9. Zoznamy a n-tice (tuple)

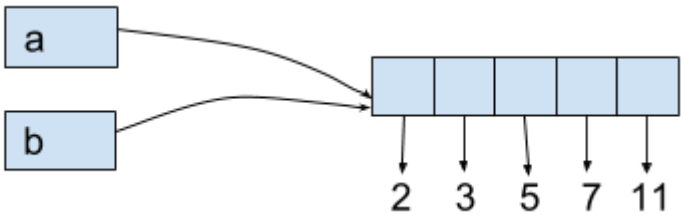
video prezentácia

zoznamy a n-tice

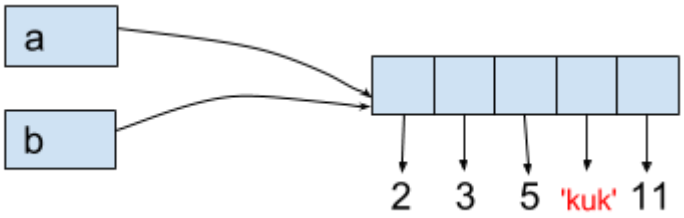
Pripomeňme si z minulej prednášky, ako dve premenné referencujú na ten istý zoznam. Priraďme:

```
>>> a = [2, 3, 5, 7, 11]
>>> b = a
>>> b[3] = 'kuk'
>>> a
    [2, 3, 5, 'kuk', 11]
```

Menili sme obsah premennej `b` (zmenili sme jej prvok s indexom `3`), ale tým sa zmenil aj obsah premennej `a`. Totiž obe premenné referencujú na ten istý zoznam:



Keď teraz meníme obsah premennej `b` pomocou **mutable operácií**, zmení sa aj obsah premennej `a`:



Toto samozrejme platí aj vo funkciách a aj s parametrami funkcií.

Zoznamy vo funkciách

Pozrime si takéto školské príklady, v ktorých dostávame chybné riešenie vďaka tomu, že si neuvedomíme, že dve premenné referencujú na ten istý zoznam.

Vytváranie dvoch zoznamov

Napíšeme funkciu, ktorá ako parameter dostáva zoznam čísel a jej úlohou je z tohto zoznamu vypísať dva zoznamy - zoznam záporných a zoznam nezáporných čísel; elegantným riešením by mohlo byť:

```
def vypis_dva_zoznamy(zoznam):
    zoz1 = zoz2 = []
    for prvok in zoznam:
        if prvok < 0:
            zoz1.append(prvok)
        else:
            zoz2.append(prvok)
    print('zaporne =', zoz1)
    print('nezaporne =', zoz2)
```

Prekvapením môže byť otestovanie tejto funkcie:

```
>>> vypis_dva_zoznamy([5, 7.3, 0, -3, 0.0, 1, -3.14])
    zaporne = [5, 7.3, 0, -3, 0.0, 1, -3.14]
    nezaporne = [5, 7.3, 0, -3, 0.0, 1, -3.14]
```

Všimnite si priradenie `zoz1 = zoz2 = []`. Týmto priradením obe premenné získali referenciu v pamäti na ten istý zoznam. Keďže ďalej s oboma premenným pracujeme len pomocou **mutable** operácií (metóda `append()`), stále sa uchovávajú referencie na ten istý zoznam. Riešením by mohlo byť, buď opraviť úvodnú inicializáciu premenných napríklad na `zoz1, zoz2 = [], []`, alebo používanie **immutable** operácií (namiesto `zoz1.append(prvok)` použijeme `zoz1 = zoz1 + [prvok]`).

Vyhadzovanie prvkov z kópie zoznamu

Ďalšia funkcia z daného reťazca vytvorí kópiu, z ktorej ale vynechá všetky hodnoty, ktoré sú znakovými reťazcami (ponechá napríklad čísla):

```
def zoznam_bez_retazcov(zoznam):
    kopia = zoznam
    for prvok in zoznam:
        if type(prvok) == str:
            kopia.remove(prvok)
    return kopia
```

Riešenie je dostatočne čitateľné. Otestujme:

```
>>> nejaky = [2, '+', 5, 'je', 7]
>>> zoznam_bez_retazcov(nejaky)
    [2, 5, 7]
>>> nejaky
    [2, 5, 7]
>>> nejaky = [1, 'prvy', 'druhy', 'treti', 'stvrty']
>>> zoznam_bez_retazcov(nejaky)
    [1, 'druhy', 'stvrty']
>>> nejaky
    [1, 'druhy', 'stvrty']
```

Prvý test dal dobrý výsledok, ale funkcia pokazila aj pôvodný zoznam. V druhom teste dokonca dostávame aj chybný výsledok aj pokazený vstupný zoznam. Aspoň trochu skúsenejší pythonista vidí problém v priradení `kopia = zoznam`. Premenná `kopia` referencuje ten istý zoznam ako je v parametri `zoznam`. Preto vyhadzovanie prvkov z premennej `kopia` ich bude vyhadzovať aj zo `zoznam`. Stačí opraviť úvodné priradenie napríklad na `kopia = list(zoznam)` a tým sa naozaj do `kopia` vyrobí nový zoznam. Teraz to už funguje, hoci skúsenejšiemu pythonistovi sa nemusí páčiť konštrukcia `kopia.remove(prvok)`. Táto metóda vyhľadá v zozname `kopia` daný prvok a jeho prvý výskyt vyhodí. Skôr by tu (namiesto vyhadzovania) zapísal skladanie nového reťazca pomocou `append()` ale už bez reťazcov, napríklad:

```
def zoznam_bez_retazcov(zoznam):
    novy = []
    for prvok in zoznam:
        if type(prvok) != str:
            novy.append(prvok)
    return novy
```

Využitie metódy pop pri prechádzaní zoznamu

Funkcia `vypis` má vypísať prvky zoznamu tak, aby v každom riadku (možno okrem posledného) bol presne zadaný počet prvkov; jedno zo študentských riešení vyzerá takto:

```
def vypis(zoznam, pocet):
    p = 0
    while zoznam:
        print(zoznam.pop(0), end=' ')
        p += 1
        if p % pocet == 0:
            print()
```

Hoci toto riešenie dáva správne výsledky, má jeden nepríjemný efekt: po výpise zoznamu zistíme, že obsah zoznamu sa zničil. Napríklad:

```
>>> a = list(range(5, 20, 2))
>>> a
    [5, 7, 9, 11, 13, 15, 17, 19]
>>> vypis(a, 3)
    5 7 9
    11 13 15
    17 19
>>> a
    []
```

Táto funkcia, hoci len vypisovala prvky zoznamu, tento zoznam vyčistila nielen vo vnútri funkcie (vďaka `zoznam.pop(0)`), ale tým aj premennú, ktorá tento zoznam referencovala pred volaním funkcie. **Mutable** operácia `pop` na parameter vždy zmení obsah tohto parametra.

Zisťovanie, či je zoznam utriedený

Ďalšia funkcia bude zisťovať, či je daný zoznam usporiadaný vzostupne, t.j. či pre každú dvojicu prvkov v zozname, ktoré sú vedľa seba platí, že prvý z nich nie je väčší ako druhý; jedným z riešení je postupne prechádzať celým zoznamom a kontrolovať túto podmienku:

```
def vzostupne(zoznam):
    for i in range(len(zoznam) - 1):
        if zoznam[i] > zoznam[i + 1]:
            return False
    return True
```

Toto je korektné riešenie, ale môžeme to vyriešiť aj trochu inak: ak mám funkciu, ktorá vie preusporiadať prvky v zozname tak, že budú vo vzostupnom poradí, potom stačí prekontrolovať, či pôvodný zoznam má prvky v rovnakom poradí ako po preusporiadaní (utriedení). Začiatočníci chcú často využiť metódu `sort` a niekedy urobia takúto chybu:

```
def vzostupne(zoznam):
    zoz1 = zoznam.sort()
    return zoz1 == zoznam
```

Tu zrejme predpokladajú, že metóda `sort` z pôvodného zoznamu vyrobí vzostupne usporiadanú postupnosť a tú potom celú porovnajú s pôvodným zoznamom. Táto funkcia ale vždy vráti `False`, lebo metóda `sort` nevyrába novú usporiadanú postupnosť, ale preusporiada v pamäti (**mutable**) pôvodný zoznam a nič nevracia - v skutočnosti vráti hodnotu `None`. Tá sa potom priradí do premennej `zoz1` a porovná s usporiadným `zoznam` (čo je zrejme `False`).

Niektorí začiatočníci to chcú opraviť takto:

```
def vzostupne(zoznam):
    zoz1 = zoznam
    zoz1.sort()
    return zoz1 == zoznam
```

Teda asi najprv si chcú vytvoriť kópiu pôvodného zoznamu (aby si ho nepokazili triedením pomocou `sort`) a až túto kópiu budú triediť metódou `sort`. Idea nie je zlá, ale realizácia je opäť chybná: priradením `zoz1 = zoznam` nevzniká kópia, ale iba ďalšia referencia na ten istý zoznam. Ďalšie volanie metódy `sort` usporiada pôvodný zoznam a na koniec skontroluje, či sa rovná samému sebe (teda to vždy bude `True`).

Už vieme, že kópia zoznamu sa najlepšie robí buď priradením `zoz1 = zoznam[:]` alebo `zoz1 = list(zoznam)`. Ak ale namiesto metódy `sort` využijeme iný variant triedenia, riešenie sa ešte zjednoduší. Triediť môžeme pomocou funkcie `sorted`, ktorá dostane ako parameter ľubovoľnú iterovateľnú postupnosť (napríklad zoznam, reťazec, …) a vráti **uzporiadaný zoznam**. Pôvodná postupnosť sa pritom nemení (je to **immutable**). Našu funkciu môžeme teraz zapísať takto:

```
def vzostupne(zoznam):
    zoz1 = sorted(zoznam)
    return zoz1 == zoznam
```

alebo ešte úspornejšie:

```
def vzostupne(zoznam):
```

```
    return sorted(zoznam) == zoznam
```

Vyprázdenie zoznamu

Nasledovná funkcia by mala vyčistiť obsah zadaného zoznamu:

```
def cisti(zoznam):
    zoznam = []
```

Samozrejme, že to nefunguje:

```
>>> ab = [1, 'dva', 3.14]
>>> cisti(ab)
>>> ab
    [1, 'dva', 3.14]
```

Volanie funkcie `cisti()` vytvorí lokálnu premennú `zoznam` a ten dostáva hodnotu skutočného parametra, teda referenciu na zoznam `[1, 'dva', 3.14]`. Lenže priradenie `zoznam = []` je **immutable** operácia, teda do lokálnej premennej `zoznam` priradíme novú referenciu na prázdny zoznam. Tým sa ale nezmení referencia pôvodnej premennej `ab`, ktorá bola skutočným parametrom volania funkcie `cisti()`. Riešením by asi bolo použitie nejakej **mutable** operácie, ktorá vyčistí obsah zoznamu, napríklad:

```
def cisti(zoznam):
    zoznam.clear()
```

Zapamätajte si, že **priradenie do premennej** pracuje vždy s **lokálnou** premennou - buď ju vytvorí (ak ešte neexistovala), alebo jej zmení hodnotu.

Podzoznamy v zoznamoch

Predpokladáme, že takéto príklady vás presvedčia, že pri práci so zoznamami vo funkciách musíte byť veľmi ostražití a uvedomovať si dôsledky referencií na zoznamy.

S referenciami môžeme mať problémy nielen v situáciách, keď dve premenné odkazujú na ten istý zoznam, ale aj v prípadoch, keď zoznam obsahuje nejaký podzoznam, t.j. referenciu na iný zoznam. Vyskúšajme:

```
>>> x = ['prvy', [2, 3], 'styri']
>>> y = x[1]
>>> y
    [2, 3]
>>> y.append('kuk')
>>> y
    [2, 3, 'kuk']
```

Zatiaľ to vyzerá dobre: do premennej `y` sme priradili prvok zoznamu `x` s indexom 1. Vidíme, že je to dvojprvkový zoznam `[2, 3]` a preto do neho na koniec pridáme nejaký reťazec

pomocou `y.append('kuk')`. Aj toto funguje dobre: v premennej `y` je teraz `[2, 3, 'kuk']`. Lenže teraz sa zmenil aj pôvodný zoznam `x`:

```
>>> x
    ['prvy', [2, 3, 'kuk'], 'styri']
```

Pekne tento efekt vidieť aj na ďalšom príklade:

```
>>> a = ['a']
>>> zoz = [a, a, a]
>>> zoz
    [['a'], ['a'], ['a']]
```

My už vieme, že zoznam `zoz` má tri prvky a všetky tri sú referenciami na ten istý **mutable** objekt. Ak sa tento zmení **mutable** operáciou, zmenia sa obsahy všetkých troch prvkov zoznamu:

```
>>> a.insert(0, 'b')
>>> a
    ['b', 'a']
>>> zoz
    [['b', 'a'], ['b', 'a'], ['b', 'a']]
```

Nielen začiatočníka prekvapí takáto chyba: plánujeme vytvoriť 100-prvkový zoznam, ktorý bude obsahovať len prázdne zoznamy `[]`. Potom budeme niektoré z týchto zoznamov meniť, napríklad:

```
>>> zoz = [[]] * 100
>>> zoz[3].append(1)
>>> zoz[7].insert(0, 0)
>>> zoz[9] += [2]
>>> zoz
    [[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
    [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
    [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
    [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
    [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
    ...]
```

Vidíme, že napriek tomu, že sme menili len niektoré tri prvky zoznamu `zoz`, malo to vplyv na všetkých 100 prvkov.

Pokračujme s týmto zoznamom. Zrejme platí:

```
>>> zoz[2] == zoz[3]
    True
>>> zoz[3] = [0, 1, 2]
>>> zoz[2] == zoz[3]
    True
```

Obsahy 2. a 3. prvkov sú rovnaké aj keď nemajú tú istú referenciu. Môžeme to vidieť, keď vymažeme obsah, napríklad 0. prvku:

```
>>> zoz[0].clear()
```

```
>>> zoz
    [[], [], [], [0, 1, 2], [], [], [], [], [], [], [], [], [], [], [], [], []
,
    [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []
,
    [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []
,
    [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []
,
    [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []
,
    [], []]
```

Všetky prvky s rovnakou referenciou sa vymazali okrem 3., ktorý mal rovnaký obsah, ale inú referenciu. V niektorých situáciách nám pomôže nová operácia `is`, pomocou ktorej vieme zistiť, či majú dva objekty identickú referenciu. Napríklad:

```
>>> zoz[1] = []
>>> zoz[0] == zoz[1]
    True
>>> zoz[0] is zoz[1]
    False
>>> zoz[0] is zoz[2]
    True
```

Zmenili sme 1. prvok na prázdny zoznam, preto platí `zoz[0] == zoz[1]`, ale sú to rôzne referencie, preto neplatí `zoz[0] is zoz[1]`.

Zoznamy a reťazce

V Pythone existuje niekoľko metód, v ktorých zoznamy spolupracujú so znakovými reťazcami. Ukážeme dve reťazcové metódy, ktoré budeme odteraz veľmi intenzívne využívať.

metóda split()

metóda `split()`

Keďže je to reťazcová metóda, má tvar:

```
reťazec.split()
```

Metóda rozbije daný reťazec na samostatné reťazce a uloží ich do zoznamu (teda vracia zoznam reťazcov). Predpokladáme, že tieto podreťazce sú navzájom oddelené „medzerovými“ znakmi (medzera, znak konca riadku, tabulátor). V helpe (napríklad `help(''.split)`) sa môžete dozvedieť ďalšie možnosti tejto funkcie.

Najlepšie to pochopíte na niekoľkých príkladoch. Metóda `split()` sa často využíva pri rozdelení prečítaného reťazca zo vstupu (`input()` alebo `subor.readline()`) na viac častí, napríklad:

```
>>> ret = input('zadaj 2 čísla: ')
    zadaj 2 čísla: 15 999
>>> zoz = ret.split()
>>> zoz
    ['15', '999']
>>> a, b = zoz
>>> ai, bi = int(zoz[0]), int(zoz[1])
>>> a, b, ai, bi
    ('15', '999', 15, 999)
```

Niekedy môžeme vidieť aj takýto zápis:

```
>>> meno, priezvisko = input('zadaj meno a priezvisko: ').split()
    zadaj meno a priezvisko: Janko Hraško
>>> meno
    'Janko'
>>> priezvisko
    'Hraško'
```

Metóda `split()` môže dostať ako parameter oddeľovač, napríklad ak sme prečítali čísla oddelené čiarkami:

```
sucet = 0
for prvok in input('zadaj čísla: ').split(','):
    sucet += int(prvok)
print('ich súčet je', sucet)

zadaj čísla: 10,20,30,40
ich súčet je 100
```

metóda join()

metóda `join()`

Opäť je to reťazcová metóda. Má tvar:

```
oddeľovač.join(zoznam)
```

Metóda zlepí všetky reťazce z daného **zoznamu reťazcov** do jedného, pričom ich navzájom oddelí uvedeným `oddeľovačom`, t. j. nejakým zadaným reťazcom. Ako zoznam môžeme uviesť ľubovoľnú postupnosť (iterovateľný objekt) reťazcov.

Ukážme to na príklade:

```
>>> zoz = ['prvý', 'druhý', 'tretí']
>>> zoz
    ['prvý', 'druhý', 'tretí']
>>> ''.join(zoz)
    'prvýdruhýtretí'
>>> '...'.join(zoz)
    'prvý...druhý...tretí'
>>> list(str(2021))
    ['2', '0', '2', '1']
>>> '.'.join(list(str(2021)))
```

```
    '2.0.2.1'
>>> '.'.join('Python')
    'P.y.t.h.o.n'
```

Preštudujte:

```
>>> veta = 'kto druhemu jamu kope'
>>> ' '.join(veta[::-1].split()[::-1])
    'otk umehurd umaj epok'
>>> ' '.join(input('? ').split()[::-1])
    ? viem ze neviem javu ale viem python
    'python viem ale javu neviem ze viem'
```

n-tice (tuple)

Sú to vlastne len nemeniteľné (**immutable**) zoznamy. Pythonovský typ `tuple` dokáže robiť skoro všetko to isté ako `list` okrem **mutable** operácií. Takže to najprv zhrňme a potom si ukážeme, ako to pracuje:

- operácia `+` na zreťazovanie (spájanie dvoch n-tíc)
- operácia `*` na viacnásobné zreťazovanie (viacnásobné spájanie jednej n-tice)
- operácia `in` na zisťovanie, či sa nejaká hodnota nachádza v n-tici
- operácia indexovania `[i]` na zistenie hodnoty prvku na zadanom indexe
- operácia rezu `[i:j:k]` na zistenie hodnoty nejakej podčasti n-tice
- relačné operácie `==`, `!=`, `<`, `<=`, `>`, `>=` na porovnávanie obsahu dvoch n-tíc
- metódy `count()` a `index()` na zisťovanie počtu výskytov, resp. indexu prvého výskytu
- prechádzanie n-tice pomocou for-cyklu (iterovanie)
- štandardné funkcie `len()`, `sum()`, `min()`, `max()` ktoré zisťujú niečo o prvkoch n-tice

Takže **n-tice**, tak ako aj zoznamy sú **štruktúrované typy**, t.j. sú to typy, ktoré obsahujú hodnoty nejakých (možno rôznych) typov (sú to tzv. **kolekcie**):

- konštanty typu n-tica uzatvárame do okrúhlych zátvoriek a navzájom oddeľujeme čiarkami
- funkcia `len()` vráti počet prvkov n-tice, napríklad:

```
>>> stred = (150, 100)
>>> zviera = ('slon', 2013, 'gray')
>>> print(stred)
    (150, 100)
>>> print(zviera)
    ('slon', 2013, 'gray')
>>> nic = ()
>>> print(nic)
    ()
>>> len(stred)
    2
>>> len(zviera)
    3
>>> len(nic)
    0
```

```
>>> type(stred)
    <class 'tuple'>
```

Vidíte, že n-tica môže byť aj prázdna, označujeme ju `()` a vtedy má počet prvkov 0 (teda `len(())` je 0). Ak n-tica nie je prázdna, hodnoty sú navzájom oddelené čiarkami.

n-tica s jednou hodnotou

n-ticu s jednou hodnotou **nemôžeme** zapísať takto:

```
>>> p = (12)
>>> print(p)
    12
>>> type(p)
    <class 'int'>
```

Ak zapíšeme ľubovoľnú hodnotu do zátvoriek, nie je to n-tica (v našom prípade je to len jedno celé číslo). Pre jednoprvkovú n-ticu musíme do zátvoriek zapísať aj čiarku:

```
>>> p = (12,)
>>> print(p)
    (12,)
>>> len(p)
    1
>>> type(p)
    <class 'tuple'>
```

Pre Python sú dôležitejšie čiarky ako zátvorky. V mnohých prípadoch si Python zátvorky domyslí (čiarky si nedomyslí nikdy):

```
>>> stred = 150, 100
>>> zviera = 'slon', 2013, 'gray'
>>> p = 12,
>>> print(stred)
    (150, 100)
>>> print(zviera)
    ('slon', 2013, 'gray')
>>> print(p)
    (12,)
```

Uvedomte si rozdiel medzi týmito dvoma priradeniami:

```
>>> a = 3.14
>>> b = 3,14
>>> print(a, type(a))
    3.14 <class 'float'>
>>> print(b, type(b))
    (3, 14) <class 'tuple'>
```

Operácie s n-ticami

Operácie fungujú presne rovnako ako fungovali s reťazcami a zoznamami:

- operácia `+` **zreťazí** dve n-tice, to znamená, že **vyrobí novú n-ticu**, ktorá obsahuje najprv všetky prvky prvej n-tice a za tým všetky prvky druhej n-tice; zrejme oba operandy musia byť n-tice: nemôžeme zreťazovať n-ticu s hodnotou iného typu
- operácia `*` zadané číslo-krát zreťazí jednu n-ticu, to znamená, že **vyrobí novú n-ticu**, ktorá požadovaný-krát obsahuje všetky prvky zadanej n-tice; operácia viacnásobného zreťazovania má jeden operand typu n-tica a druhý musí byť celé číslo
- operácia `in` vie zistiť, či sa nejaký prvok nachádza v n-tici

Napríklad:

```
>>> stred = (150, 100)
>>> zviera = ('slon', 2013, 'gray')
>>> stred + zviera
    (150, 100, 'slon', 2013, 'gray')
>>> zviera + stred
    ('slon', 2013, 'gray', 150, 100)
>>> stred * 5
    (150, 100, 150, 100, 150, 100, 150, 100, 150, 100)
>>> 50 * (1)
    50
>>> 50 * (1,)
    (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
>>> 50 * (1, 2)
    (1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
    2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
    1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
    2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

n-tica môže obsahovať ako svoje prvky aj iné n-tice:

```
>>> stred = (150, 100)
>>> p = ('stred', stred)
>>> p
    ('stred', (150, 100))
>>> len(p)
    2
>>> p = (stred, stred, stred, stred)
>>> p
    ((150, 100), (150, 100), (150, 100), (150, 100))
>>> 4 * (stred,)
    ((150, 100), (150, 100), (150, 100), (150, 100))
>>> 4 * (stred)                                    # čo je to isté ako 4 * stred
    (150, 100, 150, 100, 150, 100, 150, 100)
>>> 4 * stred,
    ((150, 100, 150, 100, 150, 100, 150, 100),)
```

Operátorom `in` vieme zistiť, či sa nejaká hodnota nachádza v n-tici ako jeden jeho prvok. Takže:

```
>>> p = (stred, stred, stred, stred)
>>> p
    ((150, 100), (150, 100), (150, 100), (150, 100))
>>> stred in p
    True
```

```
>>> 150 in p
    False
>>> 150 in stred
    True
>>> zviera = ('slon', 2013, 'gray')
>>> 2013 in zviera
    True
>>> (2013, 'gray') in zviera
    False
```

Funkcia tuple()

Vyrobí **n-ticu** z ľubovoľnej postupnosti (z iterovateľného objektu, t.j. takého objektu, ktorý sa dá prechádzať for-cyklom), napríklad zo znakového reťazca, zo zoznamu, vygenerovanej postupnosti celých čísel pomocou `range()`, ale iterovateľný je aj otvorený textový súbor. Znakový reťazec funkcia `tuple()` rozoberie na znaky:

```
>>> tuple('Python')
    ('P', 'y', 't', 'h', 'o', 'n')
```

Vytvorenie n-tice pomocou `range()`:

```
>>> tuple(range(1, 16))
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
>>> a = tuple(range(1000000))
>>> len(a)
    1000000
```

Podobne môžeme skonštruovať n-ticu z textového súboru. Predpokladajme, že súbor obsahuje tieto 4 riadky:

```
prvy
druhy
treti
stvrty
```

potom

```
>>> with open('abc.txt') as t:
        obsah = tuple(t)
>>> obsah
    ('prvy\n', 'druhy\n', 'treti\n', 'stvrty\n')
```

Riadky súboru sa postupne stanú prvkami n-tice.

for-cyklus s n-ticami

for-cyklus je programová konštrukcia, ktorá postupne prechádza všetky prvky nejakého **iterovateľného** objektu. Doteraz sme sa stretli s iterovaním pomocou

funkcie `range()`, prechádzaním prvkov reťazca `str` a zoznamu `list`, aj celých riadkov textového súboru. Ale už od 2. prednášky sme používali aj takýto zápis:

```
for i in 2, 3, 5, 7, 11, 13:
    print('prvocislo', i)
```

V tomto zápise už teraz vidíme n-ticu (`tuple`): `2, 3, 5, 7, 11, 13`. Len sme tomu nemuseli dávať zátvorky. Keďže aj n-tica je iterovateľný objekt, Môžeme ju používať vo for-cykle rovnako ako iné iterovateľné typy. To isté, ako predchádzajúci príklad, by sme zapísali napríklad aj takto:

```
cisla = (2, 3, 5, 7, 11, 13)
for i in cisla:
    print('prvocislo', i)
```

Keďže teraz už vieme manipulovať s n-ticami, môžeme zapísať napríklad:

```
>>> rozne = ('retazec', (100, 200), 3.14, len)
>>> for prvok in rozne:
        print(prvok, type(prvok))
    retazec <class 'str'>
    (100, 200) <class 'tuple'>
    3.14 <class 'float'>
    <built-in function len> <class 'builtin_function_or_method'>
```

Tu vidíme, že prvkami n-tice môžu byť najrôznejšie objekty, hoci aj funkcie (tu je to štandardná funkcia `len`).

Pomocou operácií s n-ticami vieme zapísať aj zaujímavejšie postupnosti čísel, napríklad:

```
>>> for i in 10 * (1,):
        print(i, end=' ')
    1 1 1 1 1 1 1 1 1 1
>>> for i in 10 * (1, 2):
         print(i, end=' ')
    1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
>>> for i in 10 * tuple(range(10)):
         print(i, end=' ')
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
    3 4 5 6 7 8 9
>>> for z in 'Python':
        print(z, end=' ')
    P y t h o n
>>> for z in 'Python',:
        print(z, end=' ')
    Python
```

Ďalej pomocou for-cyklu vieme n-tice skladať podobne, ako sme to robili so zoznamami. V nasledovnom príklade vytvoríme n-ticu zo všetkých deliteľov nejakého čísla:

```
cislo = int(input('zadaj cislo: '))
delitele = ()
for i in range(1, cislo+1):
```

```
    if cislo % i == 0:
        delitele = delitele + (i,)
print('delitele', cislo, 'su', delitele)
```

po spustení:

```
zadaj cislo: 124
delitele 124 su (1, 2, 4, 31, 62, 124)
```

Všimnite si, ako sme pridali jeden prvok na koniec n-tice: `delitele = delitele + (i,)`. Museli, sme vytvoriť jednoprvkovú n-ticu `(i,)` a tú sme zreťazili s pôvodnou n-ticou `delitele`. Mohli sme to zapísať aj takto: `delitele += (i,)`. Zrejme, keby sme toto riešili pomocou zoznamov, použili by sme metódu `append()`.

Funkcia enumerate()

Už poznáme použitie štandardnej funkcie `enumerate()`. Vieme ju využiť aj pre n-tice, napríklad:

```
zoz = (2, 3, 5, 7, 9, 11, 13, 17, 19)
for ix, pr in enumerate(zoz):
    print(f'{ix}. prvocislo je {pr}')
```

```
0. prvocislo je 2
1. prvocislo je 3
2. prvocislo je 5
3. prvocislo je 7
4. prvocislo je 9
5. prvocislo je 11
6. prvocislo je 13
7. prvocislo je 17
8. prvocislo je 19
```

Funkcia `enumerate()` v skutočnosti z jednej ľubovoľnej postupnosti vygeneruje postupnosť dvojíc (ktoré sú už typu `tuple`). Túto postupnosť ďalej preliezame for-cyklom. Mohli by sme to zapísať aj takto:

```
zoz = (2, 3, 5, 7, 9, 11, 13, 17, 19)
for dvojica in enumerate(zoz):
    ix, pr = dvojica
    print(f'{ix}. prvocislo je {pr} ... dvojica = {dvojica}')
```

po spustení:

```
0. prvocislo je 2 ... dvojica = (0, 2)
1. prvocislo je 3 ... dvojica = (1, 3)
2. prvocislo je 5 ... dvojica = (2, 5)
3. prvocislo je 7 ... dvojica = (3, 7)
4. prvocislo je 9 ... dvojica = (4, 9)
5. prvocislo je 11 ... dvojica = (5, 11)
6. prvocislo je 13 ... dvojica = (6, 13)
7. prvocislo je 17 ... dvojica = (7, 17)
```

```
8. prvocislo je 19 ... dvojica = (8, 19)
```

Ak by sme skúmali výsledok z tejto funkcie, dozvedeli by sme sa:

```
>>> enumerate(zoz)
    <enumerate object at 0x02F59B48>
>>> list(enumerate(zoz))
    [(0, 2), (1, 3), (2, 5), (3, 7), (4, 9), (5, 11), (6, 13), (7, 17), (8, 19)]
>>> tuple(enumerate(zoz))
    ((0, 2), (1, 3), (2, 5), (3, 7), (4, 9), (5, 11), (6, 13), (7, 17), (8, 19))
```

čo sú naozaj postupnosti skutočných dvojíc.

Indexovanie

n-tice indexujeme rovnako ako sme indexovali zoznamy:

- prvky postupnosti môžeme indexovať v `[]` zátvorkách, pričom index musí byť od 0 až po počet prvkov-1
- pomocou rezu (slice) vieme indexovať časť n-tice (niečo ako podreťazec) tak, že `[]` zátvoriek zapíšeme aj dvojbodku:
    - `ntica[od:do]` n-tica z prvkov s indexmi `od` až po `do-1`
    - `ntica[:do]` n-tica z prvkov od začiatku až po prvok s indexom `do-1`
    - `ntica[od:]` n-tica z prvkov s indexmi `od` až po koniec n-tice
    - `ntica[od:do:krok]` n-tica z prvkov s indexmi `od` až po `do-1`, pričom berieme každý `krok` prvok

Niekoľko príkladov:

```
>>> prvo = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
>>> prvo[2]
    5
>>> prvo[2:5]
    (5, 7, 11)
>>> prvo[4:5]
    (11,)
>>> prvo[:5]
    (2, 3, 5, 7, 11)
>>> prvo[5:]
    (13, 17, 19, 23, 29)
>>> prvo[::2]
    (2, 5, 11, 17, 23)
>>> prvo[::-1]
    (29, 23, 19, 17, 13, 11, 7, 5, 3, 2)
```

Vidíme, že s n-ticami pracujeme veľmi podobne ako sme pracovali so znakovými reťazcami a so zoznamami. Keď sme ale do znakového reťazca chceli pridať jeden znak (alebo aj viac), museli sme to robiť rozoberaním a potom skladaním:

```
>>> prvo = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
>>> prvo = prvo[:5] + ('fuj',) + prvo[5:]
>>> prvo
    (2, 3, 5, 7, 11, 'fuj', 13, 17, 19, 23, 29)
```

Pred 5-ty prvok vloží nejaký znakový reťazec.

Rovnako nemôžeme zmeniť ani hodnotu nejakého znaku reťazca obyčajným priradením:

```
>>> ret = 'Python'
>>> ret[2] = 'X'
    ...
    TypeError: 'str' object does not support item assignment
>>> ret = ret[:2] + 'X' + ret[3:]
>>> ret
    'PyXhon'
>>> ntica = (2, 3, 5, 7, 11, 13)
>>> ntica[2] = 'haha'
    ...
    TypeError: 'tuple' object does not support item assignment
>>> ntica = ntica[:2] + ('haha',) + ntica[3:]
>>> ntica
    (2, 3, 'haha', 7, 11, 13)
```

Všimnite si, že Python vyhlásil rovnakú chybu pre `tuple` ako pre `str`. Hovoríme, že ani reťazce ani n-tice nie sú meniteľné (teda sú **immutable**).

Porovnávanie n-tíc

Porovnávanie n-tíc je veľmi podobné ako porovnávanie reťazcov a zoznamov. Pripomeňme si, ako je to pri zoznamoch:

- postupne porovnáva i-te prvky oboch zoznamov, kým sú rovnaké; pri prvej nerovnosti je výsledkom porovnanie týchto dvoch hodnôt
- ak je pri prvej nezhode v prvom zozname menšia hodnota ako v druhom, tak prvý zoznam je menší ako druhý
- ak je prvý zoznam kratší ako druhý a zodpovedajúce prvky sa zhodujú, tak prvý zoznam je menší ako druhý

Hovoríme tomu **lexikografické** porovnávanie.

Teda aj pri porovnávaní n-tíc sa budú postupne porovnávať zodpovedajúce si prvky a pri prvej nerovnosti sa skontroluje, ktorý z týchto prvkov je menší. Treba tu ale dodržiavať jedno veľmi dôležité pravidlo: porovnávať hodnoty napríklad na menší môžeme len vtedy, keď sú zhodných typu:

```
>>> 5 < 'a'
    ...
    TypeError: unorderable types: int() < str()
>>> (1, 5, 10) < (1, 'a', 10)
    ...
    TypeError: unorderable types: int() < str()
>>> (1, 5, 10) != (1, 'a', 10)
    True
```

Najlepšie je porovnávať také n-tice, ktoré majú prvky rovnakého typu. Pri n-ticiach, ktoré majú zmiešané typy si musíme dávať väčší pozor:

```
>>> ('Janko', 'Hrasko', 'Zilina') < ('Janko', 'Jesensky', 'Martin')
```

```
    True
>>> (1, 2, 3, 4, 5, 5, 6, 7, 8) < tuple(range(1, 9))
    True
>>> ('Janko', 'Hrasko', 2008) < ('Janko', 'Hrasko', 2007)
    False
```

Viacnásobné priradenie

Najprv pripomeňme, ako funguje viacnásobné priradenie: ak je pred znakom priradenia `=` viac premenných, ktoré sú oddelené čiarkami, tak za znakom priradenia musí byť iterovateľný objekt, ktorý má presne toľko hodnôt, ako počet premenných. Iterovateľným objektom môže byť zoznam (`list`), n-tica (`tuple`), znakový reťazec (`str`), generovaná postupnosť čísel (`range()`) ale aj otvorený textový súbor (`open()`), ktorý má presne toľko riadkov, koľko je premenných v priradení.

Ak do jednej premennej priraďujeme viac hodnôt oddelených čiarkou, Python to chápe ako priradenie n-tice. Pozrite nasledovné priradenia.

Priradíme n-ticu:

```
>>> a1, a2, a3, a4 = 3.14, 'joj', len, (1, 3, 5)
>>> print(a1, a2, a3, a4)
    3.14 joj <built-in function len> (1, 3, 5)
>>> a, b, c, d, e, f = 3 * (5, 7)
>>> print(a, b, c, d, e, f)
    5 7 5 7 5 7
```

Priradíme vygenerovanú postupnosť 4 čísel:

```
>>> a, b, c, d = range(2, 6)
>>> print(a, b, c, d)
    2 3 4 5
```

Priradíme znakový reťazec:

```
>>> d, e, f, g, h, i = 'Python'
>>> print(d, e, f, g, h, i)
    P y t h o n
```

Priradíme riadky textového súboru:

```
>>> with open('dva.txt', 'w') as f:
        f.write('first\nsecond\n')
>>> with open('dva.txt') as subor:
        prvy, druhy = subor
>>> prvy
    'first\n'
>>> druhy
    'second\n'
```

Vieme zapísať aj takto:

```
>>> prvy, druhy = open('dva.txt')
```

Tento posledný príklad je veľmi umelý a v praxi sa asi priamo do premenných takto čítať nebude.

Viacnásobné priradenie používame napríklad aj na výmenu obsahu dvoch (aj viac) premenných:

```
>>> x, y, z = y, z, x
```

Aj v tomto príklade je na pravej strane priradenia (za `=`) n-tica: `(y, z, x)`.

n-tica ako návratová hodnota funkcie

V Pythone sa dosť využíva to, že návratovou hodnotou funkcie môže byť n-tica, t.j. výsledkom funkcie je naraz niekoľko návratových hodnôt. Napríklad nasledovný príklad počíta celočíselné delenie a súčasne zvyšok po delení:

```
def zisti(a, b):
    return a // b, a % b
```

Funkciu môžeme použiť napríklad takto:

```
>>> podiel, zvysok = zisti(153, 33)
>>> print('podiel =', podiel, 'zvysok =', zvysok)
    podiel = 4 zvysok = 21
```

Ak z výsledku takejto funkcie potrebujeme použiť len jednu z hodnôt, môžeme zapísať:

```
>>> print('zvysok =', zisti(153, 33)[1])
```

Ďalšia funkcia vráti postupnosť všetkých deliteľov nejakého čísla:

```
def delitele(cislo):
    vysl = ()
    for i in range(1, cislo+1):
        if cislo % i == 0:
            vysl = vysl + (i,)
    return vysl
```

Otestujeme:

```
>>> deli = delitele(24)
>>> print(deli)
    (1, 2, 3, 4, 6, 8, 12, 24)
>>> if 2 in deli:
        print('parne')
    parne
>>> print('sucet delitelov =', sum(deli))
    sucet delitelov = 60
```

```
>>> print('je prvocislo =', len(delitele(int(input('zadaj cislo: '))))==2)
    zadaj cislo: 11213
    je prvocislo = True
>>> print('je prvocislo =', len(delitele(int(input('zadaj cislo: '))))==2)
    zadaj cislo: 1001
    je prvocislo = False
```

Príklad ukazuje, že keď je výsledkom n-tica, môžeme ju ďalej rôzne spracovať alebo testovať.

Ďalšie funkcie a metódy

S n-ticami vedia pracovať nasledovné štandardné funkcie:

- `len(ntica)` - vráti počet prvkov n-tice
- `sum(ntica)` - vypočíta súčet prvkov n-tice (všetky musia byť čísla)
- `min(ntica)` - zistí najmenší prvok n-tice (prvky sa musia dať navzájom porovnať, nemôžeme tu miešať rôzne typy)
- `max(ntica)` - zistí najväčší prvok n-tice (ako pri `min()` ani tu sa nemôžu typy prvkov miešať)

Na rozdiel od zoznamov a znakových reťazcov, ktoré majú veľké množstvo metód, n-tice majú len dve:

- `ntica.count(hodnota)` - zistí počet výskytov nejakej hodnoty v n-tici
- `ntica.index(hodnota)` - vráti index (poradie) v n-tici prvého (najľavejšieho) výskytu danej hodnoty, ak sa hodnota v n-tici nenachádza, metóda spôsobí spadnutie na chybe (`ValueError: tuple.index(x): x not in tuple`)

Ukážme tieto funkcie na malom príklade. V n-tici uložíme niekoľko nameraných teplôt a potom vypíšeme priemernú, minimálnu aj maximálnu teplotu:

```
>>> teploty = (14, 22, 19.5, 17.1, 20, 20.4, 18)
>>> print('počet nameraných teplôt: ', len(teploty))
    počet nameraných teplôt:  7
>>> print('minimálna teplota: ', min(teploty))
    minimálna teplota:  14
>>> print('maximálna teplota: ', max(teploty))
    maximálna teplota:  22
>>> print('priemerná teplota: ', round(sum(teploty) / len(teploty), 2))
    priemerná teplota:  18.71
```

Ďalej môžeme zistiť, kedy bola nameraná konkrétna hodnota:

```
>>> teploty.index(20)
    4
>>> teploty.index(20.1)
    ...
    ValueError: tuple.index(x): x not in tuple
```

Môžeme zistiť, koľko-krát sa nejaká konkrétna teplota vyskytla v našich meraniach:

```
>>> teploty.count(20)
    1
>>> teploty.count(20.1)
```

```
    0
```

n-tice a grafika

Nasledovný príklad predvedie použitie n-tíc v grafickom režime. Zadefinujeme niekoľko bodov v rovine a potom pomocou nich kreslíme nejaké farebné polygóny. Začnime takto:

```
a = (70, 150)
b = (200, 200)
c = (150, 250)
d = (120, 70)
e = (50, 220)

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

canvas.create_polygon(a, b, c, d, fill='red')
```

Ak by sme chceli jedným priradením priradiť dva body do premenných `a` aj `b`, zapíšeme:

```
a, b = (100, 150), (180, 200)
```

čo je vlastne:

```
a, b = ((100, 150), (180, 200))
```

Polygónov môžeme nakresliť aj viac (zrejme väčšinou záleží na poradí ich kreslenia):

```
canvas.create_polygon(e, a, c, fill='green')
canvas.create_polygon(e, d, b, fill='yellow')
canvas.create_polygon(a, b, c, d, fill='red')
canvas.create_polygon(a, c, d, b, fill='blue')
```

Vidíme, že niektoré postupnosti bodov tvoria jednotlivé útvary, preto zapíšme:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

canvas.create_polygon(utvar1, fill='green')
canvas.create_polygon(utvar2, fill='yellow')
canvas.create_polygon(utvar3, fill='red')
canvas.create_polygon(utvar4, fill='blue')
```

Volanie funkcie `canvas.create_polygon()` sa tu vyskytuje 4-krát, ale s rôznymi parametrami. Prepíšme to do for-cyklu:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
```

```
utvar4 = a, c, d, b

for param in (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'), (utvar4, 'blu
e'):
    utvar, farba = param
    canvas.create_polygon(utvar, fill=farba)
```

Dostávame to isté. Vo for-cykle sa najprv do premennej `param` priradí dvojica s dvoma prvkami: útvar a farba, a v tele cyklu sa táto premenná s dvojicou priradí do dvoch premenných `utvar` a `farba`. Potom sa zavolá funkcia `canvas.create_polygon()` s týmito parametrami.

Už sme videli aj predtým, že pre for-cyklus existuje vylepšenie: ak sa do premennej cyklu postupne priraďujú nejaké dvojice hodnôt a tieto by sa na začiatku tela rozdelili do dvoch premenných, môžeme priamo tieto dve premenné použiť ako premenné cyklu (ako keby viacnásobné priradenie). Podobnú ideu sme mohli vidieť pri použití `enumerate()`. Predchádzajúci príklad prepíšeme:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

for utvar, farba in (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'),(utvar4
, 'blue'):
    canvas.create_polygon(utvar, fill=farba)
```

Pozrime sa na n-ticu, ktorá sa prechádza týmto for-cyklom:

```
>>> cyklus = (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'), (utvar4, 'blu
e')
>>> cyklus
    ((((50, 220), (70, 150), (150, 250)), 'green') (((50, 220), (120, 70),
    (200, 200)), 'yellow') (((70, 150), (200, 200), (150, 250), (120, 70)),
    'red') (((70, 150), (150, 250), (120, 70), (200, 200)), 'blue'))
```

Vidíme, že n-tica v takomto tvare je dosť ťažko čitateľná, ale for-cyklus jej normálne rozumie. Zrejme by sme teraz mohli zapísať:

```
for utvar, farba in cyklus:
   canvas.create_polygon(utvar, fill=farba)
```

Zoznamy a grafika

Už vieme, že väčšina grafických príkazov, napríklad `create_line()`, `create_polygon()`, … akceptujú ako parametre nielen čísla, ale aj n-tice alebo aj zoznamy čísel, resp. zoznamy/dvojice čísel, napríklad:

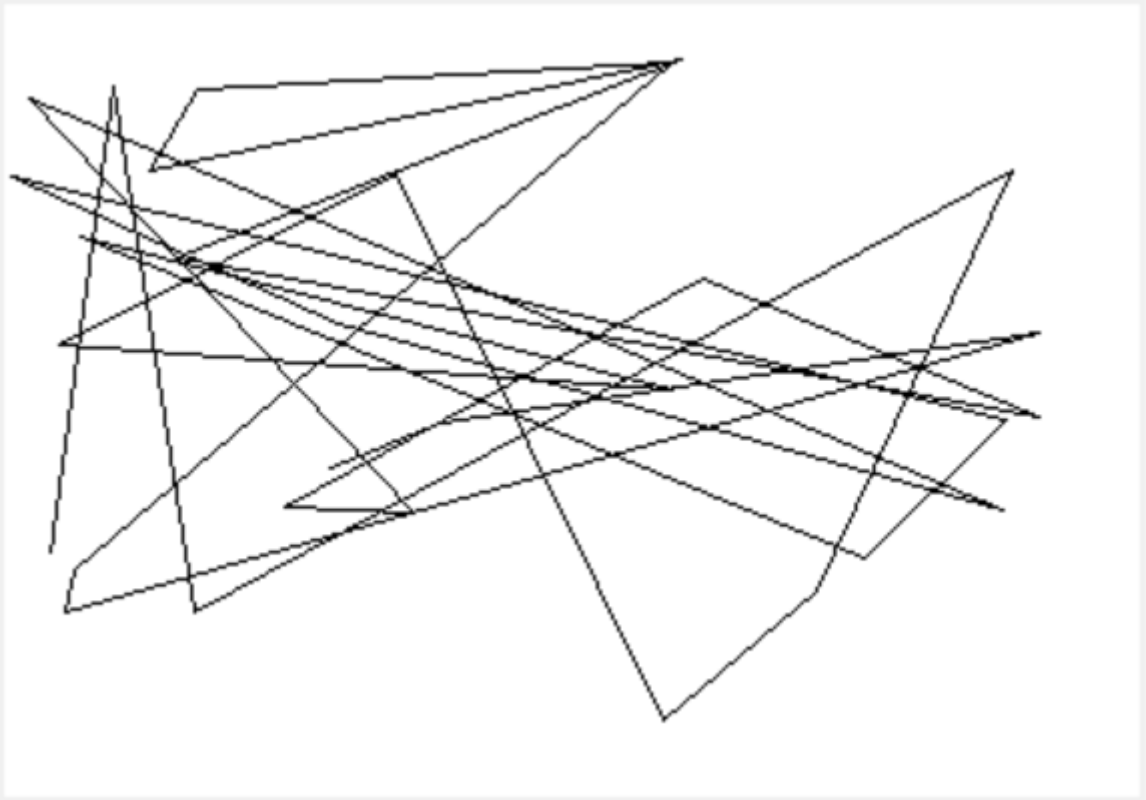
```
import tkinter

canvas = tkinter.Canvas()
```

```
canvas.pack()

utvar = ((100, 50), (200, 120))
canvas.create_rectangle(utvar, fill='blue')
canvas.create_oval(utvar, fill='yellow')
utvar2 = list(utvar)                # z n-tice sa vyrobí zoznam
utvar2.append((170, 20))
canvas.create_polygon(utvar2, fill='red')

tkinter.mainloop()
```

alebo môžeme generovať náhodnú krivku:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white')
canvas.pack()

krivka = []
for i in range(30):
    krivka.append((random.randrange(350), random.randrange(250)))
canvas.create_line(krivka)

tkinter.mainloop()
```

Po spustení dostaneme spojených 30 náhodných bodov v rovine:



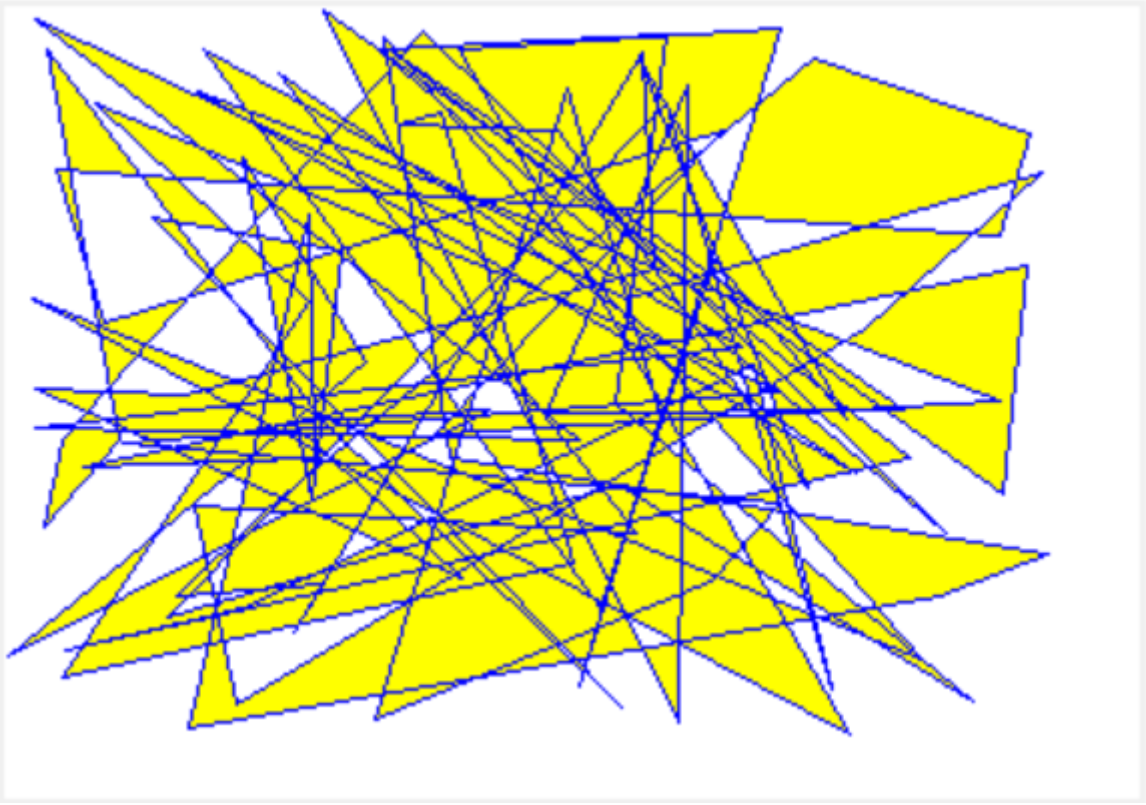
Ak by sme chceli využiť grafickú funkciu `coords()`, ktorá modifikuje súradnice nakreslenej krivky, nemôžeme jej poslať zoznam súradníc (dvojíc čísel x a y), ale táto vyžaduje plochý zoznam (alebo n-ticu) čísel. Predchádzajúci príklad mierne zmeníme:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white')
canvas.pack()

poly = canvas.create_polygon(0, 0, 0, 0, fill='yellow', outline='blue')
krivka = []
for i in range(100):
    bod = [random.randrange(350), random.randrange(250)]
    krivka.extend(bod)     # to isté ako    krivka += bod
    canvas.coords(poly, krivka)
    canvas.update()
    canvas.after(300)

tkinter.mainloop()
```

Použili sme tu dvojicu nových príkazov pre grafickú plochu `canvas`:

```
canvas.update()
canvas.after(300)
```

Po spustení sa postupne vygeneruje 100 náhodných bodov, ktoré budú tvoriť obrys polygónu:



Tieto dva príkazy označujú, že sa grafická plocha prekreslí (aktualizuje) pomocou metódy `update()` a potom sa na **300** milisekúnd (0.3 sekundy) pozdrží výpočet pomocou metódy `after()`. Vďaka tejto dvojici príkazov budeme vidieť postupné pridávanie náhodných bodov vytváranej krivky.

Okrem toho sme tu využili novú metódu `extend()`, ktorá k existujúcemu zoznamu prilepí na koniec nový zoznam. Napríklad:

```
>>> a = [1, 2, 3, 4]
>>> b = ['xx', 'yy']
>>> a.extend(b)
>>> a
    [1, 2, 3, 4, 'xx', 'yy']
```

Zrejme to isté by sme dosiahli aj takto:

```
>>> a = [1, 2, 3, 4]
>>> b = ['xx', 'yy']
>>> a += b
>>> a
    [1, 2, 3, 4, 'xx', 'yy']
```

Cvičenia

L.I.S.T.

- riešenia **aspoň 12 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- používaj len konštrukcie z doterajších prednášok
- pozri si Riešenie úloh 9. cvičenia

1. Napíš funkciu `posun(zoznam)`, ktorá posunie prvky v danom zozname tak, že prvý sa presťahuje na koniec. Funkcia nič nevracia, len zmení obsah pôvodného zoznamu. Napríklad:

```
>>> a = [2, 3, '5', 7, 11]
>>> posun(a)
>>> a
    [3, '5', 7, 11, 2]
>>> zoz = 'kto druhemu jamu kope'.split()
>>> for i in range(5):
        print(zoz)
        posun(zoz)
    ['kto', 'druhemu', 'jamu', 'kope']
    ['druhemu', 'jamu', 'kope', 'kto']
    ['jamu', 'kope', 'kto', 'druhemu']
    ['kope', 'kto', 'druhemu', 'jamu']
    ['kto', 'druhemu', 'jamu', 'kope']
```

2. Napíš funkciu `vyhod_none(ntica)`, ktorá z danej n-tice vyhodí všetky výskyty `None`. Funkcia vráti (`return`) túto novú n-ticu (nič nevypisuje). Napríklad:

```
>>> vyhod_none((None, 1, None, None))
    (1,)
```

3. Napíš funkciu `dve_kocky(pocet)`, ktorá vráti 13-prvkový zoznam celých čísel. Tento sa skonštruuje takto: zadaný `pocet`-krát hodí dvoma kockami (dve nádodné čísla od 1 do 6), pre každý takýto hod urobí ich súčet a v 13-prvkovom zozname si eviduje počet výskytov každého súčtu, napríklad `zoznam[5]` bude označovať, koľko krát pri našej simulácii padol súčet 5; zrejme na začiatku budú v zozname samé 0 a potom pri každom hode sa číslo na príslušnom indexe zvýši o 1. Funkcia nič nevypisuje, len vráti vytvorený 13-prvkový zoznam celých čísel. Mohol by si dostať napríklad, takýto zoznam:

```
>>> dve_kocky(1000)
    [0, 0, 29, 36, 90, 105, 137, 181, 125, 126, 93, 50, 28]
```

4. Napíš funkciu `desiatkova(cislo)`, ktorá vráti (`return`) zoznam cifier daného čísla v desiatkovej sústave. Využi funkcie `str` a `int`. Napríklad:

```
>>> desiatkova(11213)
    [1, 1, 2, 1, 3]
```

Všimni si, že takto sa dá vypočítať ciferný súčet čísla:

```
>>> sum(desiatkova(11213))
    8
```

5. Napíš funkciu `dvojkova(cislo)`, ktorá vráti (`return`) zoznam cifier daného čísla v dvojkovej sústave. Využi `f'{cislo:b}'`. Napríklad:

```
>>> dvojkova(11213)
    [1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1]
```

6. Napíš funkciu `z_dvojkovej(zoznam)`, ktorá dostane zoznam cifier dvojkového zápisu čísla v tvare z predchádzajúcej úlohy. Funkcia vráti celé číslo (`return`), ktorého cifry v dvojkovej sústave zodpovedajú zadanému zoznamu. Napríklad:

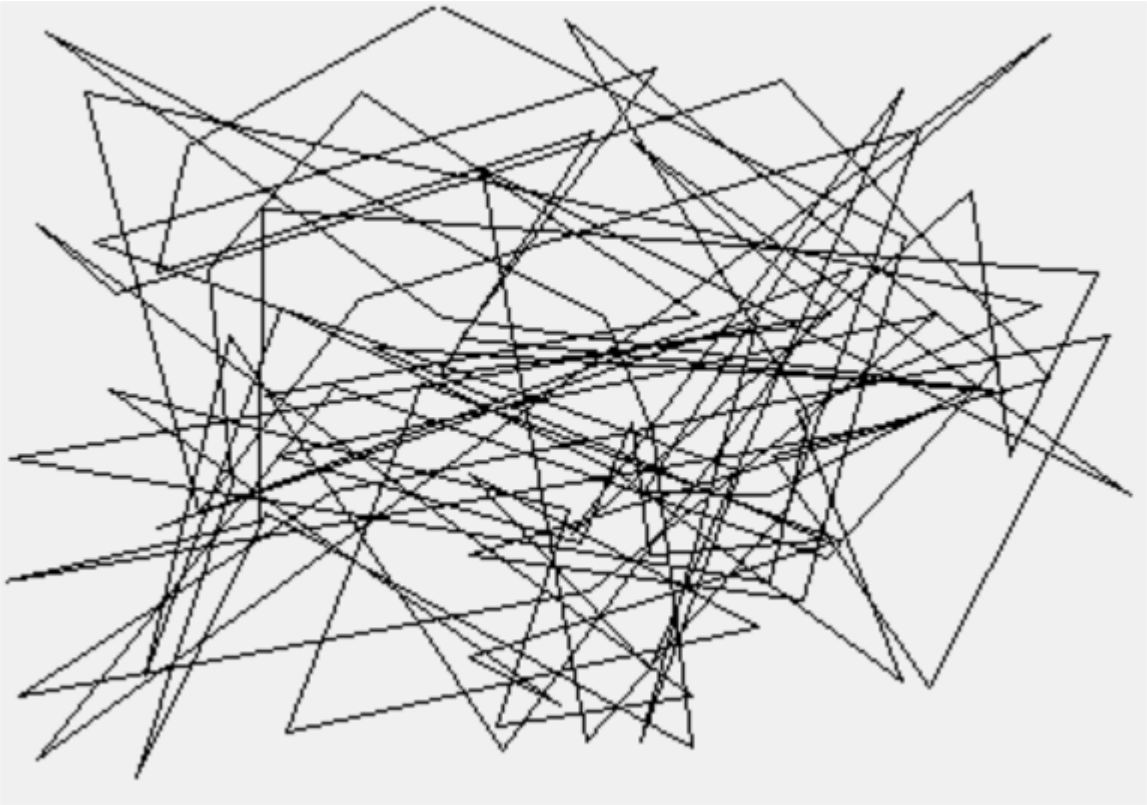
```
>>> z_dvojkovej([1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1])
    11213
>>> z = [1] + [0] * 20
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> z_dvojkovej(z)
    1048576
>>> 2 ** 20
    1048576
```

Využi funkciu `int(reťazec, 2)`, pomocou ktorej sa zo znakového reťazca s dvojkovým zápisom čísla vyrobí celé číslo.

7. Napíš funkciu `nahodne_body(pocet)`, ktorá vráti zadaný počet náhodných bodov v grafickej ploche (dvojíc čísel z 380x260). Funkcia vráti zoznam, ktorý bude obsahovať dvojprvkové n-tice celých čísel. Napríklad:

```
>>> nahodne_body(5)
    [(118, 103), (299, 27), (247, 118), (166, 237), (269, 225)]
```

8. Ak by si celú postupnosť 100 náhodných bodov z predchádzajúcej úlohy vykreslil pomocou jediného `create_line`, dostal by si nejakú čmáranicu (vyskúšaj). Ďalej otestuj, ako sa vykreslí táto postupnosť, keď sa najprv usporiada pomocou funkcie `sorted`. Napríklad, náhodná postupnosť 100 bodov:



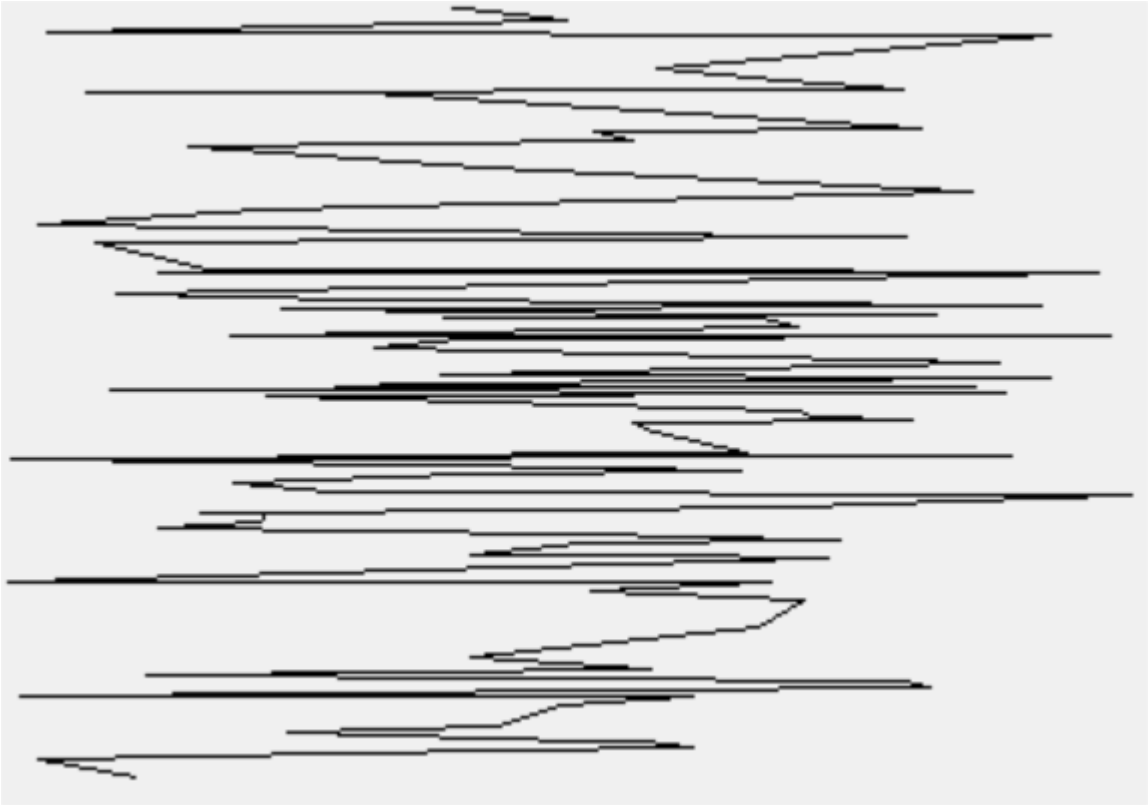
a po usporiadaní pomocou `sorted`:

9. Napíš funkciu `sort_y(zoznam)`, kde `zoznam` obsahuje dvojice (typ `tuple`) celých čísel. Tieto reprezentujú súradnice `(x, y)` nejakých bodov v rovine. Tento `zoznam` treba usporiadať od najmenšieho po najväčší, lenže podľa druhých prvkov dvojíc (`y`-ových súradníc). Uvedom si, že volanie `zoznam.sort()` by usporiadalo zoznam podľa prvých prvkov dvojíc (`x`-ových súradníc). Napríklad:

```
>>> xy = [(100, 30), (200, 10), (300, 20)]
>>> sort_y(xy)
>>> xy
    [(200, 10), (300, 20), (100, 30)]
```

Triediť môžeš takto: najprv v zozname v každej dvojici vymeníš `x` a `y`, potom normálne utriediš (napríklad pomocou metódy `sort`) a nakoniec v takto utriedenom zozname vrátiš všetky dvojice do pôvodného stavu (vymeníš `x` a `y`).

Podobne ako v predchádzajúcej úlohe vykresli takto usporiadanú postupnosť vrcholov pomocou `create_line`:



10. Napíš funkciu `prerob(cislo)`, ktorá z daného celého čísla vyrobí reťazec, ale tak, že cifry zoskupí do trojíc (sprava) a oddelí podčiarkovníkom. Využi funkciu `join`, preto z čísla najprv vyrob zoznam trojznakových reťazcov. Funkcia nič nevypisuje, ale vráti (`return`) výsledný reťazec. Napríklad:

```
>>> prerob(12345678)
    '12_345_678'
```

    Je zaujímavé, že Python aj takto zadané celé čísla považuje za korektné, napríklad:

```
>>> 12_345_678
    12345678
```

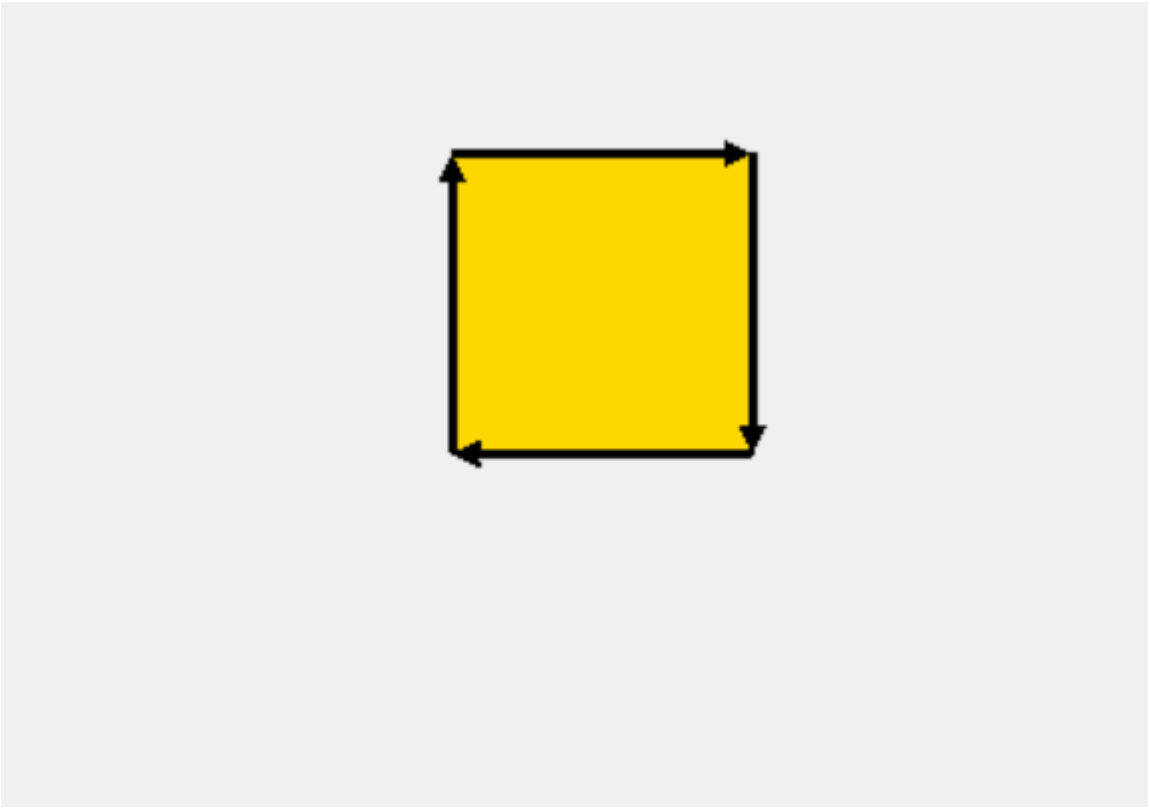
    Výsledok svojej funkcie `prerob` môžeš skontrolovať pomocou `f'{cislo:_}'`.

11. Napíš funkciu `sipka(xy1, xy2)`, ktorá do grafickej plochy nakreslí šípku z bodu `xy1` do bodu `xy2`. Oba tieto parametre sú dvojice čísel (`tuple`), t.j. súradnice bodov. Šípku kresli pomocou `create_line`, v ktorej využiješ ďalší pomenovaný parameter `arrow='last'` (podobné šípky sme kreslili v 15. úlohe z 5.cvičenia). Teraz nakresli štyri šípky, ktoré nakreslia obrys takéhoto štvorca:

```
import tkinter

def sipka(xy1, xy2):
    ...

```

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

canvas.create_rectangle(150, 50, 250, 150, fill='gold')
sipka(...)
sipka(...)
sipka(...)
sipka(...)

tkinter.mainloop()
```

Mal by si dostať takýto obrázok:

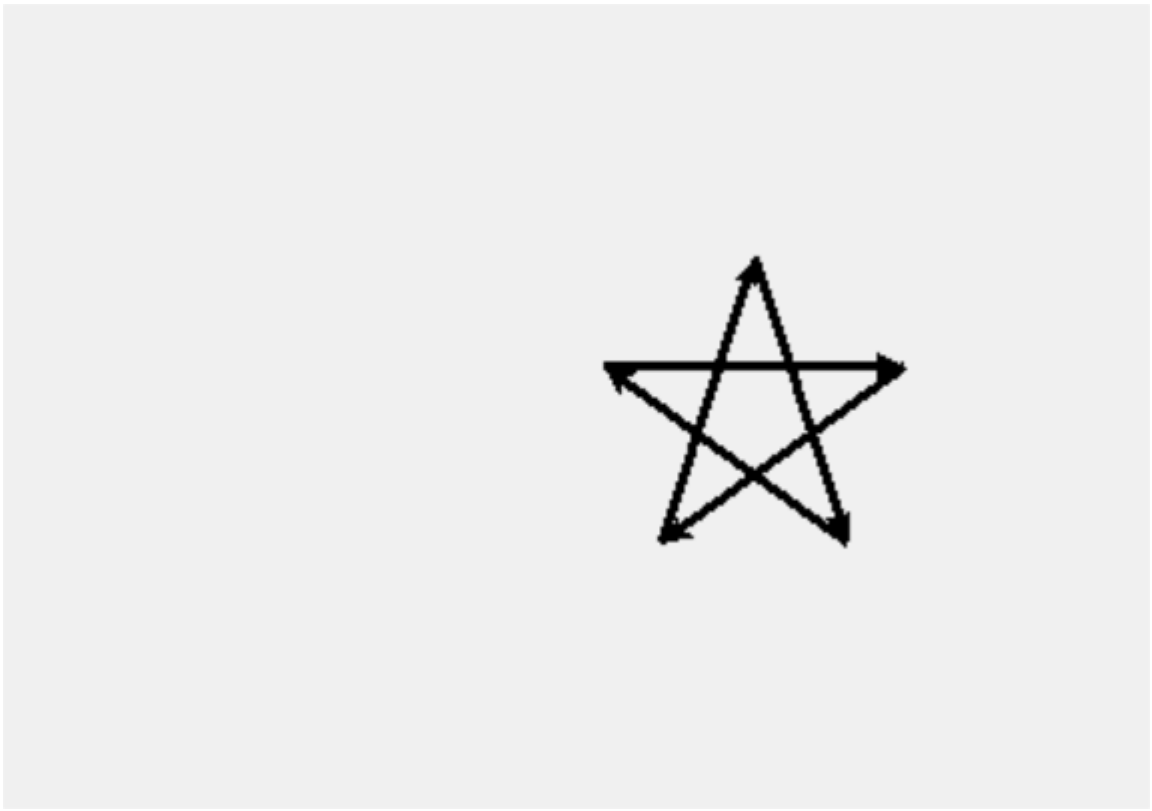


12. Napíš funkciu `vektor(xy, dlzka, uhol)`, ktorá nakreslí vektor (teda šípku) z bodu `xy` (dvojprvkový `tuple` celých čísel) s danou dĺžkou a s daným uhlom (v stupňoch). Uvedom si, že koncové body takéhoto vektora ležia na kružnici s polomerom `dlzka` a stredom `xy`. Na kreslenie šípky využi volanie funkcie `sipka` z predchádzajúcej úlohy. Funkcia `vektor` vráti (`return`) súradnicu koncového bodu vektora ako dvojicu (`tuple`) celých čísel. Napríklad:

```
import tkinter
from math import sin, cos, radians

def sipka(xy1, xy2):
    ...

def vektor(xy, dlzka, uhol):
    ...
```

```
    return ...

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

poz = (200, 120)
for uhol in range(0, 720, 144):
    poz = vektor(poz, 100, uhol)

tkinter.mainloop()
```

nakreslí:



13. Napíš funkciu `zoznam_cisel(retazec)`, ktorá z daného reťazca vyrobí zoznam celých čísel - použi metódu `split`. Napríklad:

```
>>> a = zoznam_cisel('12 345 6 -78 9000')
>>> a
    [12, 345, 6, -78, 9000]
```

14. Napíš funkciu `retazec(zoznam)`, ktorá zo zoznamu čísel vyrobí reťazec - použi metódu `join`. Napríklad:

```
>>> retazec([12, 3.45, 6, -78, 9000])
    '12 3.45 6 -78 9000'
```

15. Napíš funkciu `zadaj(pocet)`, ktorá vráti prečítanú n-ticu čísel zo vstupu (`input`). Funkcia si vypýta od používateľa, aby zadal príslušný počet čísel (napríklad `input(f'zadaj {pocet} čísla: ')`) a ak ich používateľ nezadá požadovaný počet, bude si tento `input` pýtať znovu a znovu. Funkcia vráti (`return`) n-ticu celých čísel a nie n-ticu reťazcov. Napríklad:

```
>>> tri = zadaj(3)
    zadaj 3 čísla: 12 3
    zadaj 3 čísla: 12 3 456
>>> tri
    (12, 3, 456)
```

16. Napíš funkciu `zisti(slovo1, slovo2)`, ktorá zistí (vráti `True` alebo `False`), či sú dve zadané slová zložené presne z tých istých písmen - možno v inom poradí. Napríklad:

```
>>> zisti('Skola', 'Lasko')
    True
>>> zisti('poobede', 'bopeodo')
    False
```

    Pomôcka: použi funkciu `sorted`

17. Napíš funkciu `vsetky_rozne(zoznam)`, ktorá zistí (vráti `True` alebo `False`), či sú všetky prvky zoznamu rôzne. Najprv si vyrob utriedený pomocný zoznam (nepokaz pôvodný) a v ňom zisťuj, či sa nenachádzajú dve rovnaké hodnoty za sebou. Napríklad:

```
>>> vsetky_rozne([3, 8, 7, 9, 4, 1, 6, 10, 5, 2])
    True
>>> zoz = [3, 8, 7, 9, 4, 1, 6, 3, 10, 5, 2]
>>> vsetky_rozne(zoz)
    False
>>> zoz
    [3, 8, 7, 9, 4, 1, 6, 3, 10, 5, 2]
```

18. Napíš funkciu `enum(postupnost)`, ktorá vráti (`return`) n-ticu dvojíc. V tejto n-tici je prvý prvok poradové číslo (od `0` do počet prvkov postupnosti minus 1) a druhým je prvok z danej postupnosti. Malo by to dať rovnaký výsledok ako `tuple(enumerate(postupnost))` ale bez použitia `enumerate`. Napríklad:

```
>>> enum([12, 'dva', 3.14])
    ((0, 12), (1, 'dva'), (2, 3.14))
```

19. Python ponúka ešte jednu štandardnú funkciu `zip`. Táto funkcia, keď dostane nejaké dve postupnosti (napríklad zoznam, n-ticu, reťazec, `range`, …), vytvorí z nich jednu novú postupnosť dvojíc (`tuple`): v každej takejto dvojici je jeden prvok z prvej a jeden z druhej postupnosti. Môžeš vyskúšať, napríklad:

```
>>> list(zip('python', [2, 3, 5, 7]))
```

```
    [('p', 2), ('y', 3), ('t', 5), ('h', 7)]
```

Zrejme, ak je jedna z týchto postupností kratšia, výsledok sa nastaví podľa nej. Napíš funkciu `moj_zip(post1, post2)`, ktorá z dvoch postupností (iterovateľných objektov možno rôznej dĺžky) vytvorí jeden zoznam dvojíc. Nepouži štandardnú funkciu `zip()`.

20. Prepíš funkciu `enum` z (18) úlohy tak, aby neobsahovala cyklus a využila tvoju funkciu `moj_zip`. Teda:

```
def enum(postupnost):
    return ... moj_zip ...
```

21. Napíš funkciu `od_zip(zoznam)`, ktorá bude fungovať ako opak funkcie `moj_zip`. Funkcia dostáva zoznam dvojíc a vráti dva zoznamy prvých a druhých prvkov dvojíc. Napríklad:

```
>>> z1, z2 = od_zip([(2, 'a'), ('h', 3), (5, 'o'), ('j', 7)])
>>> z1
    [2, 'h', 5, 'j']
>>> z2
    ['a', 3, 'o', 7]
```

22. Napíš funkciu `do_dvojic(zoznam)`, ktorá daný zoznam (alebo n-ticu) párnej dĺžky „rozseká“ na zoznam dvojíc (`list` s prvkami `tuple`), dvojice postupne budú (prvý, druhý), (tretí, štvrtý), … Napríklad:

```
>>> x = do_dvojic(('11', 22, '3', 4))
    [('11', 22), ('3', 4)]
```

Rieš najprv bez použitia funkcie `moj_zip()` z predchádzajúcich úloh (teda pomocou for-cyklu) a potom pomocou tejto funkcie (bez cyklu).

23. Napíš funkciu `pomiesaj(zoznam)`, ktorá náhodne pomieša poradie prvkov v pôvodnom zozname. Funkcia nič nevypisuje ani nevracia, len zmení obsah zoznamu. Funkcia by pre `n`-prvkový zoznam mala pracovať takto:
    - najprv zvolí náhodné číslo `x` od 0 do `n-1`
    - v zozname vymení `x`-tý a posledný prvok
    - potom zvolí náhodné číslo `x` od 0 do `n-2`
    - v zozname vymení `x`-tý a predposledný prvok (t.j. `zoznam[-2]`)
    - potom zvolí náhodné číslo `x` od 0 do `n-3`
    - v zozname vymení `x`-tý a a tretí od konca
    - takto to opakuje, kým sa dá

    Napríklad:

```
for i in range(4):
    p = list(range(1, 11))
    pomiesaj(p)
    print(p)
```

```
[2, 9, 6, 1, 7, 8, 10, 3, 5, 4]
[7, 2, 10, 9, 6, 8, 4, 1, 3, 5]
[3, 8, 7, 9, 4, 1, 6, 10, 5, 2]
[2, 9, 4, 1, 8, 3, 5, 7, 6, 10]
```