# 06 slovarji in množice

January 28, 2024

## 0.1 Slovar

Poglejmo še enkrat sezname. Seznam je nekakšna zbirka nekih poljubnih reči, pri čemer je vsaka reč v svojem "predalčku", predalčki pa so oštevilčeni. Prvi ima številko 0, drugi 1, tretji 2 in tako naprej. Za vsak predalček lahko pogledamo, kaj je v njem, spremenimo njegovo vsebino, predalčke lahko odvzemamo in dodajamo.

```
>>> s = [6, 9, 4, 1]
>>> s[1]
9
>>> s.append(7)
>>> s
[6, 9, 4, 1, 7]
>>> del s[3]
>>> s
[6, 9, 4, 7]
>>> s[-2:]
[4, 7]
```

Slovar (angl. *dictionary*, podatkovni tip pa se imenuje `dict`) je podoben seznamu, le da se na predalčke sklicujemo z njihovimi *ključi* (angl. *key*, ki so lahko števila, lahko pa tudi nizi, logične vrednosti, terke... Temu, kar je shranjeno v predalčku bomo rekli *vrednost* (*value*).

Seznam in slovar se že od daleč (če niste kratkovidni) razlikujeta po oglatosti. Seznam smo opisali tako, da smo v *oglatih oklepajih* našteli njihove *elemente*. Slovar opišemo tako, da v *zavitih oklepajih* naštejemo pare *ključ: vrednost*.

Stalno omenjani Benjamin si lahko takole sestavi slovar s telefonskimi številkami svojih oboževalk:

[1]:
```
stevilke = {"Ana": "041 310239", "Berta": "040 318319", "Cilka": "041 103194",␣
↪"Dani": "040 193831",
            "Ema": "051 123123", "Fanči": "040 135367", "Helga": "+49 175 4728␣
↪475"}
```

Do vrednosti, shranjenih v slovarju, pride podobno kot do vrednosti v seznamu: z indeksiranjem, le da v oglate oklepaje ne napiše zaporedne številke, temveč ključ.

[2]:
```
stevilke["Ana"]
```

[3]:
```
stevilke["Dani"]
```

Slovarji ne poznajo vrstnega reda. V slovarju ni prvega, drugega, tretjega elementa, saj ključi niso nujno primerljivi - v istem slovarju se lahko kot ključi pojavljajo števila in nizi (kar sicer ni dobra ideja), zdaj pa uredite to, če morete. Vsaj v splošnem se to ne da. Od različice 3.5 si Python neuradno (bolj "slučajno") zapomni vrstni red dodajanja, od 3.7 pa je to del standarda jezika. Ker ni vrstnega reda, tudi ni rezin.

[4]:
```
stevilke["Ana":"Cilka"]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-ee2b86ca837c> in <module>
----> 1 stevilke["Ana":"Cilka"]

TypeError: unhashable type: 'slice'
```

Tole sporočilo o napaki je sicer zelo čudno, vendar povsem smiselno. Boste razumeli, ko boste veliki. ;)

O "mehaniki" slovarjev morate vedeti le tole: slovarji so hitri, saj elementov v resnici ne iščejo. Na nek za zdaj skrivnosten način "vedo", kje je vrednost, ki pripada danemu ključu. Ne da bi iskali, vedo, kam morajo pogledati. Tudi dodajanje novih elementov v slovar je zelo hitro, enako tudi brisanje. Cena, ki jo plačamo za to, je dvojna. Ključi so lahko le podatkovni tipi, ki so nespremenljivi. Nespremenljivi podatkovni tipi so, vemo, nizi, števila, logične vrednosti in terke. Zgoraj smo kot ključ uporabili niz. To je OK. Če bi poskušali kot ključ uporabiti seznam, ne bi šlo. (V resnici je stvar malenkost bolj zapletena, ključ je lahko vsak podatkovni tip, ki je "razpršljiv", vendar to ni za bruce). Omejitve veljajo le za ključe. Vrednost je lahko karkoli.

Drugi obrok cene, ki jo plačamo za učinkovitost slovarjev, je poraba pomnilnika: slovar porabi nekako dvakrat več pomnilnika kot seznam. Koliko, točno, več, je odvisno od velikost slovarja in tega, kaj shranjujemo. Pri tem predmetu se s pomnilnikom ne obremenjujte.

Aha, še tretji obrok cene, iz drobnega tiska: vsak ključ se lahko pojavi največ enkrat. Če poskušamo prirediti neko vrednost ključu, ki že obstaja, le povozimo staro vrednost. Ampak to je tako ali tako pričakovati. Tudi v seznamu nimamo nikoli dveh različnih elementov na istem mestu.

**Kaj lahko počnemo s slovarji?** Lahko jim dodajamo nove elemente: preprosto priredimo jim nov element.

[5]:
```
stevilke["Iva"] = "040 222333"

stevilke
```

[5]:
```
{'Ana': '041 310239',
 'Berta': '040 318319',
 'Cilka': '041 103194',
 'Dani': '040 193831',
 'Ema': '051 123123',
 'Fanči': '040 135367',
 'Helga': '+49 175 4728 475',
```

```
 'Iva': '040 222333'}
```

`append` ne obstaja, saj nima smisla: ker ni vrstnega reda, ne moremo dodajati na konec. Sploh pa moramo tako ali tako povedati ključ, zato dodajamo, kot smo videli v gornjem primeru in `append` niti ni potreben. Prav tako (ali pa še bolj) ne obstaja `insert`, saj prav tako (ali pa še bolj) nima smisla.

Vprašamo se lahko, ali v slovarju obstaja določen ključ.

[6]:
```
"Cilka" in stevilke
```

[6]:
```
True
```

[7]:
```
"Jana" in stevilke
```

[7]:
```
False
```

Če se Cilka skrega z Benjaminom, jo lahko ta pobriše (mislim, pobriše njeno številko).

[8]:
```
del stevilke["Cilka"]

stevilke
```

[8]:
```
{'Ana': '041 310239',
 'Berta': '040 318319',
 'Dani': '040 193831',
 'Ema': '051 123123',
 'Fanči': '040 135367',
 'Helga': '+49 175 4728 475',
 'Iva': '040 222333'}
```

[9]:
```
"Cilka" in stevilke
False
```

[9]:
```
False
```

Če gremo prek slovarjev z zanko `for`, dobimo vrednosti ključev.

[10]:
```
for ime in stevilke:
    print(ime)
```

```
Ana
Berta
Dani
Ema
Fanči
Helga
Iva
```

Seveda lahko ob vsakem imenu izpišemo tudi številko.

[11]:
```
for ime in stevilke:
    print(ime + ":", stevilke[ime])
```

```
Ana: 041 310239
Berta: 040 318319
Dani: 040 193831
Ema: 051 123123
Fanči: 040 135367
Helga: +49 175 4728 475
Iva: 040 222333
```

Vendar to ni najbolj praktično. Pomagamo si lahko s tremi metodami slovarja, ki vrnejo vse ključe, vse vrednosti in vse pare ključ-vrednost. Imenujejo se (ne preveč presenetljivo) `keys()`, `values()` in `items()`. Delamo se lahko, da vračajo sezname ključev, vrednosti oziroma parov ... čeprav v resnici vračajo le nekaj, prek česar lahko gremo z zanko `for`. (Ja, podobna finta kot `range` in `enumerate` in `zip`. Vse to bo postalo jasneje prihodnji teden.)

Najprej poglejmo `items()`.

[12]:
```
for ime, stevilka in stevilke.items():
    print(ime + ":", stevilka)
```

```
Ana: 041 310239
Berta: 040 318319
Dani: 040 193831
Ema: 051 123123
Fanči: 040 135367
Helga: +49 175 4728 475
Iva: 040 222333
```

Tako kot sem ob zanki `for` težil, da ne uporabljajte `for i in range(len(s))` temveč `for e in s` in da že v glavi zanke razpakirajte terko, kadar je to potrebno, bom težil tudi tule: uporabljajte `items` in vaši programi bodo takoj preglednejši in s tem pravilnejši.

Metoda `values` vrne vse vrednosti, ki so v slovarju. V našem primeru torej telefonske številke. Metodo `values` človek včasih potrebuje, prav pogosto pa ne.

V nasprotju s tem metodo `keys` potrebujejo samo študenti. Ne vem točno, zakaj sploh obstaja. No, vem, zato da študenti lahko pišejo `for ime in stevilke.keys()` namesto `for ime in stevilke`. Druge uporabne vrednosti pa ne vidim. :)

Skratka: ne uporabljajte metode `keys`.

Slovar ima še dve metodi, ki smo ju videli tudi pri seznamu: metodo, ki sprazni slovar in drugo, ki naredi nov, enak slovar.

[13]:
```
stevilke2 = stevilke.copy()
stevilke2
```

[13]:
```
{'Ana': '041 310239',
 'Berta': '040 318319',
 'Dani': '040 193831',
 'Ema': '051 123123',
 'Fanči': '040 135367',
 'Helga': '+49 175 4728 475',
 'Iva': '040 222333'}
```

[14]:
```
stevilke2.clear()
stevilke2
```

[14]:
```
{}
```

Omenimo le še eno od slovarjevih metod, `get`. Ta dela podobno kot indeksiranje, `stevilke.get("Ana")` naredi isto kot `stevilke["Ana"]`. Metodo `get` uporabimo, kadar želimo v primeru, da ključa morda ni v slovarju, dobiti neko privzeto vrednost. Le-to podamo kot drugi argument.

[15]:
```
stevilke.get("Ema", "ni številke")
```

[15]:
```
'051 123123'
```

[16]:
```
stevilke.get("Greta", "ni številke")
```

[16]:
```
'ni številke'
```

Še enkrat: ne pišite `stevilke.get("Ema")`. To je čudno. Piše se `stevilke["Ema"]`. Metodo `get` uporabite le takrat, kadar niste prepričani, ali slovar vsebuje ta ključ, in bi radi podali privzeto vrednost.

**Primer: kronogrami** Neka stara domača naloga je šla takole.

Veliko latinskih napisov, ki obeležujejo kak pomemben dogodek, je napisanih v obliki kronograma: če seštejemo vrednosti črk, ki predstavljajo tudi rimske številke (I=1, V=5, X=10, L=50, C=100, D=500, M=1000), dajo letnico dogodka.

Tako, recimo, napis na cerkvi sv. Jakoba v Opatiji, CVIVS IN HOC RENOVATA LOCO PIA FVLGET IMAGO SIS CVSTOS POPVLI SANCTE IACOBE TVI, da vsoto 1793, ko je bila cerkev prenovljena (o čemer govori napis).

Pri tem obravnavamo vsak znak posebej: v besedil EXCELSIS bi prebrali X + C + L + I = 10 + 100 + 50 + 1 = 161 in ne XC + L + I = 90 + 50 + 1 = 141.

Napiši program, ki izračuna letnico za podani niz.

Očitna rešitev je:

[17]:
```
def kronogram(s):
    v = 0
    for c in s:
```

```
        if c=="I":
            v += 1
        elif c=="V":
            v += 5
        elif c=="X":
            v += 10
        elif c=="L":
            v += 50
        elif c=="C":
            v += 100
        elif c=="D":
            v += 500
        elif c=="M":
            v += 1000
    return v

napis = "CVIVS IN HOC RENOVATA LOCO PIA FVLGET IMAGO SIS CVSTOS POPVLI SANCTE␣
 ↪IACOBE TVI"
kronogram(napis)
```

[17]: 1793

Pri njej vsi, ki poznajo stavek `switch` oz. `case` postokajo, kako je možno, da Python tega stavka nima. Ne, nima ga, ker ga skoraj nikoli ne potrebujemo, vsaj ne v takšni obliki, kot ste ga navajeni iz C-ju podobnih jezikov. Tisto, kar delamo z njim, pogosto rešimo (tudi) s slovarji.

V slovar si bomo zapisali, katero številko pomeni katera črka.

[18]:
```
stevke = {"I": 1, "V": 5, "X": 10, "L": 50, "C": 100, "D": 500, "M": 1000}
```

Funkcija zdaj deluje tako, da gre prek niza in za vsako črko preveri, ali je v slovarju. Če je ni, ne pomeni številke; če je, prištejemo toliko, kolikor je vredna.

[19]:
```
def kronogram(s):
    v = 0
    for c in s:
        if c in stevke:
            v += stevke[c]
    return v

kronogram(napis)
```

[19]: 1793

Še hitreje gre z `get`; če črke ni, je njena privzeta vrednost 0.

[20]:
```
def kronogram(s):
    v = 0
```

```
    for c in s:
        v += stevke.get(c, 0)
    return v

kronogram(napis)
```

[20]: 1793

Ob tem si ne moremo kaj, da ne bi poškilili za en teden naprej, ko se bomo učili bolj funkcijskega sloga programiranja in znali biti še krajši in jedrnatejši, spet predvsem po zaslugi slovarjev.

[21]:
```
def kronogram(s):
    return sum(stevke.get(c, 0) for c in s)

kronogram(napis)
```

[21]: 1793

### 0.2 Slovarji s privzetimi vrednostmi

Slovarji so uporabne reči. V veliko primerih pa uporabimo neko različico slovarja, ki ima še en dodaten superpower.

Denimo, da smo dobili v preskušanje igralno kocko, za katero nas zanima, ali goljufa. Stokrat smo jo vrgli in zabeležili rezultate.

[22]:
```
meti = [4, 4, 4, 3, 2, 3, 5, 3, 3, 4, 6, 6, 6, 1, 3,
        6, 6, 4, 1, 4, 6, 1, 4, 4, 4, 6, 4, 6, 5, 5, 6, 6, 2, 4, 4, 6,
        3, 2, 6, 1, 3, 6, 3, 2, 6, 6, 4, 6, 4, 2, 4, 4, 1, 1, 6, 2, 6,
        6, 4, 3, 4, 2, 6, 5, 6, 3, 2, 5, 1, 5, 3, 6, 4, 6, 2, 2, 4, 1,
        4, 4, 3, 1, 4, 2, 1, 3, 1, 4, 6, 1, 1, 3, 4, 1, 4, 3, 2, 4, 6, 6]
```

Zanima nas, kolikokrat je padla katera številka. Nič lažjega. Najprej si pripravimo seznam s sedmimi ničlami.

[23]:
```
kolikokrat = [0] * 7
kolikokrat
```

[23]: [0, 0, 0, 0, 0, 0, 0]

Zdaj nam bo, recimo `kolikokrat[3]` povedal, kolikokrat je padla trojka. (Čemu sedem? Saj ima kocka samo šest ploskev. Boste videli. Finta.) Zdaj pa štejmo: pojdimo čez vse mete in povečujmo števce.

[24]:
```
for met in meti:
    kolikokrat[met] += 1

kolikokrat
```

[24]:
```
[0, 14, 12, 15, 27, 6, 26]
```

(Kje je finta? `kolikokrat[0]` bo pove, kolikokrat je padla ničla. Nikoli pač. Napišite isto reč s seznamom dolžine šest, ne sedem. Ni problem, ampak tako, kot smo nardili je preprostejše.)

(Kocka je res videti nekoliko sumljiva: štirke in šestice so nekam pogoste na račun petk. A pustimo statistikom vprašanje, ali je to še lahko naključno ali ne.)

Ups, nalogo smo rešili kar s seznamom! Da, to gre, ker so "predalčki" seznama oštevilčeni, tako kot ploskve kocke. Tole pa ne bo šlo: tule je seznam telefonskih številk deklet, ki jih je danes klical Benjamin. Katero je poklical kolikokrat?

[25]:
```
klici = ['041 103194', '040 193831', '040 318319', '040 193831', '041 310239',
         '040 318319', '040 318319', '040 318319', '040 193831', '040 193831',
         '040 193831', '040 193831', '040 193831', '040 318319', '040 318319',
         '040 318319', '040 193831', '040 318319', '041 103194', '041 103194',
         '041 310239', '040 193831', '041 103194', '041 310239', '041 310239',
         '040 193831', '041 310239', '041 103194', '040 193831', '040 318319']
```

Za tako velike številke bi moral imeti seznam zelo veliko predalčkov. Še huje bi bilo, če bi namesto seznama številk dobili seznam klicanih oseb.

[26]:
```
klici = ['Cilka', 'Dani', 'Berta', 'Dani', 'Ana', 'Berta', 'Berta',
         'Berta', 'Dani', 'Dani', 'Dani', 'Dani', 'Dani', 'Berta', 'Berta',
         'Berta', 'Dani', 'Berta', 'Cilka', 'Cilka', 'Ana', 'Dani', 'Cilka',
         'Ana', 'Ana', 'Dani', 'Ana', 'Cilka', 'Dani', 'Berta']
```

Tu smo s seznamom pogoreli. No, ne čisto; rešitev s seznami seveda obstaja, je pa zoprna - podobna je tistemu, kar smo v začetku počeli z bančnimi računi.

S slovarji je veliko lepše:

[27]:
```
pogostosti = {}
for ime in klici:
    if ime not in pogostosti:
        pogostosti[ime] = 0
    pogostosti[ime] += 1

pogostosti
```

[27]:
```
{'Cilka': 5, 'Dani': 11, 'Berta': 9, 'Ana': 5}
```

Ob vsakem novem klicu preverimo, ali je klicano `ime` že v slovarju. Če ga ni, da dodamo. Nato - najsibo ime novo ali ne - povečamo števec klicev pri tej osebi.

Ker je ta stvar resnično pogosta, nam Python pomaga z modulom `collections`, ki vsebuje podatkovni tip `defaultdict`. (Modul, ki vsebuje podatkovni tip?! Da, da; tudi to je ena od stvari, ki jih bomo našli v modulih.) Ta se obnaša tako kot slovar, z eno izjemo: če zahtevamo kak element, ki ne obstaja, si ga meni nič tebi nič izmisli. Točneje, doda ga v slovar in mu postavi privzeto vrednost. Katero, določimo. Pri tem ne podamo privzete vrednosti, temveč "funkcijo", ki bo vračala privzeto vrednost. `defaultdict` bo ustvarjal te, nove vrednosti tako, da bo poklical to funkcijo

brez argumentov in kot privzeto vrednost uporabil, kar vrne funkcija, ki jo v ta namen vsakič sproti pokliče.

Zelo pogosto bo privzeta vrednost 0 in funkcija, ki vrača 0, se imenuje, hm, `int`.

("Funkcija" `int`, je vedno sumljivejša in sumljivejša. Že od začetka smo v zvezi z njo dajali besedo "funkcija" pod narekovaje, zdaj pa vidimo, da zmore vedno več in več stvari. Pa še enako ji je ime kot tipu in funkcij, ki jim je ime enako kot tipom, je vedno več. Kakšna skrivnost se skriva za tem? To boste izvedeli v enem od prihodnjih napetih nadaljevanj Programiranja 1.)

Preštejmo torej še enkrat Benjaminove klice, tokrat s slovarjem s privzetimi vrednostmi.

[28]:
```
import collections

pogostosti = collections.defaultdict(int)
for ime in klici:
    pogostosti[ime] += 1

pogostosti
```

[28]:
```
defaultdict(int, {'Cilka': 5, 'Dani': 11, 'Berta': 9, 'Ana': 5})
```

Ni to kul?

Poglejmo si nekaj, kar je kul še bolj.

### 0.3 Števec

Preštevanje je tako pogosta reč, da obstaja zanj specializiran tip. Tako kot `defaultdict` je v modulu `collections`, imenuje pa se `Counter`.

[29]:
```
stevilo_klicev = collections.Counter(klici)
stevilo_klicev
```

[29]:
```
Counter({'Cilka': 5, 'Dani': 11, 'Berta': 9, 'Ana': 5})
```

Komentirali ne bomo veliko, ker še ne znamo. Že ob tem, da sem temu rekel tip, sem se moral ugrizniti v jezik, saj bi raje govoril o razredu. Kaj je in kako deluje, pa nas še presega. Zna pa pravzaprav še veliko stvari, tako da tem, ki jih zanima, priporočam, da si ga ogledajo. Mimogrede:

[30]:
```
napis = "CVIVS IN HOC RENOVATA LOCO PIA FVLGET IMAGO SIS" \
        "CVSTOS POPVLI SANCTE IACOBE TVI"

collections.Counter(napis)
```

[30]:
```
Counter({'C': 6,
         'V': 7,
         'I': 8,
         'S': 6,
         ' ': 12,
         'N': 3,
```

```
'H': 1,
'O': 8,
'R': 1,
'E': 4,
'A': 6,
'T': 5,
'L': 3,
'P': 3,
'F': 1,
'G': 2,
'M': 1,
'B': 1})
```

Se pravi, da lahko kronogram rešimo tudi z

[31]:
```
def kronogram(s):
    crke = collections.Counter(s)
    return crke["I"] + 5 * crke["V"] + 10 * crke["X"] + 50 * crke["L"] + \
           100 * crke["C"] + 500 * crke["D"] + 1000 * crke["M"]

kronogram(napis)
```

[31]:
```
1793
```

### 0.4 Množice

Množice so podobne seznamom, a s to razliko, da lahko vsebujejo vsak element samo enkrat. Po drugi strani (in ne le po drugi strani, tudi tehnično) pa so podobne slovarjem. Vsebujejo lahko le elemente, ki so nespremenljivi, poleg tega pa lahko zelo hitro ugotovimo, ali množica vsebuje določen element ali ne, na podoben način kot pri slovarjih hitro ugotovimo, ali vsebujejo določen ključ ali ne.

Množice zapišemo z zavitimi oklepaji, tako kot smo vajeni pri matematiki.

[32]:
```
danasnji_klici = {"Ana", "Cilka", "Eva"}
```

Tako lahko sestavimo le neprazno množico. Če napišemo le oklepaj in zaklepaj, `{}`, pa dobimo slovar. (Čemu so se odločili, naj bo to slovar, ne množica? Slovar je bil prej, množice je Python dobil kasneje. Zato. Poleg tega pa slovarje potrebujemo res velikokrat, množice pa precej redkeje.) Če hočemo narediti prazno množico, rečemo

[33]:
```
prazna = set()
```

"Funkcija" `set` je malo podobna "funkciji" `int`: damo ji lahko različne argumente, pa jih bo spremenila v množico. Damo ji lahko, recimo, seznam, pa bomo dobili množico z vsemi elementi, ki se pojavijo v njem.

[34]:
```
set([1, 2, 3])
```

[34]:
```
{1, 2, 3}
```

[35]:
```
set(range(5))
```

[35]:
```
{0, 1, 2, 3, 4}
```

[36]:
```
set([6, 42, 1, 3, 1, 1, 6])
```

[36]:
```
{1, 3, 6, 42}
```

Poleg seznamov lahko množicam podtaknemo karkoli, prek česar bi lahko šli z zanko `for`, recimo niz ali slovar.

[37]:
```
set("Benjamin")
```

[37]:
```
{'B', 'a', 'e', 'i', 'j', 'm', 'n'}
```

[38]:
```
set(stevilke)
```

[38]:
```
{'Ana', 'Berta', 'Dani', 'Ema', 'Fanči', 'Helga', 'Iva'}
```

Spremenljivka `stevilke` (še vedno) vsebuje slovar, katerega ključi so imena Benjaminovih oboževalk. Ker zanka prek slovarja "vrača" ključe, bo tudi množica, ki jo sestavimo iz slovarja, vsebovala ključe.

V množico lahko dodajamo elemente in vprašamo se lahko, ali množica vsebuje določen element.

[39]:
```
s = set("Benjamin")
```

[40]:
```
"e" in s
```

[40]:
```
True
```

[41]:
```
"u" in s
```

[41]:
```
False
```

[42]:
```
s.add("u")
```

[43]:
```
"u" in s
```

[43]:
```
True
```

[44]:
```
s.add("a")
s.add("a")
s.add("a")
s
```

[44]:
```
{'B', 'a', 'e', 'i', 'j', 'm', 'n', 'u'}
```

Na koncu smo poskušali v množico dodati element, ki ga že vsebuje. To seveda ne gre, množica vsak element vsebuje le enkrat.

Če imamo dve množici, lahko izračunamo njuno unijo, presek, razliko (elementi, ki se pojavijo v prvi, ne pa tudi v drugi množici) in simetrično razliko (elementi, ki se pojavijo v eni množici in v drugi) …

[45]:
```
u = {1, 2, 3}
v = {3, 4, 5}
u | v
```

[45]:
```
{1, 2, 3, 4, 5}
```

[46]:
```
u & v
```

[46]:
```
{3}
```

[47]:
```
u - v
```

[47]:
```
{1, 2}
```

[48]:
```
u ^ v
```

[48]:
```
{1, 2, 4, 5}
```

Tako kot lahko pri številih uporabimo `+=` za prištevanje (in namesto `x = x + a` pišemo `x += a`), in, podobno, odštejemo, primnožimo in razdelimo z `-=`, `*=` in `/=`, lahko tudi priunijamo, odsekamo ali odštejemo množico z `&=`, `|=` in `-=`. Tako je, na primer `u &= v` isto kot `u = u & v`.

Preverimo lahko tudi, ali je neka množica podmnožica (ali nadmnožica druge). To najpreprosteje storimo kar z operatorji za primerjanje.

[49]:
```
u = {1, 2, 3}
```

[50]:
```
{1, 2} <= u
```

[50]:
```
True
```

[51]:
```
{1, 2, 3, 4} <= u
```

[51]:
```
False
```

[52]:
```
{1, 2, 3} <= u
```

[52]:
```
True
```

[53]:
```
{1, 2, 3} < u
```

[53]:
```
False
```

`{1, 2, 3}`, je podmnožica `u`-ja, ni pa njegove *prava podmnožica*, saj vsebuje kar cel `u`.

Z množicami je mogoče početi še marsikaj zanimivega - vendar bodi dovolj.