# Classes in Python

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made.

## Why Use Classes?

- **Encapsulation**: Bundling data and methods that operate on the data within one unit.
- **Inheritance**: Mechanism for creating a new class using details of an existing class without modifying it.
- **Polymorphism**: Ability to define methods in a child class that have the same name as the methods in the parent class.

## Defining Classes

You can define a class using the `class` keyword followed by the class name.

```python
class MyClass:
    '''This is a simple class.'''
    pass
```

## Creating Objects

Objects are instances of a class. You can create an object by calling the class.

```python
class Dog:
    '''A simple class to represent a dog.'''

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking."

#   Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.bark())  # Output: Buddy is barking.
```

## The `__init__` Method

The `__init__` method initializes an object's attributes.

```python
class Dog:
    '''A simple class to represent a dog.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking."

#   Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.bark())  # Output: Buddy is barking.
```

## Instance Methods

Instance methods are functions defined inside a class that operate on instances of the class.

```python
class Dog:
    '''A simple class to represent a dog.'''

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking."

#   Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.bark())  # Output: Buddy is barking.
```

## Class Methods and Static Methods

Class methods are methods that are bound to the class and not the instance of the class. Static methods are methods that do not access or modify the state of the class or instance.

```python
class Dog:
    '''A simple class to represent a dog.'''

    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking."
    @classmethod
    def get_species(cls):
        return cls.species

    @staticmethod
    def is_dog_older_than(dog, age):
        return dog.age > age

#   Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

#   Calling a class method
print(Dog.get_species())  # Output: Canis familiaris

#   Calling a static method
print(Dog.is_dog_older_than(dog1, 2))  # Output: True
```

## Inheritance

Inheritance allows one class to inherit attributes and methods from another class.

```
class Animal:
    '''A simple class to represent an animal.'''

    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclasses must implement this method.")

class Dog(Animal):
    '''A simple class to represent a dog.'''

    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    '''A simple class to represent a cat.'''

    def speak(self):
        return f"{self.name} says Meow!"

#   Creating objects of Dog and Cat classes
dog1 = Dog("Buddy")
```

```
cat1 = Cat("Whiskers")

print(dog1.speak())  # Output: Buddy says Woof!
print(cat1.speak())  # Output: Whiskers says Meow!
```

## Polymorphism

Polymorphism allows methods to be used interchangeably between different classes.

```
class Animal:
    '''A simple class to represent an animal.'''

    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclasses must implement this method.")

class Dog(Animal):
    '''A simple class to represent a dog.'''

    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    '''A simple class to represent a cat.'''

    def speak(self):
        return f"{self.name} says Meow!"

def make_animal_speak(animal):
    print(animal.speak())

#   Creating objects of Dog and Cat classes
dog1 = Dog("Buddy")
cat1 = Cat("Whiskers")

#   Using polymorphism
make_animal_speak(dog1)  # Output: Buddy says Woof!
make_animal_speak(cat1)  # Output: Whiskers says Meow!
```

## Encapsulation

Encapsulation is the concept of restricting access to certain data and methods within a class.

```python
class Dog:
    '''A simple class to represent a dog.'''

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age

#  Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

#  Accessing name and age using getter methods
print(dog1.get_name())  # Output: Buddy
print(dog1.get_age())  # Output: 3

#  Modifying age using setter method
dog1.set_age(4)
print(dog1.get_age())  # Output: 4
```

## Conclusion

Classes are a fundamental part of object-oriented programming in Python. Understanding how to define classes, create objects, and use concepts like inheritance, polymorphism, and encapsulation is essential for writing efficient and maintainable Python programs.

### Stay Updated

Be sure to  this repository to stay updated with new examples and enhancements!

### License

### Contact

Panagiotis Moschos - pan.moschos86@gmail.com

*Note: This is a Python script and requires a Python interpreter to run.*

---

Happy Coding

Made with by Panagiotis Moschos (https://github.com/pmoschos)