06 slovarji in množice

January 28, 2024

0.1 Slovar

Vsak normalen jezik ima poleg tabel (seznamov v Pythonu) tudi neko obliko asociativnih tabel. Iz drugih jezikov to poznate kot `HashMap` (Java), `object` in `Map` (javascript), `array` (php), `stl::unordered_map` (C++), `Dictionary` (C#), `Map` (Kotlin).

Ob predpostavki, torej, da to že poznate, le na hitro povejmo: do elementov dostopamo s celoštevilskimi indeksi, do elementov slovarjev s ključi. Dostopanje do elementov slovarja (vključno z vstavljanjem in brisanjem) je zelo hitro in je neodvisno od velikosti slovarja. Slovar z milijon elementi je enako hiter kot tak s tremi.

Ključi slovarja so lahko nespremenljivi objekti: števila, nizi, terke in `None`, `True`, `False`. Poleg njih pa še poljubni drugi objekti, ki imajo definirano metodo `__hash__`. Vsak ključ se lahko seveda pojavi le enkrat.

Vrednosti so lahko seveda karkoli.

Slovarje zapisujemo z zavitimi oklepaji, ključ in vrednost ločimo z dvopičjem.

[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"}
```

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

[2]:
```
'041 310239'
```

Slovarji *načelno* ne poznajo vrstnega reda. Od Pythona 3.6 naprej so elementi sicer urejeni po vrstnem redu dodajanja, vendar to opazimo zgolj pri izpisovanju in prehodu z zankami. Vseeno pa slovarji nimajo metod kot sta `append` in `insert` in rezin.

Elemente dodajamo s prirejanjem.

[3]:
```
stevilke["Iva"] = "040 222333"
```

Operator `in` preveri, ali v slovarju obstaja določen ključ.

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

[4]:
```
True
```

Po vrednostih ne moremo iskati - vsaj ne z `in`.

Elemente brišemo, tako kot v seznamih, z operatorjem `del`.

[5]:
```
del stevilke["Cilka"]
```

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

[6]:
```
for ime in stevilke:
    print(ime)
```

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

Seveda lahko ob vsakem imenu izpišemo tudi številko.

[7]:
```
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()`.

[8]:
```
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.

[9]:
```
stevilke2 = stevilke.copy()
stevilke2
```

[9]:
```
{'Ana': '041 310239',
 'Berta': '040 318319',
 'Dani': '040 193831',
 'Ema': '051 123123',
 'Fanči': '040 135367',
 'Helga': '+49 175 4728 475',
 'Iva': '040 222333'}
```

[10]:
```
stevilke2.clear()
stevilke2
```

[10]:
```
{}
```

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.

[11]:
```
stevilke.get("Ema", "ni številke")
```

[11]:
```
'051 123123'
```

[12]:
```
stevilke.get("Greta", "ni številke")
```

[12]:
```
'ni številke'
```

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

0.2 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:

[13]:
```
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)
```

[13]:
```
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.

[14]:
```
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.

[15]:
```
def kronogram(s):
    v = 0
    for c in s:
        if c in stevke:
            v += stevke[c]
    return v

kronogram(napis)
```

[15]: 1793

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

[16]:
```
def kronogram(s):
    v = 0
    for c in s:
        v += stevke.get(c, 0)
    return v

kronogram(napis)
```

[16]: 1793

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

[17]:
```
def kronogram(s):
    return sum(stevke.get(c, 0) for c in s)

kronogram(napis)
```

[17]: 1793

0.3 Slovarji s privzetimi vrednostmi

Na teh, dodatnih predavanjih smo že nekajkrat uporabili `defaultdict`. Recimo, da imamo seznam Benjaminovih klicev in nas zanima, kolikokrat je poklical koga.

[18]:
```
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']

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

```
pogostosti
```

[18]:
```
{'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, si pomagamo z `defaultdict` iz modula `collections`. Ta se obnaša tako kot slovar, le da si vsakič, ko zahtevamo kak element, ki ne obstaja, vrne privzeto vrednost zanj. To vrednost dobi tako, da pokliče funkcijo, ki jo podamo kot argument `defaultdict`-u.

Zelo pogosto bo privzeta vrednost 0 ali pa prazen seznam, zato `defaultdictu` kot argument pogosto podamo `int` ali `list`.

[19]:
```
import collections

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

pogostosti
```

[19]:
```
defaultdict(int, {'Cilka': 5, 'Dani': 11, 'Berta': 9, 'Ana': 5})
```

Ni to kul?

Poglejmo si nekaj, kar je kul še bolj.

0.4 Števec

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

[20]:
```
stevilo_klicev = collections.Counter(klici)
stevilo_klicev
```

[20]:
```
Counter({'Cilka': 5, 'Dani': 11, 'Berta': 9, 'Ana': 5})
```

Kot argument mu podamo karkoli, čez kar se da iti z zanko `for`.

[21]:
```
napis = "CVIVS IN HOC RENOVATA LOCO PIA FVLGET IMAGO SIS" \
        "CVSTOS POPVLI SANCTE IACOBE TVI"

collections.Counter(napis)
```

[21]:
```
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

[22]:
```
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)
```

[22]:
```
1793
```

0.5 Množice

Množice so kot seznami, ki lahko vsebujejo vsak element samo enkrat. Po izvedbi pa so bolj podobne slovarjem: vanje lahko shranjujemo samo nespremenljive tipe, čas, potreben za dodajanje, brisanje in preverjanje, ali množica vsebuje določen element, pa je neodvisen od velikosti množice.

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

[23]:
```
danasnji_klici = {"Ana", "Cilka", "Eva"}
```

Prazne množice pa ne sestavimo z `{}`, ker bi bil to slovar. Za prazno množico pokličemo konstruktor, `set()`. (Čemu tako? Slovar je bil prej, množice je Python dobil kasneje. Zato. Poleg tega pa slovarje potrebujemo večkrat kot mmnožice).

Konstruktorju `set` lahko sicer podamo poljubno stvar, čez katero lahko gremo z zanko `for`.

[24]:
```
set([1, 2, 3])
```

[24]:
```
{1, 2, 3}
```

[25]:
```
set(range(5))
```

[25]:
```
{0, 1, 2, 3, 4}
```

[26]:
```
set([6, 42, 1, 3, 1, 1, 6])
```

[26]:
```
{1, 3, 6, 42}
```

[27]:
```
set("Benjamin")
```

[27]:
```
{'B', 'a', 'e', 'i', 'j', 'm', 'n'}
```

[28]:
```
set(stevilke)
```

[28]:
```
{'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.

[29]:
```
s = set("Benjamin")
```

[30]:
```
"e" in s
```

[30]:
```
True
```

[31]:
```
s.add("a")
s.add("a")
s.add("a")
s
```

[31]:
```
{'B', 'a', 'e', 'i', 'j', 'm', 'n'}
```

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 …

[32]:
```
u = {1, 2, 3}
v = {3, 4, 5}
u | v
```

[32]:
```
{1, 2, 3, 4, 5}
```

[33]:
```
u & v
```

[33]:
```
{3}
```

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

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

[35]:
```
{1, 2} <= u
```

[35]:
```
True
```

[36]:
```
{1, 2, 3, 4} <= u
```

[36]:
```
False
```

[37]:
```
{1, 2, 3} <= u
```

[37]:
```
True
```

[38]:
```
{1, 2, 3} < u
```

[38]:
```
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.