Accessing Attributes, Constructors and Destructors in Python

What Are Attributes?

An attribute is a variable that belongs to an object or a class. In simpler terms: Attributes store data or state about an object.

Types of Attributes

In Python, there are mainly **two kinds of attributes**:

Type	Belongs To	Defined	Accessed Using
Instance Attributes	A specific object (instance)	Inside the constructor (__init__) using self	object.attribute
Class Attributes	The class itself (shared by all objects)	Inside the class, but **outside** any method	ClassName.attribute or object.attribute

Example:

```
class Student:
    # Define an empty class first
    pass

# Create two objects
s1 = Student()
s2 = Student()

# Add attributes manually (outside the class)
s1.name = "Alice"
s1.age = 18

s2.name = "Bob"
s2.age = 20

# Access attributes
print(s1.name)   # Alice
print(s1.age)    # 18

print(s2.name)   # Bob
print(s2.age)    # 20
```

Accessing Attributes, Constructors and Destructors in Python

Explanation:
- class Student: → defines the class name.
- pass → means "do nothing for now" (a placeholder).
- s1 and s2 → are objects (instances) of Student.
- We manually add attributes to each object after creation:

How This Works?
In Python, classes are flexible — you can create new attributes **dynamically** (even outside the class).
So when you do s1.name = "Alice", Python simply **creates** a new variable called name inside the object s1.

How Attributes Are Created?
Attributes can be created in a few ways:

Where	Example	Explanation
Inside __init__	self.name = "Alice"	Most common — created when object is initialized
Inside any method	self.score = 90	Can be created later by any method
Directly from outside	obj.new_attr = value	Python allows adding attributes dynamically

Constructors and Destructors in Python

What is a Constructor?
A **constructor** is a special method that automatically runs **when you create (instantiate)** an object of a class.
It's used to **initialize** the object — that is, to give values to its attributes or perform setup tasks.

In Python, the constructor method is called:
__init__()

Accessing Attributes, Constructors and Destructors in Python

Syntax of a Constructor

```
class ClassName:
    def __init__(self, parameters):
        # initialization code
```

- __init__ → special method name (double underscores before and after)
- self → refers to the current instance (object) of the class
- You can pass additional parameters to set up object data

Simple Example of a Constructor

```
class Student:
    def __init__(self, name, age):
        # This runs automatically when a new Student is created
        self.name = name
        self.age = age
        print("Constructor is called! Student object created.")

# Creating objects
s1 = Student("Ashima", 20)
s2 = Student("Bobby", 22)

print(s1.name, s1.age)
print(s2.name, s2.age)
```

Output:

```
Constructor is called! Student object created.
Constructor is called! Student object created.
Ashima 20
Bobby 22
```

Explanation:

- Each time you create a new object, __init__() runs automatically.
- self.name and self.age store data specific to each object.

Types of Constructors in Python

Python doesn't formally define types like C++ or Java, but conceptually:

Type	Description	Example
Default constructor	Takes only self as a parameter, does basic initialization	def __init__(self):
Parameterized constructor	Takes additional arguments to initialize the object	def __init__(self, x, y):

Accessing Attributes, Constructors and Destructors in Python

What is the Dot (.) Operator in Python?

The **dot (.)** is called the **member access operator** or **attribute access operator**.
It is used to **access attributes** (variables) and **methods** (functions) that belong to an **object**, **class**, or **module**.
Think of it like saying:
"Go inside this object and get that thing."

General Syntax:

```
object.attribute
object.method()
```

- object → the instance or class name
- attribute → variable that belongs to the object/class
- method() → function that belongs to the object/class

Example: Dot Operator with a Class

```
class Student:
    def __init__(self, name, age):
        self.name = name   # "name" is an attribute
        self.age = age     # "age" is another attribute

    def display(self):     # "display" is a method
        print("Name:", self.name)
        print("Age:", self.age)

# Creating an object
s1 = Student("Zoya", 20)

# Using dot operator
print(s1.name)      # Access attribute
print(s1.age)       # Access attribute
s1.display()        # Call method
```

Output:

```
Zoya
20
Name: Zoya
Age: 20
```

Explanation:

- s1.name → accesses the variable name inside the object s1
- s1.display() → calls the display method belonging to s1

Accessing Attributes, Constructors and Destructors in Python

Example: Default vs Parameterized Constructor

```
class Example:
    # Default constructor
    def __init__(self):
        self.message = "Hello!"
        print("Default constructor called")

class Example2:
    # Parameterized constructor
    def __init__(self, name):
        self.name = name
        print("Parameterized constructor called")

obj1 = Example()
obj2 = Example2("Python")
```

Explanation:

1. **Example class**
   - Has a **default constructor** — it doesn't take any extra arguments.
   - When obj1 = Example() runs:
     - The constructor runs automatically.
     - It prints "Default constructor called".
     - Sets obj1.message = "Hello!".
2. **Example2 class**
   - Has a **parameterized constructor** — it takes one argument (name).
   - When obj2 = Example2("Python") runs:
     - The constructor runs automatically.
     - It prints "Parameterized constructor called".
     - Sets obj2.name = "Python".

Output:

```
Default constructor called
Parameterized constructor called
```

What is a Destructor?

A **destructor** is a special method that is automatically called **when an object is about to be destroyed** (i.e., when it is no longer in use or goes out of scope).
In Python, the destructor method is:

```
__del__()
```

However, destructors are **rarely needed** in Python because of **automatic garbage collection** — Python frees memory automatically when objects aren't referenced anymore.

Accessing Attributes, Constructors and Destructors in Python

Syntax of a Destructor

```
class ClassName:
    def __del__(self):
        # cleanup code
```

Example of a Destructor

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

    def __del__(self):
        print(f"Destructor called. Employee {self.name} deleted")

# Create object
emp1 = Employee("Jigyasa")

# Delete the object manually
del emp1
print("Program end")
```

Output:

```
Employee Jigyasa created
Destructor called. Employee Jigyasa deleted
Program end
```

Explanation:

- __init__() is called when the object is created.
- __del__() is called automatically when the object is deleted or garbage-collected.

Important Notes on Destructors

- Python's **garbage collector** decides *when* to actually destroy the object — so __del__() might not run immediately after del.
- Avoid relying heavily on __del__() for important cleanup (use **context managers** or the with statement instead).

Summary Table

Concept	Special Method	When Called	Purpose
Constructor	__init__(self, ...)	When object is created	Initialize object state
Destructor	__del__(self)	When object is destroyed	Clean up resources

Accessing Attributes, Constructors and Destructors in Python

Example: Bank Account

Let's create a class that represents a simple **bank account**.
We'll use:

- A **constructor (__init__)** to open the account (initialize it).
- A **destructor (__del__)** to close the account (cleanup).

Code:

```
class BankAccount:
    # Constructor
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        print(f"Account created for {self.name} with balance {self.balance}")

    # Deposit method
    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New balance: {self.balance}")

    # Withdraw method
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"{amount} withdrawn. Remaining balance: {self.balance}")
        else:
            print("Insufficient balance!")

    # Destructor
    def __del__(self):
        print(f"Account of {self.name} is now closed.")


# --- Using the class ---
acc1 = BankAccount("Alice", 5000)  # Constructor runs here

acc1.deposit(1000)
acc1.withdraw(2000)

del acc1    # Destructor runs here
```

Step-by-Step Explanation:

1. **Constructor (__init__)**
   - Called automatically when you create an object.
   - It sets the owner's name and starting balance.

Accessing Attributes, Constructors and Destructors in Python

- Prints:

"Account created for Alice with balance 5000"

2. **Deposit and Withdraw**
   - These are regular methods.
   - You can call them to change the account balance.
3. **Destructor (__del__)**
   - Called automatically when the object is deleted or goes out of use.
   - Prints:

"Account of Alice is now closed."

Output:

```
Account created for Alice with balance 5000
1000 deposited. New balance: 6000
2000 withdrawn. Remaining balance: 4000
Account of Alice is now closed.
```