OOP in Python

Part II

Polymorphism in Object-Oriented Programming

- Learning Goals
  - Polymorphism
    - Understanding Method Overriding and Method Overloading (Operator Overloading) in Python
  - Encapsulation
    - Understanding Public, Protected, and Private Members

Polymorphism

What is Polymorphism?

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to take on multiple forms. It's derived from the Greek words "poly" (meaning many) and "morph" (meaning form).

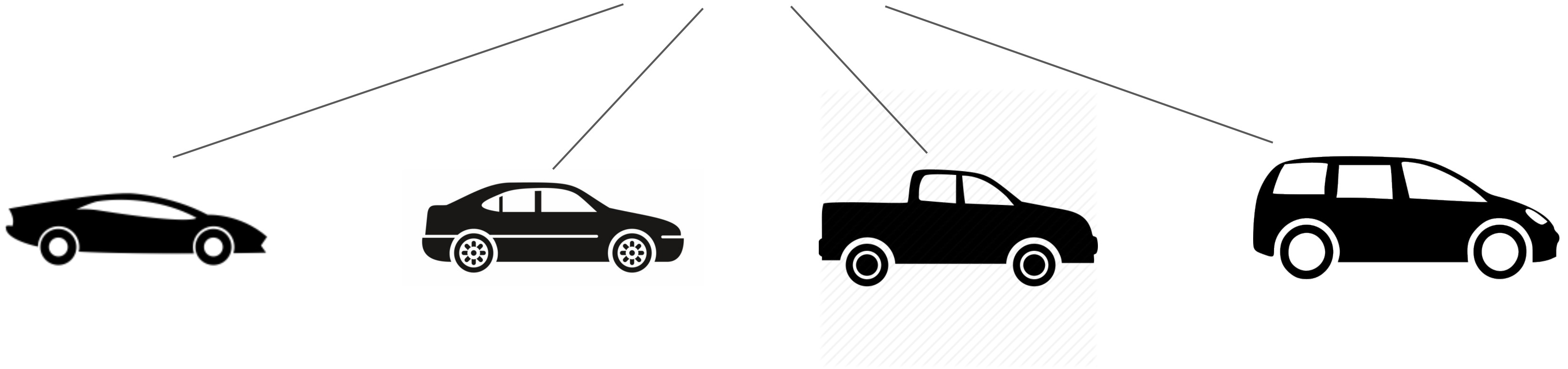
Key Points:

Enables one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

For example, if you learned driving a car, you will be able to drive on any car. It does not depend on car brand or inner implementation. It has the same driver interface.

Polymorphism

`drive()`



One Interface - Multiple Implementation

Polymorphism

Why It Matters:

Code Reusability: Allows the same piece of code to interact with objects from different classes in a unified manner. This means you can write more generic and reusable code.

Flexibility: Makes it easier to introduce new classes and objects that work seamlessly with existing code. As a result, your software can grow and evolve with minimal changes to existing functionality.

Polymorphism

- Method Overriding
  - It was covered in the OOP Part I.
- Method Overloading (Operator Overloading)

Method Overloading

- Understanding Method Overloading
  - Method overloading allows multiple methods in a class to have the same name but different parameters. It's common in many programming languages for implementing polymorphism at compile time.
- Python's Approach:
  - Unlike many object-oriented languages, Python does not support method overloading in the traditional sense. In Python, if multiple methods have the same name, only the last defined method is retained.

Achieving Method Overloading Effects

1. Default Parameter Values:

Python methods can have default values for parameters. This feature can mimic method overloading by allowing a method to be called with different numbers of arguments.

Example:

```
def display_info(name, age=None):

    if age:

        print(f"Name: {name}, Age: {age}")

    else:

        print(f"Name: {name}")

display_info("John")

display_info("John", 25)
```

Achieving Method Overloading Effects

2. Variable-length Arguments:

*args (Non-Keyword Arguments):

Allows a method to take an arbitrary number of arguments without defining them all.

Example:

```
def add(*numbers):

    return sum(numbers)

print(add(1, 2, 3, 4, 5))
```

Achieving Method Overloading Effects

**kwargs (Keyword Arguments):

Enables passing a variable number of keyword arguments to a method.

Example:

```
def describe_pet(name, **properties):
    print(f"Pet Name: {name}")
    for key, value in properties.items():
        print(f"{key}: {value}")
describe_pet("Fluffy", age=2, color="White")
describe_pet("Fluffy", age=2, color="White", species="Cat")
```

Operator Overloading

What is Operator Overloading?

Operator overloading allows operators to have different meanings depending on their operands. In Python, this is achieved by defining special methods in classes (often referred to as "**magic methods**").

Why Operator Overloading?

Flexibility and Intuitiveness: Enables custom behavior for arithmetic, comparison, and other operations on objects of user-defined classes.

Enhanced Readability: Allows for more natural and expressive code that aligns with the operations being performed.

Operator Overloading

How Python Implements Operator Overloading:

Magic Methods: Special methods that begin and end with double underscores (__). Each magic method corresponds to a specific operator.

Commonly Used Magic Methods:

Addition: `__add__(self, other)`

Subtraction: `__sub__(self, other)`

Multiplication: `__mul__(self, other)`

Equality: `__eq__(self, other)`

Operator Overloading

Example: Vector Addition

Scenario: Define a Vector class where vectors can be added using the + operator.

Implementation:

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

    # Overloading the + operator
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3.x, v3.y)  # Output: 4 6
```

Operator Overloading

Benefits of Operator Overloading:

Intuitive Operations: Allows for operations that are logical for the objects being manipulated, making code easier to understand and maintain.

Custom Behavior: Supports defining how user-defined objects should interact through familiar operators, enhancing the expressiveness of the language.

Considerations:

Clarity vs. Complexity: While operator overloading can make code more intuitive, overuse or misuse can lead to code that's hard to understand and maintain. It's essential to use this feature judiciously to ensure code clarity.

More examples

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

    def __add__(self, other):
        if isinstance(other, Car):
            return f"{self.brand} {self.model} ({self.year}) and {other.brand} {other.model} ({other.year}) are added together."
        else:
            raise TypeError("Unsupported operand type(s) for +: 'Car' and '{}'".format(type(other)))

    def __sub__(self, other):
        if isinstance(other, Car):
            return f"{self.brand} {self.model} ({self.year}) subtracted by {other.brand} {other.model} ({other.year})."
        else:
            raise TypeError("Unsupported operand type(s) for -: 'Car' and '{}'".format(type(other)))

    def __eq__(self, other):
        if isinstance(other, Car):
            return self.brand == other.brand and self.model == other.model and self.year == other.year
        else:
            return False

# Example usage:
car1 = Car("Toyota", "Camry", 2020)
car2 = Car("Honda", "Accord", 2019)

print(car1 + car2)  # Output: Toyota Camry (2020) and Honda Accord (2019) are added together.
print(car1 - car2)  # Output: Toyota Camry (2020) subtracted by Honda Accord (2019).
print(car1 == car2) # Output: False
```

```
Toyota Camry (2020) and Honda Accord (2019) are added together.
Toyota Camry (2020) subtracted by Honda Accord (2019).
False
```

List of Magic Methods for Operator Overloading

```
+     __add__(self, other)
–     __sub__(self, other)
*     __mul__(self, other)
/     __truediv__(self, other)
//     __floordiv__(self, other)
%     __mod__(self, other)
**     __pow__(self, other)
>>     __rshift__(self, other)
<<     __lshift__(self, other)
&     __and__(self, other)
|     __or__(self, other)
^     __xor__(self, other)
<     __lt__(self, other)
>     __gt__(self, other)
<=     __le__(self, other)
>=     __ge__(self, other)
==     __eq__(self, other)
!=     __ne__(self, other)
```

```
-=     __isub__(self, other)
+=     __iadd__(self, other)
*=     __imul__(self, other)
/=     __idiv__(self, other)
//=     __ifloordiv__(self, other)
%=     __imod__(self, other)
**=     __ipow__(self, other)
>>=     __irshift__(self, other)
<<=     __ilshift__(self, other)
&=     __iand__(self, other)
|=     __ior__(self, other)
^=     __ixor__(self, other)
–     __neg__(self)
+     __pos__(self)
~     __invert__(self)
```

Lab 16-1: Polymorphism through Operator Overloading

Objective: Implement a Rectangle class that supports addition using the + operator, where adding two rectangles creates a new rectangle with the combined area of the two.

Task 1: Define the Rectangle Class

Initialize Rectangle class with `width` and `height` attributes.

Implement the `__add__` method to allow two Rectangle instances to be added together. The new rectangle's width and height should be the sum of the widths and heights of the rectangles being added.

Add a method area() that returns the area of the rectangle.

Task 2: Test Your Class

Create two Rectangle objects with dimensions like this (3 x 4 and 7 x 4) .

Add these two rectangles together to create a third rectangle (10 x 8).

Print the area (80) of the resulting rectangle to verify that your __add__ implementation works correctly.

Encapsulation

What is Encapsulation?

Encapsulation is a fundamental principle in object-oriented programming (OOP) that involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or class, and restricting access to some of the object's components.

Purpose of Encapsulation:

Data Hiding: Prevents external parts of a program from directly accessing the internal mechanisms of a class.

Simplified Interface: Offers a clear, simplified interface for interaction while hiding its complex internal workings.

Encapsulation

Access Modifiers: Python uses access modifiers (public, protected, and private) to implement encapsulation, but with a more lenient approach compared to other languages.

Public Members: Accessible from anywhere.

Protected Members: Prefixed with **a single underscore** _. Intended for internal use and subclassing.

Private Members: Prefixed with **double underscores** __. Access is restricted to within the class only.

Private Member

```
class Car:
    def __init__(self, speed=0):
        self.__speed = speed  # Private variable

    def set_speed(self, speed):
        if speed < 0:
            print("Speed cannot be negative.")
        else:
            self.__speed = speed

    def get_speed(self):
        return self.__speed

    def __private_method(self):
        print("This is a private method.")

    def drive(self):
        self.__private_method()
        print(f"The car is driving at {self.__speed} speed.")

# Creating an object
my_car = Car(50)

# Accessing through public method
my_car.set_speed(60)
print(my_car.get_speed())

# Attempting direct access (will raise an error)
# print(my_car.__speed)  # AttributeError: 'Car' object has no attribute '__speed'

# Attempting to access a private method (will raise an error)
# my_car.__private_method()  # AttributeError: 'Car' object has no attribute '__private_method'
```

Protected Member

```python
class Vehicle:
    def __init__(self, brand, model):
        self._brand = brand  # Protected attribute
        self._model = model  # Protected attribute

    def _display_info(self):  # Protected method
        print(f"This is a {self._brand} {self._model}.")

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand, model)
        self.year = year  # Public attribute

    def show_details(self):
        # Accessing protected method from the parent class
        self._display_info()
        print(f"It was made in {self.year}.")

# Creating an object of the Car class
my_car = Car("Toyota", "Camry", 2019)

# Accessing protected attributes and method from outside the class is possible but not recommended
print(my_car._brand)  # Not recommended
my_car._display_info()  # Not recommended

# Using the class method which internally uses protected members
my_car.show_details()
```

```
Toyota
This is a Toyota Camry.
This is a Toyota Camry.
It was made in 2019.
```

Lab 16-2: Encapsulation

Implement a BankAccount class that encapsulates the balance as a private attribute and provides public methods to deposit and withdraw funds while ensuring the balance never becomes negative.

Task 1: Define the BankAccount Class

Initialize `BankAccount` class with a private `__balance` attribute.

Implement a public method `deposit(self, amount)` to add funds to the balance.

Implement a public method `withdraw(self, amount)` that only allows funds to be withdrawn if the balance remains non-negative after the withdrawal.

Add a public method `get_balance()` that returns the current balance.

Task 2: Test Your Class

Create a BankAccount object with an initial balance ($100).

Test depositing ($200) and withdrawing ($150 and then $400) funds, ensuring that withdrawing more than the balance is not allowed.

Print the balance ($150) after each operation to demonstrate that your encapsulation and access control mechanisms work as expected.