OOP in Python (Part 2)

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP). It allows you to create a new class that inherits the properties and behaviors (methods) of an existing class, known as the parent or base class. The new class is called the child or derived class. This helps in promoting code reusability and organizing code in a hierarchical manner.

Kalıtım

Kalıtım, nesne yönelimli programlamada (OOP) temel bir kavramdır. Varolan bir sınıfın özelliklerini ve davranışlarını (metodları) devralarak yeni bir sınıf oluşturmanıza olanak tanır; bu varolan sınıfa ebeveyn veya temel sınıf denir. Yeni sınıfa çocuk veya türetilmiş sınıf denir. Bu, kodun tekrar kullanılabilirliğini artırır ve kodu hiyerarşik bir şekilde düzenlemenize yardımcı olur.

In [ ]:

```python
# Parent class (base class)
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        pass  #"Undefined sound"

# Child class (derived class)
class Dog(Animal):
    def __init__(self, name, breed, color):
        super().__init__(name, species="Dog")
        self.breed = breed
        self.color = color

    def make_sound(self):
        return "Woof!"  # Dogs make a 'Woof!' sound

    def describe_dog(self):
        return f"{self.name} is a {self.color} {self.breed}"

# Child class (derived class)
class Cat(Animal):
    def __init__(self, name, breed, eye_color):
        super().__init__(name, species="Cat")
        self.breed = breed
        self.eye_color = eye_color

    def make_sound(self):
        return "Meow!"  # Cats make a 'Meow!' sound

    def describe_cat(self):
        return f"{self.name} is a {self.eye_color} eyed {self.breed}"

# Creating instances of the child classes and calling methods
dog_instance = Dog("Buddy", "Anatolian Shepherd Dog", "Golden")
cat_instance = Cat("Whiskers", "Siamese", "Blue")

print(dog_instance.describe_dog())  # Output: Buddy is a Golden Anatolian
print(cat_instance.describe_cat())  # Output: Whiskers is a Blue eyed Siam
print(dog_instance.make_sound())    # Output: Woof!
print(cat_instance.make_sound())    # Output: Meow!
```

```
Buddy is a Golden Anatolian Shepherd Dog
Whiskers is a Blue eyed Siamese
Woof!
Meow!
```

In [ ]:

```
# Accessing Properties and methods of the parent and child classes

print(f"{dog_instance.name} is a {dog_instance.color} {dog_instance.breed}
print(f"{dog_instance.name} says: {dog_instance.make_sound()}")

##########################################################################

print(f"{cat_instance.name} is a {cat_instance.eye_color} eyed {cat_instanc
print(f"{cat_instance.name} says: {cat_instance.make_sound()}")
```

```
Buddy is a Golden Anatolian Shepherd Dog.
Buddy says: Woof!
Whiskers is a Blue eyed Siamese.
Whiskers says: Meow!
```

In this example, we have a parent class Animal, which has a constructor (init) that initializes the name and species properties. It also has a make_sound method that is defined but does nothing (pass) since it's specific to each animal.

Then, we have two child classes, Dog and Cat, which inherit from the Animal class. They have their own constructors that call the parent class's constructor using super(), and they also override the make_sound method to provide their unique sound.

When we create instances of the child classes (dog_instance and cat_instance), they inherit the properties and methods of the parent class. We can access the properties and methods of both the parent and child classes through these instances.

Polymorphism

Polymorphism is another important concept in object-oriented programming (OOP). It allows objects of different classes to be treated as objects of a common parent class. This enables us to write code that can work with different types of objects in a uniform way.

Polimorfizm

Polimorfizm, nesne yönelimli programlamadaki (OOP) önemli kavramlardan biridir. Farklı sınıflara ait nesnelerin ortak bir ebeveyn sınıfının nesneleri gibi işlenmesine olanak tanır. Bu, farklı tipte nesnelerle birlikte çalışabilen, tutarlı bir şekilde çalışan kodlar yazmamızı sağlar.

In [ ]:

```python
# Parent Class
class Animal:
    def __init__(self, name):
        self.name = name

    def _make_sound(self):
        pass

# Child Classes
class Dog(Animal):
    def _make_sound(self):
        return "Woof!"

class Cat(Animal):
    def _make_sound(self):
        return "Meow!"

# Function that works with any Animal object
def animal_sound(animal):
    print(f"{animal.name} says: {animal._make_sound()}")

# Creating instances of the child classes
dog_instance = Dog("Buddy")
cat_instance = Cat("Silla")

# Calling the function with different types of objects
animal_sound(dog_instance)
animal_sound(cat_instance)
```

```
Buddy says: Woof!
Silla says: Meow!
```

In this example, we have a parent class Animal with a constructor (init) that initializes the name property and a make_sound method that is defined but does nothing (pass).

We also have two child classes, Dog and Cat, which inherit from the Animal class. Each of these child classes overrides the make_sound method to provide their unique sound.

The animal_sound function takes an Animal object as a parameter. However, since both Dog and Cat are subclasses of Animal, they can be passed as arguments to this function.

When we call the animal_sound function with different objects (dog_instance and cat_instance), it demonstrates polymorphism. The function doesn't need to know the specific type of the object it receives; it simply calls the make_sound method on the object, and the correct sound is returned based on the object's actual type.

This is polymorphism in action – the ability to treat objects of different classes in a uniform way, simplifying code and making it more flexible and adaptable.

Onur_Gumus