Object Oriented Programming (Using Python)

UNIT- II

Polymorphism and Inheritance: cont'd.

- Invocation of constructors and destructors in inheritance
- Aggregation and composition
- Classification hierarchies
- Unit testing
- Exceptions

Prof. R. MADANA MOHANA
Professor, Artificial Intelligence & Data Science
http://rmadanamohana.com/

Invocation of constructors and destructors

- In Python, constructors and destructors are special methods that are automatically called when an object is created and destroyed, respectively. The constructor method is called __init__() and the destructor method is called __del__().
- Here's how they are invoked in Python:

Constructor Invocation:

When an object is created, Python automatically calls the constructor method __init__() to initialize the object's attributes.

Example:

```
class MyClass:
    def __init__(self):
        print("Constructor called")
obj = MyClass()  # Output: Constructor called
```

In the above example, the constructor of the **MyClass** is invoked automatically when the object **obj** is created using the class name **MyClass()**.

Destructor Invocation:

When an object is no longer needed or when the program terminates, Python automatically calls the destructor method __del__() to perform any cleanup operations, such as releasing system resources or closing files.

Example:

```
class MyClass:
    def __del__(self):
        print("Destructor called")
obj = MyClass()  # Create object
del obj  # Delete object, Output: Destructor called
```

In the above example, the destructor of the **MyClass** is invoked automatically when the object **obj** is deleted using the **del** keyword.

Invocation of constructors and destructors

It's important to note that the destructor is not called immediately when an object is no longer needed. Instead, Python schedules the destructor to be called at a later time when it determines that the object is no longer being used.

The constructor and destructor methods in Python are automatically invoked when an object is created and destroyed, respectively.

Aggregation and composition

- Aggregation and composition are two forms of object composition in object-oriented programming that are closely related to inheritance.
- **Aggregation** is a form of object composition where one object contains references to one or more other objects, which are typically created independently of the parent object. In this way, the child objects exist independently of the parent object and can exist even after the parent object is destroyed.
- Aggregation is commonly used to represent a "has-a" relationship between objects.

Aggregation and composition

- **Composition**, on the other hand, is a stricter form of object composition where one object is composed of one or more other objects, and the child objects are created and destroyed along with the parent object. In this way, the child objects cannot exist independently of the parent object.
- **Composition** is commonly used to represent a "part-of" relationship between objects.

Aggregation and composition

Aggregation Example implemented in Python using inheritance:

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

class Company:
    def __init__(self, name, employees):
        self.name = name
        self.employees = employees

person1 = Person("Alice")
person2 = Person("Bob")
company = Company("Acme Inc.", [person1, person2])
```

Aggregation and composition

Aggregation Example implemented in Python using inheritance:

In this example, the **Company** class contains a reference to a list of **Person** objects, which are created independently of the **Company** object. This is an example of aggregation.

Aggregation and composition

Composition Example implemented in Python using inheritance:

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()

car = Car()
car.start()
```

Aggregation and composition

Composition Example implemented in Python using inheritance:

In this example, the **Car** class is composed of an **Engine** object, which is created along with the **Car** object. The **start** method of the **Car** class calls the **start** method of the **Engine** object. This is an example of composition.

Aggregation and composition

Note that while aggregation and composition are related to inheritance, they are not the same thing.

Inheritance is a mechanism for creating new classes by deriving them from existing classes, while aggregation and composition are mechanisms for composing objects by combining multiple objects into a single object.

Aggregation: Example full program

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

class Department:
    def __init__(self, name, employees):
        self.name = name
        self.employees = employees

    def add_employee(self, employee):
        self.employees.append(employee)

    def remove_employee(self, employee):
        self.employees.remove(employee)

class Manager(Person):
    def __init__(self, name, age, department):
        super().__init__(name, age)
        self.department = department

    def add_employee(self, employee):
        self.department.add_employee(employee)

    def remove_employee(self, employee):
        self.department.remove_employee(employee)

employee1 = Person("Ram", 24)
employee2 = Person("Robert", 28)
employee3 = Person("Turing", 32)

department = Department("Sales", [employee1, employee2])

manager = Manager("Alice", 40, department)
manager.add_employee(employee3)

for employee in department.employees:
    print(employee.name)
```

Output:

Ram
Robert
Turing

In this example, we have three classes: **Person**, **Department**, and **Manager**. **Person** is a base class that defines the **name** and **age** attributes of a person. **Department** is a class that defines a list of employees and provides methods to add and remove employees from the list. **Manager** is a subclass of **Person** that adds a **Department** attribute and methods to add and remove employees from the department's employee list.

We create three **Person** objects and a **Department** object with two employees. We then create a **Manager** object with the **Department** object as an argument. We use the add_employee() method of the **Manager** object to add a new employee to the **Department** object. Finally, we print out the names of all the employees in the **Department** object, including the one added by the **Manager**.

Composition: Example full program

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

class Car:
    def __init__(self, make, model, year, engine):
        self.make = make
        self.model = model
        self.year = year
        self.engine = engine

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

    def stop(self):
        print(f"{self.make} {self.model} stopped")

engine = Engine("Gasoline")
car = Car("TATA", "Nexon", 2023, engine)

car.start()
car.stop()

# Output:
# TATA Nexon started
# TATA Nexon stopped
```

In this example, we have two classes: **Engine** and **Car**. **Engine** is a class that defines the fuel_type attribute of an engine. **Car** is a class that defines the make, model, year, and engine attributes of a car.

We create an **Engine** object with a fuel type of "Gasoline" and a **Car** object with the make "TATA", model “Nexon", year 2023, and the **Engine** object as an argument. We then use the start() and stop() methods of the **Car** object to simulate starting and stopping the car.

Classification Hierarchies

- Classification hierarchies in inheritance refer to a way of organizing classes into a tree-like structure based on their relationships.
- In this hierarchy, a class at a higher level is a more general concept, while a class at a lower level is a more specific concept.
- Each class in the hierarchy inherits properties and methods from its parent class, and it may also define new properties and methods that are specific to it.

Classification Hierarchies:Example-1. Animal Classification Hierarchy

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

    def eat(self):
        print("The {} is eating.".format(self.name))

class Mammal(Animal):
    def __init__(self, name, habitat, num_legs):
        super().__init__(name, habitat)
        self.num_legs = num_legs

    def give_birth(self):
        print("The {} is giving birth to live young.".format(self.name))

class Reptile(Animal):
    def __init__(self, name, habitat, num_legs):
        super().__init__(name, habitat)
        self.num_legs = num_legs

    def lay_eggs(self):
        print("The {} is laying eggs.".format(self.name))

class Dog(Mammal):
    def __init__(self, name, habitat):
        super().__init__(name, habitat, num_legs=4)

    def bark(self):
        print("The {} is barking.".format(self.name))

class Lizard(Reptile):
    def __init__(self, name, habitat):
        super().__init__(name, habitat, num_legs=4)

    def change_color(self):
        print("The {} is changing color.".format(self.name))
```

```
dog = Dog("Rufus", "land")
dog.eat()
dog.give_birth()
dog.bark()

lizard = Lizard("Izzy", "desert")
lizard.eat()
lizard.lay_eggs()
lizard.change_color()
```

```
The Rufus is eating.
The Rufus is giving birth to live young.
The Rufus is barking.
The Izzy is eating.
The Izzy is laying eggs.
The Izzy is changing color.
```

Classification Hierarchies: Example-1. Animal Classification Hierarchy

In this example, we have a classification hierarchy for animals. The **Animal** class is the most general class, and it has two child classes: **Mammal** and **Reptile**. Each of these child classes has their own child classes, such as **Dog** and **Lizard**. The **Dog** and **Lizard** classes inherit properties and methods from their parent classes, but they also define new properties and methods specific to them.

Unit Testing

Unit testing is a process of testing individual units or components of a software application.

Inheritance plays a vital role in writing maintainable and testable code, and unit testing can help ensure that code is functioning as intended.

Unit Testing

Example: Unit Testing with Inheritance

unittesting.py - C:/Users/rmmna/AppData/Local/Programs/Python/Python310/unittesting.py (3.10.0)

File Edit Format Run Options Window Help

```python
import unittest

class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def speak(self):
        return f"{self.name} says something"

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name, "Dog")

    def speak(self):
        return f"{self.name} barks"

class Cat(Animal):
    def __init__(self, name):
        super().__init__(name, "Cat")

    def speak(self):
        return f"{self.name} meows"
```

Unit Testing

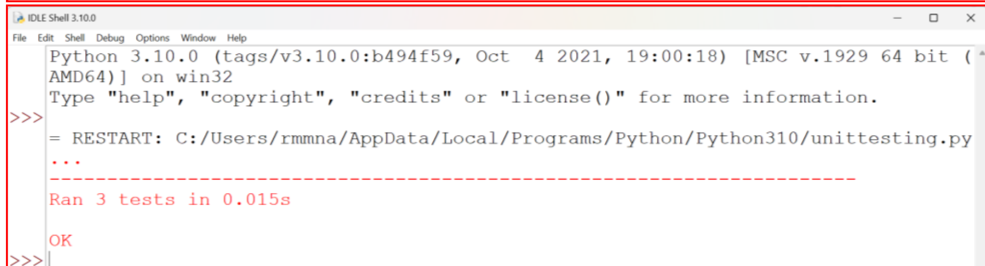
Example: Unit Testing with Inheritance

```
class AnimalTests(unittest.TestCase):
    def test_animal_speak(self):
        animal = Animal("Bob", "Raccoon")
        self.assertEqual(animal.speak(), "Bob says something")

    def test_dog_speak(self):
        dog = Dog("Fido")
        self.assertEqual(dog.speak(), "Fido barks")

    def test_cat_speak(self):
        cat = Cat("Whiskers")
        self.assertEqual(cat.speak(), "Whiskers meows")

if __name__ == "__main__":
    unittest.main()
```



Unit Testing

assertEqual() in Python is a unittest library function that is used in unit testing to check the equality of two values.

This function will take three parameters as input and return a boolean value depending upon the assert condition.

If both input values are equal **assertEqual()** will return true else return false.

Unit Testing

Syntax: assertEqual(firstValue, secondValue, message)

Parameters: assertEqual() accept three parameter which are listed below with explanation:

firstValue variable of any type which is used in the comparison by function

secondValue: variable of any type which is used in the comparison by function

message: a string sentence as a message which got displayed when the test case got failed.

Unit Testing

In this example, we define a base class **Animal** and two subclasses **Dog** and **Cat**.

We also define a unit test class **AnimalTests** that tests the **speak()** method of each class.

The **test_animal_speak()** method tests the **Animal** class, while **test_dog_speak()** and **test_cat_speak()** test the **Dog** and **Cat** classes, respectively.

We use the **assertEqual()** method to check that the output of **speak()** is equal to the expected output.