lab_13_2

March 11, 2025

Python Programming - 2301CS404

Lab - 13

OM BHUT | 23010101033 | 122

0.1 Continued..

0.1.1 10) Calculate area of a ractangle using object as an argument to a method.

[10]:
```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width


def calculate_area(obj):
    area= obj.length*obj.width
    print(f"Area calculated using object is: {area}")
r1 = Rectangle(5, 10)

calculate_area(r1)
```

```
Area calculated using object is: 50
```

0.1.2 11) Calculate the area of a square.

0.1.3 Include a Constructor, a method to calculate area named area() and a method named output() that prints the output and is invoked by area().

[8]:
```
class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        area_value = self.side ** 2
        self.output(area_value)

    def output(self, area_value):
        print(f"Area of square with side {self.side} is: {area_value}")
```

```
square = Square(4)
square.area()
```

```
Area of square with side 4 is: 16
```

0.1.4 12) Calculate the area of a rectangle.

0.1.5 Include a Constructor, a method to calculate area named area() and a method named output() that prints the output and is invoked by area().

0.1.6 Also define a class method that compares the two sides of reactangle. An object is instantiated only if the two sides are different; otherwise a message should be displayed : THIS IS SQUARE.

[ ]:
```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        area_value = self.length * self.width
        self.output(area_value)
        return area_value

    def output(self, area_value):
        print(f"Area of rectangle with length {self.length} and width {self.
 ↪width} is: {area_value}")

    @classmethod
    def create_rectangle(cls, length, width):
        if length == width:
            print("THIS IS SQUARE.")
            return

        return cls(length, width)


rect1 = Rectangle.create_rectangle(5, 3)
if rect1:
    rect1.area()

rect2 = Rectangle.create_rectangle(4, 4)
```

```
Area of rectangle with length 5 and width 3 is: 15
THIS IS SQUARE.
```

0.1.7 13) Define a class Square having a private attribute "side".

0.1.8 Implement get_side and set_side methods to accees the private attribute from outside of the class.

[13]:
```
class Square:
    def __init__(self, side):
        self.__side = side

    def get_side(self):
        return self.__side

    def set_side(self, side):
        self.__side = side

    def calculate_area(self):
        return self.__side ** 2


sq = Square(5)
print(f"Side of square: {sq.get_side()}")
print(f"Area of square: {sq.calculate_area()}")

sq.set_side(7)
print(f"New side of square: {sq.get_side()}")
print(f"New area of square: {sq.calculate_area()}")
```

```
Side of square: 5
Area of square: 25
New side of square: 7
New area of square: 49
```

0.1.9 14) Create a class Profit that has a method named getProfit that accepts profit from the user.

0.1.10 Create a class Loss that has a method named getLoss that accepts loss from the user.

0.1.11 Create a class BalanceSheet that inherits from both classes Profit and Loss and calculates the balanace. It has two methods getBalance() and printBalance().

[2]:
```
class Profit:
    def __init__(self):
        self.profit = 0

    def getProfit(self):
        self.profit = float(input("Enter profit amount: "))
        return self.profit
```

```
class Loss:
    def __init__(self):
        self.loss = 0

    def getLoss(self):
        self.loss = float(input("Enter loss amount: "))
        return self.loss

class BalanceSheet(Profit, Loss):
    def __init__(self):
        # Profit.__init__(self)
        # Loss.__init__(self)
        super().__init__()
        self.balance = 0

    def getBalance(self):
        self.balance = self.profit - self.loss
        return self.balance

    def printBalance(self):
        print(f"Profit: ${self.profit}")
        print(f"Loss: ${self.loss}")
        print(f"Balance: ${self.balance}")

balance_sheet = BalanceSheet()
balance_sheet.getProfit()
balance_sheet.getLoss()
balance_sheet.getBalance()
balance_sheet.printBalance()
```

```
Profit: $50.0
Loss: $25.0
Balance: $25.0
```

0.1.12 15) WAP to demonstrate all types of inheritance.

[19]:
```
class Parent:
    def __init__(self):
        self.parent_attribute = "This is from parent"

    def parent_method(self):
        print("This is parent method")

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.child_attribute = "This is from child"
```

```
    def child_method(self):
        print("This is child method")

class Father:
    def father_method(self):
        print("This is father method")

class Mother:
    def mother_method(self):
        print("This is mother method")

class Child2(Father, Mother):
    def child_method(self):
        print("This is child2 method")

class Grandparent:
    def grandparent_method(self):
        print("This is grandparent method")

class Parent2(Grandparent):
    def parent_method(self):
        print("This is parent2 method")

class Child3(Parent2):
    def child_method(self):
        print("This is child3 method")

class Parent3:
    def parent_method(self):
        print("This is parent3 method")

class ChildA(Parent3):
    def child_a_method(self):
        print("This is childA method")

class ChildB(Parent3):
    def child_b_method(self):
        print("This is childB method")


class Base:
    def base_method(self):
        print("This is base method")

class Derived1(Base):
    def derived1_method(self):
```

```
        print("This is derived1 method")

class Derived2(Base):
    def derived2_method(self):
        print("This is derived2 method")

class DerivedOfDerived(Derived1, Derived2):
    def derived_of_derived_method(self):
        print("This is derived of derived method")


print("\nSingle Inheritance:")
child = Child()
child.parent_method()
child.child_method()

print("\nMultiple Inheritance:")
child2 = Child2()
child2.father_method()
child2.mother_method()
child2.child_method()

print("\nMultilevel Inheritance:")
child3 = Child3()
child3.grandparent_method()
child3.parent_method()
child3.child_method()

print("\nHierarchical Inheritance:")
childA = ChildA()
childB = ChildB()
childA.parent_method()
childA.child_a_method()
childB.parent_method()
childB.child_b_method()

print("\nHybrid Inheritance:")
derived_of_derived = DerivedOfDerived()
derived_of_derived.base_method()
derived_of_derived.derived1_method()
derived_of_derived.derived2_method()
derived_of_derived.derived_of_derived_method()
```

```
Single Inheritance:
This is parent method
This is child method
```

```
Multiple Inheritance:
This is father method
This is mother method
This is child2 method

Multilevel Inheritance:
This is grandparent method
This is parent2 method
This is child3 method

Hierarchical Inheritance:
This is parent3 method
This is childA method
This is parent3 method
This is childB method

Hybrid Inheritance:
This is base method
This is derived1 method
This is derived2 method
This is derived of derived method
```

0.1.13 16) Create a Person class with a constructor that takes two arguments name and age.

0.1.14 Create a child class Employee that inherits from Person and adds a new attribute salary.

0.1.15 Override the init method in Employee to call the parent class's init method using the super() and then initialize the salary attribute.

[23]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

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

    def display_info(self):
        super().display_info()
        print(f"Salary:  {self.salary}")
```

```
# Create an employee
employee = Employee("John Doe", 30, 70000)
employee.display_info()
```

```
Name: John Doe, Age: 30
Salary:  70000
```

0.1.16 17) Create a Shape class with a draw method that is not implemented.

0.1.17 Create three child classes Rectangle, Circle, and Triangle that implement the draw method with their respective drawing behaviors.

0.1.18 Create a list of Shape objects that includes one instance of each child class, and then iterate through the list and call the draw method on each object.

[ ]:
```
class Shape:
    def draw(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Rectangle(Shape):
    def draw(self):
        print("Drawing a rectangle")

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Triangle(Shape):
    def draw(self):
        print("Drawing a triangle")

shapes = [Rectangle(), Circle(), Triangle()]

for shape in shapes:
    shape.draw()
```

```
Drawing a rectangle
Drawing a circle
Drawing a triangle
```

[7]:
```
from abc import ABC,abstractmethod
class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Rectangle(Shape):
    def draw(self):
        print("Drawing a rectangle")
```

```
class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Triangle(Shape):
    def draw(self):
        print("Drawing a triangle")

shapes = [Rectangle(), Circle(), Triangle()]

for shape in shapes:
    shape.draw()
```

```
Drawing a rectangle
Drawing a circle
Drawing a triangle
```

[ ]: