OOP in Python

Why OOP Matters?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects. Each object represents a real-world entity and contains attributes (data) and methods (functions).

Why do we use OOP?

1. **Modularity:** Code is structured in reusable classes and objects.
2. **Code Reusability:** Inheritance and polymorphism allow us to reuse code efficiently.
3. **Data Security:** Encapsulation protects data from unintended modifications.
4. **Scalability:** OOP makes it easier to maintain and scale large applications.
5. **Abstraction:** Focus on essential features while hiding unnecessary details.

1. Class Syntax and Info

A class is a blueprint for creating objects. It defines attributes (variables) and behaviors (methods) that its objects will have.

In [2]:
```
class a:
    pass

x = a()
```

In [4]:
```
class Car:

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        return f"Car Brand: {self.brand}, Model: {self.model}"

car1 = Car("Toyota", "Corolla") #instance or object
print(car1.display_info())
print(car1.brand)
print(car1.model)
```

```
Car Brand: Toyota, Model: Corolla
Toyota
Corolla
```

2. Instance Attributes and Methods

Each object (instance) of a class has attributes (data specific to the object) and methods (functions that operate on the object's data).

Why use Instance Attributes?

1. They allow each object to store unique data.
2. They are useful when creating multiple objects with different properties.

In [5]:
```
class Person:
    def __init__(self, name, age):
        self.name = name  #instance attribute
        self.age = age    #instance attribute

    def introduce(self):
        return f"Hi, my name is {self.name} and I am {self.age} years old."

person1 = Person("Alice", 25) #instance or object 1
person2 = Person("Bob", 30) #instance or object 2

print(person1.introduce())
print(person2.introduce())
```

```
Hi, my name is Alice and I am 25 years old.
Hi, my name is Bob and I am 30 years old.
```

3. Class Attributes and Methods

Class attributes are shared across all instances of a class, unlike instance attributes, which are unique to each instance. A class method is a method that is bound to the class itself, rather than an instance of the class. It is defined using the @classmethod decorator and takes cls (instead of self) as the first parameter.

Why use Class Attributes?

1. They provide a single shared value for all objects.
2. They are useful for defining constants.

Why use Class Methods?

1. Modify Class Attributes – Allows changes to class-level attributes that apply to all instances.
2. Alternative Constructors – Enables creating objects in different ways (from_string() method).
3. Access Without Instance – Can be called on the class itself, without needing an object.

In [11]:
```
class Employee:
    company = "Apple"  #class attribute is shared for all instances

    def __init__(self, name, role):
        self.name = name  #instance attribute
        self.role = role  #instance attribute

    def company_info_1(self): #instance method
```

```
        return f"This employee works at {self.company}."

    @classmethod
    def company_info(cls): #class method
        return f"This employee works at {cls.company}."

# Creating instances
emp1 = Employee("David", "Developer")
emp2 = Employee("Emma", "Designer")

print(Employee.company_info())
print(emp1.company_info())
print(emp2.company_info())
print(emp2.company_info_1())
print(emp1)
```

```
This employee works at Apple.
This employee works at Apple.
This employee works at Apple.
This employee works at Apple.
<__main__.Employee object at 0x000002646966DDC0>
```

4. Magic Methods

Magic methods (also called dunder methods) start and end with __. They allow special behaviors for objects, such as initialization, representation, or comparison.

Why use Magic Methods?

1. They allow custom behavior for built-in operations like printing, addition, etc.
2. They make objects behave more like built-in types.

In [19]:
```
print(dir(Book))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass_
_', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex
__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakr
ef__']
```

In [18]:
```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"Book('{self.title}', '{self.author}')"

#creating an instance
book1 = Book("1984", "George Orwell")
print(book1)  #calls __str__ method (string)
```

```
Book('1984', 'George Orwell')
```

5. Inheritance

Inheritance allows one class (child) to inherit the attributes and methods of another class (parent).

Why use Inheritance?

1. It helps avoid code duplication.
2. It allows extending the functionality of an existing class.

In [24]:
```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def make_sound(self):
        return "Some sound"

class Dog(Animal):  #dog inherit from animal
    def __init__(self, name, age, food):
        super().__init__(name, age)
        self.food = food #instance attribute for Dog class

    def make_sound(self):
        return "Bark!"

dog = Dog("Buddy", 10, "dry food")
animal = Animal("Dog", 5)

print(dog.name)  #inherited attribute
print(dog.food)
print(animal.name)
print(dog.make_sound())  #override method
print(animal.age)
print(dog.age)
print(animal.make_sound())
```

```
Buddy
dry food
Dog
Bark!
5
10
Some sound
```

6. Multiple Inheritance

A class can inherit from multiple parent classes, allowing it to combine behaviors from multiple sources.

Why use Multiple Inheritance?

1. It allows a class to inherit functionality from different classes.
2. It helps build complex objects with multiple behaviors.

In [29]:
```python
class Flyer:
    def fly(self):
        return "I can fly!"

class Swimmer:
    def swim(self):
        return "I can swim!"

class Duck(Flyer, Swimmer):  #duck inherits from Flyer and Swimmer
    pass

duck = Duck()
print(duck.fly())   #from flyer class
print(duck.swim())  #from swimmer class
```

```
I can fly!
I can swim!
```

7. Polymorphism

Polymorphism allows different objects to be treated the same way, even if they come from different classes.

Why use Polymorphism?

1. It enables code flexibility and reuse.
2. It allows different objects to use the same interface.

In [33]:
```python
# Base class
class PaymentMethod:
    def process_payment(self, amount):
        raise NotImplementedError("Subclasses must implement this method")

# Subclass for Credit Card Payment
class CreditCard(PaymentMethod):
    def process_payment(self, amount):
        return f"Processing ${amount} payment via Credit Card."

# Subclass for PayPal Payment
class PayPal(PaymentMethod):
    def process_payment(self, amount):
        return f"Processing ${amount} payment via PayPal."

# Subclass for Bitcoin Payment
class Bitcoin(PaymentMethod):
    def process_payment(self, amount):
        return f"Processing ${amount} payment via Bitcoin."

# Function that uses polymorphism
def make_payment(payment_method, amount):
    return payment_method.process_payment(amount)

# Creating different payment method objects
credit_card = CreditCard()
paypal = PayPal()
bitcoin = Bitcoin()
```

```
pay = PaymentMethod()
# Using the same function to process payments (Polymorphism in action)
print(make_payment(credit_card, 100))  # Processing $100 payment via Credit Card.
print(make_payment(paypal, 250))  # Processing $250 payment via PayPal.
print(make_payment(bitcoin, 500))  # Processing $500 payment via Bitcoin.
print(pay.process_payment(1000))
```

```
Processing $100 payment via Credit Card.
Processing $250 payment via PayPal.
Processing $500 payment via Bitcoin.
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[33], line 35
     33 print(make_payment(paypal, 250))  # Processing $250 payment via PayPal.
     34 print(make_payment(bitcoin, 500))  # Processing $500 payment via Bitcoin.
---> 35 print(pay.process_payment(1000))

Cell In[33], line 4, in PaymentMethod.process_payment(self, amount)
      3 def process_payment(self, amount):
----> 4     raise NotImplementedError("Subclasses must implement this method")

NotImplementedError: Subclasses must implement this method
```

8. Encapsulation

Encapsulation is the practice of hiding internal details of an object and restricting direct access.

Why use Encapsulation?

1. It protects sensitive data.
2. It ensures controlled modification of data.

In [ ]:
```
private => __hamada
protected => _hamada
public  => hamada
```

In [39]:
```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute (cannot be accessed directly)

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"Deposited {amount}, new balance: {self.__balance}"

    def get_balance(self): #getter
        return self.__balance  #controlled access

# Creating an object
account = BankAccount(1000)
#print(account.deposit(500))  # Deposited 500
#print(account.get_balance())  # Accessing balance safely

#note
print(account._BankAccount__balance)
```

```
account._BankAccount__balance = 2000
print(account._BankAccount__balance)
```

```
1000
2000
```

9. Getter & Setter

In [40]:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute

    def get_salary(self):  # Explicit getter
        return self.__salary

    def set_salary(self, new_salary):  # Explicit setter
        if new_salary >= 0:
            self.__salary = new_salary
        else:
            print("Salary cannot be negative!")


# Using getter and setter methods explicitly
emp = Employee("Alice", 5000)
print(emp.get_salary())  # Need to call as a method
emp.set_salary(6000)
print(emp.get_salary())  # 6000
```

```
5000
6000
```

Python Code Implementation

In [57]:

```
#university system
#=> student
#=> professors
#=> courses

#base class
class Person:
    university_name = "Tech University"  #class attribute (shared across all instances

    def __init__(self, name, age, id_number):
        self.name = name  #instance attribute
        self.age = age    #instance attribute
        self.id_number = id_number  #instance attribute

    def introduce(self):
        return f"Hello, I'm {self.name}."

    @classmethod
    def get_university(cls):
        return f"Welcome to {cls.university_name}"


#student class (inherits from Person)
```

```python
class Student(Person):
    def __init__(self, name, age, id_number):
        super().__init__(name, age, id_number)  #call the parent constructor
        self.courses = []  #instance attribute: stores enrolled courses

    def enroll_course(self, course):
        self.courses.append(course)
        return f"{self.name} has enrolled in {course.course_name}."

    def introduce(self):  #polymorphism (overrides parent method)
        return f"I'm {self.name}, a student at {self.university_name}."

    def __str__(self):  #magic method: string representation
        return f"Student({self.name}, {self.age}, {self.id_number})"

#employee class (for multiple inheritance)
class Employee:
    def __init__(self, salary):
        self._salary = salary  #encapsulation: protected attribute

    def get_salary(self): #getter
        return f"My salary is {self._salary}."


#professor class (inherits from person and employee)
class Professor(Person, Employee):
    def __init__(self, name, age, id_number, salary):
        Person.__init__(self, name, age, id_number)
        Employee.__init__(self, salary)
        self.courses_taught = []  #instance attribute

    def assign_course(self, course):
        self.courses_taught.append(course)
        return f"{self.name} is teaching {course.course_name}."

    def introduce(self):  #polymorphism (overrides parent method) Person
        return f"I'm Prof. {self.name}, I teach at {self.university_name}."

    def __str__(self):  #magic method: string representation
        return f"Professor({self.name}, {self.age}, {self.id_number})"


# Course Class
class Course:
    def __init__(self, course_name, course_code):
        self.course_name = course_name  #instance attribute
        self.course_code = course_code  #instance attribute
        self.students = []  #list of enrolled students

    def enroll_student(self, student):
        self.students.append(student)
        return f"{student.name} has enrolled in {self.course_name}."

    def __repr__(self):  #magic method: debug representation
        return f"Course({self.course_name}, {self.course_code})"


#university class (manages everything)
class University:
    def __init__(self):
```

```
        self.students = []
        self.professors = []
        self.courses = []

    def add_student(self, student):
        self.students.append(student)
        return f"Added Student: {student.name}"

    def add_professor(self, professor):
        self.professors.append(professor)
        return f"Added Professor: {professor.name}"

    def add_course(self, course):
        self.courses.append(course)
        return f"Added Course: {course.course_name}"

    def show_all_students(self):
        return [str(student) for student in self.students]

    def show_all_professors(self):
        return [str(professor) for professor in self.professors]

    def show_all_courses(self):
        return [repr(course) for course in self.courses]


# -------------------
# TESTING THE SYSTEM
# -------------------

#create university
uni = University()

#create courses
course1 = Course("Python Programming", "CS101")
course2 = Course("Data Structures", "CS102")

#add courses to university
uni.add_course(course1)
uni.add_course(course2)

#create students
student1 = Student("Alice", 20, "S001")
student2 = Student("Bob", 22, "S002")

#add students to university
uni.add_student(student1)
uni.add_student(student2)

#enroll students in courses
print(student1.enroll_course(course1))
print(student2.enroll_course(course2))

#create professor
professor1 = Professor("Dr. Smith", 45, "P001", 5000) #Person , Employee

#add professor to university
uni.add_professor(professor1)

#assign courses to professor
```

```
print(professor1.assign_course(course1))

#display information
print("\n=== University Students ===")
print(uni.show_all_students())

print("\n=== University Professors ===")
print(uni.show_all_professors())

print("\n=== University Courses ===")
print(uni.show_all_courses())

#test polymorphism
print("\n=== Polymorphism ===") # => Person
person = Person("ali", 44, 77)
print(person.introduce())
print(student1.introduce())  #from student class
print(professor1.introduce())  #from professor class

#test encapsulation
print("\n=== Encapsulation ===")
print(professor1.get_salary())  #access salary safely

#test class method
print("\n=== Class Method ===")
print(Person.get_university())
print(person.get_university())
```

```
Alice has enrolled in Python Programming.
Bob has enrolled in Data Structures.
Dr. Smith is teaching Python Programming.

=== University Students ===
['Student(Alice, 20, S001)', 'Student(Bob, 22, S002)']

=== University Professors ===
['Professor(Dr. Smith, 45, P001)']

=== University Courses ===
['Course(Python Programming, CS101)', 'Course(Data Structures, CS102)']

=== Polymorphism ===
Hello, I'm ali.
I'm Alice, a student at Tech University.
I'm Prof. Dr. Smith, I teach at Tech University.

=== Encapsulation ===
My salary is 5000.

=== Class Method ===
Welcome to Tech University
Welcome to Tech University
```

In [ ]:

In [ ]: