Intermediate Python Programming – Lesson 4

Facilitated by Kent State University
Topic: Advanced OOP: Inheritance and Polymorphism
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Understand and implement inheritance in Python.
- Differentiate between inheritance and composition and when to use each.
- Use method overriding and the `super()` function.
- Implement polymorphism and customize class behaviors with dunder (magic) methods.

Lesson 4: Advanced OOP – Inheritance and Polymorphism

I. Introduction to Inheritance (10 minutes)

Inheritance is one of the pillars of object-oriented programming. It allows a new class (child) to inherit the attributes and methods of an existing class (parent).

- Promotes code reuse and modular design.
- Helps in maintaining consistency across related classes.
- Supports hierarchy modeling in code.

Syntax of inheritance:

```
class Parent:
    pass

class Child(Parent):
    pass
```

This declares `Child` as a subclass of `Parent`, inheriting all its methods and attributes unless overridden.

Multiple-Choice Question

What is the main benefit of inheritance?

A. It prevents the need for functions in Python
B. It allows for code reuse and hierarchy in class design
C. It removes the need for instance attributes
D. It makes debugging harder

Answer: B. Inheritance allows for code reuse and structured class design.

Short Answer Question

How does inheritance improve code organization?

Expected Answer: Inheritance helps reduce redundancy by allowing child classes to reuse code from parent classes.

II. Implementing Inheritance (15 minutes)

To implement inheritance in Python, define a base (parent) class and then a child class that extends it. You can override methods and use the `super()` function to call the parent's method.

Example:

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

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

class Employee(Person):
    def __init__(self, name, age, job_title):
        super().__init__(name, age)
        self.job_title = job_title

    def introduce(self):
        return f"My name is {self.name}, I am {self.age}, and I work as a {self.job_title}."

# Test
e = Employee("Alice", 30, "Software Engineer")
print(e.introduce())
```

Expected Output:

```
My name is Alice, I am 30, and I work as a Software Engineer.
```

Exercise 1

Create a `Person` class with `name` and `age`. Then, create an `Employee` class that inherits from `Person` and adds a `job_title`.

Multiple-Choice Question

What does the `super()` function do?

A. Calls a method from the grandparent class
B. Calls a method from the parent class
C. Calls a method from the child class
D. Calls a method from a different module

Answer: B. `super()` calls a method from the parent class.

Short Answer Question

Why would you use `super().__init__()` in a child class?

Expected Answer: To call the parent class’s constructor and initialize inherited attributes.

III. Inheritance vs. Composition (10 minutes)

While inheritance models an "is-a" relationship, composition models a "has-a" relationship. Use inheritance when a subclass truly is a type of its superclass. Use composition when you want to build complex objects by combining simpler ones.

- **Inheritance example:** Dog is an Animal.
- **Composition example:** Car has an Engine.

Example:

```python
class Address:
    def __init__(self, street, city):
        self.street = street
        self.city = city

class Employee(Person):
    def __init__(self, name, age, job_title, street, city):
        super().__init__(name, age)
        self.job_title = job_title
        self.address = Address(street, city)

    def introduce(self):
        return f"My name is {self.name}, I work as a {self.job_title}, and I live in {self.address.city}."

# Test
e = Employee("Alice", 30, "Software Engineer", "123 Main St", "New York")
print(e.introduce())
```

Expected Output:

```
My name is Alice, I work as a Software Engineer, and I live in New York.
```

Exercise 2

Modify the previous `Employee` class to include an `Address` class instead of adding an address attribute directly.

Multiple-Choice Question

Which scenario is better suited for composition instead of inheritance?

A. An ElectricCar that is a type of Car
B. A Book that has an Author
C. A Student that is a type of Person
D. A Dog that is a type of Animal

Answer: B. Book has an Author, which is a composition relationship.

IV. Polymorphism and Method Overriding (15 minutes)

Polymorphism allows objects of different types to be treated through a common interface. In practice, this often means defining the same method name in multiple classes, each with different behavior.

Method overriding is one way polymorphism is implemented: a child class redefines a method inherited from its parent.

You can also use **dunder methods** (also known as magic methods) like `__str__`, `__len__`, and `__add__` to customize how your objects behave with built-in functions and operators.

Example:

```python
class Animal:
    def speak(self):
        return "This animal makes a sound."

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Test
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.speak())
```

Expected Output:

```
Woof!
Meow!
This animal makes a sound.
```

Exercise 3

Create an `Animal` class with a method `speak()`, and create two subclasses (`Dog`, `Cat`) that override the method.

Multiple-Choice Question

What is method overriding?

A. Creating a new method with the same name in a child class
B. Deleting a method in a class
C. Writing a method that is never used
D. Making a method private

Answer: A. Overriding means redefining a method in a child class.

Short Answer Question

How does method overriding help in polymorphism?

Expected Answer: It allows different child classes to have customized behaviors for the same method name, making code more flexible.

V. Recap and Q&A (5 minutes)

In this lesson, we:

- Introduced class inheritance and its benefits
- Used `super()` to extend and reuse parent functionality
- Distinguished between inheritance and composition
- Demonstrated polymorphism using method overriding
- Explained when and how to use dunder methods to customize behavior

Final Exercise

Create a `Shape` class with a method `calculate_area()`. Then, create `Circle` and `Rectangle` classes that override the method to compute the correct area based on their dimensions.
