Module Title

Principles of Programming

Weekly Assignment - Practical 7

Year

2025

Student Name: NIRVIK K.C.

UWE ID: 25024649

Assignment Submission Date: January 14, 2026

This assignment consists of the programming questions from practical exercises related to the topics from week 7.

Basic programming on Objects and Classes

Objective:

Practice fundamental coding, compiling, and running basic Python programs, and learn CLASS, OBJECT, METHODS, CONSTRUCTORS, CLASS and INSTANCE VARIABLES.

Questions

Exercise 1: Python Classes, Objects, and Constructors

1.

Task 1.1: In practical7, create a new file kitten.py, and it automatically opens in the editor. Write the following program and run it.

Given code:

```
class Kitten:
  pass
kitt = Kitten()
print(kitt)
```

If you can successfully run your program, you should see the following output:

<__main__.Kitten object at 0x0000023DD73646E0>

In this program Kitten is a class, and kitt is an instance/object of that class. The above output basically shows that you now have a Kitten object at 0x0000014BC3C33278. This string of letters and numbers is a memory address that indicates where the Kitten object is stored in your computer's memory. However, the address you see on your screen will be different. In this way, you can create as many objects as you like.

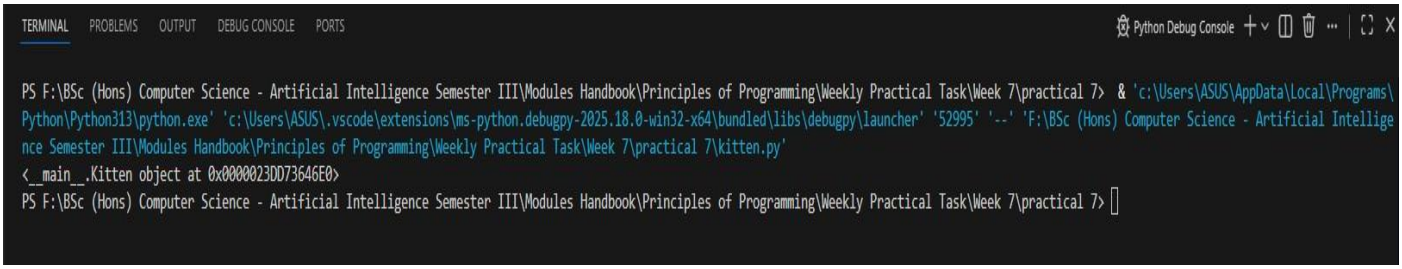
Answer:

Following code for input:

```
# Python - Classes and Objects

# Practical 7 - Exercise 1 - Task 1.1

# Nirvik K.C.

# Kitten is a class

class Kitten:

    pass

# # Create an object (instance) of the Kitten class

kitt = Kitten()

# Print the object

print(kitt)
```

Output: <__main__.Kitten object at 0x0000023DD73646E0>

Output obtained in execution:



2.

Task 1.2: Create another object kitt2, use the print statement as before to print it and run your code.

Observation: Your new Kitten instance should be located at a different memory address. That's because it's an entirely new instance and is completely unique from the first Kitten object that you instantiated.

Answer:

Following code for input:

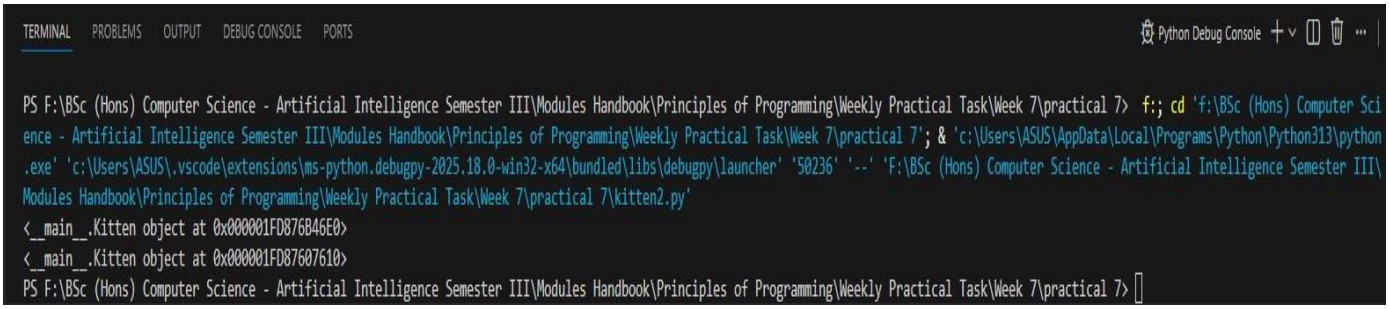
Python - Classes and Objects

Practical 7 - Exercise 1 - Task 1.1

Nirvik K.C.

```
# Kitten is a class

class Kitten:

    pass

# Create the first object of the Kitten class

kitt = Kitten()

# Print the object

print(kitt)

# Create the second object of the Kitten class

kitt2 = Kitten()

print(kitt2)


# Output: <__main__.Kitten object at 0x000001FD876B46E0>

# <__main__.Kitten object at 0x000001FD87607610>
```

Output obtained in execution:



3. Task 1.3: Member variable, method and constructor. Modify your program as below and run it.

Given code:

```
Class Kitten:

# constructor

  def __init__(self):

      # initilialising instance/member variable age

      self.age=1

# an instance/member method

def display_age(self):

    print(self.age)

# creating object of the class. This invokes constructor

kitt = Kitten()

# calling the instance/member method using the object kitt

kitt.display_age()
```

If you can successfully run your program, you should see the following output:

1

Observation:

In the above example, we have an instance variable age which we are initializing in the constructor. The constructor is being invoked when we create the object of the class, kitt in this example.

The self-parameter is a reference to the current instance/object of the class, and is used to access variables that belongs to the class. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any method in the class.

This constructor is known as a default constructor, which doesn't accept any arguments def __init__(self):

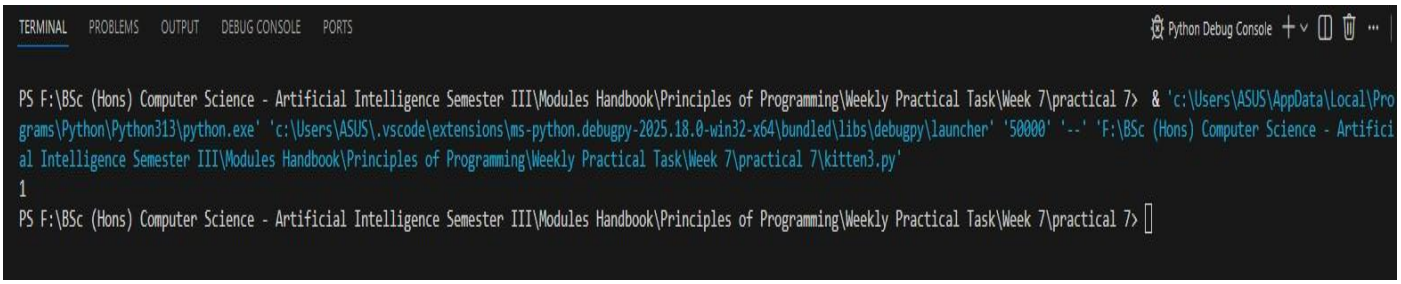
Answer:

Following code for input:

```
# Python - Member variable, method, and constructor

# Practical 7 - Exercise 1 - Task 1.3

# Nirvik K.C.
```

```
class Kitten:
    # Constructor (default constructor - no arguments)
    def __init__(self):
        # Initialise instance/member variable 'age'
        self.age = 1


        # An instance/member method
    def display_age(self):
        print(self.age)


# Creating object of the class
kitt = Kitten()


# Calling the method using the object
kitt.display_age()


# Output: 1
```

Output obtained in execution:



4.

Task 1.4: Modify your program as below and run it. Please note the commented-out constructor code. This means that we don't have any explicit constructor in this program.

Given code:

```
class Kitten:
# constructor
 # def __init__(self):
       # initilialising instance/member variable age
# self.age = 1
# An instance/member method
def display_age(self):
    print("Age unknown")
```

```
# creating object of the class. This invokes constructor
kitt = Kitten()

# calling the instance/member method using the object kitt
kitt.display_age()
```

If you can successfully run your program, you should see the following output:

```
Age unknown
```

Observation: In this example, we do not have a constructor but still we are able to create an object for the class Kitten. The program works as before because there is a default constructor implicitly inserted by python during the program compilation. The implicit default constructor looks like this:

```
def __init__(self):
    # no body, does nothing.
```

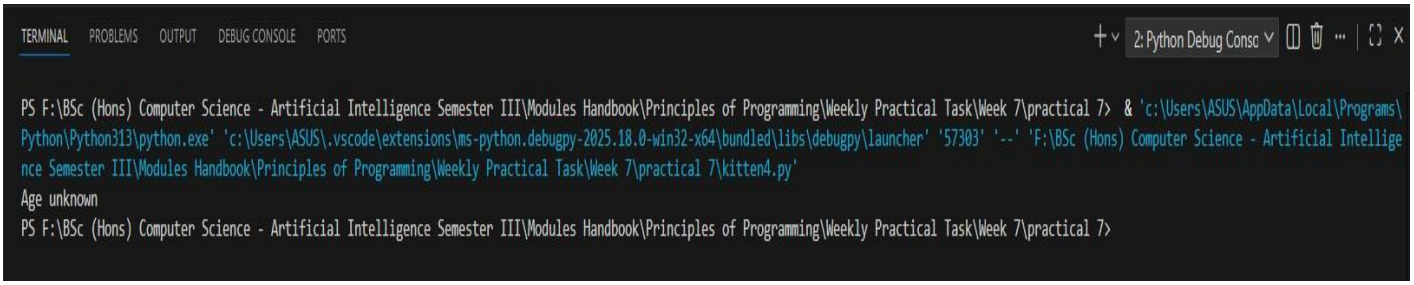
Answer:

Following code for input:

```
# Python - No use of any explicit constructor
# Practical 7 - Exercise 1 - Task 1.4
# Nirvik K.C.
```

```
class Kitten:
    # constructor
    # def __init__(self):
    #     # Initialising instance/member variable 'age'
    #     self.age = 1

    # An instance/member method
    def display_age(self):
        print("Age unknown")

# Creating object of the class. This invoked constructor
kitt = Kitten()

# Calling the method using the object
kitt.display_age()

# Output: Age unknown
```

Output obtained in execution:



5.

Task 1.5: Parameterized constructor. Modify your program as below and run it. Note the parameterised constructor and object creation by passing the value 3.

Given code:

```
Class Kitten:
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    # self.age=1


# an instance/member method
def display_age(self):
    print(self.age)
```

creating object of the class. This invokes parameterised

constructor kitt = Kitten(3)

calling the instance/member method using the object kitt

kitt.display_age()

If you can successfully run your program, you should see the following output:

3

Observation: This type of constructors is known as Parameterized constructors. It accepts the arguments during object creation. You can observe that with such type of constructors we can pass the values during object creation, which is used by the constructor to initialize the instance members of that object.

Answer:

Following code for input:

Parameterised constructor and object creation by passing value

Practical 7 - Exercise 1 - Task 1.5

Nirvik K.C.

```
class Kitten:
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    #   self.age = 1
    # Parametrised constructor
    def __init__(self, age):
        # Initialising instance/ member variable age
        self.age = age

    # Instance/member method
    def display_age(self):
        print(self.age)

# Creating object by passing a value to the parametrised constructor
kitt = Kitten(3)

# # Calling the instance/member method using the object kitt
kitt.display_age()

# Output: 3
```

Output obtained in execution:

6. Task 1.6: Create another object kitt2 by passing the value 4, call the member method, display_age(), and run your program.

```
class Kitten:
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    # self.age=1
   # parameterised constructor
  def __init__(self, value):
        # initilialising instance/member variable age
        self.age = value

  # an instance/member method
 def display_age(self):
      print(self.age)
```

```
# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)

kitt2 = Kitten(4)

# calling the instance/member method using the object kitt
kitt.display_age()

kitt2.display_age()
```

What output do you see after your program has been executed?

Observation: Every instance of a particular class has the same method(s), but the method(s) can behave differently based on the value of the instance variables.

Answer:

Following code for input:

```
# Create object kitt2 by passing the value 4, call the member method,
display_age(), and run your program.
# Practical 7 - Exercise 1 - Task 1.6
# Nirvik K.C.
```

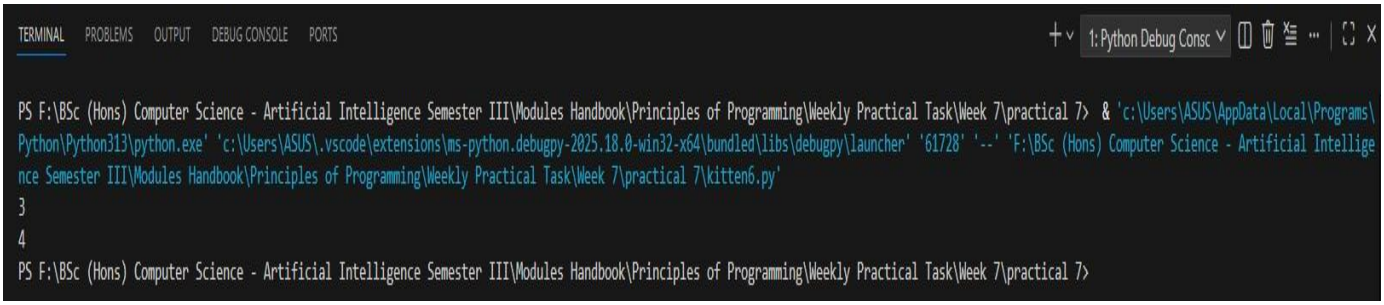
```
class Kitten:
    # Parameterised constructor
    def __init__(self, value):
        # Initialising instance/member variable age
        self.age = value

    # Instance/member method
    def display_age(self):
        print(self.age)

# Creating first object of the class. This invokes parameterised constructor
kitt = Kitten(3)

# Creating second object of the class. This invokes parameterised constructor
kitt2 = Kitten(4)

# Calling the instance/member method using the object kitt
kitt.display_age()

kitt2.display_age()
```

Output:

3

4

Output obtained in execution:



7.

Task 1.7: In your program call again the display_age() method using the kitt object and run your program.

Given code:

```
class Kitten:
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    #     self.age=1
    # parameterised constructor
```

```
def __init__(self, value):
    # initilialising instance/member variable age
    # self.age = value

# an instance/member method
def display_age(self):
    print(self.age)

# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)

# calling the instance/member method using the object
kitt kitt.display_age()

kitt2.display_age()

kitt.display_age()
```

If you can successfully run your program, you should see the following output:

```
3
4
3
```

Observation: Each object has its own copies of the instance variables.

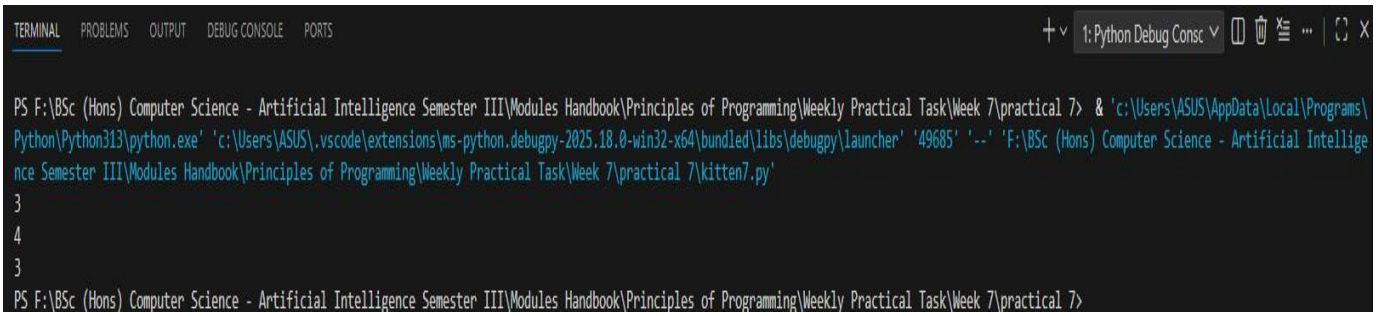
Answer:

Following code for input:

```
# Call the method using the object created and run your program
# Practical 7 - Exercise 1 - Task 1.7
# Nirvik K.C.
class Kitten:
    # Parameterised constructor
    def __init__(self, value):
        # Initialising instance/member variable age
        self.age = value

    # Instance/member method
    def display_age(self):
        print(self.age)
```

```
# Creating objects of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)

# Calling the method 'display_age()' on both objects
kitt.display_age()
kitt2.display_age()

# Calling display_age() again on the object kitt
kitt.display_age() # Prints 3 again

# Output:
# age = 3
# age = 4
# age = 3
```

Output obtained in execution:



8.

Task 1.8: Now try to create another object kitt3 without passing any value.

Given code:

```
class Kitten:
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    #    self.age=1

   # parameterised constructor
```

```
def __init__(self, value):
    # initilialising instance/member variable age
    self.age = value


# an instance/member method
def display_age(self):
    print(self.age)


# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)


# This invokes default constructor
kitt3 = Kitten()
# calling the instance/member method using the object kitt
kitt.display_age()
kitt2.display_age()
kitt.display_age()
```

Does your program work? If not, why?

Observation: (1) An object cannot be created if we don't have a constructor in our program. To create kitt3 object you need a default constructor. A default constructor is automatically generated by the compiler only if you do not explicitly define a constructor in your class. In this case you already have defined a (parameterised) constructor, so the required default constructor will not be automatically generated. (2) Even if you explicitly define another default constructor in your program, it will not work. This is because, unlike other OOP languages such as C++ and Java, Python does not support the concept of method overloading explicitly. (3) However, one way to achieve this in Python is by assigning "None" to the parameter(s):

```
def __init__(self, value=None):
        # initilialising instance/member variable age
       self.age = value
```

Answer:

Following code for input:

```
# Create another object kitt3 without passing any value
# Practical 7 - Exercise 1 - Task 1.8
# Nirvik K.C.
```

```
class Kitten:
    # Parameterized constructor
    def __init__(self, value):
        # Initialising instance/member variable age
        self.age = value

    # Instance/member method
    def display_age(self):
        print(self.age)

# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)
# This invokes parameterised constructor
# Create a third object without passing any value
kitt3 = Kitten()
# Calling the instance/member method
kitt.display_age()
```

kitt2.display_age()

kitt.display_age()

Q) Does your program work? If not, why?

Ans: No, the program does not work. It will crash before it reaches the display_age() calls. The error occurs on the line kitt3 = Kitten(). In class definition, the __init__ method (constructor) is defined with a required parameter called value.

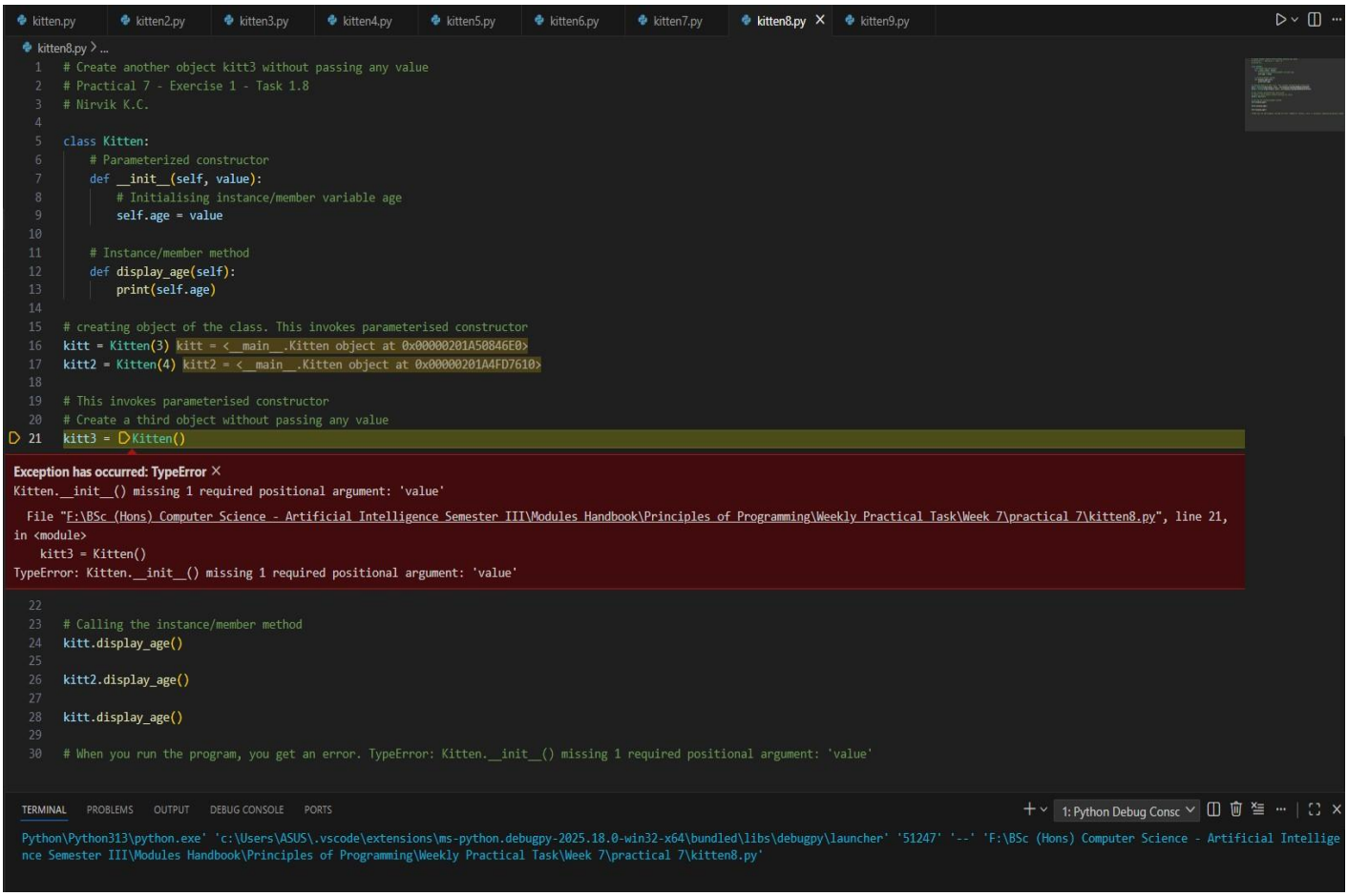
```
def __init__(self, value):
    self.age = value
```

When I called Kitten(), Python looked for that value argument. Since I didn't provide one, it doesn't know what value to assign to self.age, which resulted in a TypeError.

The Error Message:

TypeError: Kitten.__init__() missing 1 required positional argument: 'value'

The Message Shown:



9.

Task 1.9: Now modify your program as below and run it. Note the commented statement kitt3 = Kitten() and introduction and calling of a new member method set_age. Given code:

Class Kitten:

    # constructor

```
    # def __init__(self):
    # initilialising instance/member variable age
    #     self.age = 1


   # parameterised constructor
   def __init__(self, value):
        # initilialising instance/member variable age
        self.age = value


  # an instance/member method
  def set_age(self, value):
       self.age = value


 # an instance/member method
 def display_age(self):
      print(self.age)


# creating object of the class. This invokes parameterised constructor
```

```
kitt = Kitten(3)

kitt2 = Kitten(4)


# This invokes default constructor

# kitt3 = Kitten()


# calling the instance/member method using the object kitt

 kitt.display_age()


kitt2.display_age()


kitt.display_age()


kitt.set_age(5)


kitt.display_age()
```

If you can successfully run your program, you should see the following output:

3

4

3

5

What's the difference between initialising the member variables using a parameterised (or default) constructor and by calling set_age method? (Just write the answers as comments in the code)

Answer:

Following code for input:

```
# Modify kitten8.py code - commented statement kitt3 = Kitten() and introduction and calling of a new member method set_age

# Practical 7 - Exercise 1 - Task 1.9

# Nirvik K.C.
```

```
class Kitten:
    # Parameterized constructor
    def __init__(self, value):
        # Initialising instance/member variable age with passed value
        self.age = value

    # An instance/member method to change variable age after creation
    def set_age(self, value):
         self.age = value

    # An Instance/member to display the age
    def display_age(self):
        print(self.age)

# Creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)
```

```
# This invokes default constructor
# Commented out as it caused error
# kitt3 = Kitten()

# Calling the instance/member method using the object kitt

kitt.display_age()

kitt2.display_age()

kitt.display_age()

# Change the age of kitt using the new set_age method
kitt.set_age(5)

# Display the age after the change
kitt.display_age()
```

Output:

3

4

3

5

What’s the difference between initialising the member variables using a parameterised (or default) constructor and by calling set_age method? (Just write the answers as comments in the code)

Difference between parameterised (or default) constructor (__init__) and Setter Method like (set_age)

1. Constructor runs automatically only once when object is created.

1. Setter is called manually whenever you want to change the value later

2. Constructor is ideal for required initial values/initialisation.

2. Setter is ideal for optional or future changes made to an object's attributes after the object has already been created.

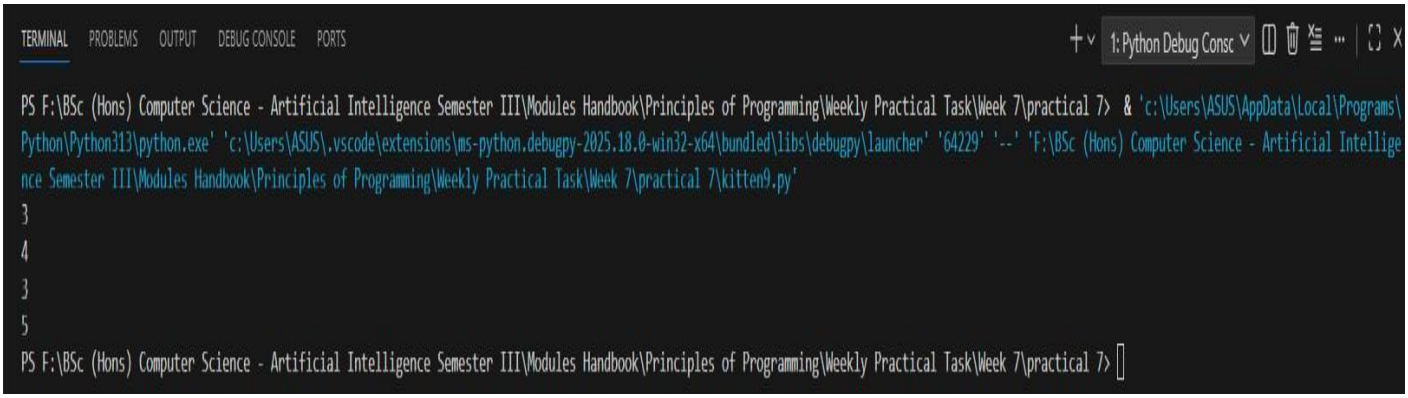
3. Constructor cannot be called again after creation.

3. Setter can be called multiple times.

4. Constructor usually has no validation logic, just set the value.

4. Setter can include validation, logic, etc.

Output obtained in execution:



Exercise 2: Class and Instance variables in Python

10.

Task 2.1 (a) : Now modify your program as below and run it. Note the introduction of the class variable breed and printing its value using kitt and kitt2 objects.

```
class Kitten:
    breed = 'Abyssinian' # class variable shared by all instances
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    #    self.age=1

    # parameterised constructor
    def __init__(self, value):
        # initilialising instance/member variable age
        self.age = value
    # an instance/member method
    def set_age(self,value):
        self.age = value
```

```
    # an instance/member method
    def display_age(self):
        print(self.age)


# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)
kitt2 = Kitten(4)
#This invokes default constructor
#kitt3 = Kitten()


# calling the instance/member method using the object kitt
kitt.display_age()
kitt2.display_age()
kitt.display_age()
kitt.set_age(5)
kitt.display_age()
print(kitt.breed)
print(kitt2.breed)
```

If you can successfully run your program, you should see the following output:

3

4

3

5

Abyssinian

Abyssinian

Observation:

There is a little more flexibility when it comes to access the class variable.

Trying to access the age variable through the class will result in an Error since instance variables are object specific and are created when __init__ constructor is invoked. This is the central distinction between the class and instance variables.

Answer:

Following code for input:

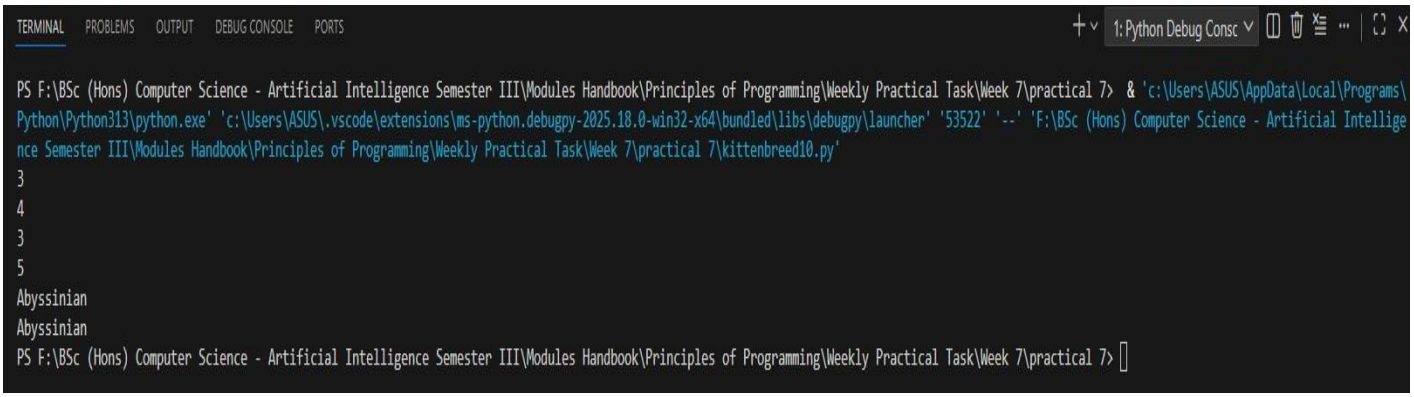
Introduction of the class variable and printing its value

```
# Practical 7 - Exercise 2 - Task 2.1 (a)
# Nirvik K.C.
class Kitten:
    # Class variable
    breed = 'Abyssinian'

    # Parameterized constructor
    def __init__(self, value):
        # Instance variable
        self.age = value

    # Instance method to change age
    def set_age(self, value):
        self.age = value

    # Instance method to display
    def display_age(self):
        print(self.age)
```

```
# Create objects using parameters constructor. This invokes parameterised
constructor
kitt = Kitten(3)
kitt2 = Kitten(4)

# This invokes default constructor
# kitt3 = Kitten()

# Display initial ages using constructor
kitt.display_age()
kitt2.display_age()
kitt.display_age()

# Change age of the first kitten using the setter method
kitt.set_age(5)

# Show the new age
kitt.display_age()   # 5
```

```
# Print the class variable using both objects

print(kitt.breed)  # Abyssinia

print(kitt2.breed) # Abyssinia


# Output:

# 3

# 4

# 3

# 5

# Abyssinian

# Abyssinian
```

Output obtained in execution:



11.

Task 2.1 (b): Now modify your program as below and run it. Note that the breed value has been changed to American Bobtail.

Given code:

```
class Kitten:
    breed = 'Abyssinian' # class variable shared by all instances
    # constructor
    # def __init__(self):
    # initilialising instance/member variable age
    #    self.age=1

    # parameterised constructor
    def __init__(self, value):
        # initilialising instance/member variable age
        self.age = value
    # an instance/member method
    def set_age(self, value):
        self.age = value
```

```
    # an instance/member method
    def display_age(self):
        print(self.age)

# creating object of the class. This invokes parameterised constructor
kitt = Kitten(3)

kitt2 = Kitten(4)

# This invokes default constructor
# kitt3 = Kitten()

# Calling the instance/member method using the object kitt
kitt.display_age()

kitt2.display_age()

kitt.display_age()
```

```
kitt.set_age(5)


kitt.display_age()


print(kitt.breed) # shared by all kittens
print(kitt2.breed) # shared by all kittens
print(Kitten.breed) #breed is accessed using the Class
#print(Kitten.age) # instance variable cannot be accessed via Class


# changing the breed value
Kitten.breed = 'American Bobtail'


print(kitt.breed) # shared by all kittens
print(kitt2.breed) # shared by all kittens
```

If you can successfully run your program, you should see the following output:

3

4

3

5

Abyssinian

Abyssinian

Abyssinian

American Bobtail

American Bobtail

Observation: What we did above will make all kittens as American Bobtail since we have modified a class variable, which will apply to all instances of Kitten class. Therefore, modifying a class variable on the class namespace affects all the instances of the class.

Answer:

Following code for input:

```
# Introduction of the class variable and printing its value -  breed value has been changed to American Bobtail

# Practical 7 - Exercise 2 - Task 2.1 (b)

# Nirvik K.C.
```

```
class Kitten:
    # Class variable which is shared by all instances of the Kitten class
    # breed value - Abyssinian
    breed = 'Abyssinian'

    # Parameterized constructor
    def __init__(self, value):
        # initilialising instance/member variable age
        self.age = value

    # An instance/member method
    def set_age(self, value):
        self.age = value

    # An instance/member method
    def display_age(self):
        print(self.age)
```

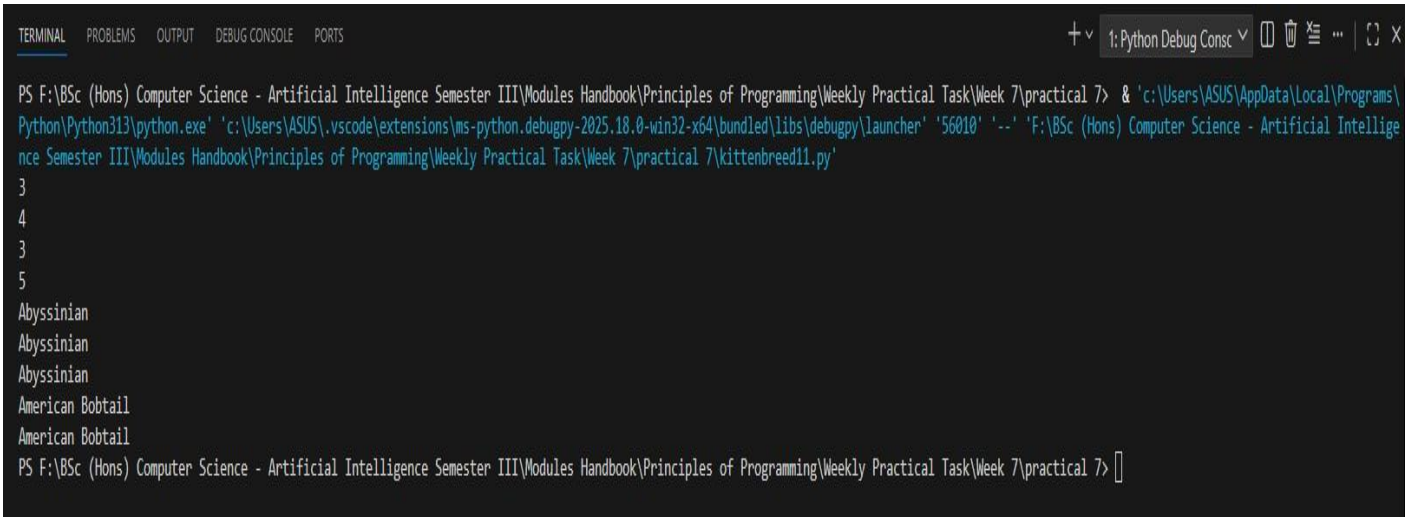
```
# Create two kitten objects using parameterized constructor
kitt = Kitten(3)
kitt2 = Kitten(4)

# Display the inital ages
kitt.display_age()
kitt2.display_age()
kitt.display_age()

# Change the age of kitt using setter method
kitt.set_age(5)
kitt.display_age()

# Print the class variable breed
print(kitt.breed)
print(kitt2.breed)
print(Kitten.breed)
```

```
# print(Kitten.age) # instance variable cannot be accessed via Class.
# It would raise AttributeError.

# Modify the class variable
Kitten.breed = 'American Bobtail'

# Print the change in variable
print(kitt.breed)
print(kitt2.breed)

# Output:
# 3
# 4
# 3
# 5
# Abyssinian
# Abyssinian
# Abyssinian
```

American Bobtail

American Bobtail

Output obtained in execution:



Exercise 3: Implement the following Rectangle class.

12. Design a class named Rectangle to represent a rectangle. The class contains:

- Two real data fields named width and height that specify the width and height of the rectangle. The default values are 1.0 for both width and height.
- A constructor that creates a rectangle with the specified width and height.

- A method named getArea() that returns the area of this rectangle.
- A method named getPerimeter() that returns the perimeter.

Write a test program that creates two Rectangle objects—one with width 4 and height 40 and the other with width 3.5 and height 35.9. Display the width, height, area, and perimeter of each rectangle in this order.

Answer:

Following code for input:

```
# Implement a Rectangle class to display the width, height, area, and perimeter
# Practical 7 - Exercise 3
# Nirvik K.C.


class Rectangle:
    """
    A class created to represent a rectangle with width and height.
    A method named getArea() that returns the area of this rectangle.
    A method named getPerimeter() that returns the perimeter.
    """
```

```
    def __init__(self, width = 1.0, height = 1.0):
        # Constructor to create a rectangle.
        self.width = width   # width of the rectangle is 1.0
        self.height = height # height of the rectangle is 1.0

    def getArea(self):
        # Returns the area of the rectangle
        return self.width * self.height

    def getPerimeter(self):
        # Returns the perimeter of the rectangle
        return 2 * (self.width + self.height)

# Create the first rectangle object (width 4, height 40)
rect1 = Rectangle(4, 40)

# Create the second rectangle object (width 3.5, height 35.9)
rect2 = Rectangle(3.5, 35.9)
```

```
def display_rectangle(name, rect):
    print(f"{name}:")
    print(f"Width: {rect.width}")
    print(f"Height: {rect.height}")
    print(f"Area: {rect.getArea():.2f}")
    print(f"Perimeter: {rect.getPerimeter():.2f}")
    print()

# Display the width, height, area, and perimeter of both rectangles
display_rectangle("Rectangle 1", rect1)
display_rectangle("Rectangle 2", rect2)

# Output:
# Rectangle 1:
# Width: 4
# Height: 40
# Area: 160.00
# Perimeter: 88.00
```

Rectangle 2:

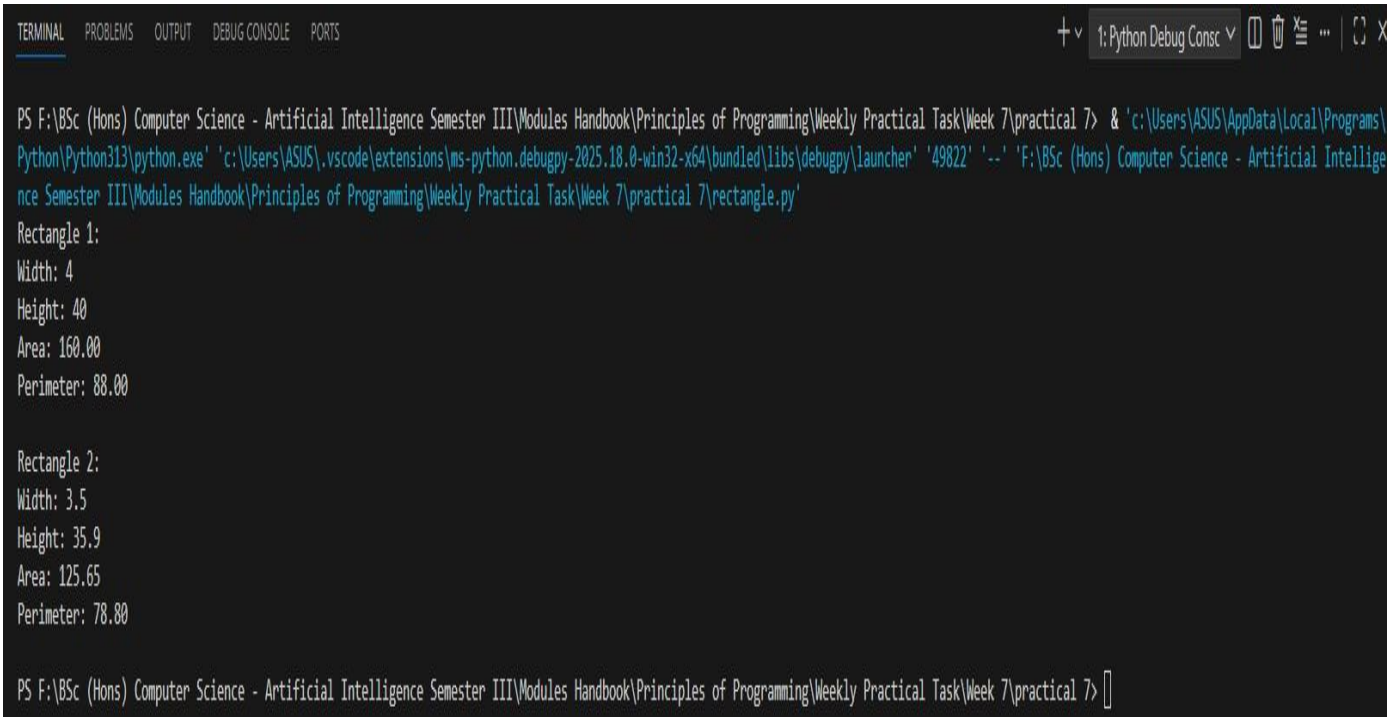
Width: 3.5

Height: 35.9

Area: 125.65

Perimeter: 78.80

Output obtained in execution:



Exercise 4: Implement the following Car class.

13. Design a class named Car to represent a car. The class contains:

- Two integer data fields representing number of doors and price of the car.
- Two string data fields representing color and brand of the car.
- A constructor that creates a car with the specified values of the data fields.
- A method named startCar() that prints "Car has started"
- A method named stopCar() that prints "Car has stopped"
- All the appropriate setters and getters methods

For example, method named setNumberOfDoors() that sets the number of doors of the car a method named getNumberOfDoors() that returns the number of doors of the car and so on.

Write a test program that creates a Car object—with number of door 5, price 20000, color White and brand ToyCar. Display the number of door, price, color and brand. By invoking appropriate methods show how your car can be started/stopped etc. Also, how you can change its price, color, brand etc.

Answer:

Following code for input:

```
# Design and implement Car class with data fields, constructor, methods, getters and setters

# Practical 7 - Exercise 4

# Nirvik K.C.

class Car:

    def __init__(self, doors, price, color, brand):

        """Constructor to initialise the attributes of the car"""

        self.doors = doors

        self.price = price

        self.color = color

        self.brand = brand


  # Methods for starting the car and stopping the car

    def startCar(self):

        print("Car has started")


    def stopCar(self):

        print("Car has stopped")
```

```
 # Using getter methods
 def getNumberOfDoors(self):
     return self.doors

 def getPrice(self):
     return self.price

def getColor(self):
     return self.color

 def getBrand(self):
     return self.brand

 # Using setter methods
 def setNumberOfDoors(self, doors):
     self.doors = doors
```

```
    def setPrice(self, price):
        self.price = price

    def setColor(self, color):
        self.color = color

    def setBrand(self, brand):
        self.brand = brand

# Create object for the Car
car_details = Car(5, 20000, "White", "ToyCar")
# Display the initial values using getters
print("Initial Car Details:")
print(f"Brand: {car_details.getBrand()}")
print(f"Doors: {car_details.getNumberOfDoors()}")
print(f"Price: {car_details.getPrice()}")
print(f"Color: {car_details.getColor()}")
print()
```

```
# To start and stop the car
car_details.startCar()
car_details.stopCar()
print()

# Change the attributes using setters
print("Updating new details for the Car")
car_details.setPrice(25000)
car_details.setColor("Red")
car_details.setBrand("SpeedToy")

# Display updated values
print("Updated Car details:")
print(f"New Brand: {car_details.getBrand()}")
print(f"New Price: {car_details.getPrice()}")
print(f"New Color: {car_details.getColor()}")
print()
```

```
# Start and stop the new car
car_details.startCar()
car_details.stopCar()
print()

# Output:
# Initial Car Details:
# Brand: ToyCar
# Doors: 5
# Price: 20000
# Color: White

# Car has started
# Car has stopped

# Updating new details for the Car
# Updated Car details:
# New Brand: SpeedToy
```

New Price: 25000

New Color: Red

Car has started

Car has stopped

Output obtained in execution: