

# Terke in spremenljivost objektov

Osnove programiranja

Nejc Ilc

# Spremenljivost objektov

Nekateri objekti se lahko spreminjajo kot plastelin, drugi so nespremenljivi kot ... kamen?

## Spremenljivi

Lahko spreminjamo vsebino (ang. *mutable*)

- seznam `list`
- množica `set`
- slovar `dict`



Fotografiji: Kilimanjaro STUDIOz in Nicolas Gras

## Nespremenljivi

Ne moremo spreminjati vsebine (ang. *immutable*)

- `bool`
- števila `int` in `float`
- niz `str`
- terka `tuple`

Primer s kamnitim kipom je slab.  
Kip namreč lahko spremeniš ...  
vendar ga s tem uničiš.



# Objekti iz kamna

Nespremenljivih objektov ne moremo spreminjati 🙅

```
>>> ime = 'jože'
>>> ime[0] = 'J'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item assignment
```

Nizu, ki je nespremenljiv objekt, smo želeli spremeniti prvi znak. Dobili smo napako, ker to ne gre.

Čakaj malo, nekje smo videli, da obstaja metoda niza, ki se imenuje `capitalize()`, in spremeni prvi znak v veliko začetnico! Glej:

```
>>> ime.capitalize()
'Jože'
```

Ja, hm, zdaj pa izpišimo `ime` ...

```
>>> ime
'jože'
```

Torej se ni nič spremenilo. Kako to? Preberi še enkrat podnaslov. Če želimo nespremenljivemu objektu kaj dodati/odvzeti/spremeniti, moramo izdelati nov objekt.

Ko kličemo metodo `ime.capitalize()`, se v resnici izdela in vrne nov niz, ki se začne z veliko začetnico. Če ga želimo "shraniti", ga moramo poimenovati (prirediti imenu):

```
>>> ime = ime.capitalize() # Nov niz prepiše starega
>>> ime
'Jože'
```

# Objekti iz plastelina

Plastelin lahko preoblikujemo v novo obliko z lahkoto. Lahko ga podaljšamo ali skrajšamo, še vedno ostane isti plastelin.

Edini spremenljivi tip objekta, ki smo ga do sedaj spoznali, je seznam. Poglejmo, kako se razlikuje od niza.

```
>>> ime = ['j', 'o', 'ž', 'e']
>>> ime[0] = 'J'
>>> ime
['J', 'o', 'ž', 'e']
```

Brez težav smo spremenili prvi element seznama. Pri tem se seznam `ime` ni na novo ustvaril. Še vedno je *isti* kot prej.

Seznami imajo precej metod, ki spreminjajo vsebino objekta. Denimo `sort()`:

```
>>> ime.sort()
>>> ime
['J', 'e', 'o', 'ž']
```

Na prejšnji strani smo videli, da metode niza ne spremenijo (ker je nespremenljiv objekt, kakopak), ampak ustvarijo in *vrnejo* nov objekt. Tu vidimo, da metode spremenijo seznam direktno - ker je seveda spremenljiv objekt. Enostavno!

# "Osebno, prosim!"

Vsak objekt ima svojo unikatno številko, EMŠO

Kako vemo, ali je nek objekt *isti* kot drug objekt. Kaj pa če mu je samo *enak*? Naj pokaže osebno izkaznico, pa bo - funkcija `id()`.

```
>>> ime = 'Jože'
>>> id(ime)
2038682205136 # Ta številka bo pri tebi drugačna
>>> ime = ime.capitalize()
>>> id(ime)
2038682205232 # Aha! Druga številka! Nov objekt.
```

```
>>> ime = ['j', 'o', 'ž', 'e']
>>> id(ime)
2038680876736
>>> ime.sort()
>>> ime
['e', 'j', 'o', 'ž']
>>> id(ime)
2038680876736 # Enak ID, torej gre za isti objekt
```



# Enak ali isti?

Poznamo razliko v pomenu besed?

## Operator prirejanja =

Kaj se zgodi ob prirejanju?

```
>>> a = 123456 # Ustvarimo objekt tipa `int` z imenom a
>>> b = a      # V imenski prostor dodamo novo ime b,
               # ki pa govori o ISTEM objektu kot ime a

>>> id(a)
2038679937936
>>> id(b)
2038679937936 # No, kaj ni res? ID se ujema.
```

Prirejanje povzroči to, da nekemu objektu na desni strani damo ime, ki je na levi strani. Isti objekt ima lahko več imen. Objekt, ki predstavlja celo število 123456 ima sedaj dve imeni: a in b.

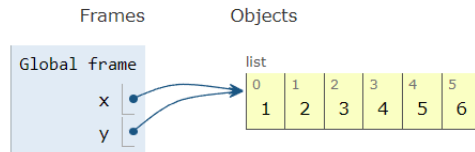
## Primer s seznamom

Je s seznamom kaj drugače?

```
1 x = [1, 2, 3, 4, 5, 6]
2 y = x
3 print(id(x))
→ 4 print(id(y))
```

Print output (drag lower right corner to resize)

```
140220131261512
140220131261512
```



Ne, prav tako vidimo, da smo istemu objektu dali dve imeni: x in y. Imata isto unikatno številko, torej gre za isti objekt.

Kaj pa, če sedaj spremenimo b ali y? Obrni stran.

# Enak ali isti?

Zdaj že poznamo razliko.

Kaj se zgodi, ko spremenimo nespremenljiv oziroma spremenljiv objekt? Poglejmo na primeru operatorja `+=`.

## Kamen

```
>>> a = 123456 # Ustvarimo objekt tipa `int` z imenom a
>>> b = a      # V imenski prostor dodamo novo ime b,
               # ki govori o ISTEM objektu kot ime a.
>>> b += 1     # Spremenimo objekt z imenom b. Ker je
>>> id(a)      # objekt tipa `int` nespremenljiv, se
2038679940112  # ustvari nov objekt, ki ima vsebino
>>> id(b)      # za 1 večjo. To vidimo tudi po ID,
2038679930992  # ki je sedaj različen za a in b.
```

Spreminjanje **nespremenljivega** objekta povzroči ustvarjanje novega objekta, ki ni več "povezan" z originalom.

## Plastelin

```
>>> x = [1, 2, 3, 4, 5, 6]
>>> y = x      # Na seznam kaže tudi ime y.
>>> y += [7]    # To je enako kot `y.extend([7])`.
>>> id(x)
2038681128832
>>> id(y)
2038681128832 # OK, gre za isti objekt.
>>> x          # Imeni x in y govorita o istem seznamu!
[1, 2, 3, 4, 5, 6, 7]
```

Spreminjanje **spremenljivega** objekta ne ustvari novega objekta, ampak obstoječi objekt samo spremeni. Sprememba seznama z imenom `y` torej spreminja tudi seznam z imenom `x`, saj imata *iste* (in ne zgolj *enake*) elemente.

# Kopija

Želiš *zares* kopirati seznam?

Kopirajmo seznam tako, da ima *enake* elemente kot original, ne pa *istih*, uporabimo metodo `copy()`.

```
>>> x = [1, 2, 3, 4, 5, 6]
>>> y = x.copy()
>>> id(x)
2038683126272
>>> id(y)
2038683228224
>>> y.remove(6)
>>> x
[1, 2, 3, 4, 5, 6]
>>> y
[1, 2, 3, 4, 5]
```

Namesto metode `copy()` lahko uporabimo rezino `[:]`, ki pa naredi isto kot `copy()`.

```
>>> x = [1, 2, 3, 4, 5, 6]
>>> y = x[:]
...
```

(Zdaj pa nekaj krutega)

```
>>> x = [[1, 2], 3] # Pazi, imamo seznam v seznamu.
>>> y = x.copy()    # Naredimo ("plitvo") kopijo.
>>> y[0][0] = 0      # Spremenimo prvi element
>>> x                # vdelanega seznama in onemimo,
[[0, 2], 3]         # ko vidimo, da smo spremenili
>>> y                # tudi original! Očitno plitva
[[0, 2], 3]         # kopija ne kopira vdelanih reči.
```

Ko ne zaleže `copy()`, uporabi `deepcopy()` iz modula `copy`.



# Funkcije in spremenljivost objektov plus

Ali lahko funkcije spreminjajo vrednosti argumentom?

Pomislimo malo na `str.capitalize()` in nato še na `list.sort()`

Odgovor je spet povezan s tem, ali so argumenti spremenljivi objekti ali ne:

Definirajmo dve funkciji.

```
def plus_1(a, b):  
    c = a + b # Uporabimo operator + in rezultatu  
    return c # priredimo novo ime.  
  
def plus_2(a, b):  
    a += b # Rezultatu damo isto ime kot ga  
    return a # ima prvi argument.
```

Sedaj uporabimo funkciji nad nizom.

```
>>> x = 'logo'  
>>> y = 'ped'  
>>> z = plus_1(x, y)  
>>> print('x:', x, ', y:', y, ', z:', z)  
x: logo , y: ped , z: logoped  
>>> z = plus_2(x, y)  
>>> print('x:', x, ', y:', y, ', z:', z)  
x: logo , y: ped , z: logoped
```

Nič presenetljivega do sedaj. Objekta `x` in `y` sta ostala nespremenjena tudi po klicu funkcije.

# Funkcije in spremenljivost objektov plus

Nadaljujmo primer, tokrat s seznamom

Imamo isti definiciji funkcij.

```
def plus_1(a, b):  
    c = a + b # Uporabimo operator + in rezultatu  
    return c # priredimo novo ime.  
  
def plus_2(a, b):  
    a += b # Rezultatu damo isto ime kot ga  
    return a # ima prvi argument.
```

Sedaj ju uporabimo nad seznamom.

```
>>> x = [1, 2, 3]  
>>> y = [4, 5]  
>>> z = plus_1(x, y)  
>>> print('x:', x, ', y:', y, ', z:', z)  
x: [1, 2, 3] , y: [4, 5] , z: [1, 2, 3, 4, 5]  
  
# Do tu nič posebnega.  
  
>>> z = plus_2(x, y)  
>>> print('x:', x, ', y:', y, ', z:', z)  
x: [1, 2, 3, 4, 5] , y: [4, 5] , z: [1, 2, 3, 4, 5]
```

Aha! Očitno funkcija lahko spreminja vrednosti **spremenljivim** argumentom in ta sprememba je vidna tudi izven funkcije.

# Primer radirka

Povadimo spremenljivost

Napišimo funkcijo, ki sprejme dva niza in iz prvega izbriše vse znake, ki se pojavijo v drugem.

```
def radirka_niz(prvi, drugi):  
    # Z zanko gremo čez drugi niz  
    for i in drugi:  
        # Brišemo vse pojavitve znaka i iz prvega niza  
        prvi = prvi.replace(i, '')  
    return prvi
```

Uporabimo funkcijo:

```
>>> a = 'brokoli'  
>>> b = 'korenje'  
>>> r = radirka_niz(a, b)  
>>> print(a, b, r)  
brokoli korenje bli
```

Napišimo funkcijo, ki sprejme dva seznama in iz prvega izbriše vse elemente, ki se pojavijo v drugem.

```
def radirka_seznam(prvi, drugi):  
    # Z zanko gremo čez drugi seznam  
    for i in drugi:  
        # Brišemo element i iz prvega, lahko jih je več  
        while i in prvi:  
            prvi.remove(i)
```

Uporabimo funkcijo:

```
>>> a = [1, 1, 2]  
>>> b = [1, 1, 3]  
>>> r = radirka_seznam(a, b)  
>>> print(a, b, r)  
[2] [1, 1, 3] None # Zakaj je izpis tak? Razmisli.
```

# Terka

Par, trojka, četverka, peterka, ...

Terka ( `tuple` ) je podatkovni tip, ki je podoben seznamu, vendar se hkrati od njega precej razlikuje. Poglejmo.

## Podobnosti s seznamom

- terka in seznam v sebi hranita zaporedje objektov

```
terka = (True, 1, 'a')  
seznam = [True, 1, 'a']
```

- terko lahko naslavljamo, delamo rezine, lepimo z drugo terko, množimo s celim številom, ...
- preko terke se lahko sprehajamo z zanko

```
for element in terka:
```

## Razlike

- terko definiramo z okroglimi oklepaji ali kar brez vseh oklepajev: `terka = True, 1, 'a'`
- **terke ni spremenljiva**, seznam je. Terka je podobna nizu, ki je tudi nespremenljiv.
- terka s samo enim elementom je malo nenavadna: `t = ('Samo',)`. Vejica za prvim elementom je obvezna. Pri seznamu ni obvezna: `s = ['Samo']`, lahko tudi `s = ['Samo',]`
- terke so učinkovitejše: porabijo manj spomina računalnika, procesor jih obdeluje hitreje

# Funkcije nad terkami

To vse smo že videli tudi pri nizih in seznamih

## tuple()

```
>>> tuple()           # Ustvarimo prazno terko
()
>>> tuple([1, 2, 3]) # Seznam pretvorimo v terko
(1, 2, 3)
```

## bool()

```
>>> bool(())          # Prazna terka je False
False
>>> bool(tuple())     # Še vedno
False
>>> bool((1, 2, 3))  # Neprazna terka je True
True
```

len(), sum(), min(), max()

# Metode terke

Samo dve sta in ne spreminjata vsebine terke

## terka.count(x)

```
# Vrne število pojavitev elementa `x`
>>> terka = (1, 2, 3, 3)
>>> terka.count(2)
1
>>> terka.count(3)
2
```

## terka.index(x)

```
# Vrne indeks prve pojavitve elementa `x`
>>> terka = (1, 2, 3, 3)
>>> terka.index(2)
1
>>> terka.index(3)
2
```

# Operacije nad terkami

Spet - to vse smo že delali z nizi in seznami

## Vsebovanost `in`, `not in`

```
>>> terka = (1, 2, 3, 3)
>>> 4 in terka    # Ali je element 4 v terki? Ni.
False
```

## Seštevanje in množenje

```
>>> (1, 2, 3) + (4,) + (5, 6) # Lepimo skupaj
(1, 2, 3, 4, 5, 6)
>>> (1, 2, 3) * 2              # Razmnožujemo
(1, 2, 3, 1, 2, 3)
```

## Primerjanje

```
>>> (1, 2, 3) > (2, 1)
False
>>> (1, 2, 3) > (1, 2, 2)
True
```

## Sprehod z zanko `for`

```
t = tuple('OP')    # Dobimo par ('O', 'P')
for element in t:
    print(element)  # Izpišemo 'O', nato še 'P'
```

## Naslavljanje in rezine

```
>>> t = (1, 2, 3)
>>> t[-1]
3
>>> t[::2]
(1, 3)
>>> t[:2] + (0,) + t[2:]
(1, 2, 0, 3)
```

# Razpakiranje elementov zaporedja

Velja za zaporedja: nize, sezname, terke,  
množice, slovarje, generatorje kot je `range` ...

Zaporedje si lahko predstavljamo kot kovček, v  
katerem so zapakirani elementi. Kovček lahko  
odpremo in ven zložimo elemente.

```
>>> niz = 'ABC'  
>>> (prvi, drugi, tretji) = niz # Na levi je terka  
>>> prvi, drugi, tretji = niz # Pišemo brez ()  
>>> prvi  
'A'  
>>> drugi  
'B'  
>>> tretji  
'C'
```

`Z` = razpakiramo desno stran v terko na levi.

# Razpakiranje elementov zaporedja

nahrbtnik

Nadaljujmo v pustolovskem duhu

Smo na lovu na zaklad in v nahrbtniku imamo nekaj predmetov. Za vsakega od njih imamo zabeleženo tudi količino. Opis predmeta in njegovo količino bomo shranili v terko, celoten nahrbtnik pa opišemo s seznamom terk. Takole:

```
nahrbtnik = [  
    ('signalna raketa', 2),  
    ('mačeta', 1),  
    ('proteinska ploščica', 5),  
    ('zavoj robčkov', 1),  
    ('rezervne nogavice', 0) # pravkar smo jih obuli  
]
```

Izpišimo vsebino nahrbtnika:

```
for predmet in nahrbtnik:  
    print(predmet[0], '(', predmet[1], 'x)', sep='')
```

Naš program lahko naredimo še za odtenek bolj enostaven za razumevanje, če namesto `predmet[0]` pišemo `opis` in namesto `predmet[1]` raje `kolicina`. Torej, razpakirajmo terko:

```
for predmet in nahrbtnik:  
    opis, kolicina = predmet  
    print(opis, '(', kolicina, 'x)', sep='')
```

Razpakiranje lahko naredimo kar v glavi zanke:

```
for opis, kolicina in nahrbtnik:  
    print(opis, '(', kolicina, 'x)', sep='')
```

Veliko lepše!



# Triki

Zanimive stvari, ki jih omogoča razpakiranje seznamov ali terke

## Hkratno prirejanje

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

Uporabili smo terko, samo njene oklepaje smo izpustili. Enakovreden zapis:

```
>>> (a, b) = (1, 2)
```

Podnapisi: terko `(1, 2)` smo razpakirali v elementa `a` in `b`. Gre tudi s seznamom:

```
>>> [a, b] = [1, 2]
```

## Menjava vrednosti

Spremenljivka `a` ima vrednost 1, `b` pa 2. Kako bi zamenjali njuni vrednosti? Prvi poskus:

```
>>> zacasno = a
>>> a = b
>>> b = zacasno
```

Gre pa tudi takole:

```
>>> b, a = a, b
# ali z oklepaji
>>> (b, a) = (a, b)
# ali s seznamom
>>> [b, a] = [a, b]
```

# Triki in Fibonacci

fibonacci\_triki

Uporabimo sveže trike na primeru programa, ki izpiše Fibonaccijeva števila

Tole smo naprogramirali par predavanj nazaj:

```
# Koliko členov izpišemo
n = 10

# Začetne vrednosti za prva dva člena
# V splošnem začnemo z 0, 1
# Fibonacci je začel z 1, 2
a = 1 # dve števili nazaj od trenutnega
b = 2 # eno število nazaj od trenutnega

for i in range(n):
    print(a)
    Fn = a + b
    a = b
    b = Fn
```

Vidimo, da gre zgolj za menjavo vrednosti vsako iteracijo zanke. Za vajo bomo še naredili funkcijo, ki izvrši izračun in izpis števil.

```
def fib(n):
    """Izpiše prvih n Fibonaccijevih števil"""
    a = 1
    b = 2

    for i in range(n):
        print(a)
        (a, b) = (b, a + b)

# Glavni program
fib(10)
```

# Znamo razpakirati. Kaj pa zapakirati?

zamrzovalnik

To navadno prepustim ženi ♥

Pripravljamo zalogo hrane, ki jo bomo dali v zamrzovalno omaro. Hrano iste vrste damo skupaj v vrečko. Zapišemo si tudi količino in enoto hrane v vrečki. Pripravimo si tri sezname:

```
opis      = ['sojini polpeti', 'bučke', 'pišč. bedrca']
kolicina  = [12,                1,                4]
enota     = ['kos',             'kg',             'kos']
```

Izpišimo vsebino zamrzovalnika. Potrebujemo indekse, uporabimo `range()`:

```
for i in range(len(opis)):
    print(opis[i], kolicina[i], enota[i])
```

```
sojini polpeti 12 kos
bučke 1 kg
pišč. bedrca 4 kos
```

Morda bi bilo pametno vse podatke o eni vrečki hrane imeti skupaj, kot en element. Kako bi to naredili?

```
zamrzovalnik = []
for i in range(len(opis)):
    # Tu se odločimo, da bomo elemente spajali v terke
    zamrzovalnik.append(
        (opis[i], kolicina[i], enota[i])
    )
print(zamrzovalnik)
```

Izpis nam da tole:

```
[
    ('sojini polpeti', 12, 'kos'),
    ('bučke', 1, 'kg'),
    ('pišč. bedrca', 4, 'kos')
]
```

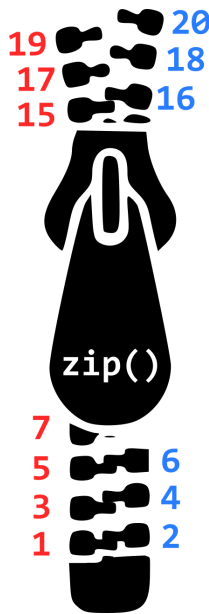
# Združevanje zaporedij po sistemu zadrge: `zip()`

Rešimo zamrzovanje hrane z vgrajeno funkcijo

Python pozna elegantno funkcijo, ki združuje več zaporedij tako, da v terke déva istoležne elemente. Poglejmo si preprost primer.

Imamo seznama `a` in `b`, ki vsebujeta vsa liha oziroma soda števila od 1 do 20. Združimo ta dva seznama z `zip()`:

```
>>> a = list(range(1,21,2))
>>> b = list(range(2,21,2))
>>> a
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> b
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> list(zip(a,b))
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10), (11, 12),
(13, 14), (15, 16), (17, 18), (19, 20)]
```



# Zapakirajmo že to hrano

zamrzovalnik

Nadaljevanje, naredimo `zip()` sami

Ups, izgubimo podatek o količini piščančjih bedrc:

```
kolicina.pop()
```

Seznam `opis` in `enota` imata 3 elemente, seznam `kolicina` samo 2. Očitno lahko združimo istoležne elemente samo dvakrat. Torej, pogledamo dolžine seznamov in vzamemo najkrajšo:

```
n = min(len(opis), len(kolicina), len(enota))

zamrzovalnik2 = []
for i in range(n):
    zamrzovalnik2.append(
        (opis[i], kolicina[i], enota[i])
    )
```

Sedaj uporabimo `zip()`, ki zna delati tudi z več zaporedji:

```
zamrzovalnik3 = list(zip(opis, kolicina, enota))
print(zamrzovalnik3)
```

Dobimo izpis, ki je enak kot pri `print(zamrzovalnik2)`:

```
[('sojini polpeti', 12, 'kos'), ('bučke', 1, 'kg')]
```

`zip()` kot argumente sprejme katerokoli zaporedje, lahko torej nize, sezname, terke, `range` itd.

# Oštevilči zaporedje z `enumerate()`

Bistveno lepši prehodi z zanko `for` preko indeksov in elementov, alternativa `range()`

Funkcija `enumerate()` sprejme zaporedje (npr. niz, seznam, terko, ...) in vrne zaporedje oštevilčenih elementov.

```
>>> enumerate('ABCČD')
<enumerate object at 0x00000172EC9D3830> # Generator!
>>> list(enumerate('ABCČD'))
[(0, 'A'), (1, 'B'), (2, 'C'), (3, 'Č'), (4, 'D')]
```

Vrne torej zaporedje terk, kjer je prvi element v terki indeks in drugi element vsebina na tem indeksu.

```
>>> ostevilcen = list(enumerate('ABCČD'))
>>> ostevilcen[0]
(0, 'A')
>>> i, v = ostevilcen[0] # Razpakiranje
>>> print(i, v)
0 A
```

## Zanka `for`

To lahko sedaj s pridom uporabimo, da pišemo lepše zanke `for`. Spomnimo: če smo znotraj zanke potrebovali indeks elementa, smo uporabili

`range()` :

```
imena = ['Ana', 'Ida', 'Eva']
for i in range(len(imena)):
    print(i, imena[i])
```

Z uporabo `enumerate()` :

```
for i, ime in enumerate(imena):
    print(i, ime)
```

Zelo pythonično!

# Koliko vrednosti lahko vrne funkcija? kaj\_vraca\_fun

Filozofsko vprašanje

Funkcija lahko ne vrne ničesar oziroma bolje rečeno *vrne nič*, kar v pythonščini zapišemo kot `return None`. To se zgodi, ko ne uporabimo stavka `return` ali ko napišemo samo `return` ali ko smo zelo jasni in napišemo `return None`. Primer:

```
def povprecje(seznam):  
    p = sum(seznam)/len(seznam)  
    print('Povprečje je', p)
```

Funkcija lahko vrne nekaj *enostavnega*, denimo eno število:

```
def povprecje(seznam):  
    p = sum(seznam)/len(seznam)  
    return p
```

Nadalje, funkcija lahko vrne nekaj *sestavljenega*, denimo niz (je to ena vrednost ali jih je več?):

```
def povprecje(seznam):  
    p = sum(seznam)/len(seznam)  
    return 'Povprečje je ' + str(p)
```

Vrne pa lahko tudi seznam ali terko:

```
def povprecje(seznam):  
    v = sum(seznam)  
    d = len(seznam)  
    p = v/d  
    n = 'Povprečje je ' + str(p)  
    return [v, d, p], n # To je terka seznama in niza
```

```
>>> povprecje([20, 50, 3, -4])  
([69, 4, 17.25], 'Povprečje je 17.25')
```

# Še malo o terkah

## Nenapisana zlata pravila





- V seznamih hranimo *homogene* elemente, torej elemente, ki imajo isti podatkovni tip. Posamezne elemente seznama obravnavamo na enak način.
- V terkah hranimo *heterogene* elemente, torej elemente, ki imajo različen podatkovni tip. Vsako izmed terk obravnavamo na isti način.

Primer:



```
[  
  ('sojini polpeti', 12, 'kos'),  
  ('bučke', 1, 'kg'),  
  ('pišč. bedrca', 4, 'kos')  
]
```

## Zakaj torej potrebujemo terke?

1. Terka je **nespremenljiva**:

-  vsebina terke je zaščitena pred dodajanjem, spreminjanjem in brisanjem;
-  če jo podamo kot argument, je funkcija ne more spreminjati. Seznam lahko.
-  lahko je element *množice* (prihodnjič);
-  lahko je *ključ* v slovarju (čez dva tedna).

2. Terka je **hitrejša** kot seznam:

-  zasede manj prostora v pomnilniku in
-  manj obremeni procesor.