A logo of a university

Description automatically generated

**كلية العلوم**

**قــســــــــــم الانـــظــــمــــة الــــطـبـيـة الـــذكــــــيـــة**

**Lecture: ( 7 )**

***(Constructor & Destructor)***

***(*** ***Encapsulation and abstraction)***

**Subject: Object oriented programming II**

**Class: Second**

**Lecturer:** **Dr. Maytham N. Meqdad**

**Constructor & Destructor in Python**

Constructor & Destructor in Python:

Constructor & Destructor are an important concept of oops in Python.

Constructor: A constructor in Python is a special type of method which is used to initialize the instance members of the class. The task of constructors is to initialize and assign values to the data members of the class when an object of the class is created.

Destructor: Destructor in Python is called when an object gets destroyed. In Python, destructors are not needed, because Python has a garbage collector that handles memory management automatically.

**Constructor:**

* The [\_\_init\_\_](https://prepinsta.com/?page_id=694374&preview=true) method is similar to **constructors**in c++ and Java.
* Constructors are used to initialize the object’s state.
* The task of constructors is to initialize(assign values) to the data members of the class when an object of class is created.

**Synatx:**

class ClassName:  
 def \_\_init\_\_( self , variables...):  
 ##body

**Types of Constructor:**

* **default constructor:**The default constructor is a simple constructor which doesn’t have any argument to pass. Its definition has only one argument which is a reference to the instance being constructed.
* **parameterized constructor:**constructor which has parameters to pass is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self.

**Code #1:**

#python Program

#Rishikesh

#constructor

#default Constructor

class A(object):

    def \_\_init\_\_(self):

        self.str1 = “PrepInsta”

        print( self.str1)

        print(‘In constructor’)

ob = A()

**Output:**

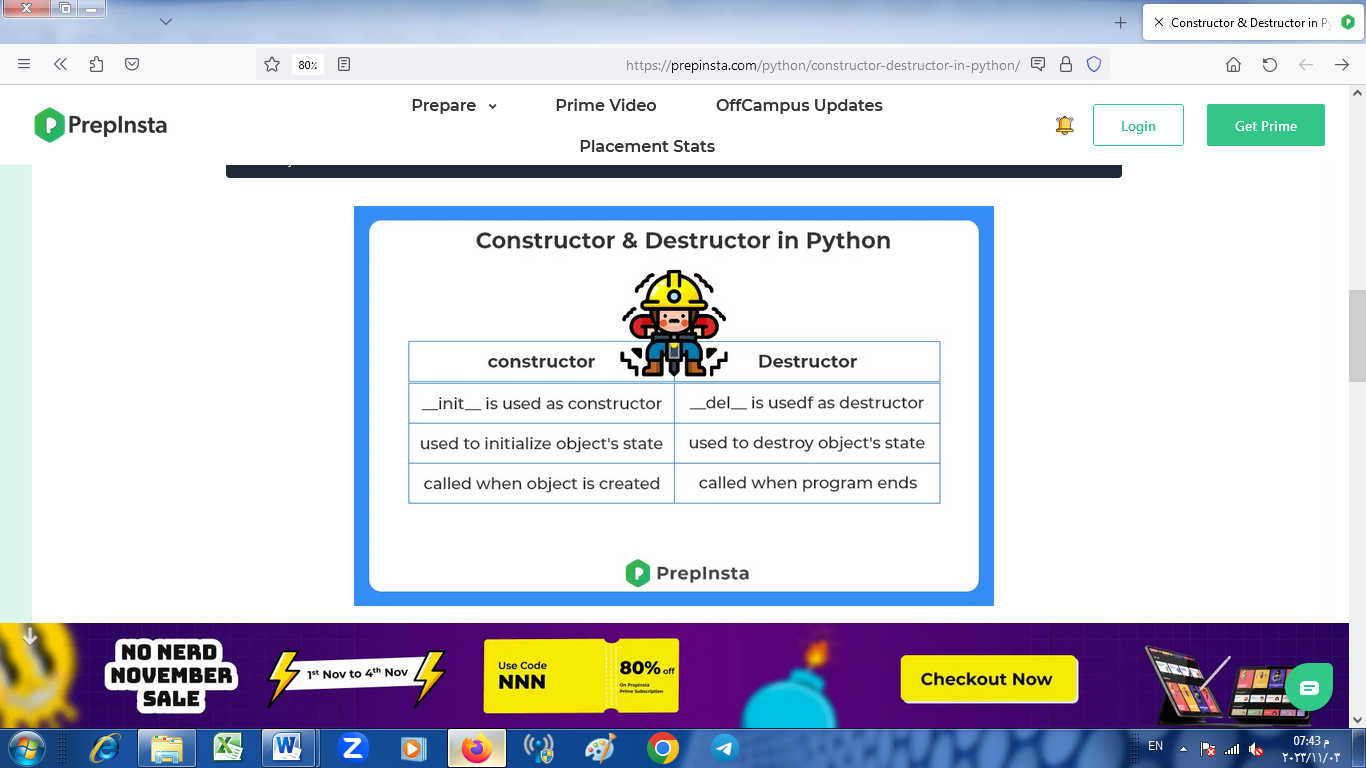
PrepInsta  
In constructor

**Destructor:**

* The \_\_del\_\_ method is similar to **destructor**in c++ and Java.
* Destructors are used to destroying the object’s state.

**Syntax:**

class ClassName:  
 def \_\_del\_\_( self ,):  
 ##body



* This program that demonstrates constructors and resource management in Python, inspired by a simple library system:

class Book:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

self.checked\_out = False

def check\_out(self):

if not self.checked\_out:

self.checked\_out = True

return True

else:

return False

def check\_in(self):

if self.checked\_out:

self.checked\_out = False

return True

else:

return False

def \_\_str\_\_(self):

return f"{self.title} by {self.author} - Checked Out: {self.checked\_out}"

# Create two book instances

book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")

book2 = Book("To Kill a Mockingbird", "Harper Lee")

# Check out and check in books

book1.check\_out()

book2.check\_out()

# Display book information

print(book1)

print(book2)

In this program:

* We have a Book class with a constructor that initializes book attributes, including title, author, and whether the book is checked out or not.
* The check\_out method allows us to check out a book, and the check\_in method allows us to return it.
* The \_\_str\_\_ method is used to create a string representation of the book, indicating its title, author, and whether it's checked out.
* We create two book instances, check them out, and then display their information.

This simple program illustrates how you can use constructors and methods to manage book objects in a basic library system.

**Constructors in Python**

Constructors in Python are used to initialize the attributes of an object with default or given values. They are defined with the \_\_init\_\_() method and are automatically called when an object is created.

Python

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Create a new Person object

person = Person("Bard", 1)

# Call the greet() method

person.greet()

Output:

Hello, my name is Bard and I am 1 years old.

**Destructors in Python**

Destructors in Python are used to clean up any resources that an object is using before it is destroyed. They are defined with the \_\_del\_\_() method and are automatically called when an object is deleted.

class File:

def \_\_init\_\_(self, filename):

self.file = open(filename, "w")

def write(self, data):

self.file.write(data)

def close(self):

self.file.close()

def \_\_del\_\_(self):

self.close()

# Create a new File object

file = File("myfile.txt")

# Write some data to the file

file.write("This is some data to write to the file.")

# Close the file

file.close()

# Delete the file object

del file

Even though we explicitly called the close() method on the file object, the \_\_del\_\_() method will still be called when the object is deleted. This ensures that the file is always closed, even if we forget to do it explicitly.

**Use cases for constructors and destructors**

Constructors and destructors are useful for a variety of tasks, such as:

* **Initializing object attributes:** Constructors can be used to initialize the attributes of an object with default or given values. This ensures that all objects of the class are properly initialized, regardless of how they are created.
* **Cleaning up resources:** Destructors can be used to clean up any resources that an object is using before it is destroyed. This can help to prevent memory leaks and other problems.
* **Enforcing encapsulation:** Constructors and destructors can be used to enforce encapsulation by hiding the internal implementation of a class from its users.

Overall, constructors and destructors are powerful features of Python that can be used to improve the quality and maintainability of your code.

**Encapsulation and abstraction** are two fundamental principles in object-oriented programming (OOP). They help in organizing and managing code, making it more understandable and maintainable. In Python, like in many other OOP languages, you can apply encapsulation and abstraction as follows:

1. **Encapsulation**:

Encapsulation is the practice of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, known as a class. This helps in controlling access to the data and ensures that the data is used and modified in a controlled manner. In Python, encapsulation is implemented by using private and protected members.

* + **Private Members**: In Python, you can mark an attribute or method as private by prefixing it with an underscore (e.g., \_variable or \_method). This is a convention and not enforced by the language, but it indicates to other developers that the attribute or method is intended for internal use and should not be accessed directly.

 class MyClass:

def \_\_init\_\_(self):

self.\_my\_variable = 10

def \_my\_method(self):

return self.\_my\_variable

 **Protected Members**: Python doesn't have a strict concept of protected members, but you can prefix an attribute or method with a double underscore (e.g., \_\_variable). This will name-mangle the member, making it harder (but not impossible) to access from outside the class.

class MyClass:

def \_\_init\_\_(self):

self.\_\_my\_variable = 10

def \_\_my\_method(self):

return self.\_\_my\_variable

 Abstraction:

Abstraction is the process of simplifying complex reality by modeling classes based on the essential attributes and behaviors of an object, while hiding the unnecessary details. In Python, you can achieve abstraction by defining classes with well-defined interfaces, i.e., a set of public methods that describe how the class can be used, while keeping the implementation details hidden.

class Shape:

def area(self):

pass

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius

class Rectangle(Shape):

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area(self):

return self.width \* self.height

* In the example above, the Shape class defines an interface with an area method that every shape subclass must implement. The subclasses Circle and Rectangle provide their own implementations, but the details of those implementations are hidden from the user of the classes.
* In summary, encapsulation and abstraction in Python are achieved through conventions for controlling access to class members and by defining well-defined interfaces that hide the implementation details. These principles help make your code more maintainable and understandable.
* This example program that demonstrates encapsulation and abstraction in Python, using a simple banking system:

class Account:

def \_\_init\_\_(self, account\_number, account\_holder, balance=0):

self.\_account\_number = account\_number # Encapsulated as a protected attribute

self.\_account\_holder = account\_holder # Encapsulated as a protected attribute

self.\_balance = balance # Encapsulated as a protected attribute

def deposit(self, amount):

if amount > 0:

self.\_balance += amount

print(f"Deposited ${amount}. New balance: ${self.\_balance}")

else:

print("Invalid deposit amount.")

def withdraw(self, amount):

if 0 < amount <= self.\_balance:

self.\_balance -= amount

print(f"Withdrew ${amount}. New balance: ${self.\_balance}")

else:

print("Invalid withdrawal amount or insufficient funds.")

def get\_balance(self):

return self.\_balance # Encapsulation allows controlled access

def account\_info(self):

return f"Account: {self.\_account\_number}, Holder: {self.\_account\_holder}, Balance: ${self.\_balance}"

class SavingsAccount(Account):

def \_\_init\_\_(self, account\_number, account\_holder, balance=0, interest\_rate=0.01):

super().\_\_init\_\_(account\_number, account\_holder, balance)

self.\_interest\_rate = interest\_rate

def apply\_interest(self):

interest = self.\_balance \* self.\_interest\_rate

self.\_balance += interest

print(f"Interest applied: ${interest}. New balance: ${self.\_balance}")

# Create a savings account

savings\_account = SavingsAccount("12345", "John Doe", 1000)

# Perform transactions and display account information

print("Account information:")

print(savings\_account.account\_info())

savings\_account.deposit(500)

savings\_account.withdraw(200)

savings\_account.apply\_interest()

print("Updated account information:")

print(savings\_account.account\_info())

In this program:

* The Account class encapsulates account details such as account\_number, account\_holder, and balance. These attributes are marked as protected. The class provides methods to deposit, withdraw, get the balance, and retrieve account information.
* The SavingsAccount class is a subclass of Account that adds an interest\_rate attribute and a method to apply interest. It inherits the encapsulated attributes and behaviors from the base class.
* We create a SavingsAccount instance, perform transactions (deposit, withdrawal, and interest application), and display account information.

This example demonstrates how encapsulation is used to protect attributes, and abstraction is achieved by providing an abstracted interface for interacting with the bank accounts. The specific implementation details are hidden from the user of the class, making it easier to use and maintain.

* This example program that demonstrates encapsulation and abstraction in Python, using a simplified employee management system:

class Employee:

def \_\_init\_\_(self, employee\_id, name):

self.\_employee\_id = employee\_id # Encapsulated as a protected attribute

self.\_name = name # Encapsulated as a protected attribute

self.\_salary = 0 # Encapsulated as a protected attribute

def calculate\_salary(self):

pass # Abstract method, to be defined in subclasses

def get\_employee\_id(self):

return self.\_employee\_id

def get\_name(self):

return self.\_name

def get\_salary(self):

return self.\_salary

def employee\_info(self):

return f"Employee ID: {self.\_employee\_id}, Name: {self.\_name}, Salary: ${self.\_salary}"

class Manager(Employee):

def calculate\_salary(self):

self.\_salary = 50000

class Developer(Employee):

def \_\_init\_\_(self, employee\_id, name, programming\_language):

super().\_\_init\_\_(employee\_id, name)

self.\_programming\_language = programming\_language

def calculate\_salary(self):

self.\_salary = 60000

def get\_programming\_language(self):

return self.\_programming\_language

# Create employees and display their information

manager = Manager("1", "Alice")

developer = Developer("2", "Bob", "Python")

print("Employee information:")

print(manager.employee\_info())

print(developer.employee\_info())

# Calculate and display salaries

manager.calculate\_salary()

developer.calculate\_salary()

print("\nUpdated employee information:")

print(manager.employee\_info())

print(developer.employee\_info())

In this program:

* The Employee class encapsulates employee details such as employee\_id, name, and salary. These attributes are marked as protected. The class provides methods to calculate the salary, get employee information, and retrieve individual attributes.
* The Manager and Developer classes are subclasses of Employee that override the calculate\_salary method to set the salary based on their roles. The Developer class also has an additional attribute, programming\_language.
* We create instances of Manager and Developer, display their information, calculate salaries, and display the updated information.

This example illustrates how encapsulation is used to protect attributes, and abstraction is achieved by providing an abstracted interface for interacting with employee objects. The specific implementation details are hidden from the user of the class, making it easier to use and maintain.

* This program that demonstrates encapsulation and abstraction in Python by modeling a school's student and teacher information:

class Person:

def \_\_init\_\_(self, name, age):

self.\_name = name # Encapsulated as a protected attribute

self.\_age = age # Encapsulated as a protected attribute

def get\_name(self):

return self.\_name

def get\_age(self):

return self.\_age

def introduce(self):

pass # Abstract method, to be defined in subclasses

class Student(Person):

def \_\_init\_\_(self, name, age, student\_id):

super().\_\_init\_\_(name, age)

self.\_student\_id = student\_id # Encapsulated as a protected attribute

def introduce(self):

return f"Hi, I'm {self.\_name}, a student with ID {self.\_student\_id}."

class Teacher(Person):

def \_\_init\_\_(self, name, age, employee\_id):

super().\_\_init\_\_(name, age)

self.\_employee\_id = employee\_id # Encapsulated as a protected attribute

def introduce(self):

return f"Hello, I'm {self.\_name}, a teacher with employee ID {self.\_employee\_id}."

# Create students and teachers and display their information

student1 = Student("Alice", 18, "S12345")

student2 = Student("Bob", 17, "S67890")

teacher1 = Teacher("Ms. Johnson", 35, "T101")

teacher2 = Teacher("Mr. Smith", 42, "T202")

print("Student and teacher information:")

print(student1.introduce())

print(student2.introduce())

print(teacher1.introduce())

print(teacher2.introduce())

In this program:

* The Person class encapsulates personal information such as name and age. These attributes are marked as protected. The class provides methods to get these attributes and an abstract introduce method.
* The Student and Teacher classes are subclasses of Person that provide concrete implementations of the introduce method and have additional attributes (student\_id and employee\_id) specific to their roles.
* We create instances of Student and Teacher, display their information by calling the introduce method, and demonstrate encapsulation by using protected attributes for name, age, student\_id, and employee\_id.

This example showcases encapsulation by protecting attributes and abstraction by providing an abstracted interface for interacting with person objects. It hides implementation details, making it easier to use and maintain.

* This program that demonstrates encapsulation and abstraction in Python, simulating a simple online shopping system with products and customers:

class Product:

def \_\_init\_\_(self, product\_id, name, price):

self.\_product\_id = product\_id # Encapsulated as a protected attribute

self.\_name = name # Encapsulated as a protected attribute

self.\_price = price # Encapsulated as a protected attribute

def get\_product\_id(self):

return self.\_product\_id

def get\_name(self):

return self.\_name

def get\_price(self):

return self.\_price

def product\_info(self):

return f"Product: {self.\_name} (ID: {self.\_product\_id}), Price: ${self.\_price}"

class Customer:

def \_\_init\_\_(self, customer\_id, name):

self.\_customer\_id = customer\_id # Encapsulated as a protected attribute

self.\_name = name # Encapsulated as a protected attribute

self.\_cart = [] # Encapsulated as a protected attribute

def add\_to\_cart(self, product):

self.\_cart.append(product)

return f"{self.\_name} added {product.get\_name()} to the cart."

def checkout(self):

total\_price = sum(product.get\_price() for product in self.\_cart)

self.\_cart = []

return f"{self.\_name} checked out. Total price: ${total\_price}"

def customer\_info(self):

return f"Customer: {self.\_name} (ID: {self.\_customer\_id})"

# Create products and customers

product1 = Product(1, "Laptop", 800)

product2 = Product(2, "Headphones", 50)

customer1 = Customer(101, "Alice")

customer2 = Customer(102, "Bob")

# Interaction

print("Online Shopping System:")

print(product1.product\_info())

print(product2.product\_info())

print(customer1.customer\_info())

print(customer2.customer\_info())

print(customer1.add\_to\_cart(product1))

print(customer2.add\_to\_cart(product2))

print(customer1.checkout())

print(customer2.add\_to\_cart(product1))

print(customer2.checkout())

In this program:

* The Product class encapsulates information about products, including their product\_id, name, and price. The attributes are marked as protected.
* The Customer class encapsulates information about customers, including their customer\_id, name, and a shopping cart. The attributes are marked as protected.
* Both classes provide methods for adding products to the cart, checking out, and providing information about the products and customers.
* The program simulates interactions between customers and products, demonstrating encapsulation by protecting attributes and abstraction by providing an abstracted interface for interacting with product and customer objects.

This example models a basic online shopping system using encapsulation and abstraction principles.

Top of Form

Bottom of Form