

19. Slovníky (dict)

video prezentácia

- [vyhľadávanie](#)
- [slovníky](#)
- [json](#)

Na 14. cvičení ste riešili úlohu, v ktorej ste zostavovali metódy pre triedu telefónny zoznam. Riešenie úlohy by mohlo vyzerat' asi takto:

```
class TelefonnyZoznam:
    def __init__(self):
        self.zoznam = []

    def __str__(self):
        vysl = []
        for meno, telefon in self.zoznam:
            vysl.append(f'{meno}: {telefon}')
        return '\n'.join(vysl)

    def pridaj(self, meno, telefon):
        for i in range(len(self.zoznam)):
            if self.zoznam[i][0] == meno:
                self.zoznam[i] = meno, telefon
                return
        self.zoznam.append((meno, telefon))
```

Vidíte, že do zoznamu ukladáme dvojice (meno, telefon).

Jednoduchý test:

```
tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
print(tz)

Jana: 0999020304
Juro: 0911111111
Jozo: 0212345678
```

Hľadanie údajov

Do tejto triedy pridajme ešte metódy `__contains__()` a `zisti()`, vďaka ktorým budeme vedieť zistiť, či sa dané meno nachádza v zozname, prípadne pre dané meno zistíme jeho telefónne číslo:

```
class TelefonnyZoznam:
    ...

    def __contains__(self, hladane_meno):
        for meno, telefon in self.zoznam:
```

```

        if meno == hladane_meno:
            return True
        return False

    def zisti(self, hladane_meno):
        for meno, telefon in self.zoznam:
            if meno == hladane_meno:
                return telefon
        raise ValueError('zadane meno nie je v zozname')

```

Opäť otestujeme:

```

tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
print(tz)
print('Juro ma telefon', tz.zisti('Juro'))
print('Fero je v zozname:', 'Fero' in tz)          # volanie tz.__contains__('Fero')
print('Fero ma telefon', tz.zisti('Fero'))

Jana: 0999020304
Juro: 0911111111
Jozo: 0212345678
Juro ma telefon 0911111111
Fero je v zozname: False
...
ValueError: zadane meno nie je v zozname

```

Zamerajme sa teraz na spôsob hľadania v tomto telefónnom zozname: všetky tri metódy `pridaj()`, `__contains__()` aj `zisti()` postupne (sekvenčne) pomocou for-cyklu prechádzajú celý zoznam a kontrolujú, či pritom nenašli hľadané meno. Zrejme pre veľký telefónny zoznam (napríklad telefónny zoznam New Yorku by mohol obsahovať aj niekoľko miliónov dvojíc (`meno`, `číslo`)) takéto **sekvenčné** prehľadávanie môže trvať už dosť dlho.

Naučme sa jednoduchý spôsob, akým môžeme odmerať čas behu nejakého programu. Hoci takéto meranie vôbec nie je presné, môže nám to dať nejaký obraz o tom, ako dlho trvajú niektoré časti programu. Na meranie použijeme takúto šablónu:

```

import time

start = time.time()          # začiatok merania času

# nejaký výpočet

koniec = time.time()         # koniec merania času

print(round(koniec - start, 3))

```

Volanie funkcie `time.time()` vráti momentálny stav nejakého počítadla sekúnd. Keď si tento stav zapamätáme (v premennej `start`) a o nejaký čas opäť zistíme stav počítadla, rozdiel týchto dvoch hodnôt nám vráti približný počet sekúnd koľko ubehlo medzi týmito dvomi volaniami `time.time()`. Túto hodnotu sa dozvedáme ako desatinné číslo, takže vidíme aj milisekundy (výsledný rozdiel zaokrúhľujeme na 3 desatinné miesta).

Začnime s meraním tohto jednoduchého programu na výpočet súčtu `n` prirodzených čísel:

```

import time

n = int(input('zadaj n: '))

```

```

start = time.time()                # začiatok merania času
sucet = 0
for i in range(1, n+1):
    sucet += i
print('cas =', round(time.time()-start, 3)) # čas trvania = koniec merania času - začiatok
print('sucet =', sucet)

```

Tento program spustíme niekoľkokrát s rôzne veľkým `n` (zrejme na rôznych počítačoch sa tieto časy budú trochu líšiť):

```

zadaj n: 10000
cas = 0.002
sucet = 50005000

zadaj n: 100000
cas = 0.021
sucet = 5000050000

zadaj n: 1000000
cas = 0.235
sucet = 500000500000

```

Môžete si všimnúť istú závislosť trvania programu od veľkosti `n`: keď zadáme 10-krát väčšiu vstupnú hodnotu, výpočet bude trvať približne 10-krát tak dlho. Zrejme je to nejaká **lineárna závislosť**: čas behu programu závisí od veľkosti `n` lineárne.

Teraz namiesto výpočtu sumy budeme merať približný čas hľadania údajov v nejakej tabuľke, pričom použijeme **sekvenčné** prehľadávanie: postupne budeme prechádzať celý zoznam pomocou for-cyklu. Pre jednoduchosť testovania budeme pracovať len so zoznamom celých čísel, ktoré najprv vygenerujeme náhodne:

```

import time
import random

def hladaj(hodnota):
    for prvok in zoznam:
        if prvok == hodnota:
            return True
    return False

n = int(input('zadaj n: '))
zoznam = []
for i in range(n):
    zoznam.append(random.randrange(2*n))
start = time.time()                # začiatok merania času
for i in range(0, 2*n, 2):
    hladaj(i)
cas = time.time() - start          # celkový čas trvania výpočtu
print('cas =', round(cas / n * 1000, 3))

```

Najprv sme vygenerovali `n`-prvkový zoznam náhodných čísel z intervalu `<0, 2n-1>`. Pre toto generovanie zoznamu sme ešte čas trvania nemerali. Samotný odmeriavaný test teraz `n`-krát hľadá v tomto zozname nejakú hodnotu z intervalu `<0, 2n-1>` (zrejme sa pokúša hľadať len párne hodnoty). Na záver vypíše čas ale vydelený počtom hľadanií (počet volaní funkcie `hladaj()`) a aby to neboli príliš malé čísla, čas ešte vynásobí 1000, teda nedostávame sekundy, ale milisekundy.

Tento test niekoľkokrát spustíme pre rôzne `n`:

```

zadaj n: 100
cas = 0.005

zadaj n: 1000

```

```
cas = 0.05
zadaj n: 10000
cas = 0.496
zadaj n: 100000
cas = 5.921
```

čo môžeme vypočítavať z týchto výsledkov:

- pre 100-prvkový zoznam trvalo jedno hľadanie približne 0.005 milisekúnd (t.j. 5 milióntin sekundy)
- pre 1000-prvkový zoznam trvalo jedno hľadanie približne 0.05 milisekúnd, to znamená, že pre 10-krát väčší zoznam aj hľadanie trvá asi 10-krát tak dlho
- pre 10000-prvkový zoznam trvalo jedno hľadanie približne 0.5 milisekundy, čo asi potvrdzuje našu hypotézu, že čas hľadania je lineárne závislý od veľkosti zoznamu
- pre 10000-prvkový zoznam jedno hľadanie trvá skoro 6 milisekúnd
- ... môžeme si zatipovať, že v 1000000-prvkovom zozname by sme hľadali 100-krát dlhšie ako pre 10000, teda asi 600 milisekúnd, čo je viac ako pol sekundy

Asi vidíme, že telefónny zoznam New Yorku s 8 miliónmi mien bude náš program priemerne prehľadávať 5 sekúnd - a to už je dosť veľa.

Zrejme múdre programy na prehľadávanie telefónnych zoznamov používajú nejaké lepšie stratégie. V prvom rade sú takéto zoznamy usporiadané podľa mien, vďaka čomu sa bude dať hľadať oveľa rýchlejšie - budeme tomu hovoriť **binárne vyhľadávanie**.

Modul time

S týmto modulom sme sa už párkrát stretli na predchádzajúcich cvičeniach. Zhrňme pre nás najzaujímavejšie funkcie z tohto modulu:

- funkcia `time.localtime()` vráti momentálny dátum a čas v špeciálnom tvare `time.struct_time`, ktorý je vlastne pomenovanou n-ticou; s touto štruktúrou môžeme pracovať podobne ako s `tuple`, napríklad:

```
>>> time.localtime()
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=27, tm_hour=8, tm_min=53, tm_sec=5
3, tm_wday=2, tm_yday=331, tm_isdst=0)
>>> time.localtime()[3:]      # momentálny dátum v tvare (rok, mesiac, deň)
(2019, 11, 27)
>>> time.localtime()[3:6]     # momentálny čas v tvare (hodiny, minúty, sekundy)
(8, 54, 17)
>>> time.localtime()[6]       # momentálny deň v týždni, kde pondelok má hodnotu 0, te
da 2 označuje stredu
2
```

- funkcia `time.time()` vráti momentálny čas a dátum v tvare desatinného čísla (`float`); v skutočnosti je to počet sekúnd, ktoré uplynuli od nejakého konkrétného času v minulosti (tzv. epocha); toto číslo vieme späť previesť na dátum a čas (pomocou `time.localtime(čas)`) ale najčastejšie sa bude používať pri zisťovaní trvania (behu) nejakého výpočtu, napríklad:

```
>>> start = time.time()      # zapamätaj si momentálny čas v premennej start
>>> # nejaký výpočet
>>> koniec = time.time()     # zapamätaj si momentálny čas v premennej koniec
>>> koniec - start           # koľko sekúnd ubehlo medzi týmito dvoma časmi
33.83050608634949
>>> time.localtime(start)[3:6] # čas start v tvare (hodiny, minúty, sekundy)
(9, 7, 1)
```

```
>>> time.localtime(koniec)[3:6] # čas koniec v tvare (hodiny, minúty, sekundy)
(9, 7, 35)
```

- funkcia `time.sleep(sekundy)` pozdrží výpočet na zadaný počet sekúnd; je to niečo podobné ako `canvas.after(milisekundy)`; napríklad:

```
>>> start = time.time(); time.sleep(2.5); koniec = time.time()
>>> koniec - start
2.500455379486084
```

Binárne vyhľadávanie

Použijeme podobnú ideu, akou sme na 4. prednáške riešili úlohu na zisťovanie druhej odmocniny nejakého čísla. V tomto prípade využijeme ten fakt, že v reálnom telefónnom zozname sú všetky mená usporiadané podľa abecedy. Vďaka tomu, ak by sme si vybrali úplne náhodnú pozíciu v takomto zozname, na základe príslušnej hodnoty v tabuľke, sa vieme jednoznačne rozhodnúť, či ďalej budeme pokračovať v hľadaní v časti zoznamu pred touto pozíciou alebo za. Nebudeme ale túto pozíciu voliť náhodne, vyberieme pozíciu v strede zoznamu.

Ukážme to na príklade: máme telefónny zoznam, ktorý je usporiadaný podľa mien. Pre lepšie znázornenie použijeme len dvojpísmenové mená, napríklad tu je utriedený 13-prvkový zoznam a ideme v ňom hľadať meno `'hj'` (možno Hraško Ján):

0	1	2	3	4	5	6	7	8	9	10	11	12
ab	ad	dd	fm	hj	jj	ka	kz	mo	pa	rd	tz	zz
zac				stred						kon		

Označili sme tu začiatok a koniec intervalu zoznamu, v ktorom práve hľadáme: na začiatku je to kompletný zoznam od indexu 0 až po 12. Vypočítame `stred` (ako $(\text{zac} + \text{kon}) // 2$) - to je pozícia, kam sa pozrieme úplne na začiatku. Keďže v strede zoznamu je meno `'ka'`, tak zrejme všetky mená v zozname za týmto stredom sú v abecede vyššie a teda `'hj'` sa medzi nimi určite nenachádza (platí `'hj' < 'ka'`). Preto odteraz bude stačiť hľadať len v prvej polovici zoznamu teda v intervale od 0 do 5. Opravíme koncovú hranicu intervalu a vypočítame nový `stred`:

0	1	2	3	4	5	6	7	8	9	10	11	12
ab	ad	dd	fm	hj	jj	ka	kz	mo	pa	rd	tz	zz
zac		stred			kon							

Stredná pozícia v tomto prípade padla na meno `'dd'`. Keďže `'hj'` je v abecede až za `'dd'` (platí `'dd' < 'hj'`), opäť prepočítame hranice sledovaného intervalu a tiež nový `stred`:

0	1	2	3	4	5	6	7	8	9	10	11	12
ab	ad	dd	fm	hj	jj	ka	kz	mo	pa	rd	tz	zz
zac			stred		kon							

Už pri tomto treťom kroku algoritmu sa nám podarilo objaviť pozíciu hľadaného mena v zozname: `stred` teraz odkazuje presne na naše hľadané slovo.

Ak by sa hľadané slovo v tomto telefónnom zozname nenachádzalo (napríklad namiesto 'hj' by sme hľadali 'gr'), tak by algoritmus pokračoval ďalšími krokmi: opäť porovnáme stredný prvok s našim hľadaným ('gr' < 'hj') a keďže je v abecede pred ním, zmeníme interval tak, že `zac == kon == stred`:

0	1	2	3	4	5	6	7	8	9	10	11	12
ab	ad	dd	fm	hj	jj	ka	kz	mo	pa	rd	tz	zz
stred												

A to je teda už definitívny koniec behu algoritmu (interval je 1-prvkový): keďže tento jediný prvok je rôzny od hľadaného 'gr', je nám jasné, že sme skončili so správou „nenašli“.

Zapíšme tento algoritmus do Pythonu:

```
def hľadaj(hodnota):
    zac = 0                                # zaciatok intervalu
    kon = len(zoznam) - 1                  # koniec intervalu
    while zac <= kon:
        stred = (zac + kon) // 2           # stred intervalu
        if zoznam[stred] < hodnota:
            zac = stred + 1
        elif zoznam[stred] > hodnota:
            kon = stred - 1
        else:
            return True
    return False
```

Ak by sme teraz spustili rovnaké testy ako pred tým so sekvenčným vyhľadávaním, boli by sme miilo prekvapení: čas na hľadanie v 1000-prvkovom zozname a 1000000-prvkovom zozname je skoro stále rovnaký a sú to len tisíciny milisekúnd.

Štandardný typ dict

Typ `dict` **slovník** (informatici to volajú **asociatívne pole**) je taká dátová štruktúra, v ktorej k prvkom neprichádzame cez poradové číslo (index) tak ako pri zoznamoch, n-ticiach a reťazcoch, ale k prvkom prichádzame pomocou **klúča**. Hovoríme, že k danému klúču je **asociovaná** nejaká hodnota (niekedy hovoríme, že hodnotu **mapujeme** na daný klúč).

Zapisujeme to takto:

```
klúč : hodnota
```

Samotný slovník zapisujeme ako kolekciu takýchto dvojíc (`klúč : hodnota`) a celé je to uzavreté v '{' a '}' zátvorkách (rovnako ako množiny). Slovník si môžeme predstaviť ako zoznam dvojíc (klúč, hodnota), pričom v takomto zozname sa nemôžu nachádzať dve dvojice s rovnakým klúčom. Napríklad:

```
>>> vek = {'Jan':17, 'Maria':15, 'Ema':18}
```

Takto sme vytvorili slovník (asociatívne pole, teda pythonovský typ `dict`), ktorý má tri prvky: 'Jan' má 17 rokov, 'Maria' má 15 a 'Ema' má 18. V priamom režime vidíme, ako ho vypisuje Python a tiež to, že pre Python to má 3 prvky (3 dvojice):

```
>>> vek
{'Jan': 17, 'Maria': 15, 'Ema': 18}
>>> len(vek)
```

Teraz zápisom, ktorý sa podobá indexovaniu zoznamu:

```
>>> vek['Jan']
17
```

získame asociovanú hodnotu pre kľúč 'Jan'.

Ak sa pokúsime zistiť hodnotu pre neexistujúci kľúč:

```
>>> vek['Juraj']
...
KeyError: 'Juraj'
```

Python nám tu oznámi `KeyError`, čo znamená, že tento slovník nemá definovaný kľúč 'Juraj'.

Rovnakým spôsobom, ako priradíme hodnotu do zoznamu, môžeme vytvoriť novú hodnotu pre nový kľúč:

```
>>> vek['Hana'] = 15
>>> vek
{'Jan': 17, 'Hana': 15, 'Maria': 15, 'Ema': 18}
```

alebo môžeme zmeniť existujúcu hodnotu:

```
>>> vek['Maria'] = 16
>>> vek
{'Jan': 17, 'Hana': 15, 'Maria': 16, 'Ema': 18}
```

Všimnite si, že poradie dvojíc `kľúč:hodnota` v samotnom slovníku je v nejakom neznámom poradí. Dokonca, ak spustíme program viackrát, môžeme dostať rôzne poradia prvkov. Podobnú skúsenosť určite máte aj s typom `set` z predchádzajúcej prednášky.

Pripomeňme si, ako funguje operácia `in` s typom zoznam:

```
>>> zoznam = [17, 15, 16, 18]
>>> 15 in zoznam
True
```

Operácia `in` s takýmto zoznamom prehľadáva hodnoty a ak takú nájde, vráti `True`.

So slovníkom to funguje trochu inak: operácia `in` neprehľadáva hodnoty ale kľúče:

```
>>> 17 in vek
False
>>> 'Hana' in vek
True
```

Vďaka tomuto vieme zabrániť, aby program spadol pre neznámy kľúč:

```
>>> if 'Juraj' in vek:
    print('Juraj ma', vek['Juraj'], 'rokov')
else:
    print('nepoznám Jurajov vek')
```

Keďže operácia `in` so slovníkom prehľadáva kľúče, tak aj for-cyklus bude fungovať na rovnakom princípe:

```
>>> for i in zoznam:
    print(i)
17
15
16
18
>>> for i in vek:
    print(i)
Jan
Hana
Maria
Ema
```

Z tohto istého dôvodu funkcia `list()` s parametrom slovník vytvorí zoznam kľúčov a nie zoznam hodnôt:

```
>>> list(vek)
['Jan', 'Hana', 'Maria', 'Ema']
```

Keď chceme zo slovníka vypísať všetky kľúče aj s ich hodnotami, zapíšeme:

```
>>> for kluc in vek:
    print(kluc, vek[kluc])
Jan 17
Hana 15
Maria 16
Ema 18
```

Zrejme slovník je rovnako ako zoznam **meniteľný** typ (**mutable**), keďže môžeme do neho pridávať nové prvky, resp. meniť hodnoty existujúcich prvkov.

Napríklad funguje takýto zápis:

```
>>> vek
{'Jan': 17, 'Hana': 15, 'Maria': 16, 'Ema': 18}
>>> vek['Jan'] = vek['Jan'] + 1
>>> vek
{'Jan': 18, 'Hana': 15, 'Maria': 16, 'Ema': 18}
```

a môžeme to využiť aj v takejto funkcii:

```
def o_1_starsi(vek):
    for kluc in vek:
        vek[kluc] = vek[kluc] + 1
```

Funkcia zvýši hodnotu v každej dvojici slovníka o 1:

```
>>> o_1_starsi(vek)
>>> vek
{'Jan': 19, 'Hana': 16, 'Maria': 17, 'Ema': 19}
```

Pythonovské funkcie teda môžu meniť (ako vedľajší účinok) nielen zoznam ale aj slovník.

Ak by sme chceli prejsť kľúče v utriedenom poradí, musíme zoznam kľúčov najprv utriediť:

```
>>> for kluc in sorted(vek):
    print(kluc, vek[kluc])
Ema 19
Hana 16
Jan 19
```


Slovníky majú definovaných viac zaujímavých metód, my si najprv ukážeme len 3 z nich. Táto skupina troch metód vráti nejaké špeciálne „postupnosti“:

- `keys()` - postupnosť kľúčov
- `values()` - postupnosť hodnôt
- `items()` - postupnosť dvojíc kľúč a hodnota

Napríklad:

```
>>> vek.keys()
dict_keys(['Jan', 'Hana', 'Maria', 'Ema'])
>>> vek.values()
dict_values([19, 16, 17, 19])
>>> vek.items()
dict_items([('Jan', 19), ('Hana', 16), ('Maria', 17), ('Ema', 19)])
```

Tieto postupnosti môžeme použiť napríklad vo for-cykle alebo môžu byť parametrami rôznych funkcií, napríklad `list()`, `max()` alebo `sorted()`:

```
>>> list(vek.values())
[19, 16, 17, 19]
>>> list(vek.items())
[('Jan', 19), ('Hana', 16), ('Maria', 17), ('Ema', 19)]
```

Metódu `items()` najčastejšie využijeme vo for-cykle:

```
>>> for prvok in vek.items():
    kluc, hodnota = prvok
    print(kluc, hodnota)
Jan 19
Hana 16
Maria 17
Ema 19
```

Alebo krajšie dvojicou premenných for-cyklu:

```
>>> for kluc, hodnota in vek.items():
    print(kluc, hodnota)
Jan 19
Hana 16
Maria 17
Ema 19
```

Jednou z najpoužívanějších metód okrem `items()` je `get()`.

metóda `get()`

Táto metóda vráti asociovanú hodnotu k danému kľúču, ale v prípade, že daný kľúč neexistuje, nespadne na chybu, ale vráti nejakú náhradnú hodnotu. Metódu môžeme volať s jedným alebo aj dvoma parametrami:

```
slovník.get(kluc)
slovník.get(kluc, nahrada)
```

V prvom prípade, ak daný kľúč neexistuje, funkcia vráti `None`, ak v druhom prípade neexistuje kľúč, tak funkcia vráti hodnotu `nahrada`.

Napríklad:

```
>>> print(vek.get('Maria'))
17
>>> print(vek.get('Mario'))
None
>>> print(vek.get('Maria', 20))
17
>>> print(vek.get('Mario', 20))
20
```

Príkaz `del` funguje nielen so zoznamom, ale aj so slovníkom:

```
>>> zoznam = [17, 15, 16, 18]
>>> del zoznam[1]
>>> zoznam
[17, 16, 18]
```

príkaz `del` so slovníkom

Príkaz `del` vyhodí zo slovníka príslušný kľúč aj s jeho hodnotou:

```
del slovník[kluc]
```

Ak daný kľúč v slovníku neexistuje, príkaz vyhlási výnimku `KeyError`

Napríklad:

```
>>> vek
{'Jan': 19, 'Hana': 16, 'Maria': 17, 'Ema': 19}
>>> del vek['Jan']
>>> vek
{'Hana': 16, 'Maria': 17, 'Ema': 19}
>>> del vek['Juraj']
...
KeyError: 'Juraj'
```

Zhrňme všetko, čo sme sa doteraz naučili pre dátovú štruktúru slovník:

- `slovník = {}`
 - prázdny slovník
- `slovník = {kľúč:hodnota, ...}`
 - priame priradenie celého slovníka
- `kľúč in slovník`
 - zistí, či v slovníku existuje daný kľúč (vráti `True` alebo `False`)
- `len(slovník)`
 - zistí počet prvkov (dvojíc `kľúč:hodnota`) v slovníku
- `slovník[kľúč]`
 - vráti príslušnú hodnotu alebo príkaz spadne na chybu `KeyError`, ak neexistuje
- `slovník[kľúč] = hodnota`
 - vytvorí novú asociáciu `kľúč:hodnota` alebo zmení existujúcu
- `del slovník[kľúč]`
 - zruší dvojicu `kľúč:hodnota` alebo príkaz spadne na chybu `KeyError`, ak neexistuje
- `for kľúč in slovník: ...`
 - prechádzanie všetkých kľúčov
- `for kľúč in sorted(slovník): ...`
 - prechádzanie všetkých kľúčov slovníka v utriedenom poradí
- `for kľúč in slovník.values(): ...`
 - prechádzanie všetkých hodnôt

- `for` kľúč, hodnota `in` `slovník.items(): ...`
 - prechádzanie všetkých dvojíc `kľúč:hodnota`
- `slovník.get(kľúč)`
 - vráti príslušnú hodnotu alebo `None`, ak kľúč neexistuje
- `slovník.get(kľúč, náhradná)`
 - vráti príslušnú hodnotu alebo vráti hodnotu parametra `náhradná`, ak kľúč neexistuje

Predstavte si teraz, že máme daný nejaký zoznam dvojíc a chceme z neho urobiť slovník. Môžeme to urobiť, napríklad `for`-cyklom:

```
>>> zoznam_dvojic = [('one', 1), ('two', 2), ('three', 3)]
>>> slovník = {}
>>> for kluc, hodnota in zoznam_dvojic:
>>>     slovník[kluc] = hodnota
>>> slovník
{'one': 1, 'two': 2, 'three': 3}
```

alebo priamo použitím štandardnej funkcie `dict()`, ktorá takto funguje ako konverzná funkcia:

```
>>> slovník = dict(zoznam_dvojic)
>>> slovník
{'one': 1, 'two': 2, 'three': 3}
```

Takýto zoznam dvojíc vieme vytvoriť aj z nejakého existujúceho slovníka pomocou metódy `items()`:

```
>>> list(slovník.items())
[('one', 1), ('two', 2), ('three', 3)]
```

Funkciu `dict()` môžeme zavolať aj takto:

```
>>> slovník = dict(one=1, two=2, three=3)
```

Vďaka tomuto zápisu nemusíme kľúče uzatvárať do apostrofov. Toto ale funguje len pre kľúče, ktoré sú znakové reťazce a majú správny formát pre identifikátory pomenovaných parametrov.

Asociatívne pole ako slovník (dictionary)

Anglický názov tohto typu `dict` je zo slova **dictionary**, teda slovník. Naozaj je „obyčajný“ slovník príkladom pekného využitia tohto typu. Napríklad:

```
slovník = {'cat':'macka', 'dog':'pes', 'my':'moj', 'good':'dobry', 'is':'je'}

def preklad(veta):
    vysl = []
    for slovo in veta.lower().split():
        vysl.append(slovník.get(slovo, '<' + slovo + '>'))
    return ' '.join(vysl)

>>> preklad('my dog is very good')
'moj pes je <very> dobry'
```

Zrejme, keby sme mali kompletnejší slovník, napríklad anglicko-slovenský s tisícami dvojíc slov, vedeli by sme veľmi jednoducho realizovať takýto „kuchársky“ preklad. V skutočnosti by sme asi mali mať slovník, v ktorom jednému anglickému slovu zodpovedá niekoľko (napríklad *n*-tíca) slovenských slov.

Slovník ako frekvenčná tabuľka

Frekvenčnou tabuľkou nazývame takú tabuľku, ktorá obsahuje informácie o počte výskytov rôznych hodnôt v nejakej postupnosti (napríklad súbor, reťazec, zoznam, ...).

Ukážeme to na príklade, v ktorom budeme zisťovať počty výskytov písmen v nejakom texte. Vytvoríme slovník, v ktorom každému prvku vstupného zoznamu (kľúču) bude zodpovedať jedno celé číslo - počítadlo výskytov tohto prvku:

```
def pocy_vyskytov(postupnost):
    vysl = {}
    for prvok in postupnost:
        vysl[prvok] = vysl.get(prvok, 0) + 1
    return vysl

pocet = pocy_vyskytov('anicka dusicka nekasli, aby ma pri tebe nenasli.')
for kluc, hodnota in pocet.items():
    print(repr(kluc), hodnota)
```

Všimnite si použitie metódy `get()`, vďaka ktorej nepotrebujeme pomocou podmieneného príkazu `if` zisťovať, či sa v slovníku príslušný kľúč už nachádza. Zápis `vysl.get(prvok, 0)` pre spracovávaný `prvok` vráti momentálny počet jeho výskytov, alebo `0`, ak sa tento `prvok` vyskytol prvýkrát.

```
'a' 7
'n' 4
'i' 5
'c' 2
'k' 3
' ' 7
'd' 1
'u' 1
's' 3
'e' 4
'l' 2
',' 1
'b' 2
'y' 1
'm' 1
'p' 1
'r' 1
't' 1
'.' 1
```

Tento výpis ukazuje, že písmeno `'a'` sa v danej vete vyskytlo 7 krát a písmeno `'d'` len raz.

Toto isté vieme zapísať aj pomocou spracovania výnimky:

```
def pocy_vyskytov(postupnost):
    vysl = {}
    for prvok in postupnost:
        try:
            vysl[prvok] += 1
        except KeyError:
            vysl[prvok] = 1
    return vysl
```

Podobne môžeme spracovať aj nejaký celý textový súbor ([dobs.txt](#) alebo [twain.txt](#)):

```
with open('dobs.txt') as subor:
```

```

    pocet = pocy_vyskytov(subor.read())
for kluc, hodnota in pocet.items():
    print(repr(kluc), hodnota)
'p' 2618
'a' 12564
'v' 3939
'o' 8740
'l' 5068
' ' 21105
'd' 4252
'b' 1838
's' 5349
'i' 6227
'n' 5006
'k' 3482
'y' 2176
'\n' 722
'r' 3983
't' 5624
'e' 8522
'u' 3220
'm' 3243
'h' 2341
'c' 2804
'j' 2083
'z' 3032
'g' 90
'f' 27
'x' 2

```

Vidíme, že pri väčších súboroch sú aj zistené počty výskytov dosť vysoké. Napríklad všetkých spracovaných znakov bolo:

```

>>> sum(pocet.values())
118057

```

Ak by sme potrebovali zistiť ovať nie počty písmen, ale počty slov, stačí zapísať:

```

with open('dobs.txt') as subor:
    pocet = pocy_vyskytov(subor.read().split())

```

Táto konkrétna štruktúra slovníka je už dosť veľká a nemá zmysel ju vypisovať celú, veď všetkých rôznych slov a teda veľkosť slovníka je:

```

>>> len(pocet)                # počet kľúčov v slovníku
5472
>>> sum(pocet.values())       # počet všetkých slov v súbore
21827

```

Ukážeme, ako môžeme zistiť 20 najčastejších slov vo veľkom texte. Najprv z frekvenčnej tabuľky `pocet` vytvoríme zoznam dvojíc (`hodnota`, `klúč`) (prehodíme poradie v dvojiciach (`klúč`, `hodnota`)). Potom tento zoznam utriedime. Robíme to preto, lebo Python triedi n-tice tak, že najprv porovnáva prvé prvky (teda počty výskytov) a potom až pri zhode porovná druhé prvky (teda samotné slová). Všimnite si, že sme tu použili metódu `sort()` pre zoznamy a pridali sme jeden pomenovaný parameter `reverse=True`, aby sa zoznam utriedil zostupne, teda od najväčších po najmenšie:

```

zoznam = []
for kluc, hodnota in pocet.items():
    zoznam.append((hodnota, kluc))

```

```
zoznam.sort(reverse=True)

print(zoznam[:20])
```

Pre súbor 'dobs.txt' dostávame:

```
[(1032, 'a'), (703, 'sa'), (436, 'na'), (238, 'ale'), (227, 'to'), (217, 'tu'),
 (213, 'ze'), (212, 'do'), (203, 'len'), (200, 'v'), (188, 'ako'), (186, 'z'),
 (184, 'si'), (166, 'tak'), (147, 'co'), (145, 'i'), (134, 'uz'), (132, 'za'),
 (132, 'mu'), (116, 's')]
```

Ešte si uvedomte, že kľúčmi nemusia byť len slová, resp. písmená. Kľúčmi v slovníku môžu byť ľubovoľné nemeniteľné (**immutable**) typy, teda okrem `str` aj `int`, `float` a `tuple`. Teda by sme zvládli zisťovať aj počet výskytov prvkov číselných zoznamov, alebo hoci dvojíc čísel (súradníc bodov v rovine) a pod.

Zoznam slovníkov

Slovník môže byť aj hodnotou v inom slovníku. Zapišme:

```
student1 = {
    'meno': 'Janko Hrasko',
    'adresa': {'ulica': 'Strukova',
               'cislo': 13,
               'obec': 'Fazulovo'},
    'narodeny': {'datum': {'den': 1, 'mesiac': 5, 'rok': 1999},
                  'obec': 'Korytovce'}
}
student2 = {
    'meno': 'Juraj Janosik',
    'adresa': {'ulica': 'Pod sibenickou',
               'cislo': 1,
               'obec': 'Liptovsky Mikulas'},
    'narodeny': {'datum': {'den': 25, 'mesiac': 1, 'rok': 1688},
                  'obec': 'Terchova'}
}
student3 = {
    'meno': 'Margita Figuli',
    'adresa': {'ulica': 'Sturova',
               'cislo': 4,
               'obec': 'Bratislava'},
    'narodeny': {'datum': {'den': 2, 'mesiac': 10, 'rok': 1909},
                  'obec': 'Vysny Kubin'}
}
student4 = {
    'meno': 'Ludovit Stur',
    'adresa': {'ulica': 'Slovenska',
               'cislo': 12,
               'obec': 'Modra'},
    'narodeny': {'datum': {'den': 28, 'mesiac': 10, 'rok': 1815},
                  'obec': 'Uhrovec'}
}

skola = [student1, student2, student3, student4]
```

Vytvorili sme 4-prvkový zoznam, v ktorom je každý prvok typu slovník. V týchto slovníkoch sú po 3 kľúče 'meno', 'adresa', 'narodeny', pričom dva z nich majú hodnoty opäť slovníky. Môžeme zapísať, napríklad:

```
>>> for st in skola:
    print(st['meno'], 'narodeny v', st['narodeny']['obec'])
Janko Hrasko narodeny v Korytovce
Juraj Janosik narodeny v Terchova
Margita Figuli narodeny v Vysny Kubin
Ludovit Stur narodeny v Uhrovec
```

Získali sme nielen mená všetkých študentov v tomto zozname ale aj ich miesto narodenia.

Ak by sme túto štruktúru vypísali pomocou `print`, dostávame takýto nečitateľný výpis:

```
>>> skola
[{'meno': 'Janko Hrasko', 'adresa': {'ulica': 'Strukova', 'cislo': 13,
'obec': 'Fazulovo'}, 'narodeny': {'datum': {'den': 1, 'mesiac': 5,
'rok': 1999}, 'obec': 'Korytovce'}}, {'meno': 'Juraj Janosik', 'adresa':
{'ulica': 'Pod sibenickou', 'cislo': 1, 'obec': 'Liptovsky Mikulas'},
'narodeny': {'datum': {'den': 25, 'mesiac': 1, 'rok': 1688}, 'obec':
'Terchova'}}, {'meno': 'Margita Figuli', 'adresa': {'ulica': 'Sturova',
'cislo': 4, 'obec': 'Bratislava'}, 'narodeny': {'datum': {'den': 2,
'mesiac': 10, 'rok': 1909}, 'obec': 'Vysny Kubin'}}, {'meno': 'Ludovit
Stur', 'adresa': {'ulica': 'Slovenska', 'cislo': 12, 'obec': 'Modra'},
'narodeny': {'datum': {'den': 28, 'mesiac': 10, 'rok': 1815}, 'obec':
'Uhrovec'}}]
```

Textové súbory JSON

Ak pythonovská štruktúra obsahuje iba:

- zoznamy `list`
- slovníky `dict` s kľúčmi, ktoré sú reťazce
- znakové reťazce `str`
- celé alebo desatinné čísla `int` a `float`
- logické hodnoty `True` alebo `False`

môžeme túto štruktúru (napríklad zoznam `skola`) zapísať do špeciálneho súboru:

```
import json

with open('subor.txt', 'w') as subor:
    json.dump(skola, subor)
```

Takto vytvorený súbor je dosť nečitateľný, čo môžeme zmeniť parametrom `indent`:

```
with open('subor.txt', 'w') as subor:
    json.dump(skola, subor, indent=2)
```

Prečítať takýto súbor a zrekonštruovať z neho celú štruktúru je potom veľmi jednoduché:

```
>>> precitane = json.load(open('subor.txt'))
>>> print(skola == precitane)
True
```

Vytvorila sa nová štruktúra `precitane`, ktorá má presne rovnaký obsah, ako do súboru zapísaný zoznam slovníkov `skola`.

Cvičenia

L.I.S.T.

- riešenia **všetkých úloh** odovzdaj na úlohový server <https://list.fmph.uniba.sk/>
- pozri si **Riešenie úloh 19. cvičenia**

1. Získali sme zoznam zvierat nejakej zoo spolu aj s ich hmotnosťami:

```
2. lev 190
3. tiger 300
4. vydra 15
5. hyena 50
6. gorila 160
7. jazvec 20
8. rys 30
9. zirafa 800
10. simpanz 50
11. diviak 180
12. hroch 1500
13. tchor 1
14. medved 300
15. los 500
16. orangutan 75
17. puma 100
18. kamzik 40
19. vlk 80
20. buvol 590
21. antilopa 200
22. liska 12
23. pyton 90
24. slon 5000
25. gibbon 10
26. daniel 90
27. jaguar 120
28. zubor 1000
29. pavian 30
30. jelen 280
31. kengura 80
```

Vytvor slovník `zoo`, v ktorom kľúčmi budú zvieratá a príslušnými hodnotami budú ich hmotnosti. Ďalej vypíš tento slovník tak, aby boli zvieratá usporiadané podľa abecedy. Tento výpis bude obsahovať aj hmotnosti zvierat. Na záver vypíš priemernú hmotnosť všetkých zvierat v zoo.

2. Pre slovník `zoo` z predchádzajúcej úlohy napíš tieto dve funkcie:

- funkcia `najtazsie()` vráti dvojicu: meno najťažšieho zvierat'a a jeho hmotnosť
- funkcia `tazsie_ako(zviera)` vypíše všetky ťažšie ako dané zviera
- funkcia `vyber_lahsie(zviera)` vráti nový slovník, v ktorom budú len zvieratá ľahšie ako zadané zviera

Napríklad:

```
>>> najtazsie()
('slon', 5000)
>>> tazsie_ako('zirafa')
hroch
slon
zubor
>>> lahsie = vyber_lahsie('jazvec')
```



```
>>> lahsie
{'vydra': 15, 'tchor': 1, 'liska': 12, 'gibon': 10}
```

3. V súbore [skladatelja.txt](#) máme zoznam hudobných skladateľov vážnej hudby aj s rokmi ich narodenia. Vytvor z tohto súboru slovník `skladatel`, v ktorom kľúčmi sú mená a hodnotami sú celé čísla rokov. Ďalej
- o vypíš meno najstaršieho a najmladšieho skladateľa aj s rokom ich narodenia,
 - o napíš funkciu `medzi(rok1, rok2)` vráti zoznam (typ `list`) skladateľov, ktorí sa narodili v danom intervale rokov `<rok1, rok2>`.

Napríklad:

```
najstarsi: xyz 1111
najmladsi: xyz 1999
>>> zoz = medzi(1720, 1730)
>>> zoz
[]
>>> zoz = medzi(1820, 1830)
>>> zoz
['Bruckner', 'Franck', 'Strauss', 'Smetana']
```

4. Zo slovníka `skladatel` z predchádzajúcej úlohy vyrob nový slovník, v ktorom kľúčmi budú roky narodenia a príslušnými hodnotami budú množiny (typ `set`) skladateľov, ktorí sa v danom roku narodili. Ďalej
- o napíš funkciu `otoc(slovník)`, ktorá otočí asociácie ľubovoľného slovníka, teda vyrobí nový slovník, v ktorom kľúčmi budú hodnoty z pôvodného slovníka (napríklad roky narodenia) a hodnotami pre tieto nové kľúče budú množiny **pôvodných kľúčov** (množiny skladateľov),
 - o prirad' `rok = otoc(skladatel)` a vypíš tento slovník utriedený podľa rokov (do každého riadka rok a príslušnú množinu)
 - o ďalej vypíš len tie roky, v ktorých sa narodil viac ako jeden skladateľ
5. V textovom súbore (napríklad [twain.txt](#)) sú slová rôznej dĺžky. Zostav slovník dĺžok slov: pre každú dĺžku sa v slovníku zapamätá **množina** slov danej dĺžky, teda `slova[1]` je množina jednopísmenových slov, `slova[2]` je množina dvojpísmenových slov, ... Teraz už budú jednoduché tieto úlohy:
- o vypíš, aká dĺžka slov je najčastejšia (má najdlhší asociovaný zoznam) a koľko rôznych je takýchto slov
 - o vypíš všetky najdlhšie slová

Napríklad:

```
najcastejsia dlzka je 5
slov dlzky 5 je 876
zoznam slov: {'stuff', 'death', ...}
najdlhsie slova: {...}
```

6. Na prednáške sa zisťovala frekvenčná tabuľka písmen, resp. slov vo vete. Napíš funkciu `dvojice(meno_saboru)`, ktorá z daného súboru slov zistí počet výskytov všetkých za sebou idúcich **dvojíc písmen**, napríklad pre slovo `'laska'` bude akceptovať tieto štyri dvojice `'la'`, `'as'`, `'sk'` a `'ka'`. Funkcia vráti frekvenčnú tabuľku ako slovník. Otestuj na súbore [twain.txt](#) a zisti 10 najčastejších sa vyskytujúcich dvojíc písmen.

7. Napíš funkciu `dve_kocky(n)`, ktorá bude simulovať hádzanie dvomi hracími kockami (s číslami od 1 do 6) a evidovať si, koľkokrát padol aký súčet. Zrejme súčty budú čísla od 2 do 12. Funkcia bude simulovať `n` takýchto hodov dvomi kockami a vráti frekvenčnú tabuľku (teda slovník). Funkcia nič nevypisuje. Napríklad:

```
8. >>> f = dve_kocky(10)
9. >>> f
10. {8: 3, 9: 3, 6: 1, 12: 1, 7: 2}
11. >>> dve_kocky(100)
12. {9: 10, 6: 16, 8: 15, 4: 13, 5: 9, 7: 18, 11: 4, 2: 3, 12: 4, 3: 4, 10: 4}
```

8. Napíš funkciu `fib(n)`, ktorá zostaví a vráti slovník s prvými `n+1` fibonacciho číslami, teda kľúčmi sú čísla od 0 do `n` a hodnotami sú príslušné fibonacciho čísla. Využi vzorec $f[i] = f[i-1] + f[i-2]$. Napríklad:

```
9. >>> f = fib(6)
10. >>> f
11. {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
```

9. Daný je slovník `sifra[písmeno] = písmeno`, v ktorom každému písmenu (od 'a' po 'z') zodpovedá nejaké iné písmeno (zrejme jednoznačne). Napíš funkciu `zasifruj(sifra, text)`, ktorá pomocou šifry v prvom parametri zašifruje zadaný text. Nepísmenové znaky v tomto text nemení. Šifrovanie textu znamená, že každé písmeno v texte sa nahradí písmenom zo slovníka `sifra`. Okrem funkcie `zasifruj` napíš aj opačnú funkciu `rozsifruj(sifra, text)`, ktorá dostáva šifrovací slovník `sifra` a zašifrovaný text, napríklad z funkcie `zasifruj`. Funkcia tento text rozšifruje. Môžeš pritom využiť ideu funkcie `otoc` z úlohy (4). Napríklad:

```
10. >>> sifra = {'a': 'c', 'b': 'd', 'c': 'e', ... }
11. >>> t = zasifruj(sifra, 'programovanie v pythone je cool')
12. >>> t
13. 'rtqitcoqxcpkg x ravjqpg lg eqqn'
14. >>> tt = rozsifruj(sifra, t)
15. >>> tt
16. 'programovanie v pythone je cool'
```

10. Napíš funkciu `pretypuj(nazov, hodnota)`, ktorá na základe názvu typu (reťazec) pretypuje danú hodnotu. Vo funkcii nepouži príkaz `if`, ale zadefinuj slovník `typ`, ktorý bude mať pre každý názov typu (kľúč) asociovanú hodnotu samotný typ. Napríklad `typ['int'] = int`. Ak sa dané pretypovanie urobiť nedá, funkcia by mala spadnúť na zodpovedajúcej chybe. Funkcia by mala akceptovať tieto názvy typov: 'int', 'float', 'list', 'tuple', 'str', 'set', 'dict'. Pre neznámy názov typu by funkcia mala spadnúť na chybu `KeyError`, inak funkcia vráti pretypovanú hodnotu v zadanom novom type. Napríklad:

```
11. >>> pretypuj('str', 3.14)
12. '3.14'
13. >>> pretypuj('set', 'Python')
14. {'t', 'y', 'P', 'h', 'o', 'n'}
15. >>> pretypuj('float', '1e5')
16. 100000.0
17. >>> pretypuj('dict', [(1, 'a'), (2, 'b')])
18. {1: 'a', 2: 'b'}
19. >>> pretypuj('novy', 123)
20. ...
21. KeyError: 'novy'
```

11. Pomocou funkcie `fib(40)` z (8) úlohy vygeneruj slovník `f`. Ulož tento slovník do súboru pomocou `json.dump(..., indent=2)` a skontroluj obsah tohto súboru. Teraz pomocou `json.load()` prečítaj jeho obsah do premennej `ff`. Mal by si dostať rovnakú štruktúru, ako si do súboru uložil. Lenže `f != ff`. Zisti prečo.
12. Napíš funkciu `rozhadz(post)`, ktorá vráti pomiešaný zoznam prvkov vstupnej postupnosti. Funkcia by mala pracovať takto:
- o vstupnú postupnosť prerobí na zoznam
 - o v cykle vyberie z tohto zoznamu náhodný prvok (`pop(index)`) a zaradí ho na koniec výsledného zoznamu (`append()`)
 - o toto opakuje, kým nebude tento zoznam prázdny
- Otestuj, napríklad:

```
>>> rozhadz('abcdefghijkl')
['d', 'k', 'f', 'l', 'c', 'h', 'u', 'g', 'j', 'a', 'e', 'b']
>>> rozhadz(range(10, 30))
[26, 29, 11, 23, 10, 12, 13, 21, 20, 27, 17, 16, 19, 22, 24, 15, 28, 18, 14, 25]
```

Teraz napíš testovaciu funkciu `test(n)`, ktorá odmeria čas v sekundách trvania tejto funkcie `rozhadz(range(n))`. Funkcia vráti tento čas zaokrúhlený na 3 desatinné miesta, napríklad:

```
>>> test(1000)
0.002
>>> test(10000)
0.027
>>> test(100000)
1.122
```

Zrejme na tvojom počítači dostaneš iné časy. Z modulu `random` použi len funkciu `randrange`.