Unit9_OOP_numpy_qb_solution

February 18, 2024

0.0.1 Program to demonstrate the use of inheritance

[1]:
```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

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

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

# Creating instances
dog_instance = Dog("Buddy")
cat_instance = Cat("Whiskers")

# Demonstrating Inheritance
print(dog_instance.speak())  # Output: Buddy says Woof!
print(cat_instance.speak())  # Output: Whiskers says Meow!
```

```
Buddy says Woof!
Whiskers says Meow!
```

0.0.2 Program to demonstrate the use of multiple inheritance

[2]:
```
class A:
    def method_A(self):
        return "Method from class A"

class B:
    def method_B(self):
        return "Method from class B"
```

```
class C(A, B):
    def method_C(self):
        return "Method from class C"

# Creating instance
obj_C = C()

# Demonstrating Multiple Inheritance
print(obj_C.method_A())  # Output: Method from class A
print(obj_C.method_B())  # Output: Method from class B
print(obj_C.method_C())  # Output: Method from class C
```

```
Method from class A
Method from class B
Method from class C
```

0.0.3 Program to demonstrate the use of multilevel inheritance

[3]:
```
class Vehicle:
    def display_type(self):
        return "Vehicle"

class Car(Vehicle):
    def display_type(self):
        return "Car"

class Sedan(Car):
    def display_type(self):
        return "Sedan"

# Creating instance
sedan_instance = Sedan()

# Demonstrating Multilevel Inheritance
print(sedan_instance.display_type())  # Output: Sedan
```

```
Sedan
```

0.1 Implement the following hierarchy . The Book function has name, n (number of authors), authors (list of authors), publisher, ISBN, and year as its data members and the derived class has course as its data member. The derived class method overrides (extends) the methods of the base class.

[1]:
```
class Book:
    def __init__(self, name, n, authors, publisher, ISBN, year):
        self.name = name
        self.n = n
        self.authors = authors
```

```
        self.publisher = publisher
        self.ISBN = ISBN
        self.year = year

    def display(self):
        print(f"Name: {self.name}")
        print(f"Number of authors: {self.n}")
        print(f"Authors: {', '.join(self.authors)}")
        print(f"Publisher: {self.publisher}")
        print(f"ISBN: {self.ISBN}")
        print(f"Year: {self.year}")


class CourseBook(Book):
    def __init__(self, name, n, authors, publisher, ISBN, year, course):
        super().__init__(name, n, authors, publisher, ISBN, year)
        self.course = course

    def display(self):
        super().display()
        print(f"Course: {self.course}")


book = Book("The Great Gatsby", 1, ["F. Scott Fitzgerald"], "Scribner",␣
 ↪"9780743273565", 1925)
book.display()

print()

course_book = CourseBook("Data Science from Scratch", 1, ["Joel Grus"],␣
 ↪"O'Reilly", "9781492041139", 2019, "Data Science")
course_book.display()
```

```
Name: The Great Gatsby
Number of authors: 1
Authors: F. Scott Fitzgerald
Publisher: Scribner
ISBN: 9780743273565
Year: 1925

Name: Data Science from Scratch
Number of authors: 1
Authors: Joel Grus
Publisher: O'Reilly
ISBN: 9781492041139
Year: 2019
Course: Data Science
```

0.2 Implement the following hierarchy . The Staff function has name and salary as its data members, the derived class Teaching has subject as its data member and the class NonTeaching has department as its data member. The derived class method overrides (extends) the methods of the base class.

[2]:
```
class Staff:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def display(self):
        print(f"Name: {self.name}")
        print(f"Salary: {self.salary}")

class Teaching(Staff):
    def __init__(self, name, salary, subject):
        super().__init__(name, salary)
        self.subject = subject

    def display(self):
        super().display()
        print(f"Subject: {self.subject}")

class NonTeaching(Staff):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def display(self):
        super().display()
        print(f"Department: {self.department}")
staff_member = Staff("John", 50000)
teaching_member = Teaching("Jane", 60000, "Math")
non_teaching_member = NonTeaching("Joe", 40000, "Finance")

staff_member.display()
print()
teaching_member.display()
print()
non_teaching_member.display()
```

```
Name: John
Salary: 50000

Name: Jane
Salary: 60000
Subject: Math
```

```
Name: Joe
Salary: 40000
Department: Finance
```

0.3 Create a class called Student, having name and email as its data members andm *init*(self, name, email) and putdata(self) as bound methods. The *init* function should assign the values passed as parameters to the requisite variables. The putdata function should display the data of the student. Create another class called PhDguide having name, email, and students as its data members. Here, the students variable is the list of students under the guide. The PhDguide class should have four bound methods: *init*, putdata, add, and remove. The *init* method should initialize the variables, the putdata should show the data of the guide, include the list of students, the add method should add a student to the list of students of the guide and the remove function should remove the student (if the student exists in the list of students of that guide) from the list of students.

[5]:
```python
class Person:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def putdata(self):
        print("Name:", self.name)
        print("Email:", self.email)

class Student(Person):
    def __init__(self, name, email):
        super().__init__(name, email)

class PhDguide(Person):
    def __init__(self, name, email):
        super().__init__(name, email)
        self.students = []

    def putdata(self):
        super().putdata()
        print("Students:", self.students)

    def add(self, student):
        self.students.append(student)

    def remove(self, student):
        if student in self.students:
            self.students.remove(student)
            print(f"{student.name} has been removed from the list of students.")
        else:
```

```
            print(f"{student.name} is not in the list of students.")
# Creating a Student object
student1 = Student("John Doe", "johndoe@example.com")
student1.putdata()  # Output: Name: John Doe, Email: johndoe@example.com

# Creating a PhDguide object with an empty list of students
guide1 = PhDguide("Jane Smith", "janesmith@example.com")
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com,␣
 ↪Students: []

# Adding the student1 to guide1's list of students
guide1.add(student1)
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com,␣
 ↪Students: [Student: John Doe]

# Removing the student1 from guide1's list of students
guide1.remove(student1)
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com,␣
 ↪Students: []
```

```
Name: John Doe
Email: johndoe@example.com
Name: Jane Smith
Email: janesmith@example.com
Students: []
Name: Jane Smith
Email: janesmith@example.com
Students: [<__main__.Student object at 0x0000026FCFB731F0>]
John Doe has been removed from the list of students.
Name: Jane Smith
Email: janesmith@example.com
Students: []
```

0.3.1 Program to demonstrate the issue of invoking init() in case of multiple inheritance

[4]:
```
class A:
    def __init__(self):
        print("Initializing class A")

class B:
    def __init__(self):
        print("Initializing class B")

class C(A, B):
    def __init__(self):
        super().__init__()
```

```
# Creating an instance of class C
obj_c = C()
```

```
Initializing class A
```

0.3.2 Overloading + and * operators for the Fraction class:

[5]:
```
class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __add__(self, other):
        common_denominator = self.denominator * other.denominator
        result_numerator = (self.numerator * other.denominator) + (other.
 ↪numerator * self.denominator)
        return Fraction(result_numerator, common_denominator)

    def __mul__(self, other):
        result_numerator = self.numerator * other.numerator
        result_denominator = self.denominator * other.denominator
        return Fraction(result_numerator, result_denominator)

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

# Example usage
fraction1 = Fraction(1, 2)
fraction2 = Fraction(3, 4)

sum_result = fraction1 + fraction2
product_result = fraction1 * fraction2

print(f"Sum: {sum_result}")
print(f"Product: {product_result}")
```

```
Sum: 10/8
Product: 3/8
```

0.4 Write program that has a class point. Define another class location which has two objects (Location and Destination) of class point. Also define function in Location that prints reflection of Destination on the x axis.

[8]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
class Location(Point):
    def __init__(self, x1, y1, x2, y2):
        super().__init__(x1, y1)
        self.destination = Destination(x2, y2)

    def reflect_destination_on_x_axis(self):
        self.destination.y = -self.destination.y
        print("Reflected Destination point on X axis: ({}, {})".format(self.
 ↪destination.x, self.destination.y))
class Destination(Point):
    pass

# Example usage
l = Location(1, 2, 3, 4)
l.reflect_destination_on_x_axis() # prints (3, -4)
```

```
Reflected Destination point on X axis: (3, -4)
```

0.4.1 Distance and Slope between two points in Cartesian coordinate system:

[7]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def calculate_distance(point1, point2):
    return ((point2.x - point1.x)**2 + (point2.y - point1.y)**2)**0.5

def calculate_slope(point1, point2):
    return (point2.y - point1.y) / (point2.x - point1.x)

# Example usage
point_a = Point(1, 2)
point_b = Point(4, 6)

distance = calculate_distance(point_a, point_b)
slope = calculate_slope(point_a, point_b)

print(f"Distance between points: {distance}")
print(f"Slope between points: {slope}")
```

```
Distance between points: 5.0
Slope between points: 1.3333333333333333
```

0.5 Write program that has classes such as Student, Course and Department. Enroll astudent in a course of particular department

[9]:
```
class Department:
    def __init__(self, name):
        self.name = name

class Course(Department):
    def __init__(self, name, department):
        super().__init__(department)
        self.course_name = name

class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

class Enroll(Student, Course):
    def __init__(self, name, roll_no, course_name, department):
        Student.__init__(self, name, roll_no)
        Course.__init__(self, course_name, department)

    def get_enrolled(self):
        print(f"{self.name} with roll no. {self.roll_no} has enrolled for {self.
 ↪course_name} in {self.name} department.")
enrollment = Enroll("Alice", 101, "Calculus", "Math")
enrollment.get_enrolled()
```

```
Math with roll no. 101 has enrolled for Calculus in Math department.
```

0.6 Create a class student with following member attributes: roll no, name, age and total marks. Create suitable methods for reading and printing member variables. Write a python program to overload '==' operator to print the details of students having same marks.

[10]:
```
class Student:
    def __init__(self, roll_no, name, age, total_marks):
        self.roll_no = roll_no
        self.name = name
        self.age = age
        self.total_marks = total_marks

    def display_student_info(self):
        print(f"Roll No: {self.roll_no}")
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Total Marks: {self.total_marks}")
```

```
    def __eq__(self, other):
        if isinstance(other, Student):
            return self.total_marks == other.total_marks
        return False
# create two student objects
s1 = Student(1, "John", 20, 90)
s2 = Student(2, "Mary", 21, 80)

# compare the students based on their total marks
if s1 == s2:
    print(f"{s1.name} and {s2.name} have the same marks!")
else:
    print(f"{s1.name} and {s2.name} do not have the same marks.")
```

```
John and Mary do not have the same marks.
```

0.7 Write a program to create a class called Data having "value" as its data member. Overload the (>) and the (<) operator for the class. Instantiate the class and compare the objects using *lt* and *gt*.

[11]:
```
class Data:
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

    def __gt__(self, other):
        return self.value > other.value

# Testing the Data class
d1 = Data(10)
d2 = Data(20)
d3 = Data(15)

print(d1 > d2) # False
print(d2 > d3) # True
print(d3 < d1) # False
print(d3 < d2) # True
```

```
False
True
False
True
```

0.8 The following illustration creates a class called data. If no argument is passed while instantiating the class a false is returned, otherwise a true is returned.

[15]:
```
class Data:
    def __init__(self, value=None):
        if value:
            self.value = value
            self.status = True
        else:
            self.status = False

    def __str__(self):
        return f"status: {self.status}"
data_obj1 = Data()
print(data_obj1)
# Output: status: False

data_obj2 = Data(10)
print(data_obj2)
# Output: status: True
```

```
status: False
status: True
```

0.8.1 Create a 5X2 integer array from a range between 100 to 200 such that the difference between each element is 10

[8]:
```
import numpy as np
arr = np.arange(100, 200, 10).reshape(5, 2)
print("5x2 Array:")
print(arr)
```

```
5x2 Array:
[[100 110]
 [120 130]
 [140 150]
 [160 170]
 [180 190]]
```

0.8.2 Following is the provided numPy array. Return array of items by taking the third column from all rows

sampleArray = numpy.array([[11 ,22, 33], [44, 55, 66], [77, 88, 99]])

[9]:
```
import numpy as np
sampleArray = np.array([[11, 22, 33], [44, 55, 66], [77, 88, 99]])
third_column = sampleArray[:, 2]
print("Array of items from the third column:")
```

```
print(third_column)
```

```
Array of items from the third column:
[33 66 99]
```

0.8.3 Return array of odd rows and even columns from below numpy array

sampleArray = numpy.array([[3 ,6, 9, 12], [15 ,18, 21, 24], [27 ,30, 33, 36], [39 ,42, 45, 48], [51 ,54, 57, 60]])"

[10]:
```
import numpy as np

sampleArray = np.array([[3, 6, 9, 12], [15, 18, 21, 24], [27, 30, 33, 36], [39,␣
↪42, 45, 48], [51, 54, 57, 60]])

odd_rows_even_columns = sampleArray[::2, 1::2]
print("Array of odd rows and even columns:")
print(odd_rows_even_columns)
```

```
Array of odd rows and even columns:
[[ 6 12]
 [30 36]
 [54 60]]
```

0.8.4 Sort following NumPy array

Case 1: Sort array by the second row Case 2: Sort the array by the second column sampleArray = numpy.array([[34,43,73],[82,22,12],[53,94,66]])"

[11]:
```
import numpy as np

sampleArray = np.array([[34, 43, 73], [82, 22, 12], [53, 94, 66]])

# Case 1: Sort array by the second row
sorted_by_second_row = sampleArray[:, sampleArray[1, :].argsort()]

# Case 2: Sort the array by the second column
sorted_by_second_column = sampleArray[sampleArray[:, 1].argsort()]

print("Sorted array by the second row:")
print(sorted_by_second_row)

print("\nSorted array by the second column:")
print(sorted_by_second_column)
```

```
Sorted array by the second row:
[[73 43 34]
 [12 22 82]
 [66 94 53]]
```

```
Sorted array by the second column:
[[82 22 12]
 [34 43 73]
 [53 94 66]]
```

0.8.5 Print max from axis 0 and min from axis 1 from the following 2-D array.

sampleArray = numpy.array([[34,43,73],[82,22,12],[53,94,66]])"

[12]:
```
import numpy as np

sampleArray = np.array([[34, 43, 73], [82, 22, 12], [53, 94, 66]])

max_axis_0 = np.max(sampleArray, axis=0)
min_axis_1 = np.min(sampleArray, axis=1)

print("Max from axis 0:")
print(max_axis_0)

print("\nMin from axis 1:")
print(min_axis_1)
```

```
Max from axis 0:
[82 94 73]

Min from axis 1:
[34 12 53]
```

0.8.6 Temperature program

"Write a NumPy array program to convert the values of Fahrenheit degrees into Celsius degrees. The numpy array to be considered is [0, 12, 45.21, 34, 99.91, 32] for Fahrenheit values. Values are stored into a NumPy array. After converting the following numpy array into Celsius, then sort the array and find the position of 0.0 (means where 0.0 value is located i.e. it's index) Formula to convert value of Fahrenheit to Celsius is: C=5F/9 - 532/9 Output: Values in Fahrenheit degrees: [0. 12. 45.21 34. 99.91 32.] Values in Centigrade degrees: [-17.77777778 -11.11111111 7.33888889 1.11111111 37.72777778 0.] [-17.77777778 -11.11111111 0. 1.11111111 7.33888889 37.72777778] (array([2], dtype=int64),)"

[13]:
```
import numpy as np

# Fahrenheit array
fahrenheit_array = np.array([0, 12, 45.21, 34, 99.91, 32])

# Convert Fahrenheit to Celsius using the given formula
celsius_array = (fahrenheit_array - 32) * 5 / 9

# Display original Fahrenheit and converted Celsius arrays
```

```
print("Values in Fahrenheit degrees:")
print(fahrenheit_array)
print("Values in Centigrade degrees:")
print(celsius_array)

# Sort the Celsius array
sorted_celsius_array = np.sort(celsius_array)

# Find the position of 0.0 in the sorted array
zero_position = np.where(sorted_celsius_array == 0.0)

# Display the sorted array and the position of 0.0
print(sorted_celsius_array)
print(zero_position)
```

```
Values in Fahrenheit degrees:
[ 0.   12.   45.21 34.   99.91 32.  ]
Values in Centigrade degrees:
[-17.77777778 -11.11111111   7.33888889   1.11111111  37.72777778
   0.        ]
[-17.77777778 -11.11111111   0.           1.11111111   7.33888889
  37.72777778]
(array([2], dtype=int64),)
```

0.8.7 Reshape program

"import numpy as np arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

Reshape arr into a 2D array and a 3D array."

[14]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

# Reshape arr into a 2D array
arr_2d = arr.reshape(2, 6)

# Reshape arr into a 3D array
arr_3d = arr.reshape(2, 3, 2)

print("Original array:")
print(arr)

print("\nReshaped 2D array:")
print(arr_2d)

print("\nReshaped 3D array:")
print(arr_3d)
```

```
Original array:
[ 1  2  3  4  5  6  7  8  9 10 11 12]

Reshaped 2D array:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]

Reshaped 3D array:
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

0.8.8 ABTRACT CLASS PROGRAM

Imagine you own a call center. Use the following abstract class template to create three more classes, Respondent, Manager, and Director that inherit this Employee Abstract Class.

from abc import ABC, abstractmethod

class Employee(ABC):

@abstractmethod def receive_call(self): pass

@abstractmethod def end_call(self): pass

@abstractmethod def is_free(self): pass

@abstractmethod def get_rank(self): pass " Create a program using the instructions given below: 1. Create a constructor in all three classes (Respondent, Manager and Director) which takes the id and name as input and initializes two additional variables, rank and free. rank should be equal to 3 for Respondent, 2 for Manager and 1 for Director. free should be a boolean variable with value True initially. (1 mark) 2. Implement rest of the methods in all three classes in the following way: (2 marks) a. receive_call(): prints the message, "call received by (name of the employee)" and sets the free variable to False. b. end_call(): prints the message, "call ended" and sets the free variable to True. c. is_free(): returns the value of the free variable d. get_rank(): returns the value of the rank variable 3. Create a class Call, with a constructor that accepts id and name of the caller and initializes a variable called assigned to False. (0.5 marks) 4. Create a class CallHandler, with three lists, respondents, managers and directors as class variables. (0.5 marks) 5. Create an add_employee() method in CallHandler class that allows you to add an employee (an object of Respondent/Manager/Director) into one of the above lists according to their rank. (1 mark) 6. Create a dispatch_call() method in CallHandler class that takes a call object as a parameter. This method should find the first available employee starting from rank 3, then rank 2 and then rank 1. If a free employee is found, call its receive_call() function and change the call's assigned variable value to True. If no free employee is found, print the message: "Sorry! All employees are currently busy." (2 marks) 7. Create 3 Respondent objects, 2 Manager objects and 1 Director object and add them into the list of available employees using the CallHandler's add_employee() method. (1 mark) 8. Create a Call object and demonstrate how it is assigned to an employee. (1 mark) "

[15]:
```python
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, employee_id, name):
        self.employee_id = employee_id
        self.name = name
        self.rank = self.get_rank()
        self.free = True

    @abstractmethod
    def receive_call(self):
        pass

    @abstractmethod
    def end_call(self):
        pass

    @abstractmethod
    def is_free(self):
        pass

    @abstractmethod
    def get_rank(self):
        pass

class Respondent(Employee):
    def __init__(self, employee_id, name):
        super().__init__(employee_id, name)

    def receive_call(self):
        print(f"Call received by {self.name}")
        self.free = False

    def end_call(self):
        print("Call ended")
        self.free = True

    def is_free(self):
        return self.free

    def get_rank(self):
        return 3

class Manager(Employee):
    def __init__(self, employee_id, name):
        super().__init__(employee_id, name)
```

```
    def receive_call(self):
        print(f"Call received by {self.name}")
        self.free = False

    def end_call(self):
        print("Call ended")
        self.free = True

    def is_free(self):
        return self.free

    def get_rank(self):
        return 2

class Director(Employee):
    def __init__(self, employee_id, name):
        super().__init__(employee_id, name)

    def receive_call(self):
        print(f"Call received by {self.name}")
        self.free = False

    def end_call(self):
        print("Call ended")
        self.free = True

    def is_free(self):
        return self.free

    def get_rank(self):
        return 1

class Call:
    def __init__(self, caller_id, caller_name):
        self.caller_id = caller_id
        self.caller_name = caller_name
        self.assigned = False

class CallHandler:
    respondents = []
    managers = []
    directors = []

    def add_employee(self, employee):
        if employee.get_rank() == 3:
            self.respondents.append(employee)
        elif employee.get_rank() == 2:
```

```
            self.managers.append(employee)
        elif employee.get_rank() == 1:
            self.directors.append(employee)

    def dispatch_call(self, call):
        for employee in self.respondents + self.managers + self.directors:
            if employee.is_free():
                employee.receive_call()
                call.assigned = True
                break
        else:
            print("Sorry! All employees are currently busy.")

# Step 7: Creating employees
respondent1 = Respondent(1, "Respondent1")
respondent2 = Respondent(2, "Respondent2")
respondent3 = Respondent(3, "Respondent3")

manager1 = Manager(4, "Manager1")
manager2 = Manager(5, "Manager2")

director1 = Director(6, "Director1")

# Adding employees to the CallHandler
call_handler = CallHandler()
call_handler.add_employee(respondent1)
call_handler.add_employee(respondent2)
call_handler.add_employee(respondent3)
call_handler.add_employee(manager1)
call_handler.add_employee(manager2)
call_handler.add_employee(director1)

# Step 8: Creating a Call object and demonstrating call dispatch
call1 = Call(101, "John Doe")
call_handler.dispatch_call(call1)
```

```
Call received by Respondent1
```

0.8.9 Write a python program to create a Bus child class that inherits from the Vehicle class.

In Vehicle class vehicle name, mileage and seatingcapacity as its data member. The default fare charge of any vehicle is seating capacity * 100. If Vehicle is Bus instance, we need to add an extra 10% on full fare as a maintenance charge. So total fare for bus instance will become the final amount = total fare + 10% of the total fare.

Sample Output: The bus seating capacity is 50. so, the final fare amount should be 5000+500=5500.

The car seating capacity is 5. so, the final fare amount should be 500."

[16]:
```python
class Vehicle:
    def __init__(self, name, mileage, seating_capacity):
        self.name = name
        self.mileage = mileage
        self.seating_capacity = seating_capacity

    def calculate_fare(self):
        return self.seating_capacity * 100

class Bus(Vehicle):
    def __init__(self, name, mileage, seating_capacity):
        super().__init__(name, mileage, seating_capacity)

    def calculate_fare(self):
        base_fare = super().calculate_fare()
        maintenance_charge = 0.1 * base_fare
        final_fare = base_fare + maintenance_charge
        return final_fare

# Example usage:
bus_instance = Bus("Bus", 10, 50)
car_instance = Vehicle("Car", 20, 5)

bus_fare = bus_instance.calculate_fare()
car_fare = car_instance.calculate_fare()

print(f"The bus seating capacity is {bus_instance.seating_capacity}. "
      f"So, the final fare amount should be {bus_fare}.")

print(f"The car seating capacity is {car_instance.seating_capacity}. "
      f"So, the final fare amount should be {car_fare}.")
```

```
The bus seating capacity is 50. So, the final fare amount should be 5500.0.
The car seating capacity is 5. So, the final fare amount should be 500.
```

0.8.10 Create an abstract class named Shape.

Create an abstract method named calculate_area for the Shape class. Create Two Classes named Rectangle and Circle which inherit Shape class. Create calculate_area method in Rectangle class. It should return the area of the rectangle object. (area of rectangle = (length * breadth)) Create calculate_area method in Circle class. It should return the area of the circle object. (area of circle = r^2)) Create objects of Rectangle and Circle class. The python Program Should also check whether the area of one Rectangle object is greater than another rectangle object by overloading > operator. Execute the method resolution order of the Circle class.

[17]:
```python
from abc import ABC, abstractmethod
import math
```

```python
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def calculate_area(self):
        return self.length * self.breadth

    def __gt__(self, other):
        return self.calculate_area() > other.calculate_area()

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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

# Creating objects of Rectangle and Circle class
rectangle1 = Rectangle(5, 10)
rectangle2 = Rectangle(3, 8)

circle = Circle(7)

# Checking whether the area of one Rectangle object is greater than another
if rectangle1 > rectangle2:
    print("Area of rectangle1 is greater than rectangle2.")
else:
    print("Area of rectangle2 is greater than rectangle1.")

# Printing the method resolution order of the Circle class
print(f"Method Resolution Order for Circle class: {Circle.mro()}")
```

```
Area of rectangle1 is greater than rectangle2.
Method Resolution Order for Circle class: [<class '__main__.Circle'>, <class
'__main__.Shape'>, <class 'abc.ABC'>, <class 'object'>]
```

0.8.11 Write a python program to demonstrate the use of super() method to call the method of base class.

[18]:
```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

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

    def speak(self):
        super().speak()  # Calling the speak method of the base class
        print(f"{self.name} barks loudly")

# Create an instance of the Dog class
dog_instance = Dog("Buddy", "Labrador")

# Call the speak method of the Dog class
dog_instance.speak()
```

```
Buddy makes a sound
Buddy barks loudly
```

0.8.12 Create a class called Matrix containing constructor that initialized the number of rows and number of columns of a new Matrix object.

The Matrix class has methods for each of the following: 1. get the number of rows 2. get the number of columns 3. set the elements of the matrix at given position (i,j) 4. adding two matrices. If the matrices are not addable, " Matrices cannot be added" will be displayed.(Overload the addition operation to perform this) 5. Multiplying the two matrices. If the matrices are not multiplied, " Matrices cannot be multiplied" will be displayed.(Overload the addition operation to perform this) "

[19]:
```
class Matrix:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.matrix = [[0 for _ in range(columns)] for _ in range(rows)]

    def get_rows(self):
        return self.rows

    def get_columns(self):
```

```
        return self.columns

    def set_element(self, i, j, value):
        if 0 <= i < self.rows and 0 <= j < self.columns:
            self.matrix[i][j] = value
        else:
            print("Invalid position. Cannot set element.")

    def add_matrices(self, other_matrix):
        if self.rows == other_matrix.get_rows() and self.columns ==␣
↪other_matrix.get_columns():
            result_matrix = Matrix(self.rows, self.columns)
            for i in range(self.rows):
                for j in range(self.columns):
                    result_matrix.set_element(i, j, self.matrix[i][j] +␣
↪other_matrix.matrix[i][j])
            return result_matrix
        else:
            print("Matrices cannot be added.")

    def multiply_matrices(self, other_matrix):
        if self.columns == other_matrix.get_rows():
            result_matrix = Matrix(self.rows, other_matrix.get_columns())
            for i in range(self.rows):
                for j in range(other_matrix.get_columns()):
                    result = 0
                    for k in range(self.columns):
                        result += self.matrix[i][k] * other_matrix.matrix[k][j]
                    result_matrix.set_element(i, j, result)
            return result_matrix
        else:
            print("Matrices cannot be multiplied.")

    def __str__(self):
        return '\n'.join([' '.join(map(str, row)) for row in self.matrix])

# Example usage:
matrix1 = Matrix(2, 3)
matrix2 = Matrix(3, 2)

matrix1.set_element(0, 0, 1)
matrix1.set_element(0, 1, 2)
matrix1.set_element(0, 2, 3)
matrix1.set_element(1, 0, 4)
matrix1.set_element(1, 1, 5)
matrix1.set_element(1, 2, 6)
```

```
matrix2.set_element(0, 0, 7)
matrix2.set_element(0, 1, 8)
matrix2.set_element(1, 0, 9)
matrix2.set_element(1, 1, 10)
matrix2.set_element(2, 0, 11)
matrix2.set_element(2, 1, 12)

print("Matrix 1:")
print(matrix1)

print("\nMatrix 2:")
print(matrix2)

result_addition = matrix1.add_matrices(matrix2)
print("\nMatrix Addition:")
print(result_addition)

result_multiplication = matrix1.multiply_matrices(matrix2)
print("\nMatrix Multiplication:")
print(result_multiplication)
```

```
Matrix 1:
1 2 3
4 5 6

Matrix 2:
7 8
9 10
11 12
Matrices cannot be added.

Matrix Addition:
None

Matrix Multiplication:
58 64
139 154
```

0.8.13 Find the MRO of class Z of below program:

[21]:
```
class A: pass
class B: pass
class C: pass
class D:pass
class E:pass
class K1(C,A,B): pass
class K3(A,D): pass
```

```
class K2(B,D,E): pass
class Z( K1,K3,K2): pass
print(Z.mro())
```

```
[<class '__main__.Z'>, <class '__main__.K1'>, <class '__main__.C'>, <class
'__main__.K3'>, <class '__main__.A'>, <class '__main__.K2'>, <class
'__main__.B'>, <class '__main__.D'>, <class '__main__.E'>, <class 'object'>]
```

0.8.14 Write a Python Program to Find the Net Salary of Employee using Inheritance.

Create three Class Employee, Perks, NetSalary. Make an Employee class as an abstract class. Employee class should have methods for following tasks. - To get employee details like employee id, name and salary from user. - To print the Employee details. - return Salary. - An abstract method emp_id. Perks class should have methods for following tasks. - To calculate DA, HRA, PF. - To print the individual and total of Perks (DA+HRA-PF). Netsalary class should have methods for following tasks. - Calculate the total Salary after Perks. - Print employee detail also prints DA, HRA, PF and net salary.

Note 1: DA-35%, HRA-17%, PF-12% Note 2: It is compulsory to create objects and demonstrating the methods with Correct output. Example: Employee ID: 1 Employee Name: John Employee Basic Salary: 25000 DA: 8750.0 HRA: 4250.0 PF: 3000.0 Total Salary: 35000.0"

[22]:
```
from abc import ABC, abstractmethod

class Employee(ABC):

    def get_employee_details(self):
        self.emp_id = int(input("Enter Employee ID: "))
        self.name = input("Enter Employee Name: ")
        self.salary = float(input("Enter Employee Basic Salary: "))

    @abstractmethod
    def emp_id(self):
        pass

    def print_employee_details(self):
        print("Employee ID:", self.emp_id)
        print("Employee Name:", self.name)
        print("Employee Basic Salary:", self.salary)

    def get_salary(self):
        return self.salary

class Perks(Employee):
    def __init__(self):
        super().__init__()
```

```
    def calculate_perks(self):
        self.da = 0.35 * self.salary
        self.hra = 0.17 * self.salary
        self.pf = 0.12 * self.salary

    def print_perks(self):
        print("DA:", self.da)
        print("HRA:", self.hra)
        print("PF:", self.pf)
        print("Total Perks:", self.da + self.hra - self.pf)

class NetSalary(Perks):
    def __init__(self):
        super().__init__()

    def calculate_net_salary(self):
        self.net_salary = self.salary + self.da + self.hra - self.pf

    def print_net_salary(self):
        self.print_employee_details()
        self.print_perks()
        print("Total Salary:", self.net_salary)

    def emp_id(self):
        # Implement the emp_id method here
        pass

# Create an object of NetSalary class and demonstrate its methods
employee = NetSalary()
employee.get_employee_details()
employee.calculate_perks()
employee.calculate_net_salary()
employee.print_net_salary()
```

```
Enter Employee ID: 12
Enter Employee Name: shp
Enter Employee Basic Salary: 35000
Employee ID: 12
Employee Name: shp
Employee Basic Salary: 35000.0
DA: 12250.0
HRA: 5950.0
PF: 4200.0
Total Perks: 14000.0
Total Salary: 49000.0
```

[ ]: