In Partial Fulfillment of the Requirements for the

CS 223 - Object-Oriented Programming

"Four Principles of Object-Oriented Programming"

Presented to:

Dr. Unife O. Cagas
Professor V

Presented by:

Jeschelle H. Bonita
BSCS 2A2 Student

" Different Animals and Different Sounds"

Project Title

Project Description

Object-oriented programming (OOP) is a paradigm centered around objects that represent real-world entities and their interactions. The key concepts in OOP include inheritance, encapsulation, polymorphism, and abstraction, which together create a robust structure for building flexible and scalable applications. This code snippet demonstrates these concepts through a simple example involving animals and their distinct sounds. In this code, we define a base class, Animal, which represents a generic animal with a common characteristic: a name. The class has an abstract method, speak(), which each subclass must implement to define their unique sound. Subclasses Dog, Cat, and Duck inherit from Animal, each providing its specific implementation of the speak() method, illustrating the concept of polymorphism. Encapsulation is demonstrated through the _name attribute, indicating that it is intended for internal use. The class provides a public method to retrieve the name, maintaining control over how the attribute is accessed and modified. The snippet also includes a function, animal_sound(), that accepts an Animal object and calls its speak() method. This function showcases polymorphism because it can work with any object derived from Animal, allowing for flexibility in handling different types of animals without specific checks or conditions.

Objectives:

1. The code aims to showcase the fundamental principles of object-oriented programming (OOP), including inheritance, encapsulation, polymorphism, and abstraction.

2. By creating a base class (Animal) and multiple subclasses (Dog, Cat, Duck), the code demonstrates how class hierarchies can be used to model relationships among different entities.

3. The code demonstrates polymorphism through the speak() method, which is overridden by subclasses to produce unique behavior. The animal_sound() function illustrates how polymorphism allows a single function to operate on objects of varying types.

4. By using encapsulated attributes and providing controlled access through methods like get_name(), the code illustrates the importance of encapsulation in managing access to internal object data.

5. The use of a base class and inheritance promotes code reusability, while abstraction and encapsulation contribute to easier maintenance and extension of the codebase.

6. The code snippet provides a straightforward example of OOP concepts, using a common theme of animals and their distinct sounds, making it accessible for learners and those new to OOP.

Importance and Contribution of the Project

The importance of this project lies in demonstrating how these OOP concepts can be combined to create a flexible, maintainable, and extensible codebase. By using encapsulation, inheritance, abstract classes, and polymorphism, the project showcases best practices for OOP design and implementation. The contribution of this project is to provide a clear and concise example of how to apply OOP concepts in a real-world scenario. This can help developers improve their understanding of OOP principles and apply them effectively in their own projects. Additionally, the code can be easily extended to support more animal classes, further demonstrating the benefits of using OOP concepts.

Four Principles of Object-Oriented Programming with code

Classes:

Classes are blueprints for creating objects. They define attributes and methods that describe the behavior of the objects they represent. This code defines several classes, Animal: The base class for all animals, providing a common structure with an encapsulated attribute (_name) and an abstract method (speak()). Dog, Cat, and Duck: These classes inherit from Animal, defining specific behaviors for each animal type.

```
class Animal:

    def __init__(self, name):
        self._name = name
```

Objects:

Objects are instances of classes, containing data and behavior defined by their class. In this code, dog, cat, and duck are instances of Dog, Cat, and Duck, respectively. They each represent individual animals with unique names.

```
dog = Dog("Budoy")
cat = Cat("Josep")
duck = Duck("Danny")
```

Inheritance:

Inheritance allows a class to derive properties and behavior from a parent class. In this code, Dog, Cat, and Duck inherit from the base class Animal, gaining access to its constructor and other methods. This allows code reuse and establishes a hierarchical relationship between classes.

```
class Dog(Animal):
    def speak(self):
        return f"{self.get_name()} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.get_name()} says Meow!"

class Duck(Animal):
    def speak(self):
        return f"{self.get_name()} says Quack!"
```

Encapsulation:

Encapsulation is the practice of restricting direct access to certain attributes and providing controlled access through methods. In this code, the _name attribute in Animal is encapsulated, indicating that it's intended for internal use. Public methods, like get_name(), provide controlled access to this attribute.

```
def get_name(self):
    return self._name
```

Polymorphism

Polymorphism allows objects of different subclasses to be treated uniformly while exhibiting unique behaviors. In this code, the speak() method is overridden in Dog, Cat, and Duck, providing different outputs for each subclass. The animal_sound() function demonstrates polymorphism by calling speak() on an Animal object, allowing it to work with any subclass without additional checks.

```
def animal_sound(animal):
    return animal.speak()
```

Abstraction:

Abstraction involves exposing only the necessary information while hiding implementation details. In this code, the Animal class serves as an abstract base class with an abstract speak() method, which must be implemented by its subclasses. This abstraction allows flexibility in defining new animal types while maintaining a consistent interface.

```
def speak(self):
    raise NotImplementedError("Subclasses must implement this method")
```

Hardware and Software Used

Hardware:

- Laptop
- Cellphone

Software:

- Visual Studio Code
- Online GDB

Output:

```
PS C:\Users\Admin\Bonita> & C:/ProgramData/anaconda3/python.exe c:/Users/Admin/Bonita/OOP
Budoy says Woof!
Josep says Meow!
Danny says Quack!
```

- Budoy says Woof!: This is the output of the dog object's speak() method, which is overridden in the Dog class to return "Woof!".
- Josep says Meow!: This is the output of the cat object's speak() method, which is overridden in the Cat class to return "Meow!".
- Danny says Quack!: This is the output of the duck object's speak() method, which is overridden in the Duck class to return "Quack!".

Each object's speak() method is called through the animal_sound() function, which demonstrates polymorphism. Despite each object being of a different subclass (Dog, Cat, Duck), they are all treated as instances of the common superclass Animal, showcasing polymorphic behavior.

Code Documentation:

```
# Base class representing a generic animal with encapsulated attributes
class Animal:
    # Constructor to initialize the name attribute
    def __init__(self, name):
        self._name = name  # Encapsulation: '_name' is internal to this class

    # Method to retrieve the name (encapsulation)
    def get_name(self):
        return self._name

    # Abstract method for subclasses to implement their own behavior
    def speak(self):
```

```
 # Abstraction
        raise NotImplementedError("Subclasses must implement this method")

 Derived class representing a dog, inheriting from Animal
class Dog(Animal):
    # Override the abstract 'speak' method to return a specific sound
    def speak(self):

# Polymorphism
 return f"{self.get_name()} says Woof!"

# Derived class representing a cat, inheriting from Animal
class Cat(Animal):
    # Override the abstract 'speak' method with a different implementation
    def speak(self):
        # Polymorphism
        return f"{self.get_name()} says Meow!"

# Derived class representing a duck, inheriting from Animal
class Duck(Animal):
    # Override the abstract 'speak' method with yet another implementation
    def speak(self):
        # Polymorphism
        return f"{self.get_name()} says Quack!"

# Function to demonstrate polymorphism by accepting an Animal instance
def animal_sound(animal):
   # Works with any class derived from Animal
    return animal.speak()
```

```
# Create instances of Dog, Cat, and Duck
dog = Dog("Budoy")
cat = Cat("Josep")
duck = Duck("Danny")

# Using polymorphism to get the sound of each animal
print(animal_sound(dog))
print(animal_sound(cat))
print(animal_sound(duck))
```

User Guide:

To use this code:

- Define classes that inherit from the Animal class and implement the speak() method with custom behavior for each animal type.
- Create instances of these classes with specific names.
- Call the animal_sound() function with instances of these classes to hear the sound each animal makes.

References:

Website: ChatGPT. URL: https://chatgpt.com/?oai-dm=1

Website: w3schools. "Python Tutorial." URL: https://www.w3schools.com/python/default.asp