Objektno programiranje

Objektno programiranje je način kako mi združimo medseboj povezane podatke in funkcije, ki delujejo na te podatke.

Primer:

Želimo napisat program "pes".

- Za psa imamo nekja podatkov. Imamo njegovo ime in starost.
- Za psa imamo tudi funkcijo, ki nam ga opiše: f"{ime} je star {starost}"
- Vse to bi radi združili v en objekt

CLASS

Z zadevami, ki so nam znane do sedaj, bi to lahko poizkusili zapakirati v dictionary.

In [1]:

```
pes = {
    "ime": "Fido",
    "starost": 9,
    "opis": "" # radi bi, da nam kle pove opis psa
}
```

Vendar to ne gre.

Vse to lahko združimo s pomočjo Class.

In [2]:

```
# Primer: Kako bi izgledal tak class.
# Kaj kera vrstica pomen si bomo pogledal v nadaljevanju
class Pes: #defines the class
    def __init__(self, ime, starost): #special function, dunder functions
        self.ime = ime #creating new variable called name inside our blank object
        self.starost = starost

    def opis(self):
        return (f'{self.ime} je star {self.starost}')
```

Defining a Class

Začnemo z keyword `class` in nato ime razreda (uporablja se CamelCase način poimenovanja).

```
class ImeRazreda:
    pass
```

Ta razred sedaj predstavlja objekt "Pes". S pomočjo tega razreda lahko sedaj ustvarjamo specifične pse (kjer ima vsak svoje specifično ime, starost, itd.)

Da sedaj ustvarimo našega psa, lahko pokličemo razred in ga shranimo v spremenljivko.

```
x = ImeRazreda()
```

x je sedaj **instanca razreda**.

In [6]:

```
class Pes:
    pass
```

In [7]:

```
a = Pes()
print(a)
print(type(a))
```

```
<__main__.Pes object at 0x00000214DBF60730>
<class '__main__.Pes'>
```

Znotraj našega Class-a definiramo metodo `__init__()` .

Ko ustvarimo novo instanco našega Class-a (novega specifičnega psa), se pokliče ta metoda.

> `__init__()` se uporablja podobno kot "konstruktor" (iz drugih jezikov), čeprov ni točno konstruktor.

In [11]:

```
class Pes:
    def __init__(self):
        print("Ustvarili smo novega psa")

fido = Pes()
print(fido)
print(type(fido))

print()

rex = Pes()
print(rex)
print(type(rex))
```

```
Ustvarili smo novega psa
<__main__.Pes object at 0x00000214DBF99460>
<class '__main__.Pes'>

Ustvarili smo novega psa
<__main__.Pes object at 0x00000214DBF99730>
<class '__main__.Pes'>
```

Znotraj našega razreda lahko ustvarimo spremenljivke, specifične posamezni instanci razreda.

Da ustvarimo spremenljivko specifično instanci razreda se uporabi sledeča sintaksa:

```
self.ime = "Fido"
```

Do spremenljivke dostopamo na sledeč način:

```
print(moj_pes.ime)
Output: Fido
```

V našem primeru bomo vsakemu specifičnemu psu pripisali njegovo ime. Psu bomo ime določili takoj, ko ga ustvarimo. Zato bomo njegovo ime posredovali __init__ metodi.

In [13]:

```
class Pes: #defines the class
    def __init__(self, ime, starost): #ime, starost so argumenti, k jih posredujemo
        self.ime = ime #creating new variable called name inside our blank object
        self.starost = starost
```

In [14]:

```
fido = Pes("Fido", 9)
print(fido.ime)
print(fido.starost)
print()

rex = Pes("Rex", 12)
print(rex.ime)
print(rex.starost)
```

```
Fido
9

Rex
12
```

`self` parameter je instanca razreda in je avtomatično posredovana kot prvi parameter vsaki metodi našega Class-a.

`self.ime = ime` v naši kodi tako pomeni, da naši instanci pripišemo to spremenljivko. Če bi za razliko napisali `Pes.ime = ime` pa bi spremenljivko name spremenili za celoten Class (ker pa vsi psi nimajo enakega imena ostanemo pri `self.ime = ime`).

> method definitions have self as the first parameter, and we use this variable inside the method bodies – but we don't appear to pass this parameter in. This is because whenever we call a method on an object, the object itself is automatically passed in as the first parameter. This gives us a way to access the object's properties from inside the object's methods.

> In some languages this parameter is implicit – that is, it is not visible in the function signature – and we access it with a special keyword. In Python it is explicitly exposed. It doesn't have to be called self, but this is a very strongly followed convention.

In [15]:

```
class Pes: #defines the class
    def __init__(self, ime, starost): #ime, starost so argumenti, k jih posredujemo
        print(self) # dobiš isto, k če daš print(fido)
        print(type(self))
        print(id(self))
        self.ime = ime #creating new variable called name inside our blank object
        self.starost = starost

fido = Pes("Fido", 9)
print(fido.ime)
print(fido.starost)
print()

print(fido)
print(type(fido))
print(id(fido))
```

```
<__main__.Pes object at 0x00000214DBF991F0>
<class '__main__.Pes'>
2288613167600
Fido
9

<__main__.Pes object at 0x00000214DBF991F0>
<class '__main__.Pes'>
2288613167600
```

In [ ]:

Naloga:

Ustvarite razred Vozilo. Vsaka instanca naj ima svojo specifično hitrost in kilometrino.

Izpišite njegove lastnosti na sledeč način:

```
"Max hitrost -vozila-: -hitrost-. Prevozenih je -kilometrina- km."
```

Primeri:

```
Input:
avto = Vozilo("avto", 300, 80)

Output:
Max hitrost avta: 300. Prevozenih je 80 km.

Input:
motor = Vozilo("motor" ,180, 33)

Output:
Max hitrost motor: 180. Prevozenih je 33 km.
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

Znotraj razreda lahko definiramo tudi naše metode s katerimi lahko dostopamo in obdelujemo podatke naših instanc.

Vsaka funkcija/metoda ima najmanj 1 parameter in to je `self` , ki predstavlja instanco razreda in je avtomatično posredovana kot prvi parameter vsaki metodi našega Class-a.

In [16]:

```
class Pes:
    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost

    def opis(self):
        print("Metoda razreda Pes")
```

In [18]:

```
fido = Pes("Fido", 9)
rex = Pes("Rex", 12)

fido.opis()
rex.opis()
```

```
Metoda razreda Pes
Metoda razreda Pes
```

In [22]:

```
class Pes:
    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost

    def opis(self):
        print(f"{self.ime} je star {self.starost}")

fido = Pes("Fido", 9)
rex = Pes("Rex", 12)

fido.opis()
rex.opis()
```

```
Fido je star 9
Rex je star 12
```

In [23]:

```
# Se prav mi lahko uporabmo našo instanco objekta in kličemo njeno metodo na sledeč
fido.opis()

print()

# Oziroma, lahko kličemo direktno Class metodo opis() in sami posredujemo "self" pa
Pes.opis(fido)
```

```
Fido je star 9

Fido je star 9
```

In [ ]:

```

```

Metoda lahko tudi vrne vrednost.

In [21]:

```
class Pes:
    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost

    def opis(self):
        print(f"{self.ime} je star {self.starost}")

    def vrni_starost(self):
        return self.starost

fido = Pes("Fido", 9)
rex = Pes("Rex", 10)

print(fido.vrni_starost())
print(rex.vrni_starost())

if fido.vrni_starost() > rex.vrni_starost():
    print("Fido je starejši")
else:
    print("Rex je starejši")
```

```
9
10
Rex je starejši
```

In [ ]:

```

```

Razredi imajo lahko tudi skupne spremenljivke - spremenljivke, ki so enake vsaki instanci.

> Vsak pes ima 4 noge. Vsak pes ima rad svinjino.

Če želimo, da je spremenljivka enotna celotnemu razredu:

In [29]:

```
class Pes:
    hrana = ["svinjina"]
    #set_ = {1,2,3,3,4,5} #sets are modifyable (mutable)

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        #self.vrsta += "X"

    def opis(self):
        print(f'{self.ime} je star {self.starost}')
```

Do spremenljivke lahko sedaj dostopamo preko razreda samege:

In [31]:

```
print(f"Psi najraje jejo {Pes.hrana}")
```

```
Psi najraje jejo ['svinjina']
```

Oziroma, spremenljivka je dostopna preko vsake instance.

In [32]:

```
fido = Pes("Fido", 9)
rex = Pes("Rex", 10)

print(f'{fido.ime} najraje je {fido.hrana}.')
print(f'{rex.ime} najraje je {rex.hrana}.')
```

```
Fido najraje je ['svinjina'].
Rex najraje je ['svinjina'].
```

In [ ]:

```

```

Spremenljivko lahko tudi spremenimo in jo tako spremenimo tudi za vse instance razreda.

In [35]:

```
class Pes:
    hrana = ["svinjina"]
    #set_ = {1,2,3,3,4,5} #sets are modifyable (mutable)

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        #self.vrsta += "X"

    def opis(self):
        print(f'{self.ime} je star {self.starost}')

fido = Pes("Fido", 9)
rex = Pes("Rex", 10)

print(f'{fido.ime} najraje je {fido.hrana}.')
print(f'{rex.ime} najraje je {rex.hrana}.')

Pes.hrana = ["teletina"]

print(f'{fido.ime} najraje je {fido.hrana}.')
print(f'{rex.ime} najraje je {rex.hrana}.')
```

```
Fido najraje je ['svinjina'].
Rex najraje je ['svinjina'].
Fido najraje je ['teletina'].
Rex najraje je ['teletina'].
```

In [ ]:

```

```

In [8]:

```
class Pes:
    vrsta = "pes"
    hrana = ["svinjina"]
    #set_ = {1,2,3,3,4,5} #sets are modifyable (mutable)

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        #self.vrsta += "X"

    def opis(self):
        print(f'{self.ime} je star {self.starost}')

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta # to nrdi instance variable, ki overwrida Class variable

    def dodaj_hrano(self, hrana):
        self.hrana.append(hrana) # to modify-a variable. In ker je list mutable to

    #def add_to_set(self, el):
        #self.set_ = el
        #self.set_.add(el)


fido = Pes("Fido", 9)
rex = Pes("Rex", 10)
ace = Pes("Ace", 3)

print(f'{fido.ime} je {fido.starost} let star in je {fido.vrsta}. Najraje je {fido.
print(f'{rex.ime} je {rex.starost} let star in je {rex.vrsta}. Najraje je {rex.hran
print(f'{ace.ime} je {ace.starost} let star in je {ace.vrsta}. Najraje je {ace.hran

print(30*"*")
Pes.vrsta = "kuščar" # tuki spremenimo variable celotnemu razredu. Vsi, ki so insta
fido.spremeni_vrsto("opica") # to naredu self.vrsta = opica za fido instanco razred

rex.dodaj_hrano("teletina")



print(f'{fido.ime} je {fido.starost} let star in je {fido.vrsta}. Najraje je {fido.
print(f'{rex.ime} je {rex.starost} let star in je {rex.vrsta}. Najraje je {rex.hran
print(f'{ace.ime} je {ace.starost} let star in je {ace.vrsta}. Najraje je {ace.hran

#print(30*"*")
#print(Pes.vrsta)
#print(Pes.hrana)
#ace.add_to_set(66)
#print(f'{fido.set_} \n{rex.set_} \n{ace.set_}')
```

```
Fido je 9 let star in je pes. Najraje je ['svinjina'].
Rex je 10 let star in je pes. Najraje je ['svinjina'].
Ace je 3 let star in je pes. Najraje je ['svinjina'].
******************************
Fido je 9 let star in je opica. Najraje je ['svinjina', 'teletina'].
Rex je 10 let star in je kuščar. Najraje je ['svinjina', 'teletina'].
Ace je 3 let star in je kuščar. Najraje je ['svinjina', 'teletina'].
```

Treba bit pozoren, če za spremenljivko instance uporabimo enako ime kot za spremenljivko razreda, potem bo spremenljivka instance override class spremenljivko.

Če je spremenljivka mutable (list, itd..) in jo **modify-amo** (dodajamo elemente, odvzemamo, itd..) potem jo spremenimo za celoten razred.

> When we set an attribute on an instance which has the same name as a class attribute, we are overriding the class attribute with an instance attribute, which will take precedence over it. We should, however, be careful when a class attribute is of a mutable type – because if we modify it in-place, we will affect all objects of that class at the same time. Remember that all instances share the same class attributes:

Decorators

S pomočjo dekoratorjev lahko vplivamo na atribute našega razreda.

S pomočjo `@classmethod` lahko definiramo class method. Ko kličemo metodo je `self` parameter dejanski razred. Tako lahko urejamo spremenljivke razreda. Praksa je, da parameter poimenujemo `cls` namesto `self` . Koda ne bo delovala nič drugače.

> What are class methods good for? Sometimes there are tasks associated with a class which we can perform using constants and other class attributes, without needing to create any class instances. If we had to use instance methods for these tasks, we would need to create an instance for no reason, which would be wasteful. Sometimes we write classes purely to group related constants together with functions which act on them – we may never instantiate these classes at all.

In [10]:

```
class Pes:
    vrsta = "pes"
    hrana = ["svinjina"]

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        self.neki = 1

    def opis(self):
        return (f'{self.ime} je star {self.starost}')

    @classmethod
    def spremeni_vrsto(cls, vrsta):
        print(cls)
        cls.vrsta = vrsta # to nrdi instance variable, ki overwrida Class variable.

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina") # to modify-a variable. In ker je list mutabl


fido = Pes("Fido", 9)
rex = Pes("Rex", 10)
ace = Pes("Ace", 3)

#Pes.spremeni_vrsto("opica") # obe metodi delujeta, obe pošljeta razred kot prvi ar
fido.spremeni_vrsto("opica")

print(f'{fido.ime} je {fido.starost} let star in je {fido.vrsta}. Najraje je {fido.
print(f'{rex.ime} je {rex.starost} let star in je {rex.vrsta}. Najraje je {rex.hran
print(f'{ace.ime} je {ace.starost} let star in je {ace.vrsta}. Najraje je {ace.hran
```

```
<class '__main__.Pes'>
Fido je 9 let star in je opica. Najraje je ['svinjina'].
Rex je 10 let star in je opica. Najraje je ['svinjina'].
Ace je 3 let star in je opica. Najraje je ['svinjina'].
```

`@staticmethod` Statična metoda nima `self` oziroma `cls` parametra, kar pomeni, da nima dostopa do spremenljivk. Ponavadi je uporabljena za kakšno helper al pa utility funkcijo razreda.

> The advantage of using static methods is that we eliminate unnecessary cls or self parameters from our method definitions. The disadvantage is that if we do occasionally want to refer to another class method or attribute inside a static method we have to write the class name out in full, which can be much more verbose than using the cls variable which is available to us inside a class method.

In [11]:

```
class Pes:
    vrsta = "pes"
    hrana = ["svinjina"]

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        self.neki = 1

    def opis(self):
        return (f'{self.ime} je star {self.starost}')

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta # to nrdi instance variable, ki overwrida Class variable

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina") # to modify-a variable. In ker je list mutabl

    @staticmethod
    def dolzina_imena(string):
        return f'Dolzina {string} je {len(string)}.'


fido = Pes("Fido", 9)
rex = Pes("Rex", 10)
ace = Pes("Ace", 3)

#print(f'{fido.ime} je {fido.starost} let star in je {fido.vrsta}. Najraje je {fido
#print(f'{rex.ime} je {rex.starost} let star in je {rex.vrsta}. Najraje je {rex.hra
#print(f'{ace.ime} je {ace.starost} let star in je {ace.vrsta}. Najraje je {ace.hra

print(Pes.dolzina_imena("Rex"))
#print(rex.dolzina_imena("Rex")) #oboje isto dela.
```

```
Dolzina Rex je 3.
```

`@property` dekorator nam omogoča, da se metoda obnaša kot atribut.

> Something which is often considered an advantage of setters and getters is that we can change the way that an attribute is generated inside the object without affecting any code which uses the object. For example, suppose that we initially created a Person class which has a fullname attribute, but later we want to change the class to have separate name and surname attributes which we combine to create a full name. If we always access the fullname attribute through a setter, we can just rewrite the setter – none of the code which calls the setter will have to be changed.

> But what if our code accesses the fullname attribute directly? We can write a fullname method which returns the right value, but a method has to be called. Fortunately, the @property decorator lets us make a method behave like an attribute:

In [27]:

```
class Pes:
    vrsta = "pes"
    hrana = ["svinjina"]

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        self.neki = 1

    @property
    def opis(self):
        return (f'{self.ime} je star {self.starost}')

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta # to nrdi instance variable, ki overwrida Class variable

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina") # to modify-a variable. In ker je list mutabl

fido = Pes("Fido", 9)

print(f'{fido.opis}.')
```

```
Fido je star 9.
```

In [28]:

```
print(fido.opis)
```

```
Fido je star 9
```

Python Object Inheritance

S pomočjo dedovanja (inheritance) lahko iz že obstoječih razredov ustvarimo nove, bolj specifične razrede.

Tako novo ustvarjeni razredi so imenovani "child classes" in so izpeljani iz "parent classes".

Child-classes podedujejo vse attribute in metode parent-class-a, katere lahko tudi prepišemo (override) ali pa dodamo nove, bolj specifične attribute in metode.

In [30]:

```
class Pes:
    vrsta = "pes"
    hrana = ["svinjina"]
    #set_ = {1,2,3,3,4,5} #sets are modifyable (mutable)

    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost
        #self.vrsta += "X"

    def opis(self):
        return (f'{self.ime} je star {self.starost}')

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta # to nrdi instance variable, ki overwrida Class variable

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina") # to modify-a variable. In ker je list mutabl

    #def add_to_set(self, el):
        #self.set_ = el
        #self.set_.add(el)


fido = Pes("Fido", 9)

print(f'{fido.ime} je {fido.starost} let star in je {fido.vrsta}. Najraje je {fido.
```

```
Fido je 9 let star in je pes. Najraje je ['svinjina'].
```

In [31]:

```
# Sedaj ustvarimo child class, ki bo dedoval iz class Pes

class Bulldog(Pes):
    pass

spencer = Bulldog("Spencer", 15) # ustvarimo novo instanco class Bulldog, ki deduje
print(type(spencer)) # vidimo, da je instanca class Bulldog
print(spencer)
print(spencer.opis()) # vidimo, da smo dedovali metodo opis() iz class Pes
# če deluje metoda opis pol mamo tud .ime in .starost spremenljivko
```

```
<class '__main__.Bulldog'>
<__main__.Bulldog object at 0x00000123A232DC18>
Spencer je star 15
```

Extending child class

Child class lahko tudi naprej razvijemo z novimi metodami.

In [32]:

```
class Bulldog(Pes):
    def bark(self): # dodali smo metodo, ki jo ima samo Bulldog class, ne pa Pes cl
        return(f'Woof, woof.')
```

In [34]:

```
spencer = Bulldog("Spencer", 15)
print(spencer.bark())

fido = Pes("Fido", 9)
print(fido.bark())
```

```
Woof, woof.

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
 last)
<ipython-input-34-27c093936b16> in <module>
      3
      4 fido = Pes("Fido", 9)
----> 5 print(fido.bark())

AttributeError: 'Pes' object has no attribute 'bark'
```

Overriding methods and attributes

Metode in attribute parentclass-a lahko tudi prepišemo.

In [35]:

```
class Bulldog(Pes):
    vrsta = "Bulldog"

    def opis(self):
        return (f'{self.ime} je star {self.starost} in je Bulldog.')

    def bark(self): # dodali smo metodo, ki jo ima samo Bulldog class, ne pa Pes cl
        return(f'Woof, woof.')

spencer = Bulldog("Spencer", 15)
print(spencer.vrsta)  # prepisali smo vrsto in sedaj so vsi Bulldogi, vrste Bulldog
print(spencer.bark()) # še vedno imamo to metodo, ki je specifična za Bulldog class
print(spencer.opis()) # prepisali smo metodo opis. Sedaj je ta drugačna za class Bu

print()

fido = Pes("Fido", 9)
print(fido.vrsta)
print(fido.opis())
```

```
Bulldog
Woof, woof.
Spencer je star 15 in je Bulldog.

pes
Fido je star 9
```

Multiple inheritance

In [37]:

```
# Multiple inheritance
class SuperA:
    VarA = 10
    def funa(self):
        return 11

class SuperB:
    VarB = 20
    def funb(self):
        return 21

class Sub(SuperA,SuperB):
    pass

object_ = Sub() # podeduje metode in atribute razreda A in razreda B

print(object_.VarA, object_.funa())
print(object_.VarB, object_.funb())
# kle ni problem, ker se nobena stvar ne prekriva (ne instance, ne metode)
```

```
10 11
20 21
```

In [40]:

```
# Left to right
class A:
    def fun(self):
        print('a')

class B:
    def fun(self):
        print('b')


class C(B,A):
    pass
object_ = C()
object_.fun() # prvo dedujemo iz najbl desnega, pol proti levi in prepisujemo stvar
```

```
b
```

In [41]:

```
# override the entities of the same names
class Level0:
    Var = 0
    def fun(self):
        return 0

class Level1(Level0):
    Var = 100
    def fun(self):
        return 101

class Level2(Level1):
    pass

object_ = Level2() # razred Level0 je parent. Level1 deduje iz Level0 in "overrida"
print(object_.Var, object_.fun())
```

```
100 101
```

isinstance() function

s pomočjo funkcijo `python isinstance()` lahko preverimo, če je naša instanca res instanca določenega razreda oziroma razreda, ki od njega deduje.

In [57]:

```
# override the entities of the same names
class Level0:
    Var = 0
    def fun(self):
        return 0

class Level1(Level0):
    Var = 100
    def fun(self):
        return 101

class Level2(Level1):
    pass

l0 = Level0()
l1 = Level1()
l2 = Level2()

print(isinstance(l2, Level2)) #ali je instanca level2 del razreda Level2
print(isinstance(l2, Level1))
print(isinstance(l2, Level0))
print()
```

```
True
True
True
```