Week 5: Inheritance and Polymorphism in Python

Authors: Refat Othman and Diaeddin Rimawi.

Lecture 1: Inheritance and Polymorphism

1. Inheritance

Definition: Inheritance is a mechanism in object-oriented programming that allows a class (child or subclass) to inherit attributes and methods from another class (parent or superclass).

Types of Inheritance in Python:

- **Single Inheritance**: A child class inherits from one parent class.
- **Multiple Inheritance**: A child class inherits from more than one parent class.
- **Multilevel Inheritance**: A class inherits from a child class which in turn inherits from a parent class.
- **Hierarchical Inheritance**: Multiple child classes inherit from one parent class.
- **Hybrid Inheritance**: A combination of two or more types of inheritance.

Key Concepts:

- `super()` function: Used to call the constructor, methods, and access data members from the parent class.

Example:

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

    def greet(self):
        print(f"Hello, my name is {self.name}.")

class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)  # Calling constructor of the parent class
        self.student_id = student_id

    def show_id(self):
        print(f"My ID is {self.student_id}.")

    def greet(self):
        super().greet()  # Calling method from the parent class
        print("I am also a student.")

# Usage
s = Student("Alice", "S123")
s.greet()         # Accessing parent method
s.show_id()       # Accessing subclass method
print(s.name)     # Accessing data member from parent class
```

- Overriding methods: A child class can provide its own implementation of methods defined in the parent class.

Example:

```
class Vehicle:
    def start_engine(self):
        print("Starting the engine...")

class Car(Vehicle):
    def start_engine(self):
        print("Car engine starts with a key.")

# Usage
v = Vehicle()
v.start_engine()   # Output: Starting the engine...

c = Car()
c.start_engine()   # Output: Car engine starts with a key.
```

This demonstrates method overriding, where the `Car` class provides its own implementation of the `start_engine` method defined in the `Vehicle` class.

Example 1: Single Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Usage
d = Dog()
d.speak()
d.bark()
```

Example 2: Multiple Inheritance

```
class Father:
    def skills(self):
        print("Gardening, Programming")

class Mother:
    def skills(self):
```

```
        print("Cooking, Art")

class Child(Father, Mother):
    def skills(self):
        print("Child's own skills:")
        super().skills()  # This will follow MRO (Method Resolution Order)
        print("Sports")

# Usage
c = Child()
c.skills()
```

Explanation: In the example above, Python uses the Method Resolution Order (MRO) to decide which `skills()` method to call first when `super().skills()` is used. Since `Father` appears before `Mother` in the inheritance list of `Child`, `Father.skills()` is called.

MRO of Child:

```
print(Child.__mro__)
# Output: (<class '__main__.Child'>, <class '__main__.Father'>, <class
'__main__.Mother'>, <class 'object'>)
```

UML Example: Multiple Inheritance

```
  Father      Mother
     \          /
       \      /
        Child
```

Example: Multilevel Inheritance

```
class Grandparent:
    def house(self):
        print("Grandparent's house")

class Parent(Grandparent):
    def car(self):
        print("Parent's car")

class Child(Parent):
    def bike(self):
        print("Child's bike")

# Usage
c = Child()
c.house()
```

```
c.car()
c.bike()
```

UML: Multilevel Inheritance

```
Grandparent
    |
  Parent
    |
  Child
```

Example: Hierarchical Inheritance

```
class Animal:
    def sound(self):
        print("Some generic animal sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

class Cat(Animal):
    def sound(self):
        print("Meow")

# Usage
animals = [Dog(), Cat()]
for animal in animals:
    animal.sound()
```

UML: Hierarchical Inheritance

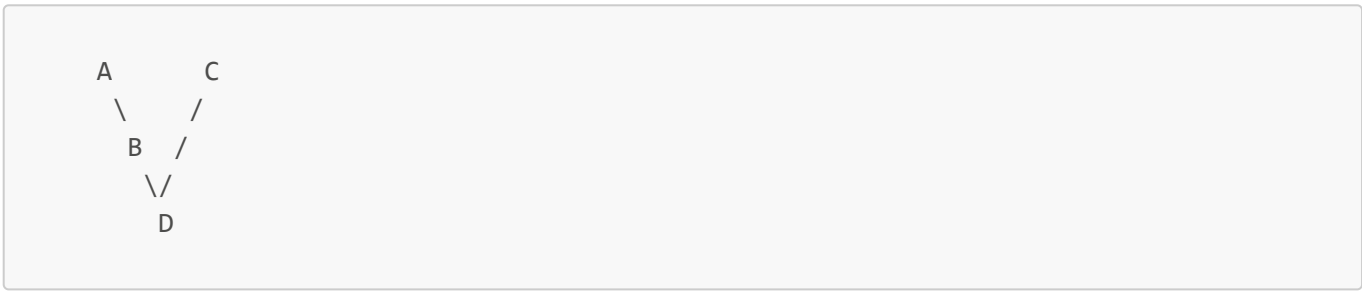
```
     Animal
     /    \
   Dog    Cat
```

Example: Hybrid Inheritance

```
class A:
    def method(self):
        print("Method from A")

class B(A):
    def method(self):
        print("Method from B")
```

```python
class C:
    def method(self):
        print("Method from C")

class D(B, C):
    def method(self):
        super().method()  # Calls B.method() due to MRO
        print("Method from D")

# Usage
d = D()
d.method()
```

UML: Hybrid Inheritance



---

2. Polymorphism

Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It mainly refers to the ability to call the same method on different objects and have each of them respond in their own way.

Types of Polymorphism:

- **Compile-time (Method Overloading)**: Not supported in Python directly, can be mimicked.
- **Runtime (Method Overriding)**: Supported through inheritance.

Polymorphism in Other Languages: Java Example

To understand how polymorphism is implemented in other programming languages, consider this Java example:

```java
// Base class Person
class Person {

    // Method that displays the
    // role of a person
    void role() {
        System.out.println("I am a person.");
    }
}
```

```java
// Derived class Father that
// overrides the role method
class Father extends Person {

    // Overridden method to show
    // the role of a father
    @Override
    void role() {
        System.out.println("I am a father.");
    }
}

public class Main {
    public static void main(String[] args) {

        // Creating a reference of type Person
        // but initializing it with Father class object
        Person p = new Father();

        // Calling the role method. It calls the
        // overridden version in Father class
        p.role();  // Output: I am a father.
    }
}
```

Comparison between Python and Java Polymorphism:

Feature	Python	Java
Method Overloading	Not natively supported	Supported using method signatures
Method Overriding	Supported via inheritance	Supported via inheritance
Type Declaration	Dynamic (no type declaration needed)	Static (type declaration required)
Super Keyword	`super()`	`super` keyword

This comparison highlights that while Python supports polymorphism, it does so with more flexibility due to its dynamic typing, whereas Java enforces more structure through static typing and method signatures.

Example 1: Runtime Polymorphism

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
```

```
        print("Cat meows")

# Usage
for animal in [Dog(), Cat()]:
    animal.speak()
```

Example 2: Polymorphism with Functions and Classes

```
def add(x, y, z=0):
    return x + y + z

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

Lecture 2: Exercises

Exercise Set 1: Inheritance

1. **Exercise 1**: Class Hierarchy with `Person`, `Student`, and `Teacher`

Specification:

- Base Class: `Person`
  - Attributes: `name`, `age`
  - Methods: `introduce()`
- Subclass: `Student`
  - Additional Attributes: `student_id`, `major`
  - Additional Methods: `study()`
- Subclass: `Teacher`
  - Additional Attributes: `employee_id`, `subject`
  - Additional Methods: `teach()`

Diagram:

```
        Person
         /   \
    Student   Teacher
```

2. Extend the above example to use `super()` and override the `introduce()` method in `Student`. The overridden method should call the base class version of `introduce()` using `super()`, then print an additional message specific to the student.

3. Add a new class `Staff` and demonstrate multiple inheritance by combining `Person` and `Staff` into a new subclass `Teacher`.

Specification:

- Class: `Staff`
  - Attributes: `department`, `salary`
  - Methods: `display_staff_info()`
- Class: `Teacher` (inherits from both `Person` and `Staff`)
  - Attributes: `employee_id`, `subject` (in addition to inherited ones)
  - Methods: `teach()` and override `introduce()`

UML Diagram:

```
Person    Staff
   \       /
   Teacher
```

Exercise Set 2: Polymorphism

Specification of Classes:

- **Base Class**: `Shape`
  - **Attributes:** None (acts as an abstract base class)
  - **Methods:** `area()` - meant to be overridden in subclasses
- **Subclass**: `Circle`
  - **Attributes:** `radius`
  - **Methods:** Overrides `area()` to return the area using π * r^2
- **Subclass**: `Square`
  - **Attributes:** `side`
  - **Methods:** Overrides `area()` to return the area using side^2
- **Subclass**: `Rectangle`
  - **Attributes:** `length`, `width`
  - **Methods:** Overrides `area()` to return the area using length * width

These classes demonstrate polymorphism through overriding the `area()` method in a way that reflects the behavior of each geometric shape.

1. Create a base class `Shape` with a method `area()`. Implement it in `Circle`, `Square`, and `Rectangle`.
2. Implement a function `describe_shape(shape)` that accepts any shape and prints its area and type.
3. Using the `Shape` classes, create a polymorphic list and iterate over it.

Shape Hierarchy:

```
        Shape
       /  |   \
Circle Square Rectangle
```

Assignment: Vending Machine Simulation Using Inheritance and Polymorphism

Assignment Instructions: Design and implement a Python program simulating a vending machine using inheritance and polymorphism. Define a base class `Product` and derive multiple product types from it, each overriding relevant methods.

Class Design:

- Base Class: **`Product`**
  - Attributes: `name`, `price`
  - Method: `display_info()` – displays basic product details.
- **Subclasses:**
  - `Drink`: adds attribute `volume`, overrides `display_info()`
  - `Snack`: adds attribute `calories`, overrides `display_info()`
  - `Candy`: adds attribute `flavor`, overrides `display_info()`

UML Diagram:

```
       Product
      /   |    \
Drink Snack Candy
```

Runtime Behavior Example:

Extended Feature – Loading Products from a File: To simulate a realistic vending machine, extend the program to load products from a file.

Example File Structure (`products.txt`):

```
Drink,Cola,1.50,500
Snack,Chips,2.00,250
Candy,Gummy Bears,1.20,Strawberry
Drink,Water,1.00,600
Snack,Cookies,1.75,300
```

Each line contains: `Type,Name,Price,Attribute`

- For `Drink`: the last value is volume in ml
- For `Snack`: the last value is calories
- For `Candy`: the last value is flavor

Student Task:

- Parse this file line by line
- Create the correct object (`Drink`, `Snack`, or `Candy`) based on the type
- Store all objects in a list
- Show a menu based on the loaded products

Expected Menu Output:

```
Welcome to the Python Vending Machine!

Please select what you want:
1. Drink - Cola
2. Snack - Chips
3. Candy - Gummy Bears
4. Drink - Water
5. Snack - Cookies

> 1

Product Information:
Product: Cola, Price: $1.50
Volume: 500ml
```

Hint for Students: Use `open()`, `readlines()` or `csv.reader` to process the file. Implement a simple factory method or conditional logic to instantiate the correct subclass.

```
Welcome to the Python Vending Machine!

Please select what you want:
1. Drink
2. Candy
3. Snack

> 1

Product Information:
Product: Cola, Price: $1.50
Volume: 500ml
```

Note: You must:

- Use inheritance and polymorphism.
- Use `super()` to reuse the base class methods.

- Override `display_info()` in each subclass.
- Present the user with a menu and respond accordingly.