

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.

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.

```
fido = ImeRazreda()
```

fido je sedaj **instanca razreda**.

In [1]:

```
class Pes:  
    pass
```

In [2]:

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

```
<__main__.Pes object at 0x7f3c94527250>
<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), čeprav ni točno konstruktor.

In [3]:

```
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 0x7f3c944e0310>
<class '__main__.Pes'>
```

```
Ustvarili smo novega psa
<__main__.Pes object at 0x7f3c944e0640>
<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 [4]:

```
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 [5]:

```
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 [6]:

```
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 0x7f3c944e07c0>
<class '__main__.Pes'>
139898162907072
Fido
9
```

```
<__main__.Pes object at 0x7f3c944e07c0>
<class '__main__.Pes'>
139898162907072
```

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:

```
kamion = Vozilo("kamion", 90, 33)
```

Output:

Max hitrost kamion: 90. Prevozenih je 33 km.

In []:

Class methods

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 [7]:

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

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

In [8]:

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

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

Metoda razreda Pes
Metoda razreda Pes

In [9]:

```
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 [10]:

```
# 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

Metoda lahko tudi vrne vrednost.

In [12]:

```
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_v_cloveskih_letih(self):  
        return self.starost * 7  
  
fido = Pes("Fido", 9)  
rex = Pes("Rex", 10)  
  
print(fido.vrni_starost_v_cloveskih_letih())  
print(rex.vrni_starost_v_cloveskih_letih())
```

63

70

In []:

Naloga:

Ustvarite razred Vozilo. Vsaka instanca naj ima svojo specifično hitrost in kilometrino *in koliko goriva je bilo porabljenega do sedaj*.

Razred Vozilo naj ima funkcija **poraba()**, ki vrne koliko je povprečna poraba tega vozila.

Primeri:

Input:

```
avto = Vozilo("avto", 300, 80, 100)
print(avto.poraba())
```

Output:

Vozilo porabi 6.25l/km

Input:

```
kamion = Vozilo("kamion", 90, 5500, 125000)
print(f"Vozilo porabi {kamion.poraba()}l/km")
```

Output:

Vozilo porabi 22.73l/km

In []:

In []:

Class variables

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 [19]:

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

    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 samega:

In [20]:

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

Psi najraje jejo ['svinjina']

Oziroma, spremenljivka je dostopna preko vsake instance.

In [21]:

```
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 [22]:

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

    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", 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 []:

Oziroma, lahko ustvarimo class method, ki bo spremenila našo class spremenljivko.

In [23]:

```

class Pes:
    hrana = ["svinjina"]

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

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

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

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

print(f'{fido.ime} je {fido.starost} let star. Najraje je {fido.hrana}.')
print(f'{rex.ime} je {rex.starost} let star. Najraje je {rex.hrana}.')
print(f'{ace.ime} je {ace.starost} let star. Najraje je {ace.hrana}.')

print(30*"")

rex.dodaj_hrano("teletina")

print(f'{fido.ime} je {fido.starost} let star. Najraje je {fido.hrana}.')
print(f'{rex.ime} je {rex.starost} let star. Najraje je {rex.hrana}.')
print(f'{ace.ime} je {ace.starost} let star. Najraje je {ace.hrana}.')

```

```

Fido je 9 let star. Najraje je ['svinjina'].
Rex je 10 let star. Najraje je ['svinjina'].
Ace je 3 let star. Najraje je ['svinjina'].
*****
Fido je 9 let star. Najraje je ['svinjina', 'teletina'].
Rex je 10 let star. Najraje je ['svinjina', 'teletina'].
Ace je 3 let star. Najraje je ['svinjina', 'teletina'].

```

In []:

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

Če je spremenljivka mutable (list, itd..) in jo **modify-amo** (dodajamo elemente, odvezujemo, 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:

In [27]:

```

class Pes:
    hrana = ["svinjina"]

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

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

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

    def spremeni_hrano(self, hrana):
        self.hrana = hrana

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

print(f'{fido.ime} je {fido.starost} let star. Najraje je {fido.hrana}.')
print(f'{rex.ime} je {rex.starost} let star. Najraje je {rex.hrana}.')
print(f'{ace.ime} je {ace.starost} let star. Najraje je {ace.hrana}.')

print(30*"")

rex.dodaj_hrano("teletina")
ace.spremeni_hrano(["piščanec"])

print(f'{fido.ime} je {fido.starost} let star. Najraje je {fido.hrana}.')
print(f'{rex.ime} je {rex.starost} let star. Najraje je {rex.hrana}.')
print(f'{ace.ime} je {ace.starost} let star. Najraje je {ace.hrana}.')

```

```

Fido je 9 let star. Najraje je ['svinjina'].
Rex je 10 let star. Najraje je ['svinjina'].
Ace je 3 let star. Najraje je ['svinjina'].
*****
Fido je 9 let star. Najraje je ['svinjina', 'teletina'].
Rex je 10 let star. Najraje je ['svinjina', 'teletina'].
Ace je 3 let star. Najraje je ['piščanec'].

```

In []:

In []:

Naloga:

Ustvarite razred Vozilo. Vsaka instanca naj ima svojo specifično hitrost in kilometrino in koliko goriva je bilo porabljenega do sedaj.

Razred Vozilo naj ima funkcija **poraba()**, ki vrne koliko je povprečna poraba tega vozila.

Dodaite **class variable** razredu Vozilo. Spremenljivki naj bo ime **st gum** in njena vrednost naj bo **4**.

*Dodajte metodo **opis()**, ki naj izpiše opis vozila.*

Primeri:

Input:

```
avto = Vozilo("avto", 300, 80, 500)
avto.opis()
```

Output:

Max hitrost avto: 300. Prevozenih je 80 km. Poraba vozila je 6.25. Vozilo ima 4 gum.

Input:

```
kamion = Vozilo("kamion", 90, 5500, 125000)
print(f"Vozilo porabi {kamion.poraba()}l/km")
```

Output:

Max hitrost kamion: 180. Prevozenih je 5500 km. Poraba vozila je 22.73. Vozilo ima 4 gum.

In []:

In []:

Decorators

S pomočjo dekoratorjev lahko vplivamo na attribute 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 nradi 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_vrstu(self, vrsta):
self.vrsta = vrsta # to nradi 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

In []:

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 prepíšemo (override) ali pa dodamo nove, bolj specifične attribute in metode.

In [46]:

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

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

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

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina")

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

Fido je star 9

In [47]:

```
# 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 mammo tud .ime in .starost spremenljivko
```

```
<class '__main__.Bulldog'>
<__main__.Bulldog object at 0x7f3c9447e070>
Spencer je star 15
```

Extending child class

Child class lahko tudi naprej razvijemo z novimi metodami.

In [48]:

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

In [49]:

```

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

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

```

Woof, woof.

```

-----
-----
AttributeError                                Traceback (most recent call
last)
<ipython-input-49-27c093936b16> 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 [51]:

```

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

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

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

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina")

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

```

Fido je star 9

In [52]:

```
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 class
        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 Bulldog

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

Uporaba metod parent class-a

Sedaj želimo dodati najljubši hrano vsakega Bulldoga.

In [60]:

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

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

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

    def spremeni_vrsto(self, vrsta):
        self.vrsta = vrsta

    def dodaj_hrano(self, hrana):
        self.hrana.append("teletina")

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

Fido je star 9

TO lahko dosežemo tako, da prepisemo `__init__` metodo Bulldog class-a:

In [61]:

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

    def __init__(self, ime, starost, najljubsa_hrana):
        self.ime = ime
        self.starost = starost
        self.najljubsa_hrana = najljubsa_hrana

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

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

spencer = Bulldog("Spencer", 15, "čevapi")
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. Najraje je čevapi

pes

Fido je star 9

Vendar tako ponavljamo določeno kodo:

```
self.ime = ime
self.starost = starost
```

Namesto tega lahko uporabimo `super()` funkcijo s katero dostopamo do metod razreda iz katerega smo dedovali.

In [73]:

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

    def __init__(self, ime, starost, najljubsa_hrana):
        super().__init__(ime, starost)
        self.najljubsa_hrana = najljubsa_hrana

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

spencer = Bulldog("Spencer", 15, "čevapi")  
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. Najraje je čevapi

pes

Fido je star 9

In []:

Naloga:

Ustvarite razred Vozilo. Vsaka instanca naj ima svojo specifično hitrost in kilometrino in koliko goriva je bilo porabljenega do sedaj.

Razred Vozilo naj ima funkcija **poraba()**, ki vrne koliko je povprečna poraba tega vozila.

Dodajte **class variable** razredu Vozilo. Spremenljivki naj bo ime **st_gum** in njena vrednost naj bo **4**.

Dodajte metodo **opis()**, ki naj izpiše opis vozila.

Ustvarite podrazreda **Avto** in **Motor**. Razreda naj dedujete od razreda Vozila. Motor razred naj prepiše spremenljivko **st_gum** v **2**. Oba razreda naj

Primeri:

Input:

```
avto = Avto(300, 80, 500)
avto.opis()
```

Output:

Max hitrost avto: 300. Prevozenih je 80 km. Poraba vozila je 6.25 l/km. Vozilo ima 4 gum.

Input:

```
motor = Motor(90, 220, 520)
motor.opis()
```

Output:

Max hitrost motor: 90. Prevozenih je 220 km. Poraba vozila je 2.36 l/km. Vozilo ima 2 gum.

In []:

In []:

In []:

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

In [7]:

```
# 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

inspect.getmro(class_name)

S pomočjo te funkcije lahko izpiše strukturo dedovanja.

In [79]:

```
import inspect
# 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

inspect.getmro(Level2)
```

Out[79]:

```
(__main__.Level2, __main__.Level1, __main__.Level0, object)
```

In []: