # protected
        self._age = age      # protected

    def display(self):
        print(f"Name: {self.__name}, Age: {self._age}")

s = Student("Sonia", 22)
s.display()

# Technically accessible, but not recommended
print(s.__name)  # ⚠️ Allowed, but against convention
```

```
Name: Sonia, Age: 22
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipython-input-1957145286.py in <cell line: 0>()
     11
     12 # Technically accessible, but not recommended
---> 13 print(s.__name)  # ⚠️ Allowed, but against convention

AttributeError: 'Student' object has no attribute '__name'
```

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance   # private

    # Getter method
    def get_balance(self):
        return self.__balance

    # Setter method
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Invalid deposit amount!")
```

```
        def withdraw(self, amount):
            if 0 < amount <= self.__balance:
                self.__balance -= amount
            else:
                print("Invalid or insufficient funds!")

    # Testing
    acc = BankAccount("Aman", 1000)
    print(acc.owner)
    # print(acc.__balance)   # ❌ AttributeError
    print(acc.get_balance())  # ✅ Correct way

    acc.deposit(500)
    print(acc.get_balance())
    acc.withdraw(200)
    print(acc.get_balance())
```

⚡ Name Mangling in Python

Even private members can be accessed using name mangling:

```
    print(acc._BankAccount__balance)  # ⚠️ Not recommended
```

```
    class Student:
        def __init__(self, name):
            self.__name = name
            self.__marks = {}

        def set_mark(self, subject, mark):
            if 0 <= mark <= 100:
                self.__marks[subject] = mark
            else:
                print("Invalid mark!")

        def get_marks(self):
            return self.__marks

        def get_average(self):
            if self.__marks:
                return sum(self.__marks.values()) / len(self.__marks)
            return 0

    # Testing
    s = Student("Rahul")
    s.set_mark("Math", 95)
    s.set_mark("Science", 88)
    s.set_mark("English", 76)

    print(s.get_marks())
    print("Average:", s.get_average())
```

🔄 Polymorphism

📌 What is Polymorphism?

The word **Polymorphism** comes from Greek:

- **Poly** → many
- **Morph** → forms

👉 In OOP, **Polymorphism** allows the **same function or operator to behave differently based on the object or data type**.

🎯 Why Polymorphism?

- ✅ Makes code more flexible & reusable.
- ✅ Reduces redundancy.
- ✅ Helps in designing extensible systems.
- ✅ Encourages abstraction.

🤷 Example 1: Built-in Polymorphism

Some Python functions/operators work on multiple data types.

```
print(len("Python"))     # Works on string → 6
print(len([10, 20, 30])) # Works on list   → 3
print(len({"a": 1, "b": 2})) # Works on dict → 2
```

```
print(5 + 10)        # Addition → 15
print("AI" + " ML")  # String concatenation → "AI ML"
```

👷 Example 2: Polymorphism with Functions

A function can take different types of inputs and still work.

```
def add(x, y):
    return x + y

print(add(10, 20))        # int + int
print(add(3.5, 2.5))      # float + float
print(add("Hello ", "AI")) # str + str
```

👷 Example 3: Polymorphism with Class Methods

Different classes can have the same method name but with different implementations.

```
class Dog:
    def sound(self):
        return "Woof! 🐶"

class Cat:
    def sound(self):
        return "Meow! 🐱"

class Cow:
    def sound(self):
        return "Moo! 🐮"

# Using polymorphism
for animal in (Dog(), Cat(), Cow()):
    print(animal.sound())
```

👷 Example 4: Polymorphism with Inheritance (Method Overriding)

Child classes can override methods of the parent class.

```
class Bird:
    def fly(self):
        print("Birds can fly ✈️")

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly ❌")

b1 = Bird()
p1 = Penguin()

b1.fly()
p1.fly()  # Overridden method
```

👷 Example 5: Polymorphism with Abstract Classes

We can enforce method overriding using abc module.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

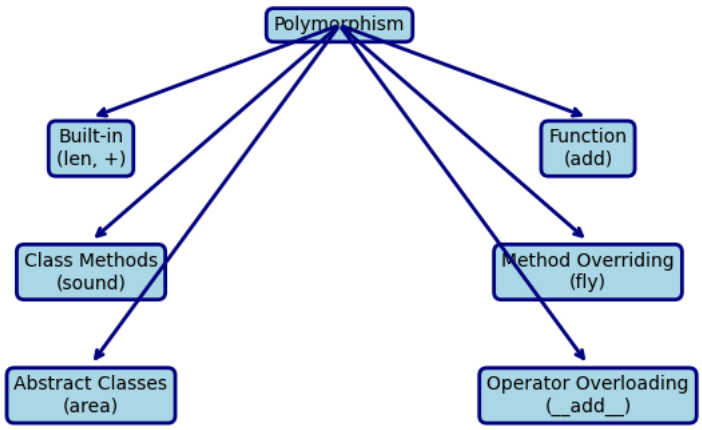
```
class Rectangle(Shape):
    def __init__(self, l, w):
        self.l = l
        self.w = w

    def area(self):
        return self.l * self.w

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r * self.r

# Using polymorphism
shapes = [Rectangle(5, 3), Circle(4)]
for shape in shapes:
    print(shape.area())
```

```
15
50.24
```

🎮 Mini Project: Payment System

```
class Payment:
    def pay(self, amount):
        raise NotImplementedError("Subclass must implement pay method")

class CreditCard(Payment):
    def pay(self, amount):
        return f"Paid {amount} using Credit Card 💳"

class PayPal(Payment):
    def pay(self, amount):
        return f"Paid {amount} using PayPal 🅿️"

class Crypto(Payment):
    def pay(self, amount):
        return f"Paid {amount} using Cryptocurrency ₿"

# Polymorphism in action
payments = [CreditCard(), PayPal(), Crypto()]

for method in payments:
    print(method.pay(500))
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8,6))
ax.axis("off")

# Nodes
nodes = {
    "Polymorphism": (0.5, 0.9),
    "Built-in\n(len, +)": (0.2, 0.7),
    "Function\n(add)": (0.8, 0.7),
    "Class Methods\n(sound)": (0.2, 0.5),
    "Method Overriding\n(fly)": (0.8, 0.5),
    "Abstract Classes\n(area)": (0.2, 0.3),
    "Operator Overloading\n(__add__)": (0.8, 0.3),
}

# Draw nodes
for text, (x, y) in nodes.items():
    ax.text(x, y, text, ha="center", va="center",
            bbox=dict(boxstyle="round,pad=0.4", fc="lightblue", ec="navy", lw=2))

# Arrows
for (x1, y1) in [nodes["Built-in\n(len, +)"], nodes["Function\n(add)"],
                 nodes["Class Methods\n(sound)"], nodes["Method Overriding\n(fly)"],
                 nodes["Abstract Classes\n(area)"], nodes["Operator Overloading\n(__add__)"]]:
    ax.annotate("", xy=(x1, y1+0.05), xytext=nodes["Polymorphism"],
                arrowprops=dict(arrowstyle="->", lw=2, color="navy"))

plt.show()
```


Polymorphism
Built-in
(len, +)
Function
(add)
Class Methods
(sound)
Method Overriding
(fly)
Abstract Classes
(area)
Operator Overloading
(__add__)