HACKTIV8

Python: Object Oriented Programming

FTDS Phase 0 - Week 1 Day 4 AM

Contents

HACKTIV8

Introduction to

Object Oriented Programming

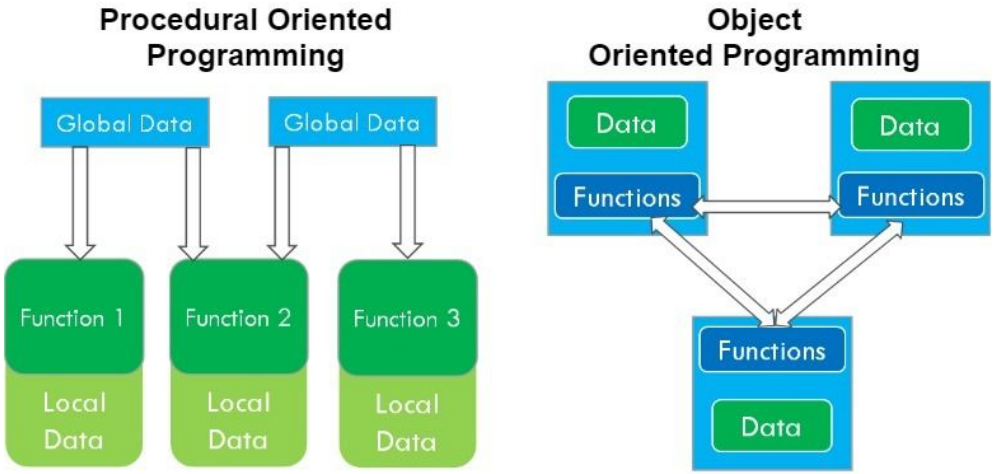
Object-oriented programming (OOP) is a programming paradigm that organizes code into **objects**, which are instances of classes. It provides a way to structure and design software by representing real-world entities as objects and defining their properties (attributes) and behaviors (methods).

In OOP, the key concepts are:

- Objects
- Class
- Encapsulation
- Inheritance
- Polymorphism

Introduction to OOP

Procedural vs Object Oriented Programming

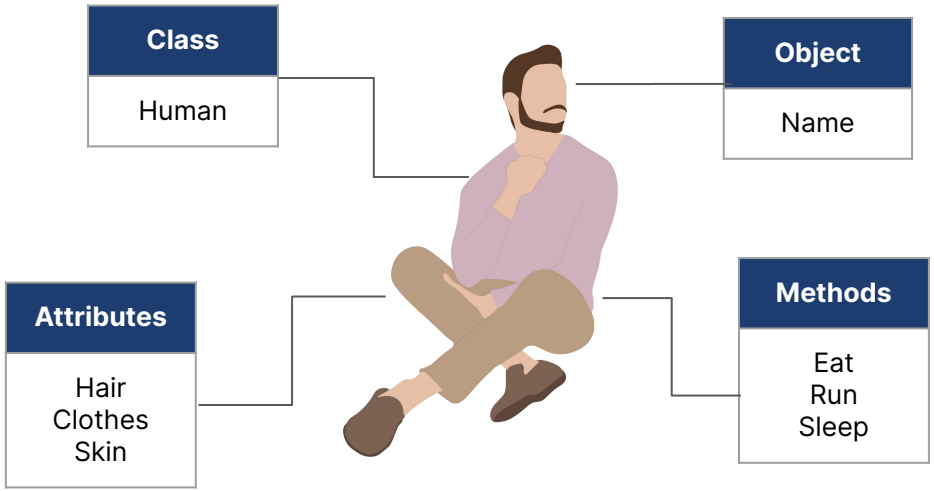


Procedural programming focuses on step-by-step instructions operating on data, using functions for code reusability and modularity. Object-oriented programming emphasizes organizing code around objects with their own state and behavior, promoting code reuse, encapsulation, and modularity through the use of classes and objects.

Introduction to

Objects

Objects are instances that have properties (attributes) and behaviours (methods) that describe themself.



Introduction to

Class

In Python, a class is a blueprint for creating objects (instances) that have certain attributes (variables) and behaviors (methods). It serves as a template or a definition for creating objects of a particular type. To define a class in Python, you can use the class keyword followed by the class name.

```
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.")

# Creating an instance of the Person class
person = Person("Alice", 25)
person.greet()
```

Class

Self and __init__() Method

Self is an agent that behave like a representative of a method. When we call a method of this object as **myobject.method(arg1, arg2)**, this is automatically converted by Python into **MyClass.method(myobject, arg1, arg2)**.

__init__() method is a constructor that contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

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

    def say_hello(self):
        print(f"Hello, my name is {self.name}")


alice = Person("Alice")
alice.say_hello()
```

Class

__str__() Method

__str__() method is used to define how a class object should be represented as a string. It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information.

```
class friends:
    def __init__(self, name1, name2):
        self.name1 = name1
        self.name2 = name2

    def __str__(self):
        return f"{self.name1} and {self.name2} are friends."


bff = friends("John", "Willy")
print(bff)
```

Introduction to

Inheritance

Inheritance is a concept that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. To inherit classes, you need parent class and child class.

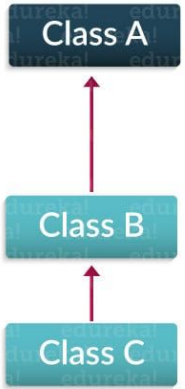
```
# Parent Class
class Student(object):

  # Constructor
  def __init__(self, name, batch):
    self.name = name
    self.batch = batch

  def Display(self):
    print(self.name, self.batch)


std = Student("Fahmi", "RMT-57")
std.Display()
```

```
#Child Class
class FTDS(Student):

  def __init__(self, lc1, lc2, lc3):
    self.lc1 = lc1
    self.lc2 = lc2
    self.lc3 = lc3

std1 = FTDS("Fahmi", "RMT-57")

# calling parent class function
std1.Display()

# Calling child class function
std1.Print()
```

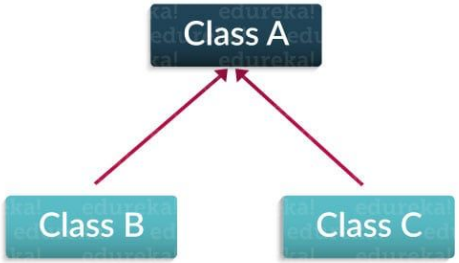
Inheritance

Type of Inheritance

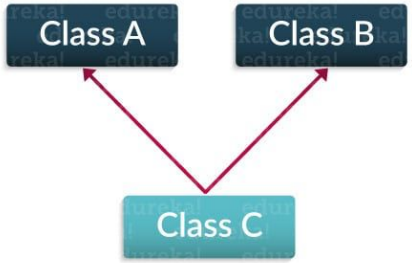
Single Inheritance



Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance

Introduction to

Encapsulation

Encapsulation is a concept in object-oriented programming that combines data (attributes) and methods (functions) into a single unit called a class. It allows you to bundle related data and behavior together, providing a way to organize and structure your code.

In Python, encapsulation is achieved by defining classes and using access modifiers to control the visibility of attributes and methods. Although Python doesn't have strict access modifiers like public, private, and protected, there are conventions that can be followed to indicate the intended visibility.

By convention, a single leading *underscore* _ is used to indicate that an attribute or method is intended to be treated as internal or "private" to the class. It's a way of signaling to other developers that they should not access or modify those members directly. However, Python does not enforce this restriction, and these members can still be accessed from outside the class.

Encapsulation

Example

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number # Private attribute
        self.__balance = balance # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print ("Insufficient funds")

    def get balance (self):
        return self.__balance

# Creating an instance of the BankAccount class
account = BankAccount ("1234567890", 1000)

# Accessing and modifying the private attribute indirectly
account.deposit(500)
account.withdraw(200)

#Accessing the private attribute using a getter method
print( "Account Balance:", account. get_balance() )
```

Encapsulation

Example Explanation

In the previous example, we define a `**BankAccount**` class that encapsulates the attributes `**account_number**` and balance. These attributes are marked as private by prefixing them with a single underscore `**_**`.

The class provides methods deposit and withdraw to modify the balance of the account. The `**get_balance**` method is a getter method that allows external code to access the value of the balance attribute.

We create an instance of the `**BankAccount**` class and perform deposit and withdrawal operations on the account. The private attributes are accessed and modified indirectly through the public methods.

Finally, we use the `**get_balance**` method to retrieve the updated account balance and print it.

Introduction to

Polymorphism

Polymorphism in Python refers to the ability of objects of different types to be treated as if they belong to a common type. It allows you to write code that can work with objects of multiple classes, as long as they share a common interface or superclass. In Python, polymorphism is achieved through **method overriding** and **method overloading.**

1. Method Overriding: When a subclass provides its own implementation of a method that is already defined in its superclass, it overrides the superclass method. When the method is called on an object of the subclass, the overridden method in the subclass is executed.
2. Method Overloading: Although Python does not support method overloading directly, you can achieve similar functionality by defining multiple methods with the same name but different parameter lists. This allows you to have different versions of the method that can be called based on the arguments provided.

Polymorphism allows you to write more flexible and reusable code by abstracting away the specific types of objects and focusing on their common behaviors or interfaces. It enables you to treat different objects as interchangeable entities, providing a way to write more generic and adaptable code.

Polymorphism

Overriding - Example Code

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Cat(Animal):
    def sound(self):
        print("Cat meows")

class Dog(Animal):
    def sound(self):
        print("Dog barks")

# Polymorphic behavior
animals = [Cat(), Dog()]

for animal in animals:
    animal.sound()
```

In the beside example, we define a superclass Animal with a method sound. The Cat and Dog classes inherit from Animal and override the sound method with their own implementations. We create a list of animals containing both Cat and Dog objects. When the sound method is called on each object, the appropriate implementation based on the actual object type is executed, demonstrating polymorphic behavior.

Polymorphism

Overloading - Example Code

```
class MathOperations:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

# Polymorphic behavior
math = MathOperations()

print(math.add(2, 3))
print(math.add(2, 3, 4))
```

In the beside example, we define a MathOperations class with two add methods. One method takes two parameters, and the other method takes three parameters. By providing different numbers of arguments, we can invoke the appropriate method based on the arguments passed, achieving polymorphic behavior.

Additional

If __name__ == “__main__” Idiom

The **if __name__ == "__main__"** idiom in Python is a commonly used construct to check if the current module is being run as the main program, rather than being imported as a module into another program. It allows you to differentiate between the two scenarios and execute specific code accordingly.

```
def some_function():
    # Function code here

def main():
    # Code to be executed when the module is run as the main program
    # It can call other functions, perform operations, etc.
    print("Running as the main program")
    some_function()

if __name__ == "__main__":
    main()
```

Exercise

Real-Life Case 1

Write a Python class BankAccount with attributes like account_number, balance, date_of_opening and customer_name, and methods like deposit, withdraw, and check_balance.

Exercise

Real-Life Case 2

Write a Python class Employee with attributes like emp_id, emp_name, emp_salary, and emp_department and methods like calculate_emp_salary, emp_assign_department, and print_employee_details.

Sample Employee Data:
"ADAMS", "E7876", 50000, "ACCOUNTING"
"JONES", "E7499", 45000, "RESEARCH"
"MARTIN", "E7900", 50000, "SALES"
"SMITH", "E7698", 55000, "OPERATIONS"

- Use 'assign_department' method to change the department of an employee.
- Use 'print_employee_details' method to print the details of an employee.
- Use 'calculate_emp_salary' method takes two arguments: salary and hours_worked, which is the number of hours worked by the employee. If the number of hours worked is more than 50, the method computes overtime and adds it to the salary. Overtime is calculated as following formula:

Overtime = hours_worked - 50

Overtime amount = (overtime * (salary / 50))

HACKTIV8

External References

Colab Link _____ *Visit Here*