In the name of Allah

Hello Everybody

Let’s get started!

Object-Oriented Programming

Object-Oriented Programming (OOP) in Python is a programming paradigm that utilizes objects and classes to structure and organize code. OOP is a way of designing and building software by modeling real-world entities and their interactions in a more intuitive and modular manner.

Class

A class is a template or prototype that describes what an object will be. It defines its attributes(data) and behavior(methods). We must design a class before creating an object.

Object

An object is an instance of a class. When we create an object, we create real-world entities such as cars, bicycles, or dogs with their own attributes and own behaviors.

Define a Class

To define a class in Python, you use the class keyword followed by the name of the class. Here's the basic syntax for defining a class

```
class ClassName:
    # Class attributes and methods go here
```

Inside the Class

Inside the class block, you can define attributes (data) and methods (functions) that will belong to objects created from this class.

```
class Person:
    # Class attribute
    species = "Homo sapiens"

    # Constructor method (__init__)
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Instance method
    def introduce(self):
        return f"Hi, I'm {self.name}, and I'm
{self.age} years old."
```

Properties

Properties are attributes of an object that you can access and sometimes modify.

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

    def bark(self):
        return f"{self.name} barks loudly."

dog1 = Dog("Buddy", "Golden Retriever")
result = dog1.bark()  # Calling the bark
method
```

Methods

- Methods are functions defined within a class.
- They define the behaviors and actions that objects created from the class can perform.
- Methods can take parameters and can operate on the attributes of the object (instance variables).
- Methods are defined within the class, and you can call them on objects of that class.

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

    def bark(self):
        return f"{self.name} barks loudly."

dog1 = Dog("Buddy", "Golden Retriever")
result = dog1.bark()  # Calling the bark
method
```

Constructor

A constructor is a special method within a class that gets called automatically when an object of the class is created. It is used to initialize the attributes (data members) of the object. The constructor method is named __init__

```
class ClassName:
    def __init__(self, parameter1, ...):
        # Initialize attributes here
```

Dunder methods

Dunder methods, short for "double underscore" methods, are special methods in Python that have double underscores at the beginning and end of their names. These methods are also known as magic methods or special methods. They have specific purposes and are automatically called by the Python interpreter under certain circumstances

__init__(self, ...)

The constructor method. It is called automatically when an object is created from the class and is used to initialize object attributes.

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

person = Person("Alice", 30)
```

__len__(self)

The length method. It is called when you use the len(object) function to determine the number of items in an object, such as a list or a custom container class.

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

    def __len__(self):
        return len(self.items)

my_list = MyList([1, 2, 3, 4, 5])
print(len(my_list))  # Output: 5
```

__getitem__(self, key)

The item retrieval method. It is called when you access an item in an object using indexing, like object[key]. You can use this method to define custom behavior for item retrieval.

```
class MyDictionary:
    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        if key in self.data:
            return self.data[key]
        else:
            return f"Key '{key}' not found."

    def __setitem__(self, key, value):
        self.data[key] = value

    def __delitem__(self, key):
        if key in self.data:
            del self.data[key]
```

__setitem__(self, key, value)

The item assignment method. It is called when you assign a value to an item in an object using indexing, like object[key] = value. You can use this method to define custom behavior for item assignment.

```
class MyDictionary:
    def __init__(self):
        self.data = {}

    def __setitem__(self, key, value):
        self.data[key] = value
```

__delitem__(self, key)

The item deletion method. It is called when you delete an item from an object using the del statement, like del object[key]. You can use this method to define custom behavior for item deletion.

```
class MyDictionary:
    def __init__(self):
        self.data = {}

    def __delitem__(self, key):
        if key in self.data:
            del self.data[key]
```

__str__(self)

The string representation method. It is called when you use the str(object) function or print(object) to obtain a human-readable string representation of the object. You can customize this method to return a formatted string.

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

    def __str__(self):
        return f"{self.name}, {self.age}
years old"

person = Person("Alice", 30)
print(str(person))  # Output: Alice, 30
years old
```

__repr__(self)

The representation method. It is called when you use the repr(object) function to obtain an unambiguous string representation of the object. It's typically used for debugging and should return a string that, when evaluated, creates an equivalent object.

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

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

point = Point(2, 3)
print(repr(point))  # Output: Point(2, 3)
```

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (subclass or derived class) based on an existing class (superclass or base class).

Subclass

The subclass inherits attributes and behaviors (attributes and methods) from the superclass, allowing for code reuse and the creation of more specialized classes. In Python, you can implement inheritance using the class keyword and by specifying the superclass in parentheses after the subclass name.

```
# Superclass (Base Class)
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

# Subclass (Derived Class)
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

# Create instances of the subclasses
dog = Dog("Buddy")
print(dog.speak())  # Output: Buddy says
Woof!
```

Is OOP complete?

No, it's just the beginning. While we've covered the essentials for this course, there's a whole world of knowledge waiting for you to explore. I encourage you to continue your learning journey and delve deeper into this fascinating topic.

That’s all for today!