Object-Oriented Programming (OOP)

Class and Object

Class: template for creating objects. It defines a set of attributes and methods that the created objects will have.

Object: An instance of a class

```
class Car:
    def __init__(self, model, year):
        self.model = model
        self.year = year
my_car = Car("Toyota", 2020)
```

Magic methods (dunder methods): predefined methods that you can override to customize the behavior of your classes. surrounded by double underscores (__), such as __init__, __str__ .

example:

__init__(self, ...)

Constructor; initializes a new object.

__str__(self)

Defines the string representation .

__repr__(self)

Defines the formal string representation of an object for debugging and repr().

Inheritance

Inheritance: new class (child) acquires the methods and attributes of an existing class (parent).

```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
   def display_details(self):
        print(f"Name: {self.name}, Age: {self.age}, Salary: ${self.salary}")
class Developer(Employee):
    def __init__(self, name, age, salary, programming_language):
        super().__init__(name, age, salary)
        self.programming_language = programming_language
    def display_details(self):
        super().display_details()  # Call the parent class's display_details method
        print(f"Programming Language: {self.programming_language}")

# Example usage
dev = Developer("Alice", 30, 80000, "Python")
dev.display_details()
```

```
output: Name: Alice, Age: 30, Salary: $80000
Programming Language: Python
```

Encapsulation

Encapsulation help prevent the accidental modification of data.

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # Private attribute
        self.__age = age     # Private attribute
    def get_name(self):
        return self.__name  # Public method to access the private attribute
    def set_age(self, age):
        if age > 0:
            self.__age = age  # Public method to modify the private attribute


# Creating an object of Person class
person = Person("Alice", 30)
person.__age=20 #error
```

Private methods in Python are used to hide implementation details within a class and prevent them from being accessed directly from outside the class.

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance
   def __add_to_balance(self, amount):
        self.__balance += amount
account = BankAccount("123456", 1000)
# account.__add_to_balance(100)  # Error
```

Decorators

decorators allow you to change the behavior of a function without modifying the function itself.

```
def my_decorator(func):
    def wrapper():
        print(" before the function is called.")
        func()
        print("after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")
say_hello()
```

```
output:
before the function is called.
Hello!
after the function is called.
```

Class & Static Methods & Property

Class methods are bound to the class and not the instance of the class. defined using the @classmethod decorator and take cls as their first parameter, which refers to the class itself.

```
class Car:
    number_of_cars = 0  # Class attribute

    def __init__(self, model):
        self.model = model
        Car.number_of_cars += 1  # Increment class attribute
@classmethod
    def get_number_of_cars(cls):
        return cls.number_of_cars

car1 = Car( "Corolla")
car2 = Car( "Civic")
print(Car.get_number_of_cars())  # Output: 2 (Two cars created)
```

Static methods are defined using the @staticmethod decorator.
They do not take self or cls as their first parameter.Static methods can be called directly from the class or via an instance of the class.

Object-Oriented Programming (OOP)

```
class TemperatureConverter:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        """Convert Celsius to Fahrenheit."""
        return (celsius * 9/5) + 32

# Using static methods
celsius_temp = 25
fahrenheit_temp =
TemperatureConverter.celsius_to_fahren
```

*regular method need self as arguments

Properties: enable you to define methods that can be accessed like attribute

```
class BankAccount:
    def __init__(self, balance=0):
        self._balance = balance
  @property
    def balance(self):
  @balance.setter
    def balance(self, amount):
        if amount < 0:
            raise ValueError("Balance cannot
be negative")
        self._balance = amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount
must be positive")
        self._balance += amount
# Usage
account = BankAccount(100)
print(account.balance)   # Output: 100
account.deposit(50)
print(account.balance)   # Output: 150
```

balance is method but with property we ca call it without () like attribute

Abstraction

Abstract classes and methods provide a clear blueprint for what the derived classes should implement, making maintenance easier.

```
the child class sould have abstract methods

from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

    def sleep(self):
        print("This animal is sleeping")

class Dog(Animal):
    def sound(self):
        print("Bark")
```

Polymorphism

Polymorphism allows objects of different classes to be treated through a common interface.

```
class Dog:
from abc import ABC, abstractmethod
class Language(ABC):
    @abstractmethod
    def say_hello(self):
        pass
class English(Language):
    def say_hello(self):
        return "Hello!"

class French(Language):
    def say_hello(self):
        return "Bonjour!"

def greet(language):
    print(language.say_hello())

# Usage
english = English()
```