Object Oriented Programming (Using Python)

UNIT- II

Polymorphism and Inheritance:

- Type conversions
- Base classes and derived classes
- Various types of classes (including Metaclass/ abstract classes)
- Invocation of constructors and destructors in inheritance
- Aggregation and composition
- Classification hierarchies
- Unit testing and Exceptions

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

Type conversions in python classes

- In Python, you can convert one data type to another using type conversion functions.
- Here are some commonly used type conversion functions:

1. **int()** - converts a string or a float to an integer.
2. **float()** - converts a string or an integer to a float.
3. **str()** - converts an integer, float or any other data type to a string.
4. **bool()** - converts any data type to a boolean.

Type conversions in python classes

- When it comes to Python classes, you can define your own conversion functions by implementing the special methods **__int__()**, **__float__()**, **__str__()**, and **__bool__()** in your class.

Type conversions in python classes

Here is an example of a Python class that defines these conversion methods:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def __int__(self):
        return int(self.radius)
    def __float__(self):
        return float(self.radius)
    def __str__(self):
        return f"Circle with radius {self.radius}"
    def __bool__(self):
        return self.radius > 0
```

Type conversions in python classes

Here is an example of a Python class that defines these conversion methods:

```
# create an object of the Circle class
circle = Circle(5)
# calling the conversion methods
print(int(circle))   #output: 5
print(float(circle)) #output: 5.0
print(str(circle))   #output: Circle with radius 5
print(bool(circle))   # output: True
```

Type conversions in python classes

Here is an example of a Python class that defines these conversion methods:

OUTPUT:

5

5.0

Circle with radius 5

True

Type conversions in python classes

Here is an example of a Python class that defines these conversion methods:

In this example, we have defined a **`Circle`** class that has a **`radius`** attribute.
We have implemented the **`__int__()`**, **`__float__()`**, **`__str__()`**, and **`__bool__()`** methods to convert the circle object to an integer, float, string, and Boolean.

Base classes and derived classes

- Inheritance is one of the fundamental concepts of object-oriented programming in Python.
- In inheritance, a **derived class** is created by inheriting the attributes and methods of a **base class**.
- The **derived class** can then add or modify its own attributes and methods.
- The **base class** is also known as the **parent class**, while the **derived class** is also known as the **child class**.

Base classes and derived classes

Here's an example of how to create a derived class in Python:

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def eat(self):
        print(f"{self.name} is eating.")
class Dog(Animal):
    def bark(self):
        print("Woof!")
```

Base classes and derived classes

Here's an example of how to create a derived class in Python:

```
# create objects of the classes
animal = Animal("Simba", 3)
dog = Dog("Buddy", 5)
# calling the methods
animal.eat()     # output: Simba is eating.
dog.eat()        # output: Buddy is eating.
dog.bark()       # output: Woof!
```

Base classes and derived classes

Here's an example of how to create a derived class in Python:

OUTPUT:

```
Simba is eating.
Buddy is eating.
Woof!
```

Base classes and derived classes

Here's an example of how to create a derived class in Python:

In this example, we have defined an **Animal** class with an **__init__()** method and an **eat()** method.

We then define a **Dog** class that inherits from the **Animal** class using the syntax

```
class Dog(Animal):
```

The **Dog** class also has a **bark()** method.

Base classes and derived classes

Here's an example of how to create a derived class in Python:

When we create objects of these classes, we can see that the **Dog** object inherits the **name** and **age** attributes as well as the **eat()** method from the **Animal** class. Additionally, the **Dog** object has its own **bark()** method.

By using inheritance, we can reuse code and avoid duplication of code. We can also create hierarchies of classes, where a derived class can be further inherited by another derived class.

Various types of classes

- In Python, there are several types of classes that you can create depending on your needs.
- Some of the common types of classes are:
  1. Regular Classes
  2. Abstract Classes
  3. Singleton Classes
  4. Metaclasses
  5. Inner Classes

Various types of classes

1. Regular Classes: Regular classes are the most common type of class in Python. They define a blueprint for creating objects with a set of attributes and methods.

Example:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self):
        print(f"{self.name} says Woof!")
d1=Dog("DOG")
d1.bark()#Output: DOG says Woof!
```

Various types of classes

2. Abstract Classes: These are classes that cannot be instantiated, but are used as a base class for other classes. They contain abstract methods that need to be implemented in the derived class.

Example:

```
from abc import ABC, abstractmethod # abc - Abstract Base Classes module
class Shape(ABC): # ABC - Abstract Base Classes
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
```

Various types of classes

2. Abstract Classes: Example: cont'd.

```
class Square(Shape):
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side * self.side
    def perimeter(self):
        return 4 * self.side
s=Square(5)
print(s.area())
print(s.perimeter())
```

OUTPUT:

```
25
20
```

Various types of classes

3. Singleton Classes: These are classes that allow only one instance to be created throughout the program. They are useful when you want to restrict the number of instances of a class.

Example:

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            return cls._instance
s=Singleton()
print(s)
s1=Singleton()
print(s1)
```

Various types of classes

3. Singleton Classes: Example cont'd.

'Output:

<__main__.Singleton object at 0x0000029CCD3868C0>
None

The __new__() static method creates a new instance of a class cls and takes in the class (of which the instance was requested) as the first argument.

Syntax

```
class ClassName:
  __new__(cls, *args, **kwargs):
    obj = super().__new__(cls)
    return obj
```

Various types of classes

3. Singleton Classes:

Object creation is typically done by invoking the superclass' __new__() in two ways. The first argument of __new__() is always the class which is passed in as the first parameter automatically. The name cls is not a keyword; it's used to reference the class (the first parameter) by convention. When __new__() returns an instance of cls it automatically invokes the __init__() method of that instance with the arguments passed to it. The first argument passed to __init__() will be the instance itself (this happens automatically) which is by convention referenced by self.

Various types of classes

4. Metaclasses: These are classes that are used to create other classes. They define the behavior and structure of a class.

Example:

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        attrs['x'] = 1
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMeta):
    pass

print(MyClass.x) # Output: 1
```

Various types of classes

5. Inner Classes: These are classes that are defined inside another class. They can be used to group related classes together.

Example:

Various types of classes

5. Inner Classes: cont'd.

```
class OuterClass:
    def __init__(self):
        self.x = 10

    def outer_method(self):
        print("Outer method called")

        class InnerClass:
            def __init__(self):
                self.y = 20
                self.x = 10

            def inner_method(self):
                print("Inner method called")
                print("x = ", self.x)
                print("y = ", self.y)

        inner_obj = InnerClass()
        inner_obj.inner_method()

outer_obj = OuterClass()
outer_obj.outer_method()
```

Various types of classes

5. Inner Classes: cont'd.

```
OUTPUT:
Outer method called
Inner method called
x =  10
y =  20
```

Various types of classes

5. Inner Classes: cont'd.

In the above example, the **OuterClass** contains an inner class I**nnerClass**. When the **outer_method** is called, it creates an instance of the **InnerClass** and calls its **inner_method**. The **inner_method** accesses the attributes **x** and **y** of both the outer and inner classes.

- **Note** that the inner class can only be accessed within the scope of the outer class. It is not visible outside the outer class, and it can't be instantiated directly. The inner class can access all the attributes and methods of the outer class, including private ones.
- Inner classes can be used to organize and encapsulate code within a larger class, and they can access the attributes and methods of the outer class.