Basic Programming with Python

Prepared By:

Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

8

OOP Concepts in Python

CONTENTS

8.9 CONSTRUCTOR IN PYTHON

In Python, a constructor is a special method within a class that is automatically called when a new instance of the class is created. The purpose of a constructor is to initialize the object's attributes.

In Python, the constructor method is named **__init__()**. It's defined within the class like any other method, but it always takes at least one parameter, typically named **self**, which refers to the instance of the class being created.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Creating the Constructor in Python

Creating a constructor in Python involves defining the **__init__()** method within a class. Here's a step-by-step guide to creating a constructor:

1. Define the class.
2. Define the **__init__()** method within the class.
3. Within the **__init__()** method, initialize the attributes of the class using the self keyword.

Here's an example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of the Person class
person1 = Person("Rahman", 30)

# Accessing attributes of the object
print(person1.name)  # Output: Rahman
print(person1.age)   # Output: 30
```

Counting the number of objects of a class

We can count the number of objects created from a class by utilizing the constructor method (__init__()). Inside the constructor, we can update a class variable each time a new object is instantiated.

Here's how we can do it:

```
class MyClass:
    count = 0       # Class variable

    def __init__(self):
        MyClass.count += 1          # Increment the count each time an object is created

# Creating instances of the MyClass class
obj1 = MyClass()
obj2 = MyClass()
obj3 = MyClass()

# Accessing the count of objects
print("Number of objects created:", MyClass.count)       # Output: 3
```

In this example, each time an object of MyClass is created, the __init__() method is called, and it increments the count class variable by 1. This way, we can keep track of the total number of objects created from the class.

8.9.1 Non-Parameterized Constructor in Python

In Python, a non-parameterized constructor, also known as a default constructor, is a constructor that doesn't accept any parameters other than self. It's called when an object of the class is instantiated without passing any arguments. Non-parameterized constructors are typically used when we don't need to initialize any attributes with specific values upon object creation.

Non-parameterized constructors are useful when we simply want to perform some initialization tasks common to all instances of the class without needing any specific parameters.

Here's an example of a non-parameterized constructor:

```
class Car:
    def __init__(self):
        print("A car object is created.")

# Creating an instance of the Car class
car1 = Car()

# Output: "A car object is created."
```

Example: Class with a message display in the constructor

```
class Greeting:
    def __init__(self):
        print("Hello, welcome!")

# Creating an instance of the Greeting class
greeting1 = Greeting()

# Output: "Hello, welcome!"
```

Example: Class with initialization of class variables in a non-parameterized constructor

```
class Circle:
    def __init__(self):
        self.radius = 5                    # Default radius for all circle objects
        self.area = 3.14 * self.radius ** 2

# Creating an instance of the Circle class
circle1 = Circle()

# Accessing attributes
print("Radius:", circle1.radius)           # Output: 5
print("Area:", circle1.area)               # Output: 78.5
```

Example: Class with non-parameterized constructor performing some initialization tasks

```
class Person:
    def __init__(self):
```

```
        self.name = "MM Rahman"
        self.age = 30
        self.gender = "Male"
        print("A person object is created with default values.")

# Creating an instance of the Person class
person1 = Person()

# Accessing attributes
print("Name:", person1.name)            # Output: MM Rahman
print("Age:", person1.age)              # Output: 30
print("Gender:", person1.gender)        # Output: Male
```

Example: Class with default values for attributes

```
class Rectangle:
    def __init__(self):
        self.width = 10
        self.height = 5

# Creating an instance of the Rectangle class
rectangle1 = Rectangle()

# Accessing attributes
print("Width:", rectangle1.width)   # Output: 10
print("Height:", rectangle1.height) # Output: 5
```

Example: Class with initialization of a list attribute

```
class ShoppingCart:
    def __init__(self):
        self.items = []  # Initializing an empty list for items

    def add_item(self, item):
        self.items.append(item)

# Creating an instance of the ShoppingCart class
cart1 = ShoppingCart()

# Adding items to the shopping cart
cart1.add_item("Apple")
cart1.add_item("Banana")

# Accessing items in the cart
print("Items in the cart:", cart1.items)            # Output: ['Apple', 'Banana']
```

8.9.2 Parameterized Constructor in Python

In Python, a parameterized constructor, also known as an initialized constructor, is a constructor that accepts parameters other than just self. It allows you to initialize the attributes of an object with specific values passed during object creation. Parameterized constructors are useful when you want to customize the initialization of objects based on different input values.

Example: Simple class with a parameterized constructor

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of the Person class with specific values
person1 = Person("Sumi", 25)

# Accessing attributes
print("Name:", person1.name)          # Output: Sumi
print("Age:", person1.age)            # Output: 25
```

Example: Class with validation in the parameterized constructor

```
class Student:
    def __init__(self, name, age):
        if age < 0:
            raise ValueError("Age cannot be negative.")
        self.name = name
        self.age = age

# Creating an instance of the Student class with validation
try:
    student1 = Student("Sumi", -20)
except ValueError as e:
    print(e)                          # Output: "Age cannot be negative."
```

Example: Class with multiple parameters in the constructor

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

# Creating an instance of the Point class with specific coordinates
point1 = Point(3, 5)

# Accessing attributes
print("X coordinate:", point1.x)  # Output: 3
print("Y coordinate:", point1.y)  # Output: 5
```

Example: Class with default values and parameterized constructor

```
class Rectangle:
    def __init__(self, length=1, width=1):
        self.length = length
        self.width = width

# Creating instances of the Rectangle class with different parameters
rectangle1 = Rectangle()
rectangle2 = Rectangle(5)
rectangle3 = Rectangle(3, 4)

# Accessing attributes
print("Rectangle 1:", rectangle1.length, rectangle1.width)  # Output: 1 1
print("Rectangle 2:", rectangle2.length, rectangle2.width)  # Output: 5 1
print("Rectangle 3:", rectangle3.length, rectangle3.width)  # Output: 3 4
```

Example: Class with initialization of a list attribute in the constructor

```
class ShoppingList:
    def __init__(self, items):
        self.items = items

# Creating an instance of the ShoppingList class with a list of items
shopping_list1 = ShoppingList(["Apples", "Bananas", "Oranges"])

# Accessing the list attribute
print("Items in the shopping list:", shopping_list1.items)  # Output: ['Apples', 'Bananas', 'Oranges']
```

8.9.3 Multiple Constructors in Single class

In Python, a class can't have more than one constructor in the traditional sense. However, we can achieve similar functionality by using default parameter values or by using class methods as alternative constructors.

Example using Default Parameter Values:

```
class MyClass:
    def __init__(self, x=None, y=None):
        if x is not None and y is not None:
            self.x = x
            self.y = y
        else:
            self.x = 0
            self.y = 0

# Creating instances of MyClass with different initialization
obj1 = MyClass()
```

```
obj2 = MyClass(3, 5)

print("Object 1:", obj1.x, obj1.y)  # Output: Object 1: 0 0
print("Object 2:", obj2.x, obj2.y)  # Output: Object 2: 3 5
```

Example using Class Methods as Alternative Constructors:

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def from_string(cls, str_val):
        x, y = map(int, str_val.split(","))
        return cls(x, y)

# Creating instances of MyClass using different constructors
obj1 = MyClass(3, 5)            # Using the default constructor
obj2 = MyClass.from_string("6,8")   # Using the class method as an alternative constructor

print("Object 1:", obj1.x, obj1.y)  # Output: Object 1: 3 5
print("Object 2:", obj2.x, obj2.y)  # Output: Object 2: 6 8
```

8.9.4 Instance Variable and Instance Method

In Python, instance variables and instance methods are associated with instances of a class. They are unique to each individual object created from the class.

Instance variables and instance methods play a crucial role in object-oriented programming in Python, allowing for encapsulation and defining behavior specific to individual instances of a class.

Instance Variable

An instance variable is a variable that is defined within a class and belongs to each instance of that class individually. Each object or instance of the class has its own copy of the instance variables. These variables are accessed using the syntax **self.variable_name** within instance methods.

Example: Instance Variables

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

# Creating instances of the Circle class
circle1 = Circle(5)
circle2 = Circle(3)
```

```
# Accessing instance variables
print("Radius of circle1:", circle1.radius)  # Output: Radius of circle1: 5
print("Radius of circle2:", circle2.radius)  # Output: Radius of circle2: 3
```

In this example, **radius** is an instance variable of the Circle class. Each instance of the Circle class (circle1 and circle2) has its own radius instance variable.

Instance Method

An instance method is a function defined within a class that operates on instances of that class. It can access and modify instance variables and perform operations specific to each instance. Instance methods are defined using the **def** keyword within the class and always take **self** as the first parameter, which refers to the instance itself.

Example: Instance Methods

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

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

# Creating an instance of the Rectangle class
rectangle1 = Rectangle(4, 5)

# Calling the instance method
area = rectangle1.calculate_area()
print("Area of rectangle1:", area)  # Output: Area of rectangle1: 20
```

In this example, calculate_area() is an instance method of the Rectangle class. It calculates the area of the rectangle using the length and width instance variables of the object on which it is called.

⌘⌘⌘