Programowanie zorientowane obiektowo

Real Python

PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO

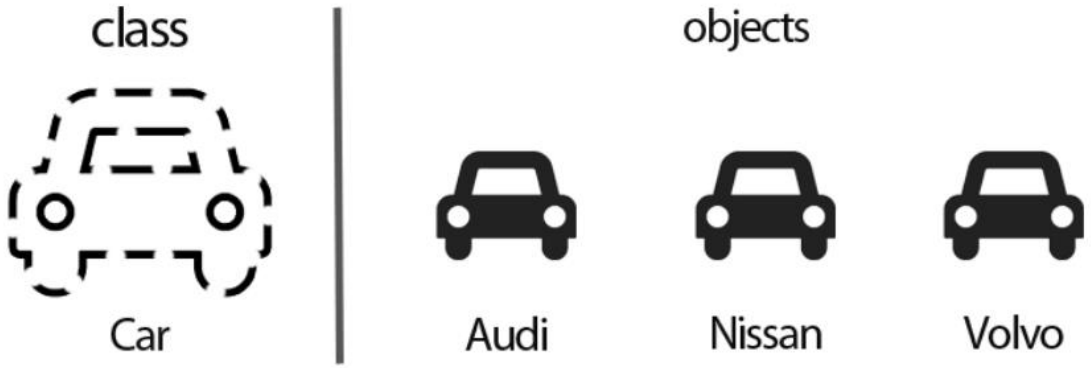
Programowanie zorientowane obiektowo jest oparte na koncepcie **obiektów**. Obiekty te zawierają pola reprezentujące ich **stan** (jako zmienne) i **metody** (jako funkcje), które mogą czytać i modyfikować **stan** obiektu.

KLASA I OBIEKT

Klasa jest zasadniczo szablonem definiującym obiekt, podającym jakie ten obiekt powinien mieć pola, metody i stan domyślny.

Obiekt jest rezultatem stworzenia instancji klasy. Ma on pola i metody zdefiniowane w klasie, z której pochodzi. Może być nieskończona ilość obiektów danej klasy w programie.

KLASA I OBIEKT - ANALOGIA



DEFINICJA KLASY

- Klasę tworzy się poprzez użycie słowa kluczowego **class**.
- Wnętrze klasy (z polami i metodami) tworzy blok kodu, należy więc korzystać z wcięć.
- **__init__(self)** to specjalna metoda (funkcja), wołana za każdym razem, gdy tworzymy obiekt na podstawie danej klasy.
- Metodę **__init__** nazywa się konstruktorem.
- Parametr **self** jest koniecznym pierwszym parametrem w metodach, stosowanym by metody mogły odnosić się do obiektu, na którym są wołane.

DEFINICJA KLASY

```
class Animal:
    NAME = ""  # zmienna klasowa
    AGE = 0    # zmienna klasowa

    def __init__(self):
        self.name = "John"  # ustawienie domyslnej wartosci pola name obiektu klasy
        self.age = 2

    def print_details(self):  # metoda wypisujaca stan obiektu
        print(f"Imie: {self.name}, wiek: {self.age}.")
```

TWORZENIE OBIEKTU

- Obiekt można utworzyć dopiero po zdefiniowaniu klasy.
- Tworzenie obiektu uruchamia metodę **__init__** klasy.
- By dostać się do wartości pola lub wykonać metodę na obiekcie, należy użyć operatora kropki **.** .

```
class Animal:
    ...

my_dog = Animal()  # Tutaj tworzymy obiekt, wolany jest konstruktor __init__
my_dog.print_details()  # Uruchomienie metody print_details() na obiekcie my_dog
print(my_dog.name) # Dostep do pola name obiektu my_dog
my_dog.age = 3  # Aktualizacja pola age obiektu my_dog
```

NIEZALEŻNOŚĆ OBIEKTÓW

Każdy obiekt ma swój stan. Zmieniając wartości pól jednego obiektu, nie wpływamy na te same pola innych.

```
class Animal:
    ...

puppy = Animal()
dog = Animal()  # Stworzenie drugiego obiektu klasy Animal
puppy.age = 1
puppy.name = "Rex Junior"
dog.age = 10
dog.name = "Rex Senior"

print(f"Mój piesek: {puppy.name}, {puppy.age} a starszy pies: {dog.name}, {dog.age}
```

KONSTRUKTOR __INIT__

Zamiast ustawiać wartości domyślne pól obiektów, możemy przekazać je w momencie tworzenia obiektu do konstruktora.

```
class Animal:
    def __init__(self, name="Rex", age=2):
        self.name = name
        self.age = age
```

KONSTRUKTOR __INIT__

Zamiast ustawiać wartości domyślne pól obiektów, możemy przekazać je w momencie tworzenia obiektu do konstruktora.

```
class Animal:
    def __init__(self, name="Rex", age=2):
        self.name = name
        self.age = age

my_cat = Animal("Bonifacy", 5)
my_parrot = Animal("Ara")  # age ustawiony na 2 domyslnie
my_turtle = Animal()  # Ustawienie domyslnych wartosci pol name i age
```

POLA CHRONIONE

- W klasie Animal byliśmy w stanie odwołać się do każdego z pól obiektu i go zmienić - co więcej, każdy mógł to zrobić. Nie zawsze tak chcemy (co jeżeli ktoś ustawi omyłkowo wiek na -10?).
- Dodając na początek nazwy pola _ sugerujemy innym programistom, że ta zmienna może być aktualizowana tylko przez metody danego obiektu.

```
class Animal:
    def __init__(self, name="Rex", age=2):
        self._name = name
        self._age = age

my_dog = Animal()
print(my_dog._name)  # Da sie, wypisze wartosc pola name obiektu my_dog
```

POLA PRYWATNE

By wprowadzić prawdziwą prywatność i chronienie pól, należy dodać __ przed nazwą zmiennej obiektowej.

```
class Animal:
    def __init__(self, name="Rex", age=2):
        self.__name = name
        self.__age = age

my_dog = Animal()
print(my_dog.__name)  # Python wyrzuci blad!!!
```

POLA PRYWATNE - AKTUALIZACJA STANU OBIEKTU

```
class Animal:
    def __init__(self, name="Rex", age=2):
        self.__name = name
        self.__age = age

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age must be grater than 0.")

    def get_age(self):
        return self.__age

my_dog = Animal()
my_dog.set_age(3)
print(my_dog.get_age())  # Wyswietli zaktualizowana wartosc pola __age obiektu my_d
```

PROPERTIES

- Zamiast tworzyć osobne metody takie jak set_age, get_age, można wykorzystać tzw. properties (specjalne atrybuty), które pomagają z enkapsulacją pól w bardziej pythonowy sposób.
- Property może mieć metody **getter**, **setter** i **deleter**.
- Nazwa metody korzystającej z mechanizmu property dla operacji get, set i delete musi być taka sama!

PROPERTIES - GETTER SETTER

```python
class Animal:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property  # getter - wyciaga wartosc pola
    def age(self):
        return self.__age

    @age.setter  # setter - ustawia nowa wartosc pola
    def age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age must be greater than 0.")

my_dog = Animal('Dog', 13)
my_dog.age = 3  # Ustawia wiek - korzysta z settera
print(my_dog.age)  # Odczytuje wiek - korzysta z gettera
```

PROPERTIES - DELETER

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

    @property
    def age(self):
        return self.__age

    @age.deleter  # deleter - usuwa pole
    def age(self):
        del self.__age

my_dog = Animal('Dog', 12)
del my_dog.age  # Usuwa pole - korzysta z deletera
```

DZIEDZICZENIE

- Pozwala klasom potomnym dziedziczyć metody i atrybuty klas bazowych.
- Dzięki temu, jeżeli chcemy napisać podobną klasę do już istniejącej, nie musimy przepisywać czy kopiować raz już napisanego kodu - wystarczy, że nowa klasa odziedziczy (przejmie) kod starej.
- Dziedziczenie wyraża relację **jest**.

```
class Vehicle:
    pass

class Car(Vehicle):
    pass
```

DZIEDZICZENIE - PRZYKŁAD

```python
class Human:
    def __init__(self, name, height, weigth):
        self.name = name
        self.height = height
        self.weigth = weigth

class Programist(Human):
    def __init__(self, name, height, weigth, languages):
        super().__init__(name, height, weigth)
        self.languages = languages

bob = Programist("Bob", 180, 100, ["Python", "Java"])
print(bob.name)  # Dostep do odziedziczonego po klasie Human pola name
```

WIELODZIEDZICZENIE

W szczególnych sytuacjach może istnieć potrzeba implemetnacji klasy dziedziczącej po więcej niż jednej klasie bazowej.

```
class Bat:
    pass

class Man:
    pass

class Batman(Bat, Man):
    pass
```

- Abstrakcja
- Hermetyzacja
- Dziedziczenie
- Polimorfizm