Chapter 3 - Constructors and Destructors

1. Introduction

In object-oriented programming, **constructors** and **destructors** are special methods that handle object initialization and cleanup.

- The **constructor** (`__init__()`) is called when an object is created.
- The **destructor** (`__del__()`) is called when an object is deleted or goes out of scope.

These methods allow us to manage resources, such as database connections, files, or memory, effectively.

---

2. Constructors (`__init__()`)

2.1 What is a Constructor?

A **constructor** is a special method that initializes an object's attributes when it is created. In Python, the constructor is the `__init__()` method.

2.2 Syntax and Example

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        print(f"{self.brand} {self.model} created!")

car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)
```

Output:

```
Toyota Corolla created!
Honda Civic created!
```

2.3 Constructor Overloading (Using Default Values)

Python does not support **method overloading**, but we can achieve similar behavior using default values.

```
class Laptop:
    def __init__(self, brand="Unknown", model="Unknown", price=0):
        self.brand = brand
        self.model = model
        self.price = price

l1 = Laptop("Dell", "XPS", 1200)
l2 = Laptop()  # Uses default values

print(l1.brand, l1.model, l1.price)  # Output: Dell XPS 1200
print(l2.brand, l2.model, l2.price)  # Output: Unknown Unknown 0
```

2.4 Using `super().__init__()` in Inheritance

If a class inherits from another class, we use `super().__init__()` to call the parent class's constructor.

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand)  # Calls parent constructor
        self.model = model
        self.year = year

car1 = Car("Ford", "Mustang", 2022)
print(car1.brand, car1.model, car1.year)  # Output: Ford Mustang 2022
```

---

3. Destructors (`__del__()`)

3.1 What is a Destructor?

A **destructor** is a special method that is automatically called when an object is destroyed. It helps in **resource management**, like closing database connections or file handles.

3.2 Syntax and Example

```
class Person:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} is created!")

    def __del__(self):
        print(f"{self.name} is deleted!")

p1 = Person("Alice")
del p1  # Manually deleting object
```

Output:

```
Alice is created!
Alice is deleted!
```

3.3 When is the Destructor Called?

- When an object **goes out of scope** (e.g., at the end of a function).
- When we **explicitly delete** an object using `del`.
- When **Python's garbage collector** removes an unreferenced object.

Example:

```
def create_object():
    obj = Person("John")  # Object created
    return obj

p = create_object()
print("Function ended")
```

Output:

```
John is created!
Function ended
John is deleted!  # Destructor called automatically
```

3.4 Using Destructor to Release Resources

Destructors are useful for managing **files, database connections, or network resources**.

```
class FileHandler:
    def __init__(self, filename):
        self.file = open(filename, "w")
        print(f"File {filename} opened.")

    def write_data(self, data):
        self.file.write(data)

    def __del__(self):
        self.file.close()
        print("File closed.")

# Using the class
file = FileHandler("test.txt")
file.write_data("Hello, World!")
del file  # File closed automatically
```

---

4. Constructors and Destructors in Django

4.1 Using `__init__()` in Django Models

In Django, the `__init__()` method is used to **customize** object creation.

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print(f"Product {self.name} created!")

# Creating an object in Django shell
# python manage.py shell
# >>> p = Product(name="Laptop", price=1200)
# >>> p.save()
# Output: Product Laptop created!
```

4.2 Destructor in Django

Django does not directly use destructors like in Python classes because Django models interact with the database. However, we can use **signals** to execute actions before deletion.

Example using the `pre_delete` signal:

```
from django.db.models.signals import pre_delete
from django.dispatch import receiver

@receiver(pre_delete, sender=Product)
def product_deleting(sender, instance, **kwargs):
    print(f"Deleting product: {instance.name}")

# When deleting a product:
# >>> p.delete()
# Output: Deleting product: Laptop
```

---

5. Exercises

Exercise 1: Implement a Constructor

Create a class `Book` with attributes `title`, `author`, and `price`. Initialize these attributes using a constructor and create an object.

Exercise 2: Implement a Destructor

Modify the `Book` class by adding a destructor that prints `"Book deleted!"` when an object is destroyed.

Exercise 3: Using `super().__init__()`

Create a class `Electronic` with an attribute `category`. Create a subclass `Phone` that inherits from `Electronic` and uses `super().__init__()`.

Exercise 4: Manage a File with a Class

Create a Python class called `FileWriter` that:

1. **Opens a file** in write mode when an object is created. The filename should be passed as an argument to the constructor.

2. **Has a method** `write_data(self, data)` that writes the given text to the file.
3. **Closes the file automatically** when the object is deleted (in the destructor).
4. **Demonstrates the behavior** by creating an instance, writing some data, and then deleting the object.