6. Znakové reťazce

video prezentácia

funkcie

Typ string

Čo už vieme o znakových reťazcoch:

- reťazec je postupnosť znakov uzavretá v apostrofoch `''` alebo v úvodzovkách `""`
- vieme priradiť reťazec do premennej
- zreťaziť (zlepiť) dva reťazce
- násobiť (zlepiť viac kópií) reťazca
- načítať zo vstupu (pomocou `input()`) a vypisovať (pomocou `print()`)
- vyrobiť z čísla reťazec (`str()`), z reťazca číslo (`int()`, `float()`)
- rozobrať reťazec vo `for`-cykle

Postupne prejdeme tieto možnosti práce s reťazcami a doplníme ich o niektoré novinky.

Keďže znakový reťazec je postupnosť znakov uzavretá v apostrofoch `''` alebo v úvodzovkách `""`, platí:

- môže obsahovať ľubovoľné znaky (okrem znaku apostrof `'` v `''` reťazci, a znaku úvodzovka `"` v úvodzovkovom `""` reťazci)
- musí sa zmestiť do jedného riadka (nesmie prechádzať do druhého riadka)
- môže obsahovať špeciálne znaky (zapisujú sa dvomi znakmi, ale pritom v reťazci reprezentujú len jeden), vždy začínajú znakom `\` (opačná lomka):
  - `\n` - nový riadok
  - `\t` - tabulátor
  - `\'` - apostrof
  - `\"` - úvodzovka
  - `\\` - opačná lomka

Napríklad

```
>>> 'Monty\nPython'
    'Monty\nPython'
>>> print('Monty\nPython')
    Monty
    Python
>>> print('Monty\\nPython')
    Monty\nPython
```

Viacriadkové reťazce

platí:

- reťazec, ktorý začína trojicou buď apostrofov `'''` alebo úvodzoviek `"""` môže obsahovať aj `'` a `"`, môže prechádzať cez viac riadkov (automaticky sa sem doplní `\n`)
- musí byť ukončený rovnakou trojicou `'''` alebo `"""`

```
>>> macek = '''Išiel Macek
    do Malacek
    šošovičku mlácic'''
>>> macek
    'Išiel Macek\ndo Malacek\nšošovičku mlácic'
>>> print(macek)
    Išiel Macek
    do Malacek
    šošovičku mlácic
>>> '''tento retazec obsahuje " aj ' a funguje'''
    'tento retazec obsahuje " aj \' a funguje'
>>> print('''tento retazec obsahuje " aj ' a funguje''')
    tento retazec obsahuje " aj ' a funguje
```

Dĺžka reťazca

Štandardná funkcia `len()` vráti dĺžku reťazca (špeciálne znaky ako `'\n'`, `'\''`, a pod. reprezentujú len 1 znak):

```
>>> a = 'Python'
>>> len(a)
    6
>>> len('Peter\'s dog')
    11
>>> len('\\\\\\')
    3
```

Túto funkciu už vieme naprogramovať aj sami, ale v porovnaní so štandardnou funkciou `len()` bude oveľa pomalšia:

```
def dlzka(retazec):
    pocet = 0
    for znak in retazec:
        pocet += 1
    return pocet

>>> dlzka('Python')
    6
>>> a = 'x' * 100000000
>>> dlzka(x)
    100000000
>>> len(x)
    100000000
```

Operácia in

Aby sme zistili, či sa v reťazci nachádza nejaký konkrétny znak, doteraz sme to museli riešiť takto:

```
def zisti(znak, retazec):
    for z in retazec:
        if z == znak:
            return True
    return False

>>> zisti('y', 'Python')
    True
>>> zisti('T', 'Python')
    False
```

Pritom existuje binárna operácia `in`, ktorá zisťuje, či sa zadaný podreťazec nachádza v nejakom konkrétnom reťazci. Jej tvar je

```
podretazec in retazec
```

Najčastejšie sa bude využívať v príkaze `if` a v cykle `while`, napríklad:

```
>>> 'nt' in 'Monty Python'
    True
>>> 'y P' in 'Monty Python'
    True
>>> 'tyPy' in 'Monty Python'
    False
>>> 'pyt' in 'Monty Python'
    False
```

Na rozdiel od našej vlastnej funkcie `zisti()`, operácia `in` funguje nielen pre zisťovanie jedného znaku, ale aj pre ľubovoľne dlhý podreťazec.

Ak niekedy budeme potrebovať negáciu tejto podmienky, môžeme zapísať:

```
if not 'a' in retazec:
    ...
if 'a' not in retazec:
    ...
```

Pričom sa odporúča druhý spôsob zápisu `not in`.

Operácia indexovania [ ]

Pomocou tejto operácie vieme pristupovať k jednotlivým znakom postupnosti (znakový reťazec je postupnosť znakov). Jej tvar je

```
reťazec[číslo]
```

Celému číslu v hranatých zátvorkách hovoríme **index**:

- znaky v reťazci sú indexované od `0` do `len()-1`, t.j. prvý znak v reťazci má index 0, druhý 1, … posledný má index `len()-1`
- výsledkom indexovania je vždy 1-znakový reťazec (čo je nový reťazec s kópiou 1 znaku z pôvodného reťazca) alebo chybová správa, keď indexujeme mimo znaky reťazca

Očíslujme znaky reťazca:

indexy v reťazci

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

Napríklad do premennej `abc` priradíme reťazec 12 znakov a pristupujeme ku niektorým znakom pomocou indexu:

```
>>> abc = 'Monty Python'
>>> abc[3]
   't'
>>> abc[9]
   'h'
>>> abc[12]
    ...
    IndexError: string index out of range
>>> abc[len(abc)-1]
    'n'
```

Vidíme, že posledný znak v reťazci má index **dĺžka reťazca-1**. Ak indexujeme väčším číslom ako 11, vyvolá sa chybová správa **IndexError: string index out of range**.

Často sa indexuje v cykle, kde premenná cyklu nadobúda správne hodnoty indexov, napríklad:

```
a = 'Python'
for i in range(len(a)):
    print(i, a[i])
```

```
0 P
1 y
2 t
3 h
4 o
5 n
```

Funkcia `range(len(a))` zabezpečí, že cyklus prejde postupne pre všetky `i` od `0` do `len(a)-1`.

Indexovanie so zápornými indexmi

Keďže často potrebujeme pristupovať ku znakom na konci reťazca, môžeme to zapisovať pomocou záporných indexov:

```
abc[-5] == abc[len(abc)-5]
```

Znaky reťazca sú indexované od `-1` do `-len()` takto:

záporné indexy

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Napríklad:

```
>>> abc = 'Monty Python'
>>> abc[len(abc)-1]
    'n'
>>> abc[-1]
    'n'
>>> abc[-7]
    ' '
>>> abc[-13]
    ...
    IndexError: string index out of range
```

alebo aj for-cyklom:

```
a = 'Python'
for i in range(1, len(a)+1):
    print(-i, a[-i])
```

```
-1 n
-2 o
-3 h
-4 t
-5 y
-6 P
```

alebo for-cyklom so záporným krokom:

```
a = 'Python'
for i in range(-1, -len(a)-1, -1):
    print(i, a[i])
```

```
-1 n
-2 o
-3 h
-4 t
-5 y
-6 P
```

Podreťazce

Indexovať môžeme nielen jeden znak, ale aj nejaký podreťazec celého reťazca. Opäť použijeme operátor indexovania, ale index bude obsahovať znak `':'`:

```
reťazec[prvý : zaposledný]
```

kde

- `prvý` je index začiatku podreťazca
- `zaposledný` je index prvku **jeden za**, t.j. musíme písať index prvku o 1 viac
- takejto operácii hovoríme **rez** (alebo po anglicky **slice**)
- ak takto indexujeme mimo reťazec, **nenastane** chyba, ale prvky mimo sú prázdny reťazec

Ak indexujeme rez od 6. po 11. prvok:

podreťazec

M	o	n	t	y		P	y	t	h	o	n
						^					^
0	1	2	3	4	5	6	7	8	9	10	11

prvok s indexom 11 už vo výsledku nebude:

```
>>> abc = 'Monty Python'
>>> abc[6:11]
    'Pytho'
>>> abc[6:12]
    'Python'
>>> abc[6:len(abc)]
    'Python'
>>> abc[6:12]
    'Python'
>>> abc[10:16]
    'on'
```

Podreťazce môžeme vytvárať aj v cykle:

```
a = 'Python'
for i in range(len(a)):
    print(f'{i}:{i+3} {a[i:i+3]}')
```

```
0:3 Pyt
1:4 yth
2:5 tho
3:6 hon
4:7 on
5:8 n
```

alebo

```
a = 'Python'
for i in range(len(a)):
    print(f'{i}:{len(a)} {a[i:len(a)]}')
```

```
0:6 Python
1:6 ython
2:6 thon
3:6 hon
4:6 on
```

```
5:6 n
```

Ešte otestujte aj takéto zápisy:

```
>>> 'Python'[1:-1]
    'ytho'
>>> 'Python'[-5:4]
    'yth'
>>> 'Python'[-3:3]
    ''
>>> 'Python'[1:-1][1:-1]
    'th'
```

Dobre sa zamyslite nad každým z týchto reťazcových výrazov.

Predvolená hodnota

Ak neuvedieme prvý index v podreťazci, bude to označovať rez **od začiatku reťazca**. Zápis je takýto:

```
reťazec[ : zaposledný]
```

Ak neuvedieme druhý index v podreťazci, označuje to, že chceme rez **až do konca reťazca**. Teda:

```
reťazec[prvý : ]
```

Uvedomte si, že zápisy `reťazec[:počet]` a `reťazec[-počet:]` budú veľmi často označovať buď prvých alebo posledných `počet` znakov reťazca.

Ak neuvedieme ani jeden index v podreťazci, označuje to, že chceme **celý reťazec**, t.j. vytvorí sa kópia pôvodného reťazca:

```
reťazec[ : ]
```

kladné aj záporné indexy

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

napríklad

```
>>> abc = 'Monty Python'
>>> abc[6:]                 # od 6. znaku do konca
    'Python'
>>> abc[:5]                 # od zaciatku po 4. znak = prvých 5 znakov reťazca
    'Monty'
>>> abc[-4:]                # od 4. od konca az do konca = posledné 4 znaky reťaz
ca
```

```
    'thon'
>>> abc[16:]                # indexujeme mimo retazca
    ''
```

Podreťazce s krokom

Podobne ako vo funkcii `range()` aj pri indexoch podreťazca môžeme určiť aj krok indexov:

```
reťazec[prvý : zaposledný : krok]
```

kde `krok` určuje o koľko sa bude index v reťazci posúvať od `prvý` po `posledný`. Napríklad:

```
>>> abc = 'Monty Python'
>>> abc[2:10:2]
    'nyPt'
>>> abc[::3]
    'MtPh'
>>> abc[9:-7:-1]
    'htyP'
>>> abc[::-1]
    'nohtyP ytnoM'
>>> abc[6:] + ' ' + abc[:5]
    'Python Monty'
>>> abc[4::-1] + ' ' + abc[:5:-1]
    'ytnoM nohtyP'
>>> (abc[6:] + ' ' + abc[:5])[::-1]
    'ytnoM nohtyP'
>>> 'kobyla ma maly bok'[::-1]
    'kob ylam am alybok'
>>> abc[4:9]
    'y Pyt'
>>> abc[4:9][2]          # aj podretazce mozeme dalej indexovat
    'P'
>>> abc[4:9][2:4]
    'Py'
>>> abc[4:9][::-1]
    'tyP y'
```

Reťazce sú v pamäti nemenné (nemeniteľné)

Typ `str` (znakové reťazce) je nemeniteľný typ (hovoríme aj **immutable**). To znamená, že hodnota reťazca sa v pamäti nedá zmeniť. Ak budeme potrebovať reťazec, v ktorom je nejaká zmena, budeme musieť skonštruovať nový. Napríklad:

```
>>> abc[6] = 'K'
    TypeError: 'str' object does not support item assignment
```

Všetky doterajšie manipulácie s reťazcami nemenili reťazec, ale zakaždým vytvárali úplne nový (niekedy to bola len kópia pôvodného), napríklad:

```
>>> cba = abc[::-1]
>>> abc
    'Monty Python'
```

```
>>> cba
    'nohtyP ytnoM'
```

Takže, keď chceme v reťazci zmeniť nejaký znak, budeme musieť skonštruovať nový reťazec, napríklad takto:

```
>>> abc[6] = 'K'
    ...
    TypeError: 'str' object does not support item assignment
>>> novy = abc[:6] + 'K' + abc[7:]
>>> novy
    'Monty Kython'
>>> abc
    'Monty Python'
```

Alebo, ak chceme opraviť prvý aj posledný znak:

```
>>> abc = 'm' + abc[1:-1] + 'N'
>>> abc
'monty PythoN'
```

Porovnávanie jednoznakových reťazcov

Jednoznakové reťazce môžeme porovnávať relačnými operátormi `==`, `!=`, `<`, `<=`, `>`, `>=`, napríklad:

```
>>> 'x' == 'x'
    True
>>> 'm' != 'M'
    True
>>> 'a' > 'm'
    False
>>> 'a' > 'A'
    True
```

Python na porovnávanie používa vnútornú reprezentáciu Unicode (UTF-8). S touto reprezentáciou môžeme pracovať pomocou funkcií `ord()` a `chr()`:

- funkcia `ord(znak)` vráti vnútornú reprezentáciu znaku (kódovanie v pamäti počítača)

```
>>> ord('a')
    97
>>> ord('A')
    65
```

- opačná funkcia `chr(číslo)` vráti jednoznakový reťazec, pre ktorý má tento znak danú číselnú reprezentáciu

```
>>> chr(66)
    'B'
>>> chr(244)
    'ô'
```

Pri porovnávaní dvoch znakov sa porovnávajú ich vnútorné reprezentácie, t.j.

```
>>> ord('a') > ord('A')
    True
>>> 97 > 65
    True
>>> 'a' > 'A'
    True
```

Vnútornú reprezentáciu niektorých znakov môžeme zistiť, napríklad pomocou for-cyklu:

```
for i in range(ord('A'), ord('J')):
    print(i, chr(i))
```

```
65 A
66 B
67 C
68 D
69 E
70 F
71 G
72 H
73 I
```

Vyskúšajte:

```
>>> chr(8984)
>>> chr(9819)
>>> chr(9860)
>>> chr(9328)
```

Prípadne vyskúšajte vypísať tieto znaky do grafickej plochy pomocou nejakého väčšieho fontu v `create_text`.

Porovnávanie dlhších reťazcov

Dlhšie reťazce Python porovnáva postupne po znakoch:

- kým sú v oboch reťazcoch rovnaké znaky, preskakuje ich
- pri prvom rôznom znaku, porovná tieto dva znaky

Napríklad pri porovnávaní dvoch reťazcov ‚kocur' a ‚kohut':

- porovná 0. znaky: `'k' == 'k'`
- porovná 1. znaky: `'o' == 'o'`
- porovná 2. znaky: `'c' < 'h'` a tu aj skončí porovnávanie týchto reťazcov

Preto platí, že `'kocur' < 'kohut'`. Treba si dávať pozor **na znaky s diakritikou**, lebo, napríklad `ord('č') = 269 > ord('h') = 104`. Napríklad:

```
>>> 'kocúr' < 'kohút'
    True
>>> 'kočka' < 'kohut
    False
>>> 'PYTHON' < 'Python' < 'python'
    True
```

alebo

```
>>> 'pytagoras' < 'python' < 'pytliak' < 'pyton'
    True
```

Prechádzanie reťazca v cykle

Už sme videli, že prvky znakového reťazca môžeme prechádzať for-cyklom, v ktorom indexujeme celý reťazec postupne od `0` do `len()-1`:

```
a = 'Python'
for i in range(len(a)):
    print('.' * i, a[i])
```

```
 P
. y
.. t
... h
.... o
..... n
```

Tiež vieme, že for-cyklom môžeme prechádzať nielen postupnosť indexov (t.j. `range(len(a))`), ale priamo postupnosť znakov, napríklad:

```
for znak in 'python':
    print(znak * 5)
```

```
ppppp
yyyyy
ttttt
hhhhh
ooooo
nnnnn
```

Zrejme reťazec vieme prechádzať aj while-cyklom, napríklad:

```
a = '.....veľa bodiek'
print(a)
while len(a) != 0 and a[0] == '.':
    a = a[1:]
print(a)
```

```
.....veľa bodiek
veľa bodiek
```

Cyklus sa opakoval, kým bol reťazec neprázdny a kým boli na začiatku reťazca znaky bodky `'.'`. Vtedy sa v tele cyklu reťazec skracoval o prvý znak. Uvedený while-cyklus môžeme zapísať aj takto:

```
while a and a[0] == '.':
    a = a[1:]
```

Zamyslite sa, prečo musíme v podmienke kontrolovať, či je reťazec neprázdny. Hoci v tomto prípade by sme to vedeli zapísať aj takto (menej prehľadne):

```
while a[0:1] == '.':
    a = a[1:]
```

Automatické číslovanie prechodov vo for-cykle

Už sme si zvykli, že ak by sme potrebovali v takomto cykle:

```
for znak in 'Python':
    print(znak)
```

ku každému vypisovanému znaku pridať aj jeho poradové číslo, musíme použiť pomocnú premennú a zyšovať ju v cykle o 1, napríklad takto:

```
i = 0
for znak in 'Python':
    print(i, znak)
    i += 1
```

Keďže teraz už vieme znakové reťazce aj indexovať, vieme to prepísať aj takto:

```
retazec = 'Python'
for i in range(len(retazec)):
    print(i, retazec[i])
```

Existuje ešte jeden spôsob, ktorý je z týchto „najpythonovejší“ (pythonic). Využijeme ďalšiu štandardnú funkciu `enumerate`, pomocou ktorej môžeme for-cyklus zapísať novým štýlom:

```
for i, znak in enumerate('Python'):
    print(i, znak)
```

Táto funkcia očakáva, že ako parameter dostane nejakú postupnosť (napríklad postupnosť znakov) a jej úlohou bude každý prvok tejto postupnosti **očíslovať**. Vďaka tomuto for-cyklus dokáže nastavovať naraz dve premenné cyklu: premennú pre očíslovanie (premenná `i`) a premennú pre hodnotu (u nás je to premenná `znak`). Tento cyklus teda prejde 6-krát, ale zakaždým nastaví naraz dve premenné:

```
0 P
1 y
2 t
3 h
4 o
5 n
```

Všimnite si, že číslovanie prechodov cyklu prebieha od `0` a nie od `1`, presne tak, ako je to v Pythone zvykom.

Ukážme ešte jeden príklad s `enumerate`. Najprv obyčajný for-cyklus:

```
for cislo in range(5, 25, 6):
    print(cislo)

5
```

```
11
17
23
```

Ak teraz potrebujeme **očíslovať** každý prechod tohto for-cyklu, môžeme zapísať:

```
for i, cislo in enumerate(range(5, 25, 6)):
    print(i, cislo)
```

```
0 5
1 11
2 17
3 23
```

Pripočítavacia šablóna

Pripomeňme si pripočítavaciu šablónu z 2. prednášky, pomocou ktorej sme prešli prvky (znaky) znakového reťazca a poskladali sme z nich dva ďalšie raťazce:

```
vstup = input('zadaj: ')
pocet = 0
retazec1 = retazec2 = ''
for znak in vstup:
    retazec1 = retazec1 + znak
    retazec2 = znak + retazec2
    pocet = pocet + 1
print('počet znakov reťazca =', pocet)
print('retazec1 =', retazec1)
print('retazec2 =', retazec2)
```

Táto šablóna znamená to, že pred cyklom inicializujeme premenné, ktoré sa budú v cykle meniť. Najčastejšie budeme tieto premenné v cykle meniť pripočítavaním, alebo odpočítavaním, pridávaním, zreťazovaním a pod. Po skončení cyklu bude v týchto premenných výsledok. V našom príklade bude v premennej `retazec1` kópia vstupného reťazca `vstup`, v `retazec2` bude prevrátená hodnota vstupu a v `pocet` počet prechodov cyklu, teda počet znakov v reťazci.

Reťazcové funkcie

Už poznáme tieto štandardné funkcie:

- `len()` - dĺžka reťazca
- `int()`, `float()` - prevod reťazca na celé alebo desatinné číslo
- `bool()` - prevod reťazca na `True` alebo `False` (ak je prázdny, výsledok bude `False`)
- `str()` - prevod čísla (aj ľubovoľnej inej hodnoty) na reťazec
- `ord()`, `chr()` - prevod do a z **Unicode**

Okrem nich existujú ešte aj tieto tri užitočné štandardné funkcie:

- `bin()` - prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v dvojkovej sústave
- `hex()` prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v šestnástkovej sústave
- `oct()` - prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v osmičkovej sústave

Napríklad

```
>>> bin(123)
    '0b1111011'
>>> hex(123)
    '0x7b'
>>> oct(123)
    '0o173'
```

Zápisy celého čísla v niektorej z týchto sústav fungujú ako celočíselné konštanty:

```
>>> 0b1111011
    123
>>> 0x7b
    123
>>> 0o173
    123
```

Vlastné funkcie

Môžeme vytvárať vlastné funkcie, ktoré majú aj reťazcové parametre, resp. môžu vracať reťazcovú návratovú hodnotu. Niekoľko námetov:

- funkcia vráti `True` ak je daný znak (jednoznakový reťazec) číslicou:

```
def je_cifra(znak):
    return '0' <= znak <= '9'
```

alebo inak

```
def je_cifra(znak):
    return znak in '0123456789'
```

- funkcia vráti `True` ak je daný znak (jednoznakový reťazec) malé alebo veľké písmeno (anglickej abecedy)

```
def je_pismeno(znak):
    return 'a' <= znak <= 'z' or 'A' <= znak <= 'Z'
```

- parametrom funkcie je reťazec s menom a priezviskom (oddelené sú práve jednou medzerou) - funkcia vráti reťazec, v ktorom bude najprv priezvisko a až za tým meno (oddelené medzerou)

```
def meno(r):
    ix = 0
```

```
    while ix < len(r) and r[ix] != ' ':     # najde medzeru
        ix += 1
    return r[ix+1:] + ' ' + r[:ix]
```

- funkcia vráti prvé slovo vo vete; predpokladáme, že obsahuje len malé a veľké písmená (využijeme funkciu `je_pismeno`)

```
def slovo(veta):
    for i in range(len(veta)):
        if not je_pismeno(veta[i]):
            return veta[:i]
    return veta
```

Reťazcové metódy

Je to špeciálny spôsob zápisu volania funkcie (bodková notácia):

```
reťazec.metóda(parametre)
```

kde `metóda` je meno niektorej z metód, ktoré sú v systéme už definované pre znakové reťazce. My si ukážeme niekoľko užitočných metód, s niektorými ďalšími sa zoznámime neskôr:

- `reťazec.count(podreťazec)` - zistí počet výskytov podreťazca v reťazci
- `reťazec.find(podreťazec)` - zistí index prvého výskytu podreťazca v reťazci
- `reťazec.lower()` - vráti reťazec, v ktorom prevedie všetky písmená na malé
- `retazec.upper()` - vráti reťazec, v ktorom prevedie všetky písmená na veľké
- `reťazec.replace(podreťazec1, podreťazec2)` - vráti reťazec, v ktorom nahradí všetky výskyty `podreťazec1` iným reťazcom `podreťazec2`
- `reťazec.strip()` - vráti reťazec, v ktorom odstráni medzery na začiatku a na konci reťazca (odfiltruje pritom aj iné oddeľovacie znaky ako `'\n'` a `'\t'`)
- `reťazec.format(hodnoty)` - vráti reťazec, v ktorom nahradí formátovacie prvky `'{}'` zadanými hodnotami

Ak chceme o niektorej z metód získať **help**, môžeme zadať, napríklad:

```
>>> help(''.find)
Help on built-in function find:

find(...) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    ...
```

metóda `reťazec.count()`

`reťazec.count(`*`podreťazec`*`)`

Parametre

- **reťazec** – reťazec, v ktorom sa budú hľadať všetky výskyty nejakého zadaného podreťazca
- **podreťazec** – hľadaný podreťazec

Metóda zistí počet všetkých výskytov podreťazca v danom reťazci. Napríklad:

```
>>> 'Python'.count('th')
    1                    # reťazec 'th' sa nachádza v 'Python' iba raz
>>> 'Python'.count('to')
    0                    # reťazec 'to' sa v 'Python' nenachádza ani raz
>>> 'Pyp ypY Ypy yPy yPY'.count('Py')
    2                    # reťazec 'Py' sa tu nachádza na 2 miestach
```

metóda `reťazec.find()`

`reťazec.find(`*`podreťazec`*`)`

Parametre

- **reťazec** – reťazec, v ktorom sa bude hľadať prvý výskyt nejakého zadaného podreťazca
- **podreťazec** – hľadaný podreťazec

Metóda nájde prvý najľavejší výskyt podreťazca v danom reťazci alebo vráti `-1`, keď ho nenájde. Napríklad:

```
>>> 'Python'.find('th')
    2                        # retazec 'th' sa nachadza v 'Python' od 2. indexu
>>> 'Python'.find('to')
    -1                       # retazec 'to' sa v 'Python' nenachadza
>>> 'abcd ce abced'.find('ce')
    5                        # prvy vyskyt retazca 'ce' je na indexe 5
```

metóda `reťazec.lower()`

`reťazec.lower()`

Parametre

reťazec – reťazec, z ktorého sa vyrobí nový, ale s malými písmenami

Metóda vyrobí kópiu daného reťazca, v ktorej všetky veľké písmená prerobí na malé. Nepísmenové znaky nemení. Napríklad:

```
>>> 'PyTHon'.lower()
    'python'
>>> '1+2'.lower()
    '1+2'
```

metóda `reťazec.upper()`

`reťazec.upper()`

Parametre

reťazec – reťazec, z ktorého sa vyrobí nový, ale s veľkými písmenami

Metóda vyrobí kópiu daného reťazca, v ktorej všetky malé písmená prerobí na veľké. Nepísmenové znaky nemení. Napríklad:

```
>>> 'PyTHon'.upper()
    'PYTHON'
>>> '1+2'.upper()
    '1+2'
```

metóda `reťazec.replace(podreťazec1, podreťazec2)`

`reťazec.replace()`

Parametre

reťazec – reťazec, z ktorého sa vyrobí nový, ale s nahradenými výskytmi podreťazcov

Metóda vyrobí kópiu daného reťazca, v ktorej všetky veľké výskyty `podreťazec1` prerobí na `podreťazec2`. Napríklad:

```
>>> 'Monty Python'.replace('y', '***')
    'Mont*** P***thon'
>>> 'abradabra'.replace('ra', 'y').replace('by', 'ko')
    'akodako'
```

Najprv sa nahradia výskyty `'ra'` na jednoznakové `'y'` (dostaneme `'abydaby'`) a v tomto novom reťazci sa výskyty `'by'` nahradia reťazcom `'ko'`. Všimnite si, že bude fungovať aj:

```
>>> (10 * 'abc').replace('bc', '=')
    'a=a=a=a=a=a=a=a=a=a='
```

metóda `reťazec.strip()`

`reťazec.strip()`

Parametre

reťazec – reťazec, z ktorého sa odfiltrujú medzerové znaky na začiatku a na konci reťazca

Metóda vyrobí kópiu daného reťazca, v ktorej vyhodí všetky medzerové znaky (medzery, ale aj `'\n'` a `'\t'`) zo začiatku aj konca reťazca. Napríklad:

```
>>> '          Python'.strip()
    'Python'
>>> 'Python\n\n\n'.strip()
    'Python'
>>> 'P y  t h  o n'.strip()
    'P y  t h  o n'
```

Formátovanie reťazca

Možnosti formátovania pomocou formátovacích reťazcov `f'{x}'` sme už videli predtým. Teraz ukážeme niekoľko užitočných formátovacích prvkov. Volanie má tvar:

```
f'formátovací reťazec s hodnotami v {}'
```

Takýto zápis, ale funguje až od verzie Pythonu **3.6** a vyššie. V starších verziách budeme musieť použiť tvar `reťazec.format(parametre)`, ale ten tu rozoberať nebudeme, hoci sa veľmi podobá novšiemu zápisu.

Formátovací reťazec obsahuje formátovacie prvky, ktoré sa nachádzajú v `{}` zátvorkách: v týchto zátvorkách sa **musí** náchadzať priamo nejaká hodnota (môže to byť ľubovoľný pythonovský výraz) a za ňou sa **môže** nachádzať špecifikácia oddelená znakom `':'`. Napríklad:

```
>>> x = 1237
>>> a = f'vysledok pre x={x} je {x + 8}'
>>> ahoj = 'hello'
```

```
>>> b = f'ahoj po anglicky je "{ahoj}"'
```

Python v tomto prípade najprv vyhľadá všetky výskyty zodpovedajúcich `'{}'` a vyhodnotí (vypočíta hodnoty) výrazov, ktoré sú vo vnútri týchto zátvoriek. Tieto hodnoty potom dosadí do reťazca namiesto zápisu `{...}`. Zrejme, ak to neboli reťazcové hodnoty (napríklad to boli čísla), tak ich najprv prevedie na reťazce.

V ďalšom príklade sme v `{}` použili aj nejaké špecifikácie, t.j. upresnenie formátovania:

```
>>> r, g, b = 100, 150, 200
>>> farba = f'#{r:02x}{g:02x}{b:02x}'
```

Špecifikácia formátu

V zátvorkách `'{}'` sa môžu nachádzať rôzne upresnenia formátovania, ktorými určujeme detaily, ako sa budú vypočítané hodnoty prevádzať na reťazce. Prvé číslo za `:` väčšinou označuje šírku (počet znakov), do ktorej sa vloží reťazec a ďalej tam môžu byť znaky na zarovnanie (`'<'`, `'>'`, `'^'`) a znaky na typ hodnoty (`'d'`, `'f'`, …). Napríklad:

- `'{hodnota:10}'` - šírka výpisu 10 znakov
- `'{hodnota:>7}'` - šírka 7, zarovnané vpravo
- `'{hodnota:<5d}'` - šírka 5, zarovnané vľavo, parameter musí byť celé číslo (bude sa vypisovať v 10-ovej sústave)
- `'{hodnota:12.4f}'` - šírka 12, parameter desatinné číslo vypisované na 4 desatinné miesta
- `'{hodnota:06x}'` - šírka 6, zľava doplnená nulami, parameter celé číslo sa vypíše v 16-ovej sústave
- `'{hodnota:^20s}'` - šírka 20, vycentrované, parametrom je reťazec

Zhrňme najpoužívanejšie písmená pri označovaní typu parametra:

- `d` - celé číslo v desiatkovej sústave
- `b` - celé číslo v dvojkovej sústave
- `x` - celé číslo v šestnástkovej sústave
- `s` - znakový reťazec
- `f` - desatinné číslo (možno špecifikovať počet desatinných miest, inak default 6)
- `g` - desatinné číslo vo všeobecnom formáte

Dokumentačný reťazec pri definovaní funkcie

Ak funkcia vo svojom tele hneď ako **prvý riadok** obsahuje znakový reťazec (zvykne byť viacriadkový s `'''`), tento sa stáva, tzv. **dokumentačným reťazcom** (**docstring**). Pri vykonávaní tela funkcie sa takéto reťazce ignorujú (preskakujú). Tento reťazec (docstring) sa, ale môže neskôr vypísať, napríklad štandardnou funkciou `help()`.

Zadefinujme reťazcovú funkciu a hneď do nej dopíšeme aj niektoré základné informácie:

```
def pocet_vyskytov(podretazec, retazec):
    '''funkcia vráti počet výskytov podreťazca v reťazci

    prvý parameter podretazec - ľubovoľný neprázdny reťazec, o ktorom sa
                                bude zisťovať počet výskytov
    druhý parameter retazec - reťazec, v ktorom sa hľadajú výskyty
```

```
    ak je prvý parameter podretazec prázdny reťazec, funkcia vráti len(retazec)
    '''
    pocet = 0
    for ix in range(len(retazec)):
        if retazec[ix:ix+len(podretazec)] == podretazec:
            pocet += 1
    return pocet
```

Takto definovaná funkcia funguje rovnako, ako keby žiaden dokumentačný reťazec neobsahovala, ale teraz bude fungovať aj:

```
>>> help(pocet_vyskytov)
Help on function pocet_vyskytov in module __main__:

pocet_vyskytov(podretazec, retazec)
    funkcia vráti počet výskytov podreťazca v reťazci

    prvý parameter podretazec - ľubovoľný neprázdny reťazec, o ktorom sa
                                bude zisťovať počet výskytov
    druhý parameter retazec - reťazec, v ktorom sa hľadajú výskyty

    ak je prvý parameter podretazec prázdny reťazec, funkcia vráti len(retazec)
```

Tu môžeme vidieť užitočnú vlastnosť Pythonu: programátor, ktorý vytvára nejaké nové funkcie, môže hneď vytvárať aj malú dokumentáciu o jej používaní pre ďalších programátorov. Asi ľahko uhádneme, ako funguje napríklad aj toto:

```
>>> help(hex)
Help on built-in function hex in module builtins:

hex(number, /)
    Return the hexadecimal representation of an integer.

    >>> hex(12648430)
    '0xc0ffee'
```

Pri takomto spôsobe samodokumentácie funkcií si treba uvedomiť, že Python v tele funkcie ignoruje nielen všetky reťazce, ale aj iné konštanty:

- ak napríklad zavoláme funkciu, ktorá vracia nejakú hodnotu a túto hodnotu ďalej nespracujeme (napríklad priradením do premennej, použitím ako parametra inej funkcie, …), vyhodnocovanie funkcie takúto návratovú hodnotu ignoruje
- ak si uvedomíme, že meno funkcie bez okrúhlych zátvoriek nespôsobí volanie tejto funkcie, ale len hodnotu referencie na funkciu, tak aj takýto zápis sa ignoruje

Napríklad všetky tieto zápisy sa v tele funkcie (alebo aj v programovom režime mimo funkcie) ignorujú:

```
s.replace('a', 'b')
print
g.pack
pocet + 1
i == i + 1
math.sin(uhol)
```

Python pri nich nehlási ani žiadnu chybu.

Príklad s kreslením a reťazcami

Navrhnime malú aplikáciu, v ktorej budeme pohybovať mysleným perom. Toto pero sa bude hýbať v jednom zo štyroch smerov: `'s'` pre sever, `'v'` pre východ, `'j'` pre juh, `'z'` pre západ. Dĺžka kroku pera nech je nejaká malá konštanta, napríklad `10`:

```
import tkinter

def kresli(retazec):
    x, y = 100, 100
    for znak in retazec:
        x1, y1 = x, y
        if znak == 's':
            y1 -= 10
        elif znak == 'v':
            x1 += 10
        elif znak == 'j':
            y1 += 10
        elif znak == 'z':
            x1 -= 10
        else:
            print('nerozumiem "' + znak + '"')
            return
        canvas.create_line(x, y, x1, y1)
        x, y = x1, y1

canvas = tkinter.Canvas()
canvas.pack()

kresli('ssvvjjzz')

tkinter.mainloop()
```

Po spustení dostaneme:

Zrejme rôzne reťazce znakov, ktoré obsahujú len naše štyri písmená pre smery pohybu, budú kresliť rôzne útvary. Napríklad:

```
kresli('vvvvvvvjjjjjjjzzzzzzzsssssss')
```

nakreslí trochu väčší štvorec. Toto vieme zapísať napríklad aj takto:

```
kresli('v'*7 + 'j'*7 + 'z'*7 + 's'*7)
```

Alebo

```
def stvorec(n):
    return 'v'*n + 'j'*n + 'z'*n + 's'*n

kresli(stvorec(7))
```

Na cvičeniach budeme rôzne vylepšovať túto ideu funkcie `kresli()`.

Cvičenia

L.I.S.T.

- riešenia **aspoň 10 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- používaj len konštrukcie z doterajších prednášok
- pozri si Riešenie úloh 6. cvičenia

1. Ručne bez počítača zisti, čo sa vypíše:

```
>>> x, y = 'Bratislava', 'Košice'
>>> y[1] + x[4] + y[3] + x[-4] + y[-5]
    ...
>>> x[5:8] + 3 * x[3] + y[2:]
    ...
>>> y[:2] + x[-2:]
    ...
>>> x[1::2] + y[2::2] + x[2::3]
    ...
>>> x.replace('a', 'e') + y.replace('ic', 'm')
    ...
>>> (y + x).replace('i', '').replace('a', 'xa')
    ...
```

   Potom to skontroluj pomocou Pythonu.

2. Napíš funkciu `sucet(retazec)`, ktorá dostáva znakový reťazec s dvomi celými číslami oddelenými znakom `'+'`. Funkcia vráti (nič nevypisuje) celé číslo, ktoré je súčtom dvoch čísel v reťazci. Použi metódu `retazec.find('+')` a funkciu `int()`. Napríklad:

```
>>> x = sucet('12+9')
>>> x
    21
>>> sucet('987654321+99999')
    987754320
```

3. Zovšeobecni funkciu `sucet(retazec)` z predchádzajúcej úlohy: vstupný reťazec obsahuje aspoň jedno číslo a keď ich je viac, sú oddelené znakom `'+'`. Funkcia vypočíta súčet. Napríklad:

```
>>> x = sucet('12+9')
>>> x
    21
>>> sucet('1+2+3+4')
    10
>>> sucet('1234')
    1234
```

4. Napíš funkciu `postupnost(start, koniec)`, ktorá vytvorí znakový reťazec z postupnosti čísel `range(start, koniec)`. Čísla v tejto postupnosti budú oddelené znakom medzera `' '`. Napríklad:

```
>>> p = postupnost(5, 13)
>>> p
'5 6 7 8 9 10 11 12'
```

   Do funkcie pridaj ešte jeden parameter `postupnost(start, koniec, krok=1)` tak, aby fungovalo napríklad:

```
>>> p = postupnost(13, 5, -2)
>>> p
'13 11 9 7'
```

5. Napíš funkciu `rozsekaj(text, sirka)`, ktorá vypíše zadaný text do viacerých riadkov, pričom každý (možno okrem posledného) má presne `sirka` znakov. Napríklad:

```
>>> ret = rozsekaj('Anicka dusicka, kde si bola', 10)
>>> ret
    'Anicka dus\nicka, kde \nsi bola'
>>> print(ret)
    Anicka dus
    icka, kde
    si bola
```

6. Napíš funkciu `stvorec(n, znak)`, ktorá vráti jeden dlhý znakový reťazec. Znakový reťazec by po vypísaní pomocou `print` vytvoril obvod štvorca z daného znaku. Môžeš predpokladať, že `n` nebude menšie ako 2. Napríklad:

```
>>> r = stvorec(5, '#')
>>> r
    '#####\n#   #\n#   #\n#   #\n#####'
>>> print(r)
    #####
    #   #
    #   #
    #   #
    #####
```

Pokús sa to vyriešiť tak, že telo funkcie bude obsahovať jediný riadok `return`:

```
def stvorec(n, znak):
    return ...
```

7. Napíš funkciu `vyhod_duplikaty(retazec)`, ktorá z daného reťazca vyhodí všetky za sebou idúce opakujúce sa znaky (nechá len jeden z nich). Napríklad:

```
>>> x = vyhod_duplikaty('Braatisssllavaaaaa')
>>> x
'Bratislava'
```

8. Napíš funkciu `ozatvorkuj(retazec, podretazec)`, ktorá v zadanom reťazci `retazec` všetky výskyty daného podreťazca ozátvorkuje. Napríklad:

```
>>> b = ozatvorkuj('Bratislava', 'a')
>>> b
    'Br(a)tisl(a)v(a)'
>>> ozatvorkuj('prospešné programovanie v prologu', 'pro')
```

```
    '(pro)spešné (pro)gramovanie v (pro)logu'
```

9. Znakový reťazec vieme prevrátiť pomocou zápisu `retazec[::-1]`. Napíš funkciu `prevrat(retazec)`, ktorá len pomocou cyklu a zreťazovania prevráti zadaný reťazec. Napríklad:

```
>>> x = prevrat('tseb eht si nohtyP')
>>> x
    'Python is the best'
```

10. Napíš funkcie `male(retazec, i)` a `velke(retazec, i)`, ktoré `i`-te písmeno v reťazci prerobia na malé (resp. veľké). Napríklad:

```
>>> r = male('PYTHON', 3)
>>> r
    'PYThON'
>>> r = velke('python', 5)
>>> r
    'pythoN'
```

11. Napíš funkciu `riadky(retazec)`, ktorá vypíše daný viacriadkový reťazec, ale pritom každý riadok očísluje číslami od 1 do počet riadkov. Napríklad:

```
>>> riadky('prvy riadok\n\ntreti je posledny')
    1. prvy riadok
    2.
    3. treti je posledny
>>> riadky('len \\n jeden riadok')
    1. len \n jeden riadok
```

12. Napíš funkciu `posun_znak(znak, posun)`, ktorá posunie daný znak v abecede o `p` znakov vpravo (resp. vľavo, ak je záporné). Na konci abecedy sa pokračuje od začiatku. Funkcia posúva len písmená malej abecedy, ostatné znaky nemení. Napríklad:

```
>>> posun_znak('c', 4)
    'g'
>>> posun_znak('g', -4)
    'c'
>>> posun_znak('x', 10)
    'h'
>>> posun_znak('A', 10)
    'A'
```

13. Napíš funkciu `zakoduj(text, posun)`, ktorá posunie v abecede všetky znaky (pomocou funkcie `posun_znak`). Napríklad:

```
>>> x = zakoduj('pyThon', 10)
```

```
>>> x
    'ziTryx'
>>> zakoduj(x, -10)
    'pyThon'
>>> zakoduj(x, 16)
    'pyThon'
```

14. Napíš funkciu `je_palindrom(reťazec)`, ktorá zistí (vráti `True` alebo `False`), či je zadaný reťazec palindróm. Funkcia ignoruje medzery a nerozlišuje medzi malými a veľkými písmenami. Napríklad:

```
>>> je_palindrom('Python')
    False
>>> je_palindrom('tahat')
    True
>>> je_palindrom('Jelenovi Pivo Nelej')
    True
```

15. Metóda `'reťazec'.count(podreťazec)` zisťuje počet výskytov podreťazca v reťazci. Napíš funkciu `pocet(retazec, podretazec)`, ktorá robí to isté, ale bez použitia tejto metódy. Napríklad:

```
>>> pocet('mama ma emu a ema ma mamu', 'ma ')
    4
>>> pocet('mama ma emu a ema ma mamu', 'am')
    2
```

16. Napíš funkciu `usporiadaj(h1, h2, h3)`, ktorá z troch zadaných hodnôt (všetky tri sú rovnakého typu, napríklad reťazce) vytvorí reťazec (vráti ho ako výsledok funkcie) zlepením týchto troch hodnôt v utriedenom poradí: najprv najmenšia (napríklad reťazec prvý v abecede), potom väčšia a na koniec najväčšia. Medzi zlepené reťazce vloží medzeru. Napríklad:

```
>>> x = usporiadaj('python', 'pytliak', 'pytagoras')
>>> x
    'pytagoras python pytliak'
>>> usporiadaj(345, 123, 234)
    '123 234 345'
```

17. Napíš funkciu `nazov(n)`, pomocou ktorej sa bude dať vygenerovať názov hudobnej skupiny. Chceme, aby toto meno začínalo a končilo rovnakou samohláskou a medzi týmito samohláskami by sa malá `n` krát objaviť nejaká spoluhláska. Prvé písmeno mena by malo byť veľké ostatné malé. Zrejme využiješ ideu z 2. prednášky, pomocou ktorej sa náhodne generovali písmená. Môžeš dostať napríklad:

```
for i in range(5):
    print(nazov(2))
    print(nazov(3))
```

```
Uxxu
Uxxxu
Ippi
Idddi
Atta
Ottto
Yggy
Odddo
Aqqa
Ibbbi
```

18. Unicode `0x2654` a ďalších päť za ním sú obrázky šachových figúrok. Napíš program, ktorý do grafickej plochy nakresli vedľa seba všetkých 6 figúrok náhodnými farbami väčším fontom (napríklad `'arial 50'`). Môžeš dostať takýto obrázok:

19. Napíš funkciu `stvorce(vel, retazec)`, ktorá dostáva dva parametre: veľkosť štvorca a znakový reťazec s menami farieb. Funkcia nakreslí rad farebných štvorcov veľkosti `vel`, ktoré budú zafarbené farbami z reťazca. Zrejme štvorcov bude toľko, koľko farieb je v reťazci. Pre takéto volanie:

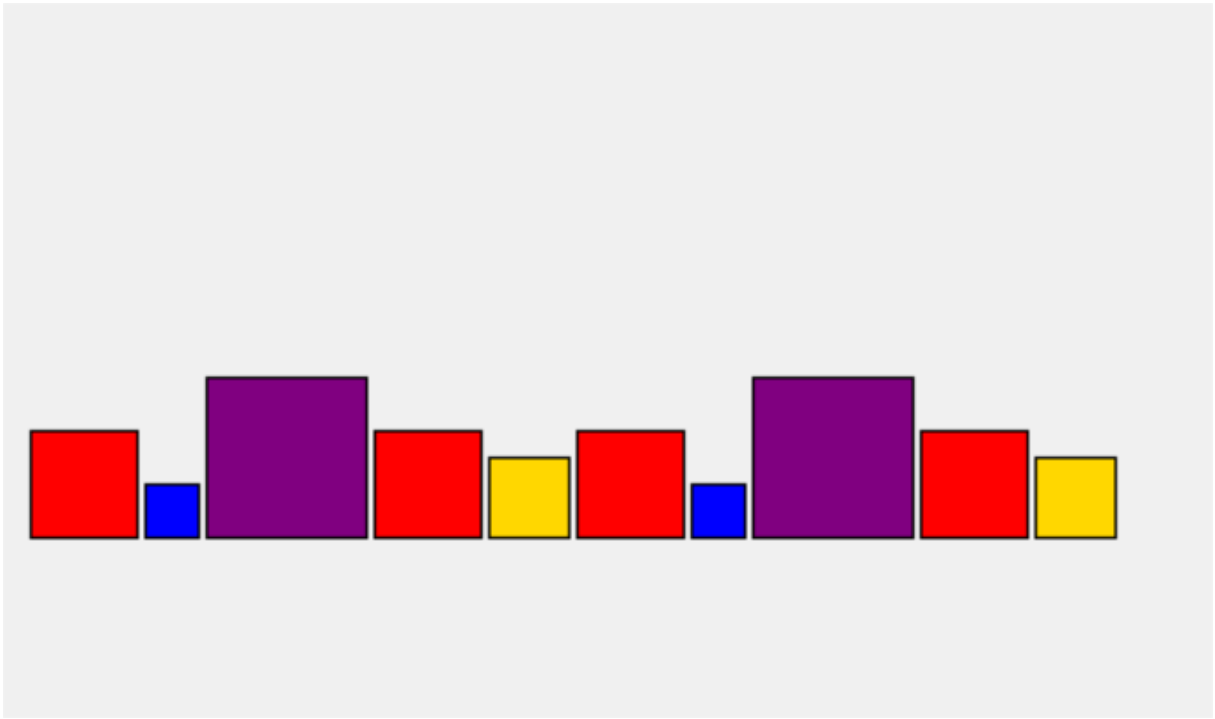
```
stvorce(40, 'red blue purple red gold')
```

by si mohol dostať takýto obrázok:

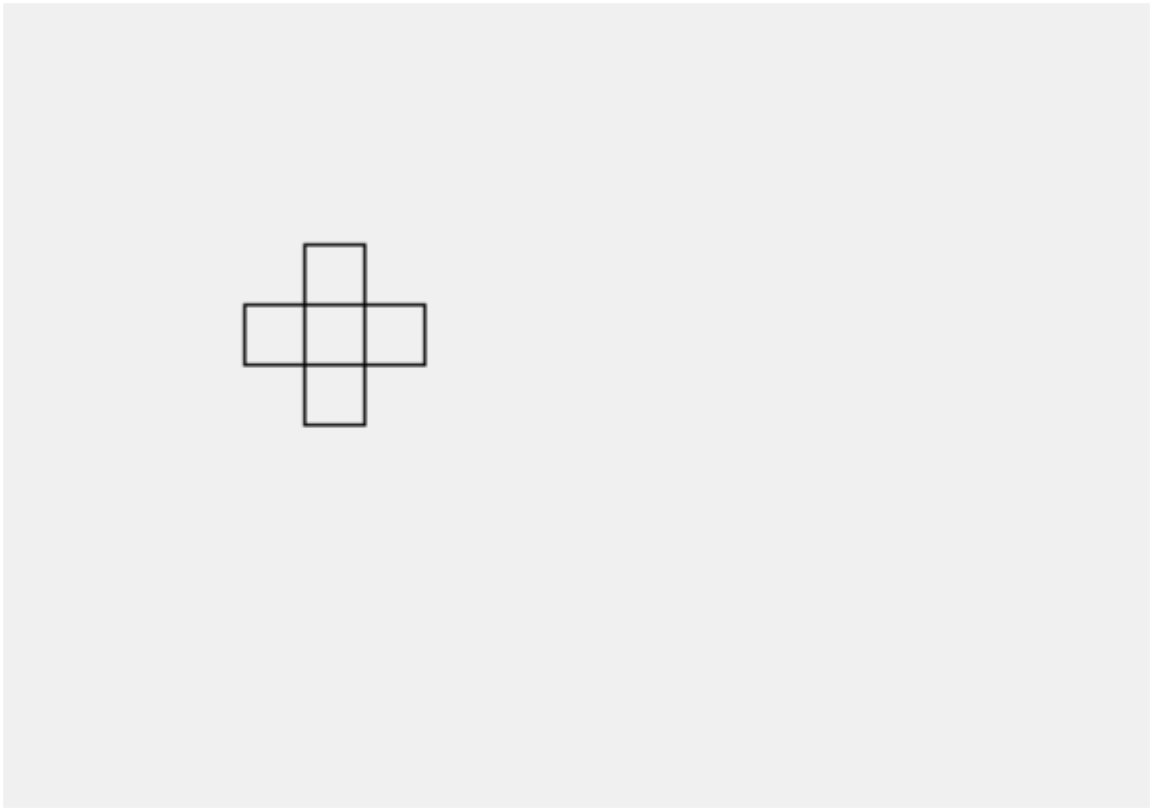
20. Úloha bude podobná predchádzajúcej: napíš funkciu `stvorce(retazec)`, ktorá dostáva v reťazci informáciu o veľkosti a farbe štvorcov. Funkcia bude tieto štvorce kresliť vedľa seba, ale len dovtedy, kým by nasledovný nevypadol z grafickej plochy (tento reťazec sa stále opakuje od začiatku). Do premennej `sirka` nastav nejakú šírku grafickej plochy a zavolaj funkciu, napríklad takto:

```
21. sirka = 450
22. canvas = tkinter.Canvas(width=sirka)
23. canvas.pack()
24. 
25. stvorce('40 red 20 blue 60 purple 40 red 30 gold')
```

Mohol by si dostať takýto obrázok:



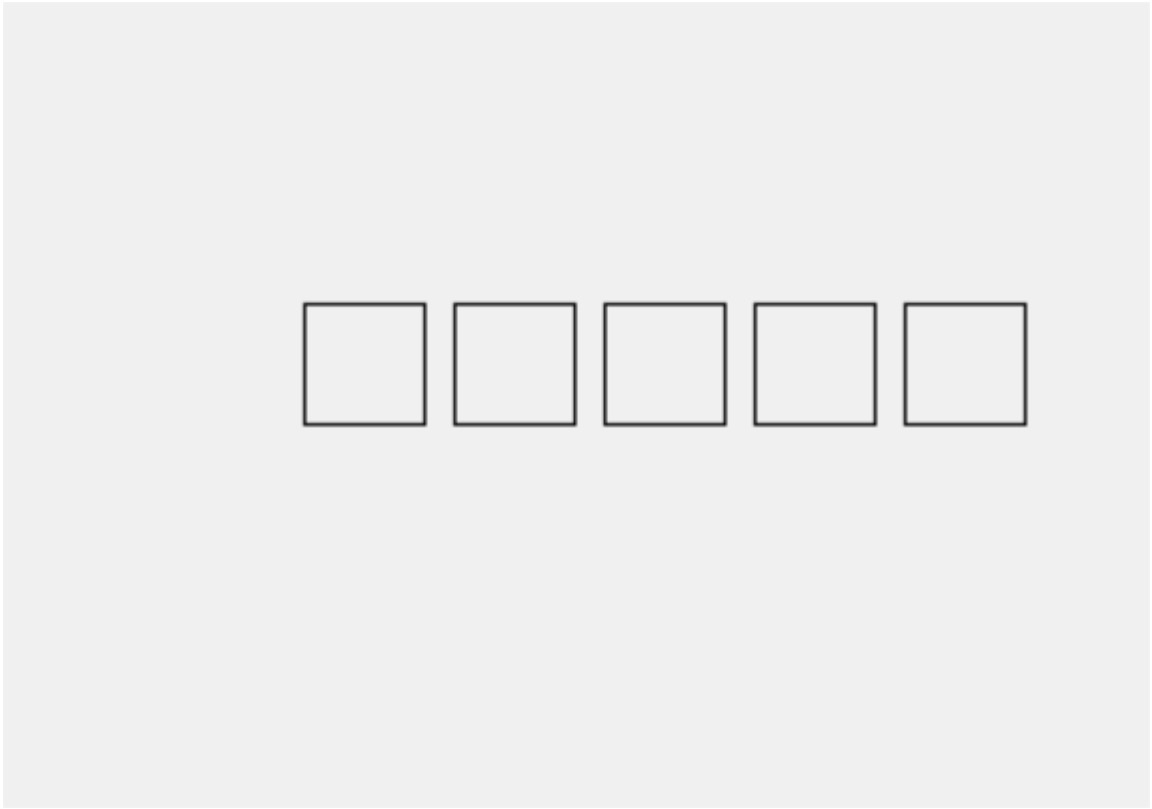
21. Na konci prednášky je funkcia `kresli(retazec)`, pomocou ktorej môžeme vytvárať nejakú kresbu zakódovanú písmenami `'svjz'`. Nakresli pomocou tejto funkcie takýto obrázok:



22. Dopíš do tejto funkcie spracovanie týchto ďalších znakov:
    - `'h'` - kresliace pero sa bude odteraz pohybovať bez kreslenia (pero hore)
    - `'d'` - kresliace pero bude odteraz pri pohybe kresliť (pero dole)
    - číslice od `'1'` do `'9'` - nasledovný príkaz (jeden z `'svjz'`) sa vykoná príslušný počet krát
    - napríklad:

```
23. >>> kresli('4v4j4z4sh5vd'*5)
```

nakreslí vedľa seba 5 štvorcov:



3. Týždenný projekt

L.I.S.T.

- riešenie odovzdaj na úlohový server https://list.fmph.uniba.sk/

Napíš pythonovský skript, ktorý bude definovať tieto 4 funkcie:

- funkcia `pocet_dni_v_mesiaci(mesiac, priestupny=False)` - vráti číslo od `28` do `31`, pričom parameter mesiac je znakový reťazec, v ktorom sú dôležité len prvé tri znaky, tie sú `'jan'`, `'feb'`, `'mar'`, `'apr'`, `'maj'`, `'jun'`, `'jul'`, `'aug'`, `'sep'`, `'okt'`, `'nov'`, `'dec'`,
    - zrejme `pocet_dni_v_mesiaci('feb', True)` vráti `29`

- funkcia `pocet_dni_medzi(datum1, datum2)` - pre dva dátumy vypočíta počet dní, ktoré sú medzi nimi
    - oba dátumy sú zadané ako znakové reťazce v tvare `'mesiac.rok'`, pričom pre mesiac sú dôležité len prvé tri znaky
    - prvý dátum je vlastne `1.mesiac1.rok1`, druhý dátum je `1.mesiac2.rok2`
    - môžeš predpokladať, že prvý dátum je pred alebo rovný druhému dátumu; keď sa oba dátumy rovnajú, funkcia vráti `0`
    - oba roky budú v intervale `<1901, 2099>`
    - napríklad:

```
>>> print(pocet_dni_medzi('sep.2020', 'okt.2020'))        # medzi 1.9.2020 a 1.10.2020
    30
>>> print(pocet_dni_medzi('okt.2019', 'okt.2020'))        # medzi 1.10.2019 a 1.10.2020
    366
>>> print(pocet_dni_medzi('januar.1999', 'oktober.2020')) # medzi 1.1.1999 a 1.10.2020
    7944
```

    - zrejme využiješ funkciu `pocet_dni_v_mesiaci()` a to, že priestupný rok (`rok%4==0`) má `366` dní a nepriestupný `365`
- funkcia `den_v_tyzdni(datum)`, kde `datum` je znakový reťazec vo formáte `'den.mesiac.rok'`
    - mesiac v tomto dátume je znakový reťazec, v ktorom sú dôležité len prvé tri znaky
    - funkcia vráti deň v týždni ako trojznakový reťazec, jeden z `'pon'`, `'uto'`, `'str'`, `'stv'`, `'pia'`, `'sob'`, `'ned'`
    - môžeš to počítať tak, že najprv zistíš počet dní, ktoré uplynuli od dátumu **1.január.1901** a keďže vieme, že vtedy bol **utorok**, ľahko z toho vypočítaš deň v týždni (bude to nejaký zvyšok po delení `7`)
    - môžeš predpokladať, že dátum bude zadaný korektne a bude z intervalu `<1.jan.1901, 31.dec.2099>`
    - napríklad:

```
>>> den_v_tyzdni('8.oktober.2021')
    'pia'
>>> den_v_tyzdni('1.jan.1901')
    'uto'
>>> den_v_tyzdni('23.jun.1912')
    'ned'
```

    - Vedel by si zistiť, čím je dátum **23. júna 1912** zaujímavý?
- `kalendar(datum)` - vypíše (pomocou `print()`) kalendár pre jeden mesiac v takomto tvare

- dátum je zadaný ako znakový reťazec v tvare `'mesiac.rok'`, pričom pre mesiac sú dôležité len prvé tri znaky
- napríklad:

```
>>> kalendar('oktober.2021')
    pon uto str stv pia sob ned
                      1   2   3
      4   5   6   7   8   9  10
     11  12  13  14  15  16  17
     18  19  20  21  22  23  24
     25  26  27  28  29  30  31
>>> kalendar('maj.1945')
    pon uto str stv pia sob ned
          1   2   3   4   5   6
      7   8   9  10  11  12  13
     14  15  16  17  18  19  20
     21  22  23  24  25  26  27
     28  29  30  31
```

- prvý riadok obsahuje mená dní v týždni presne v tomto tvare, ďalej nasledujú dni v mesiaci, ktoré sú naformátované presne do zodpovedajúcich stĺpcov
- treba si dať pozor na medzery

Na otestovanie správnosti svojich funkcií môžeš využiť, napríklad tento štandardný modul:

```
>>> import calendar
>>> calendar.weekday(2021, 10, 8)      # 8.oktober 2021, vráti: 0=nedela, 1=pondelok, ...4=piatok
    4
>>> calendar.day_name[calendar.weekday(2021, 10, 8)]
    'Friday'
>>> calendar.prmonth(2021, 10, 3)      # oktober 2021, čísla na šírku 3 znaky
            October 2021
    Mon Tue Wed Thu Fri Sat Sun
                      1   2   3
      4   5   6   7   8   9  10
     11  12  13  14  15  16  17
     18  19  20  21  22  23  24
     25  26  27  28  29  30  31
```

Tvoj odovzdaný program s menom `riesenie.py` musí začínať tromi riadkami komentárov (zmeň na svoje meno a dátum odovzdania):

```
# 3. zadanie: kalendár
# autor: Janko Hraško
# datum: 22.10.2021
```

V programe používaj len konštrukcie jazyka Python, ktoré sme sa učili na prvých 6 prednáškach. Nepoužívaj príkaz `import`.

Súbor `riesenie.py` odovzdaj na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **22. októbra**, kde ho môžeš nechať otestovať. Môžeš zaň získať **5 bodov**.