Python

Class 7

# Introduction to Python

Azmain Adel

September 1, 2024

# 01.

# Review of Previous Class

# Review Topics

- Classes and objects
- Polymorphism
- Abstraction
- Encapsulation
- Inheritance

# Solution to Problem 1

Write a Python program to create a class representing a Circle. Include methods to calculate its area and perimeter.

```python
import math

Codiumate: Options | Test this class
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

# Solution to Problem 2

Write a Python program to create a class that represents a **shape**. Include methods to calculate its area and perimeter.

Implement subclasses for different shapes like **circle**, **triangle**, and **square**.

```python
import math

Codiumate: Options | Test this class
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

Codiumate: Options | Test this class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

```python
class Triangle(Shape):
    Codiumate: Options | Test this method
    def __init__(self, side1, side2, side3):
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    Codiumate: Options | Test this method
    def area(self):
        # Using Heron's formula
        s = (self.side1 + self.side2 + self.side3) / 2
        return math.sqrt(s * (s - self.side1) * (s - self.side2) * (s - self.side3))

    def perimeter(self):
        return self.side1 + self.side2 + self.side3

Codiumate: Options | Test this class
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

    def perimeter(self):
        return 4 * self.side
```

# 02.

# Attributes

# Class Attributes

- The properties or variables defined inside a class are called as Attributes.
- An attribute provides information about the type of data a class contains.
- There are two types of attributes in Python namely **instance** attribute and **class** attribute.

# Modifying Attributes

```
class Employee:
   # class attribute
   empCount = 0
   def __init__(self, name, age):
      self.__name = name
      self.__age = age
      # modifying class attribute
      Employee.empCount += 1
      print ("Name:", self.__name, ", Age: ", self.__age)
      # accessing class attribute
      print ("Employee Count:", Employee.empCount)

e1 = Employee("Bhavana", 24)
print()
e2 = Employee("Rajesh", 26)
```

# Class Attribute

- They are used to define those properties of a class that should have the **same value for every object** of that class.
- Class attributes can be used to set default values for objects.
- This is also useful in creating singletons. They are objects that are instantiated only once and used in different parts of the code.

# Built-in Attributes

- **__dict__** : Dictionary containing the class's namespace.
- **__doc__** : Class documentation string or none, if undefined.
- **__name__** : Class name.
- **__module__** : Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

# Built-in Attributes

```
class Employee:
   def __init__(self, name="Bhavana", age=24):
      self.name = name
      self.age = age
   def displayEmployee(self):
      print ("Name : ", self.name, ", age: ", self.age)

print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

# Instance Attributes

- An instance attribute in Python is a variable that is **specific to an individual object** of a class.
- It is defined inside the **__init__**() method.

# Instance Attributes

```
class Student:
   def __init__(self, name, grade):
      self.__name = name
      self.__grade = grade
      print ("Name:", self.__name, ", Grade:", self.__grade)

# Creating instances
student1 = Student("Ram", "B")
student2 = Student("Shyam", "C")
```

# 03.

# Static Methods

# Static Methods

- A static method is a type of method that does not require any instance to be called.
- Static methods are used to access static fields of a given class.
- They cannot modify the state of a class since they are bound to the class, not instance.

# Writing Static Methods

- Using **staticmethod**() Function
- Using @**staticmethod** Decorator

```
class Employee:
   empCount = 0
   def __init__(self, name, age):
      self.__name = name
      self.__age = age
      Employee.empCount += 1

   # creating staticmethod
   def showcount():
      print (Employee.empCount)
      return
   counter = staticmethod(showcount)

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.counter()
Employee.counter()
```

```python
class Student:
    stdCount = 0
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        Student.stdCount += 1

    # creating staticmethod
    @staticmethod
    def showcount():
        print (Student.stdCount)

e1 = Student("Bhavana", 24)
e2 = Student("Rajesh", 26)
e3 = Student("John", 27)

print("Number of Students:")
Student.showcount()
```

# 04.

# Interfaces

# Interfaces

- An interface is a software architectural pattern.
- It is similar to a class but its methods just have prototype signature definition without any executable code or implementation body.

# Abstract Method

- The method defined without any executable code is known as abstract method.

# Writing Interfaces

- Methods defined inside an interface must be abstract.
- Creating object of an interface is not allowed.
- A class implementing an interface needs to define all the methods of that interface.
- In case, a class is not implementing all the methods defined inside the interface, the class must be declared abstract.

# Example: Interface

```
class demoInterface:
    def displayMsg(self):
        pass

class newClass(demoInterface):
    def displayMsg(self):
        print ("This is my message")

# creating instance
obj = newClass()

# method call
obj.displayMsg()
```

# 05.

# Access Modifiers

# Access Modifiers

- Access modifiers are used to restrict access to class members (i.e., variables and methods) from outside the class.

# Types of Access Modifiers

1. **Public** members – A class member is said to be public if it can be accessed from anywhere in the program.
2. **Protected** members – They are accessible from within the class as well as by classes derived from that class.
3. **Private** members – They can be accessed from within the class only.

# Example: Access Modifiers

```
class Employee:
    Codiumate: Options | Test this method
    def __init__(self, name, age, salary):
        self.name = name # public variable
        self.__age = age # private variable
        self._salary = salary # protected variable
    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.__age, ", salary: ", self._salary)

e1=Employee("Bhavana", 24, 10000)

print (e1.name)
print (e1._salary)
print (e1.__age)
```

# 06.

# Enums

# Packages

- **Enum** assigns constant values to a set of strings.
- Created from inheriting the Enum built-in class.

# Example: Enums

```
# importing enum
from enum import Enum

class subjects(Enum):
    ENGLISH = 1
    MATHS = 2
    SCIENCE = 3
    SANSKRIT = 4

obj = subjects.MATHS
print (type(obj))
```

# 07.

# Method Overriding

# Method Overriding

- Method overriding refers to defining a method in a subclass with the same name as a method in its superclass.

# Example: Method Overriding

```
# define parent class
class Parent:
   def myMethod(self):
      print ('Calling parent method')

# define child class
class Child(Parent):
   def myMethod(self):
      print ('Calling child method')

# instance of child
c = Child()
# child calls overridden method
c.myMethod()
```

# 08.

# Method Overloading

# Method Overloading

- Method overloading is when a class can have multiple methods with the same name **but different parameters.**

# Example: Method Overloading

```
class example:
   def add(self, a, b):
      x = a+b
      return x
   def add(self, a, b, c):
      x = a+b+c
      return x

obj = example()

print (obj.add(10,20,30))
print (obj.add(10,20))
```

# 09.

# Recap and Q&A

Open floor for questions and clarifications

# Thank you.

azmainadel47@gmail.com
linkedin.com/in/azmainadel
azmainadel.com