Object-Oriented Programming (OOP)

Concepts, Principles, and Implementation in Python

Lenny Pelhate

CentraleSupélec

April 5, 2025

Table of Contents

Table of Contents

What is Object-Oriented Programming?

- A programming paradigm based on the concept of **objects**
- Objects contain data (attributes) and code (methods)
- Organizes software design around data rather than functions and logic
- Aims to increase flexibility and maintainability of code

Procedural vs. Object-Oriented Programming

Procedural Programming

- Focuses on procedures/functions
- Data shared across functions
- Top-down approach
- Examples: C, Fortran

Object-Oriented Programming

- Focuses on objects
- Data encapsulated within objects
- Bottom-up approach
- Examples: Python, Java, C++

Table of Contents

Classes and Objects

- **Class**: A blueprint for creating objects
  - Defines attributes and methods
  - Template that describes the behaviors/states
- **Object**: An instance of a class
  - Has state (attributes) and behavior (methods)
  - Occupies memory space when created

Class and Object Example in Python

```python
# Class definition
class Car:
    # Constructor (initializer)
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    # Method
    def drive(self):
        print(f"{self.brand} {self.model} is driving.")

    def get_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Creating objects
my_car = Car("Toyota", "Corolla", 2020)
your_car = Car("Honda", "Civic", 2019)

# Using object methods
my_car.drive()  # Output: Toyota Corolla is driving.
print(your_car.get_info())  # Output: 2019 Honda Civic
```

The Four Pillars of OOP

1. **Encapsulation**: Bundling data with methods that operate on that data
2. **Inheritance**: Mechanism for code reuse and establishing relationships
3. **Polymorphism**: Ability to process objects differently based on their data type
4. **Abstraction**: Hiding complexity by exposing only necessary parts

The Four Pillars of OOP

1. **Encapsulation**: Bundling data with methods that operate on that data
2. **Inheritance**: Mechanism for code reuse and establishing relationships
3. **Polymorphism**: Ability to process objects differently based on their data type
4. **Abstraction**: Hiding complexity by exposing only necessary parts

Encapsulation

- Bundling of data and methods that operate on that data
- Restricting direct access to some components
- **Information hiding**: Implementation details hidden
- In Python:
  - Convention-based: Single underscore (_) for protected
  - Double underscore (__) for private (name mangling)
  - Getters and setters using @property decorator
- Benefits:
  - Control over data access and modification
  - Improved maintainability and flexibility

Encapsulation Example in Python

```python
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number  # Private
        self.__balance = initial_balance        # Private

    @property # This makes this method a getter (the method will work like an attribute)
    def balance(self):
        """Getter for balance"""
        return self.__balance # Returns the value of the private attribute __balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

account = BankAccount("123456", 1000)
print(account.balance)          # Uses getter: 1000
account.deposit(500)            # Controlled access
print(account.balance)          # 1500
# print(account.__balance)      # Error: attribute doesn't exist
```

The Four Pillars of OOP

1. **Encapsulation**: Bundling data with methods that operate on that data
2. **Inheritance**: Mechanism for code reuse and establishing relationships
3. **Polymorphism**: Ability to process objects differently based on their data type
4. **Abstraction**: Hiding complexity by exposing only necessary parts

Inheritance

- Mechanism for code reuse and establishing relationships
- Classes can inherit attributes and methods from other classes
- Terminology:
  - **Parent/Base/Super class**: The class being inherited from
  - **Child/Derived/Sub class**: The class that inherits
- Types of inheritance:
  - Single inheritance
  - Multiple inheritance (Python supports this)
  - Multilevel inheritance
  - Hierarchical inheritance

Inheritance

- **Key lines for inheritance:**
  - `class ChildClass(ParentClass):`
    - *Defines the relationship between the child and parent class.*
  - `super().__init__()`
    - *Calls the constructor of the parent class to initialize inherited attributes.*
- The child class inherits all methods and attributes from the parent class unless overridden.

Inheritance Example in Python

```
class Vehicle:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        print(f"The {self.brand} {self.model}'s engine is starting...")

    def get_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Car inherits from Vehicle
class Car(Vehicle):
    def __init__(self, brand, model, year, doors):
        # Call parent class constructor
        super().__init__(brand, model, year)
        self.doors = doors

    def drive(self):
        print(f"The {self.brand} {self.model} is being driven.")

# Motorcycle inherits from Vehicle
class Motorcycle(Vehicle):
    def __init__(self, brand, model, year, has_sidecar):
        super().__init__(brand, model, year)
        self.has_sidecar = has_sidecar

car = Car("Toyota", "Camry", 2020, 4)
car.start_engine()  # Inherited method
car.drive()         # Car-specific method
```

The Four Pillars of OOP

1. **Encapsulation**: Bundling data with methods that operate on that data
2. **Inheritance**: Mechanism for code reuse and establishing relationships
3. **Polymorphism**: Ability to process objects differently based on their data type
4. **Abstraction**: Hiding complexity by exposing only necessary parts

Polymorphism

- Ability to process objects differently based on their data type
- Types of polymorphism:
  - **Method overriding**: Redefining methods in child classes
  - **Method overloading**: Same method name, different parameters
    - *Unlike languages like Java or C++, Python does not support method overloading in the traditional sense. Python allows only one method with a particular name, but one can achieve similar behavior by using default arguments or variable-length argument lists (*args and **kwargs).*
  - **Duck typing**: The type or class of an object is determined by its behavior (methods and properties) rather than its actual type.
    - *Instead of checking the object's type, Python checks if the object has the required methods or properties. If it does, the object is considered to be of the expected type.*
    - *Example: "If it walks like a duck and quacks like a duck, then it must be a duck."*
- Benefits:
  - Code flexibility and reusability
  - Interface consistency
  - Runtime determination of method calls

Polymorphism Example in Python

Method Overriding:

```
class Animal:
    def speak(self):
        pass  # Abstract method to be overridden

class Dog(Animal):
    def speak(self): # Polymorphism (child method overrides parent)
        return "Woof!"
```

Method Overloading:

```
class Calculator:
    def add(self, *args):  # Method Overloading (simulated with *args)
        return sum(args)

calc = Calculator()
print(calc.add(1, 2))           # Output: 3
print(calc.add(1, 2, 3, 4))      # Output: 10
```

Duck Typing:

```
def animal_speak(animal):
    animal.speak()  # It doesn't matter if it's a Dog, Cat, or Duck

class Person:
    def speak(self):
        return "Hello!"

# Even though Person isn't an Animal subclass, it works because it has a speak() method
animal_sound(Person())  # Output: Hello!
```

The Four Pillars of OOP

1. **Encapsulation**: Bundling data with methods that operate on that data
2. **Inheritance**: Mechanism for code reuse and establishing relationships
3. **Polymorphism**: Ability to process objects differently based on their data type
4. **Abstraction**: Hiding complexity by exposing only necessary parts

Abstraction

- Hiding complex implementation details
- Showing only the necessary features of an object
- In Python:
  - **Abstract Base Classes (ABC module)**:
    - Classes that cannot be instantiated on their own and are meant to be inherited.
  - **Abstract methods**:
    - Method declared in an ABC that must be implemented by any non-abstract subclass.
    - @abstractmethod decorator to declare abstract methods
    - It has no body (or just a pass).
- Benefits:
  - Reduces complexity and isolates impact of changes
  - Forces derived classes to implement specific methods
  - Provides a common interface for related classes

Abstraction Example in Python

```python
from abc import ABC, abstractmethod

class Shape(ABC):   # This is an abstract base class (ABC)
    @abstractmethod
    def area(self): # This is an abstract method
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# shape = Shape()  # Error: Can't instantiate abstract class
rect = Rectangle(5, 3)

print(f"Rectangle area: {rect.area()}")
```

Table of Contents

Advanced OOP Concepts

Composition vs. Inheritance

Inheritance ("is-a" relationship)

- Car is a Vehicle
- Circle is a Shape
- Derived class inherits properties

Composition ("has-a" relationship)

- Car has an Engine
- Library has Books
- Object contains other objects

Python principle: *"Composition over inheritance"*

Composition Example in Python

```
class Engine:
    def __init__(self, power):
        self.power = power

    def start(self):
        return "Engine started!"

class Wheel:
    def __init__(self, size):
        self.size = size

class Car:
    def __init__(self, model, engine_power, wheel_size):
        self.model = model
        # Composition: Car has an Engine
        self.engine = Engine(engine_power)
        # Composition: Car has Wheels
        self.wheels = [Wheel(wheel_size) for _ in range(4)]

    def start(self):
        print(f"{self.model}: {self.engine.start()}")

    def get_specs(self):
        return f"{self.model} with {self.engine.power}hp engine and {self.wheels[0].size
     }\" wheels"

my_car = Car("Toyota Corolla", 132, 16)
my_car.start()  # Toyota Corolla: Engine started!
print(my_car.get_specs())
```

Static Methods and Class Methods

- **Instance methods**: Regular methods that operate on instance data
  - First parameter is always `self`
- **Class methods**: Methods that operate on class data
  - First parameter is always `cls`
  - Decorated with `@classmethod`
  - Can be called on the class itself
- **Static methods**: Methods that don't need instance or class data
  - No mandatory first parameter
  - Decorated with `@staticmethod`
  - Utility functions related to the class

Static and Class Methods Example

```python
class MathUtility:
    pi = 3.14159

    def __init__(self, value):
        self.value = value

    def square(self): # Instance method: uses self
        return self.value ** 2

    @classmethod
    def circle_area(cls, radius): # Class method: uses cls.pi
        return cls.pi * radius ** 2

    @staticmethod
    def is_prime(num): # Static method: doesn't use self or cls (like a normal function)
        if num < 2:
            return False
        for i in range(2, int(num ** 0.5) + 1):
            if num % i == 0:
                return False
        return True

# Using class methods/static methods directly
area = MathUtility.circle_area(5)
print(f"Area of circle: {area}")
print(f"Is 17 prime? {MathUtility.is_prime(17)}")

# Using instance methods
math = MathUtility(4)
print(f"Square of 4: {math.square()}")
```

Advanced OOP Concepts

Special Methods (Magic/Dunder Methods)

- Special methods surrounded by double underscores (dunders): `__method__`
- Allow classes to emulate built-in types behavior
- Key special methods:
  - `__init__`: Constructor/Initializer
  - `__str__`: String representation (user-friendly)
  - `__repr__`: String representation (developer-friendly)
  - `__len__`: Length of object
  - `__add__`, `__sub__`, etc.: Arithmetic operations
  - `__eq__`, `__lt__`, etc.: Comparison operations
  - `__getitem__`, `__setitem__`: Indexing operations

Special Methods Example

```python
class Vector:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        """User-friendly string representation"""
        return f"Vector({self.x}, {self.y}, {self.z})"

    def __repr__(self):
        """Developer-friendly string representation"""
        return f"Vector(x={self.x}, y={self.y}, z={self.z})"

    def __add__(self, other):
        """Vector addition with + operator"""
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y, self.z + other.z)
        return NotImplemented

    def __len__(self):
        """Makes len(vector) return dimensionality"""
        return 3

v1 = Vector(1, 2, 3)
v2 = Vector(4, 5, 6)
v3 = v1 + v2  # Uses __add__
print(v3)  # Uses __str__: Vector(5, 7, 9)
```

Properties and Descriptors

- **Properties**: Control access to attributes
  - `@property`: Getter
    - A method that retrieves (gets) the value of a private attribute.
  - `@attribute.setter`: Setter
    - A method that modifies (sets) the value of a private attribute.
  - `@attribute.deleter`: Deleter
    - A special method used to delete a property safely, and optionally define custom behavior when deletion happens.
- Benefits:
  - Data validation
  - Computed properties
  - Encapsulation with natural syntax

Properties Example

Example without Getters/Setters:

```python
class Person:
    def __init__(self, age):
        self.age = age

p = Person(25)
p.age = -5  # Invalid age, no protection
```

Example with Getters/Setters:

```python
class Person:
    def __init__(self, age):
        self.__age = age  # private attribute

    @property
    def age(self): # Getter method
        return self.__age

    @age.setter
    def age(self, value): # Setter method
        if value >= 0:
            self.__age = value
        else:
            raise ValueError("Age cannot be negative")

p = Person(25)
print(p.age)      # Uses getter: 25
p.age = 30        # Uses setter: valid
p.age = -10       # Uses setter: raises ValueError
```

Multiple Inheritance and MRO

- Python supports inheriting from multiple parent classes
- **Method Resolution Order (MRO)**: Determines method lookup path
  - C3 linearization algorithm
  - Can be viewed using `Class.__mro__`
- Diamond problem: Ambiguity in multiple inheritance
- Best practices:
  - Use composition over multiple inheritance when possible
  - Keep inheritance hierarchies shallow
  - Use `super()` correctly

Multiple Inheritance Example

Multiple Inheritance with no conflict:

```python
class Printer:
    def print_doc(self):
        print("Printing...")
class Scanner:
    def scan(self):
        print("Scanning...")

class MultiFunctionDevice(Printer, Scanner):
    pass

mfd = MultiFunctionDevice()
mfd.print_doc()
mfd.scan()
```

Multiple Inheritance with conflict (automatically solved by MRO):

```python
class A:
    def greet(self):
        print("Hello from A")
class B:
    def greet(self):
        print("Hello from B")

class C(A, B):
    pass

c = C()
c.greet()              # Uses A's method
print(C.__mro__)       # Method Resolution Order
```

Table of Contents

Common Design Patterns

- **Creational Patterns**:
  - Singleton: One instance of a class
  - Factory: Create objects without specifying exact class
  - Builder: Construct complex objects step by step
- **Structural Patterns**:
  - Adapter: Interface compatibility between classes
  - Decorator: Add responsibilities to objects dynamically
  - Facade: Simplified interface to a complex subsystem
- **Behavioral Patterns**:
  - Observer: One-to-many dependency between objects
  - Strategy: Family of algorithms, each encapsulated
  - Iterator: Sequential access to elements

Singleton Pattern Example

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

class DatabaseConnection(Singleton):
    def __init__(self, host="localhost", port=5432):
        # Initialize only once
        if not hasattr(self, "initialized"):
            self.host = host
            self.port = port
            self.initialized = True
            print(f"Connecting to database at {host}:{port}")

    def execute_query(self, query):
        print(f"Executing: {query}")

# Both variables reference the same instance
db1 = DatabaseConnection("localhost", 5432)
db2 = DatabaseConnection("127.0.0.1", 5555)  # Parameters ignored

print(db1 is db2)  # True
print(db1.host)    # "localhost" (not "127.0.0.1")
```

OOP Best Practices

- **SOLID Principles**:
  - **S**ingle Responsibility: Class has one reason to change
  - **O**pen/Closed: Open for extension, closed for modification
  - **L**iskov Substitution: Subclasses should substitute base classes
  - **I**nterface Segregation: Many specific interfaces over one general
  - **D**ependency Inversion: Depend on abstractions, not concretions
- Other best practices:
  - Favor composition over inheritance
  - Keep classes focused and cohesive
  - Implement proper encapsulation
  - Write clear method and class names
  - Follow language conventions (PEP 8 for Python)

Table of Contents

OOP Terms and Concepts (1/2)

- **Class**: Blueprint for creating objects
- **Object/Instance**: Concrete entity created from a class
- **Attribute/Property**: Data stored in a class or instance
- **Method**: Function defined inside a class
- **Constructor**: Special method for initializing objects (__init__)
- **Inheritance**: Mechanism to derive a class from another class
- **Encapsulation**: Bundling data with methods that operate on that data
- **Polymorphism**: Ability to process objects differently based on their data type
- **Abstraction**: Hiding implementation details
- **Interface**: Contract defining behavior

OOP Terms and Concepts (2/2)

- **Composition**: Building complex objects from simpler ones
- **Method Overriding**: Redefining method in a subclass
- **Superclass/Base class**: Parent class in inheritance
- **Subclass/Derived class**: Child class in inheritance
- **Instance variable**: Variable defined in instance method
- **Class variable**: Variable shared across all instances
- **Static method**: Method that doesn't access instance or class data
- **Class method**: Method that operates on class rather than instance
- **Abstract class**: Class that cannot be instantiated
- **Design pattern**: Reusable solution to common problems
- **Duck typing**: "If it walks like a duck and quacks like a duck..."

Table of Contents

Conclusion: Benefits of OOP

- **Modularity**: Objects can be maintained independently
- **Reusability**: Inheritance promotes code reuse
- **Productivity**: Libraries of objects can be purchased
- **Maintainability**: Implementation details hidden
- **Scalability**: Programs can grow through composition
- **Collaboration**: Teams can work on different classes
- **Flexibility**: Polymorphism enables extension