Cvičenia k Programovaniu v Pythone

časti 1 a 2

Andrej Blaho

sep 12, 2016

Obsah

Fakulta matematiky, fyziky a informatiky

Univerzita Komenského v Bratislave

Autor Andrej Blaho

Názov Cvičenia k Programovaniu v Pythone (k predmetom Programovanie (1) 1-AIN-130/13 a Programovanie (2) 1-AIN-170/13)

Vydavateľ Knižničné a edičné centrum FMFI UK

Rok vydania 2016

Miesto vydania Bratislava

Vydanie prvé

Počet strán 111

Internetová adresa http://python.input.sk/

ISBN 978-80-8147-071-4

Cvičenie

1.1 Štart Pythonu

1. Program si najprv vypýta priezvisko, potom si vypýta meno a na záver vypíše pozdrav v tvare `Vitaj Meno Priezvisko,ako sa máš?`.

```
Zadaj priezvisko: Hraško
Zadaj meno: Janko
Vitaj Janko Hraško, ako sa máš?
```

riešenie:

```
priezvisko = input('Zadaj priezvisko: ')
meno = input('Zadaj meno: ')
print('Vitaj', meno, priezvisko + ', ako sa máš?')
```

2. Program prečíta 2 celé čísla a vypíše výsledok po celočíselnom delení aj zvyšku po delení - delíme prvé číslo druhým:

```
Zadaj 1. číslo: 258369
Zadaj 2. číslo: 1475
celočíselné delenie = 175
zvyšok po delení = 244
```

riešenie:

```
cislo1 = int(input('Zadaj 1. číslo: '))
cislo2 = int(input('Zadaj 2. číslo: '))
print('celočíselné delenie =', cislo1 // cislo2)
print('zvyšok po delení =', cislo1 % cislo2)
```

3. Program od používateľa zistí tri rozmery hranatej nádoby v tvare kvádra v cm. Potom vypíše objem tejto nádoby v centimetroch kubických a tiež objem v decimetroch kubických zaokrúhlene na celé číslo.

```
zadaj 1. rozmer: 12.5
zadaj 2. rozmer: 8
zadaj 3. rozmer: 29
objem v cm3 je 2900
objem v dm3 je priblizne 3
```

riešenie:

```
a = float(input('zadaj 1. rozmer: '))
b = float(input('zadaj 2. rozmer: '))
c = float(input('zadaj 3. rozmer: '))
print('objem v cm3 je', round(a*b*c))
print('objem v dm3 je priblizne', round(a*b*c/1000))
```

4. Program zistí, približne koľko dní už žijete. Najprv do programu zadáte dve celé čísla `roky` a `mesiace`:
   - predpokladajte, že rok má 365 dní a každý mesiac má 30 dní
   - potom to prepočítajte na sekundy
   - potom to zaokrúhlite na celé milióny

```
zadaj roky: 19
zadaj mesiace: 5
počet dní = 7085
počet sekúnd = 612144000
zaokrúhlené sekundy = 612000000
```

riešenie:

```
roky = int(input('zadaj roky: '))
mesiace = int(input('zadaj mesiace: '))
dni = roky * 365 + mesiace * 30
print('počet dní =', dni)
sekundy = dni*24*3600
print('počet sekúnd =', sekundy)
print('zaokrúhlené sekundy =', round(sekundy, -6))
```

1.2 Priradenia, for-cykly

5. Program si vypýta dve čísla a zo znaku `'*'` vypíše obdĺžnik týchto rozmerov:

```
Zadaj šírku: 7
Zadaj výšku: 5
*******
*     *
*     *
*     *
*******
```

Predpokladajte, že šírka aj výška obdĺžnika je aspoň 2.

riešenie:

```
sirka = int(input('Zadaj šírku: '))
vyska = int(input('Zadaj výšku: '))
print('*' * sirka)
for i in range(vyska-2):
    print('*' + ' '*(sirka-2) + '*')
print('*' * sirka)
```

6. Program prečíta nejaké slovo a jedno celé číslo `n`, potom vypíše `n`-riadkov: v prvom je zadané slovo raz, v druhom je 2-krát (slová sú oddelené medzerou), v treťom 3-krát (tiež s medzerou medzi slovami), atď.

```
Zadaj slovo: slon
Zadaj počet: 4
slon
slon slon
slon slon slon
slon slon slon slon
```

Použite násobenie reťazca celým číslom.

riešenie:

```
slovo = input('Zadaj slovo: ')
pocet = int(input('Zadaj počet: '))
for i in range(1, pocet+1):
    print((slovo+' ')*i)
```

7. Program prečíta 3 celé čísla `a`, `b` a `c` a do jedného riadka postupne vypíše všetky hodnoty postupnosti `range(a,b,c)`:

```
Zadaj 1. číslo: 7
Zadaj 2. číslo: 47
Zadaj 3. číslo: 6
postupnost = 7 13 19 25 31 37 43
```

Použite parameter `end=' '` vo funkcii `print()`.

riešenie:

```
a = int(input('Zadaj 1. číslo: '))
b = int(input('Zadaj 2. číslo: '))
c = int(input('Zadaj 3. číslo: '))
print('postupnost =', end=' ')
for i in range(a, b, c):
    print(i, end=' ')
print()
```

8. Program prečíta jedno celé číslo `n` a potom vypíše `n` riadkov, pričom v každom bude `n` celých čísel: v prvom čísla od 1 do `n`, v druhom od 2 do `n+1`, v treťom od 3 do `n+2`, ...

```
zadaj počet: 5
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

riešenie:

```
n = int(input('Zadaj počet: '))
for i in range(1, n+1):
    for j in range(i, i+n):
        print(j, end=' ')
    print()
```

KAPITOLA 2

Cvičenie

2.1 Grafika

1. Modul `math` obsahuje množstvo užitočných matematických funkcií. Aby ste ich mohli používať, musíte zadať príkaz `import math`. Goniometrické funkcie `math.sin()` a `math.cos()` ale pracujú v radiánoch a nie v stupňoch. Napíšte program, ktorý vypíše tabuľku hodnôt: v prvom stĺpci je uhol v stupňoch (z intervalu od 0 do 360 s krokom 30), v druhom je vypočítaný `sin()` a v treťom `cos()` tohto uhla. Na prevod medzi stupňami a radiánmi môžete použiť hodnotu premennej `math.pi`. Výsledok zaokrúhľujte na 2 desatinné miesta (`round(číslo,2)`). :

```
0 0.0 1.0
30 0.5 0.87
60 0.87 0.5
90 1.0 0.0
120 0.87 -0.5
150 0.5 -0.87
180 0.0 -1.0
...
```

riešenie:

```
import math

for uhol in range(0, 361, 30):
    rad = uhol/180*math.pi
    print(uhol, round(math.sin(rad), 2), round(math.cos(rad), 2))
```

2. Body na kružnici so stredom `(x0,y0)` a polomerom `r` sa dajú vyjadriť vzorcom:

```
x = x0 + r * cos(uhol)
y = y0 + r * sin(uhol)
```

kde `uhol` je číslo od 0 do 360 stupňov (pozor na radiány). Ak budete takto vypočítané body postupne spájať úsečkami (napr. pomocou `canvas.create_line()`), dostanete kružnicu. Nakreslite týmto postupom kružnicu, pričom otestujte kreslenie pre rôznu hustotu bodov na kružnici (pre rôzne hodnoty zväčšovania uhla, napr. s krokom 30, alebo 10 alebo 2, ...).

riešenie:

```
import math, tkinter

canvas = tkinter.Canvas(width=300, height=300)
```

```
canvas.pack()
x0, y0, r = 150, 150, 100
xx, yy = x0+r, y0
for uhol in range(10, 361, 10):
    rad = uhol/180*math.pi
    x = x0 + r * math.cos(rad)
    y = y0 + r * math.sin(rad)
    canvas.create_line(x, y, xx, yy)
    xx, yy = x, y
```

3. Idea programu z úlohy (2) sa dá využiť aj na kreslenie pravidelného `n`-uholníka. Napíšte program, ktorý si vypýta `n` a do stredu grafickej plochy nakreslí pravidelný n-uholník

```
zadaj n: 7
# nakreslí pravidelný 7-uholník
```

riešenie:

```
import math, tkinter

canvas = tkinter.Canvas(width=300, height=300)
canvas.pack()
n = int(input('zadaj n: '))
x0, y0, r = 150, 150, 100
xx, yy = x0+r, y0
uhol = 360/n
for i in range(n):
    rad = uhol/180*math.pi
    x = x0 + r * math.cos(rad)
    y = y0 + r * math.sin(rad)
    canvas.create_line(x, y, xx, yy)
    xx, yy = x, y
    uhol += 360/n
```

4. Ak v programe z úlohy (3) nebudeme spájať susedné vrcholy, ktoré ležia na obvode kružnice, ale budeme spájať tieto vrcholy so stredom kružnice (zvoľte žlté hrubé pero) a na koniec nakreslíme žltý kruh (`canvas.create_oval()`) s rovnakým stredom ako naša kružnica ale s menším polomerom, dostaneme slnko s lúčmi. Napíšte program:

```
pocet lucov: 10
dlzka lucov od stredu: 150
velkost slnka: 80
# nakreslí žlté slnko
```

riešenie:

```
import math, tkinter

canvas = tkinter.Canvas(width=300, height=300, bg='navy')
canvas.pack()
n = int(input('pocet lucov: '))
r = int(input('dlzka lucov od stredu: '))
vel = int(input('velkost slnka: '))
x0, y0 = 150, 150
uhol = 360/n
for i in range(n):
    rad = uhol/180*math.pi
    x = x0 + r * math.cos(rad)
```

```
    y = y0 + r * math.sin(rad)
    canvas.create_line(x0, y0, x, y, width=8, fill='yellow')
    uhol += 360/n
canvas.create_oval(x0-vel, y0-vel, x0+vel, y0+vel, fill='yellow', outline=
↪'')
```

5. Napíšte program, ktorý si najprv vyžiada nejaké meno a potom ho 10-krát vypíše na náhodné pozície nejakým väčším fontom

```
zadaj meno: Bonifac
# vypíše toto slovo 10-krát na náhodné pozície
```

Zabezpečte, aby sa náhodné polohy výpisu slova generovali len vo vnútri grafickej plochy a aby z nej žiadne slovo nevypadlo. Môžete počítať s tým, že slovo bude mať max. 8 znakov.

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
meno = input('zadaj meno: ')
for i in range(10):
    x = random.randint(100, 500)
    y = random.randint(30, 420)
    canvas.create_text(x, y, text=meno, font='arial 30 bold')
```

6. Progran z úlohy (5) doplňte tak, aby sa každé výpísané slovo objavilo vo farebnom obdĺžniku (zvoľte nejaké vhodné rozmery). Farby výplne týchto obdĺžnikov (zrejme sa nakreslia príkazom `canvas.create_rectangle()`) sa budú striedať medzi nejakými vami zvolenými tromi farbami, napr. `'blue'`, `'black'`, `'gray'` pričom samotný text môže byť žltý.

```
zadaj meno: Bonifac
zadaj počet: 20
# vypíše toto slovo 20-krát na náhodné pozície aj s rámikmi
```

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
meno = input('zadaj meno: ')
pocet = int(input('zadaj počet: '))
farba, farba1, farba2 = 'red', 'navy', 'darkgreen'
for i in range(pocet):
    x = random.randint(100, 500)
    y = random.randint(30, 420)
    canvas.create_rectangle(x-100, y-30, x+100, y+30, fill=farba)
    canvas.create_text(x, y, text=meno, font='arial 30 bold', fill='yellow
↪')
    farba, farba1, farba2 = farba1, farba2, farba
```

7. V priečinku, v ktorom je nainštalovaný Python, nájdite súbor `'rgb.txt'` a otestujte z neho aspoň 5 nejakých zaujímavých farieb: nakreslite pod seba 5 rovnako veľkých obdĺžnikov a každý zafarbite nejakou inou farbou (inou, ako už známe základné farby)

```
# nakreslí 5 farebných obdĺžnikov
```

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
y, vel = 10, 80
for farba in 'indian red','maroon1','purple','MistyRose1','SkyBlue1':
    canvas.create_rectangle(100, y, 400, y+vel, fill=farba)
    y += vel
```

2.2 Podmienené príkazy

8. Napíšte program, ktorý pomocou príkazu `canvas.create_polygon()` nakreslí nejaký n-uholník (apoň so 4 stranami), ktorý nemá žiadnu stranu rovnobežnú s osami. Tento útvar má na začiatku priesvitnú výplň. Program si potom vypýta meno nejakej farby a touto farbou vyplní obsah n-uholníka. Program bude toto opakovať v nekonečnom cykle.

```
farba: azure
# zmení výplň polygonu
farba: pink
# zmení výplň polygonu
farba: navy
# zmení výplň polygonu
...
```

Na vygenerovanie vrcholov polygonu môžete použiť aj náhodný generátor, napr. do 4 premenných priradíte 4 dvojice náhodných súradníc a potom tieto body použijete v príkaze na nakreslenie polygonu. Na zmenu farby polygonu použite príkaz `canvas.itemconfig()`.

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
a = (100, 100)
b = (300, 150)
c = (50, 200)
d = (100, 300)
e = (300, 250)
i = canvas.create_polygon(a, b, c, d, e)
while True:
    canvas.update()
    farba = input('farba: ')
    canvas.itemconfig(i, fill=farba)
```

9. Zvoľte si nejaký obrázok vo formáte `.png` najlepšie do rozmeru 100x100 pixelov. Program tento obrázok prečíta a vykreslí ho 10-krát na náhodné pozície. Pritom si zapamätá identifikačné číslo prvého nakresleného obrázka z týchto desiatich (v ploche sa nachádza pod všetkými ostatnými). Potom si v nekonečnom cykle vypýta 2 čísla `dx` a `dy` a posunie (`canvas.move()`) o tieto hodnoty tento jeden zapamätaný obrázok.

```
# nakreslí obrázok 10-krát
zadaj dx: 50
zadaj dy: -20
# posunie obrázok o (50, -20)
zadaj dx: -100
zadaj dy: 0
# posunie obrázok o (-100, 0)
...
```

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
obrazok = tkinter.PhotoImage(file='auto.png')
for i in range(10):
    x = random.randint(100, 500)
    y = random.randint(30, 420)
    if i == 0:
        prvy = canvas.create_image(x, y, image=obrazok)
    else:
        canvas.create_image(x, y, image=obrazok)
while True:
    canvas.update()
    dx = int(input('zadaj dx: '))
    dy = int(input('zadaj dy: '))
    canvas.move(prvy, dx, dy)
```

10. Zvoľte si nejaké dva malé rôzne obrázky. Vykreslite ich na opačné konce grafickej plochy: prvý úplne vľavo, druhý úplne v pravo, oba s tou istou y-ovou súradnicou niekde v strede plochy. Potom si program vypýta 2 čísla: rýchlosť pohybu pre 1. obrázok a rýchlosť pohybu pre 2. obrázok. Po zadaní týchto čísel sa oba obrázky pomaly rozbehnú smerom ku sebe, každý svojou rýchlosťou (t.j. krokom, ktorým sa mení ich x-ová súradnica). Toto vzájomné približovanie zastane, keď x-ová súradnica 1. obrázku preskočí x-ovú súradnicu 2. obrázku (t.j. zastane, keď x1 > x2).

```
# nakreslí dva obrázky na ich štartové pozície
zadaj 1. rýchlost': 7
zadaj 2. rýchlost': 3
# obrázky sa hýbu ku sebe
```

riešenie:

```
import random, tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
obrazok1 = tkinter.PhotoImage(file='auto1.png')
obrazok2 = tkinter.PhotoImage(file='auto2.png')
x1, x2 = 50, 550
prvy = canvas.create_image(x1, 200, image=obrazok1)
druhy = canvas.create_image(x2, 200, image=obrazok2)
canvas.update()
dx1 = int(input('zadaj 1. rýchlost': '))
dx2 = int(input('zadaj 2. rýchlost': '))
while x1 < x2:                          # while x1+50 < x2-50:
    canvas.move(prvy, dx1, 0)
```

```
    x1 += dx1
    canvas.move(druhy, -dx2, 0)
    x2 -= dx2
    canvas.update()
    canvas.after(100)
```

KAPITOLA 3

Cvičenie

3.1 Funkcie

1. Napíšte funkciu `max(a,b)`, ktorá vráti väčšiu z hodnôt `a` a `b`, napr.

```
>>> max(55, 59)
59
>>> max('abc', 'ABC')
'abc'
```

Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu. Nepoužívajte štandardnú funkciu `max()`, ktorá robí to isté ako vaša funkcia `max()`.

riešenie:

```
def max(a, b):
    if a > b:
        return a
    return b
```

2. Napíšte funkciu `cs(cislo)`, ktorá vypočíta ciferný súčet zadaného čísla, napr.

```
>>> cs(2506)
13
```

Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu. Úlohu riešte dvomi rôznymi spôsobmi:

- pomocou `cislo % 10` získame poslednú cifru čísla (tú pripočítame k výsledku) a číslo o poslednú cifru skrátime `cislo // 10`, toto sa opakuje, kým je číslo nenulové
- najprv vytvoríme z čísla znakový reťazec `str(cislo)`, ten pomocou for-cyklu prejdeme znak za znakom, každý prevedieme na číslo `int(znak)` a toto pripočítame k výsledku

riešenie:

```
def cs(cislo):
    sucet = 0
    while cislo != 0:
        sucet += cislo % 10
        cislo //= 10
    return sucet
```

```
# druhe riesenie

def cs(cislo):
    sucet = 0
    for znak in str(cislo):
        sucet += int(znak)
    return sucet
```

3. Napíšte funkciu `nsd(a,b)`, ktorá počíta **najväčší spoločný deliteľ** dvoch čísel, napr.

```
>>> nsd(21, 15)
3
>>> nsd(12, 35)
1
```

Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu.

riešenie:

```
def nsd(a, b):
    for i in range(a, 0, -1):
        if a%i == 0 and b%i == 0:
            return i

# druhe riesenie - euklidov algoritmus

def nsd(a, b):
    while b > 0:
        a, b = b, a%b
    return a
```

4. Napíšte grafickú funkciu `sestuholnik(dlzka,farba)`, ktorá nakreslí vyfarbený pravidelný 6-uholník so stranou `dlzka`. Stred tohto 6-uholníka nech je `(150,150)`. Zrejme využijete funkciu `canvas.create_polygon()`, ktorému ako parametre pošlete 6 bodov v rovine, napr.

```
b1 = (150+dlzka, 150)
b2 = (...)
...
canvas.create_polygon(b1, b2, ...)
```

Uvedomte si, že každá strana 6-uholníka tvorí so stredom 6-uholníka rovnostranný trojuholník so stranou `dlzka`.

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=400, height=300, bg='white')
canvas.pack()

def sestuholnik(dlzka, farba):
    vyska = dlzka * 3**.5 / 2     # vyska trojuholnika
    a = (150+dlzka, 150)
    b = (150+dlzka/2, 150-vyska)
    c = (150-dlzka/2, 150-vyska)
    d = (150-dlzka, 150)
    e = (150-dlzka/2, 150+vyska)
    f = (150+dlzka/2, 150+vyska)
```

```
    canvas.create_polygon(a, b, c, d, e, f, fill=farba)

sestuholnik(100, 'blue')
```

5. Napíšte funkciu `kruhy(pocet,y=100,r=20,farba='blue')`, ktorá nakreslí vedľa seba zadaný `pocet` zafarbených kruhov (s čiernym obrysom) s polomerom `r`, ktoré všetky majú rovnakú `y`-ovú súradnicu a x-ová súradnica prvého kruhu je `r`. Otestujte napr.

```
>>> kruhy(5, r=80)
>>> kruhy(10, farba='red')
>>> kruhy(8, r=50, y=50, farba='')
```

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=300, bg='white')
canvas.pack()

def kruhy(pocet, y=100, r=20, farba='blue'):
    for x in range(r, 2*pocet*r+1, 2*r):
        canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba)

kruhy(5, r=45)
kruhy(10, farba='red')
kruhy(8, r=50, y=50, farba='')
```

3.2 Znakové reťazce

6. Napíšte funkcie `nn()` a `nf()` - obe bez parametrov, ktoré pri každom zavolaní vrátia náhodné číslo od 0 do 99 (funkcia `nn()`) alebo náhodnú farbu (funkcia `nf()`). Otestujte napr.

```
>>> nn()
37
>>> nn()*3
135
>>> 20 + nn()//5
21
>>> nf()
'#13ad7f'
```

riešenie:

```
import random

def nn():
    return random.randrange(100)

def nf():
    return '#{:06x}'.format(random.randrange(256**3))

print(nn())
print(nn()*3)
print(20 + nn()//5)
print(nf())
```

7. Napíšte funkciu `text(slovo)`, ktorá na náhodnú pozíciu vypíše zadané `slovo`, pričom veľkosť písma nech je tiež náhodné číslo medzi 20 a 40 a farba textu nech je tiež náhodná. Pre generovanie všetkých náhodných hodnôt vo funkcii využite funkcie `nn()` a `nf()` z predchádzajúceho príkladu. Otestujte napr.

```
>>> for i in range(10):
        text('PYTHON')
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=400, bg='white')
canvas.pack()

def nn():
    return random.randrange(100)

def nf():
    return '#{:06x}'.format(random.randrange(256**3))

def text(slovo):
    x, y = 100+nn()*4, 50+nn()*3
    canvas.create_text(x, y, text=slovo,
                       font='arial '+str(20+nn()//5),
                       fill=nf())

for i in range(10):
    text('Python')
```

8. Napíšte funkciu `pi1(n)`, ktorá bude počítať približnú hodnotu čísla **pi** podľa tohto radu:

```
pi = 4/1 - 4/3 * 4/5 - 4/7 + 4/9 - 4/11 + ...
```

V čitateľoch všetkých zlomkov je číslo `4`, v menovateľoch sú nepárne čísla menšie ako `n`, pričom sa striedajú znamienka + a −. Otestujte pre rôzne veľké `n`, napr.

```
>>> for n in 10, 100, 1000, ...:
        print(n, pi1(n))
10 3.3396825396825403
100 3.121594652591011
1000 3.139592655589785
...
```

riešenie:

```
def pi1(n):
    vysl = 0
    citatel = 4
    for i in range(1, n, 2):
        vysl += citatel/i
        citatel = -citatel
    return vysl

for n in 10, 100, 1000, 10000, 100000, 1000000:
    print(n, pi1(n))
```

9. Konštantu **pi** môžeme počítať aj iným spôsobom (tzv. Monte Carlo): do grafickej plochy veľkosti napr. 200x200 budeme náhodne generovať zadaný `pocet` náhodných bodiek. Tie z nich, ktorých vzdialenosť od stredu štvorca

je menšia ako 100, zafarbíme na červeno. Na záver funkcia vráti počet červených bodiek. Z dvoch informácií: celkový počet bodiek a počet bodiek v kruhu (červených bodiek) vieme približne počítať **pi**:

- pi = 4 * počet červených / počet všetkých

Matematicky sa to dá ukázať veľmi jednoduchou úvahou (vzťah medzi obsahom štvorca a obsahom kruhu). Vaša funkcia `pi2(pocet)` nemusí farebné bodky naozaj kresliť, stačí, keď ich bude náhodne generovať a počítať počet v kruhu. Otestujte pre rôzne počty generovaných bodov, napr.

```
>>> for n in 100, 1000, 10000, 100000, 1000000:
        p = pi2(n)
        print(n, 4*p/n)
100 3.08
1000 3.152
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=200, height=200, bg='white')
canvas.pack()

def pi2(pocet):
    cervene = 0
    #canvas.delete('all')
    for i in range(pocet):
        x = random.randrange(200)
        y = random.randrange(200)

        if (x-100)**2 + (y-100)**2 <= 100**2:
            cervene += 1
            farba = 'red'
        else:
            farba = 'yellow'
        canvas.create_oval(x-1,y-1,x+1,y+1, fill=farba, outline='')
    return cervene

def pi2(pocet):      # bez grafiky
    cervene = 0
    for i in range(pocet):
        x = random.randrange(200)
        y = random.randrange(200)
        if (x-100)**2 + (y-100)**2 <= 100**2:
            cervene += 1
    return cervene

for n in 10, 100, 1000, 10000, 100000, 1000000:
    p = pi2(n)
    print(n, 4*p/n)
```

10. Na predchádzajúcom cvičení ste kreslili pravidelný n-uholník spájaním vypočítaných bodov, ktoré ležia na nejakej kružnici. Napíšte funkciu `n_uholnik(n,polomer=100)`, ktorá nakreslí pravideľný n-uholník. Jeho vrcholy ležia na kružnici so stredom `(150,150)` a s polomerom `polomer`. Funkcia okrem toho vráti súčet všetkých dĺžok nakreslených strán n-uholníka (obvod útvaru). Z hodnoty `polomer` a vypočítaného obvodu sa dá približne počítať konštanta **pi**: čím je `n` väčšie, tým sa nakreslený útvar stále viac blíži ku kružnici. Poznáme vzorec na výpočet obvodu kruhu: `obvod = 2 * pi * polomer`, z čoho vieme vypočítať **pi**, ak poznáme obvod a polomer. Otestujte takto vypočítané **pi** pre rôzne veľké `n`. Pre urýchlenie výpočtu pre veľké `n` nemusíte naozaj n-uholník kresliť, ale stačí generovať body a počítať obvod. Napr.

```
>>> for n in 10, 100, 1000, ...:
        obvod = n_uholnik(n)
        print(n, obvod, obvod/200)
```

riešenie:

```
import tkinter, math

canvas = tkinter.Canvas(width=300, height=300, bg='white')
canvas.pack()

def n_uholnik(n, polomer=100):
    x0, y0 = 150, 150
    xx, yy = x0+polomer, y0
    uhol, obvod = 360/n, 0
    for i in range(n):
        x = x0 + polomer*math.cos(uhol/180*math.pi)
        y = y0 + polomer*math.sin(uhol/180*math.pi)
        canvas.create_line(xx, yy, x, y)
        obvod += math.sqrt((x-xx)**2 + (y-yy)**2)
        xx, yy = x, y
        uhol += 360/n
    return obvod

for n in 10, 100, 1000:
    obvod = n_uholnik(n)
    print(n, obvod, obvod/200)
```

11. Idea príkladu (10) sa dá trochu zjednodušiť: n-uholník nebudeme ani kresliť ani počítať všetky jeho vrcholy. Stačí vypočítať dĺžku jednej strany n-uholníka, keďže je pravidelný všetky strany by predsa mali byť rovnako dlhé. Tiež, jedna strana n-uholníka by sa nemusela počítať ako vzdialenosť dvoch bodov v rovine, ale ako dĺžka základne rovnoramenného trojuholníka, v ktorom poznáme dĺžky ramien (`polomer`) a uhol, ktorý zvierajú (v stupňoch `360/n`). Ak zadáme, že polomer kružnice je `1`, zjednoduší sa aj celkový vzorec: `pi = n * strana / 2`, kde `strana` je vypočítaná strana n-uholníka. Napíšte funkciu `pi4(n)`, ktorá počíta **pi** z n-uholníka. Otestujte, napr.

```
>>> for n in 10, 100, 1000, ...:
        print(n, pi4(n))
```

riešenie:

```
import math

def pi4(n):
    uhol = 360/n
    strana = 2*math.sin(uhol/2/180*math.pi)
    return n * strana / 2

for n in 10, 100, 1000, 10000, 100000, 1000000:
    print(n, pi4(n))
```

12. Napíšte funkciu `vyhod_samo(retazec)`, ktorá vráti zadaný `retazec` ale bez samohlások, napr.

```
>>> vyhod_samo('jelenovi pivo nelej')
'jlnv pv nlj'
>>> vyhod_samo('strc prst skrz krk')
'strc prst skrz krk'
```

riešenie:

```
def vyhod_samo(retazec):
    for znak in 'aeiouy':
        retazec = retazec.replace(znak, '')
    return retazec

print(vyhod_samo('jelenovi pivo nelej'))
print(vyhod_samo('strc prst skrz krk'))
```

13. Napíšte funkciu `palindrom(retazec)`, ktorá zistí (vráti `True` alebo `False`), či je daný reťazec palindrom - číta sa rovnako odpredu aj odzadu, napr.

```
>>> palindrom('jelenovipivonelej')
True
>>> palindrom('jelenovi pivo nelej')
False
```

Môžete porozmýšľať nad verziou funkcie, v ktorej sa ignorujú medzery a druhý príklad potom tiež vráti `True`.

riešenie:

```
def palindrom(retazec):
    n = len(retazec)
    for i in range(n//2):
        if retazec[i] != retazec[n-i-1]:
            return False
    return True

print(palindrom('jelenovipivonelej'))
print(palindrom('jelenovi pivo nelej'))

# druha verzia ignoruje medzery

def palindrom2(retazec):
    return palindrom(retazec.replace(' ', ''))

print(palindrom2('jelenovipivonelej'))
print(palindrom2('jelenovi pivo nelej'))
```

14. Napíšte funkciu `koduj(slovo)`, pomocou ktorej dokážeme zakódovať aj rozkódovať ľubovoľný znakový reťazec. Zakódovanie/odkódovanie pracuje na tomto princípe: každé písmeno v reťazci sa cyklicky posunie o 13 pozícií v abecede, ostatné nepísmenové znaky sa nemenia. Napr.

```
>>> koduj('Python je Super!')
'Clguba wr Fhcre!'
>>> koduj('Clguba wr Fhcre!')
'Python je Super!'
```

riešenie:

```
def koduj(slovo):
    vysl = ''
    for znak in slovo:
        if 'a'<=znak<='z':
            vysl += chr((ord(znak)-ord('a')+13)%26+ord('a'))
        elif 'A'<=znak<='Z':
            vysl += chr((ord(znak)-ord('A')+13)%26+ord('A'))
```

```
        else:
            vysl += znak
    return vysl

print(koduj('Python je Super!'))
print(koduj('Clguba wr Fhcre!'))
```

KAPITOLA 4

Cvičenie

4.1 Súbory

Varovanie: V riešeniach všetkých príkladov dnešného cvičenia nepoužívajte metódu `split()`.

1. Napíšte funkciu `sucet(retazec)`, ktorá spočíta hodnoty všetkých čísel v danom znakovom reťazci, napr.

```
>>> sucet('15 -7 104')
112
>>> sucet('')
0
```

Čísla v reťazci sú navzájom oddelené aspoň jednou medzerou.

riešenie:

```
def sucet(retazec):
    vysl = 0
    cislo = ''
    for z in retazec+' ':
        if z == ' ':
            if cislo:
                vysl += int(cislo)
            cislo = ''
        else:
            cislo += z
    return vysl

print(sucet('15 -7 104'))
print(sucet(''))
```

2. Napíšte funkciu `sucty(subor)`, ktorá číta textový súbor s menom `subor` a pre každý riadok súboru vypíše súčet čísel:

   - v každom riadku vstupného súboru je niekoľko čísel oddelených jednou medzerou (riadok môže byť aj prázdny)
   - program pre každý riadok spočíta tieto čísla (napr. funkciou `sucet()`) a tento súčet vypíše pomocou `print()`

   Ak súbor `'cisla.txt'` obsahoval:

```
1 2

15 -7 104
```

dostávame:

```
>>> sucty('cisla.txt')
3
0
112
```

riešenie:

```
def sucty(subor):
    with open(subor) as t:
        for riadok in t:
            print(sucet(riadok.strip()))

sucty('cisla.txt')
```

3. Napíšte funkciu `spracuj(subor1,subor2)`, ktorá číta textový súbor s menom `subor1` a vyrobí z neho nový súbor s menom `subor2`:
   - v každom riadku vstupného súboru je niekoľko čísel oddelených jednou medzerou (riadok môže byť aj prázdny)
   - program pre každý riadok: najprv tieto čísla prekopíruje do výstupného súboru, tiež ich spočíta (napr. funkciou `sucet()`) a do výstupného súboru na koniec riadka pripíše tento súčet

   Ak súbor `'cisla.txt'` obsahoval:

```
1 2

15 -7 104
```

   po zavolaní `spracuj('cisla.txt','cisla2.txt')` v súbore `'cisla2.txt'` dostávame:

```
1 2 3
 0
15 -7 104 112
```

riešenie:

```
def spracuj(subor1, subor2):
    with open(subor1) as vstup, open(subor2, 'w') as vystup:
        for riadok in vstup:
            riadok = riadok.strip()
            print(riadok, sucet(riadok), file=vystup)

spracuj('cisla.txt', 'cisla2.txt')
```

4.2 n-tice (tuple)

4. Napíšte funkciu `ntica(retazec)`, ktorá zo znakového reťazca čísel oddelených medzerou vytvorí n-ticu čísel, napr.

```
>>> ntica('15 -7 104')
(15, -7, 104)
>>> ntica('42')
(42,)
>>> ntica('')
()
```

riešenie:

```
def ntica(retazec):
    vysl = ()
    cislo = ''
    for z in retazec+' ':
        if z == ' ':
            if cislo:
                vysl += (int(cislo),)
            cislo = ''
        else:
            cislo += z
    return vysl

print(ntica('15 -7 104'))
print(ntica('42'))
print(ntica(''))
```

5. Prepíšte funkciu `sucet(retazec)` z prvého príkladu tak, aby ste využili funkciu `ntica()` zo (4) príkladu a štandardnú funkciu `sum()`

```
>>> sucet('15 -7 104')
112
>>> sucet('')
0
```

riešenie:

```
def sucet(retazec):
    return sum(ntica(retazec))

print(sucet('15 -7 104'))
print(sucet(''))
```

6. Napíšte funkciu `ntica_str(param)`, ktorá dostáva ako parameter n-ticu celých čísel a vytvorí z neho reťazec čísel oddelených medzerou - funkcia by mala byť opačnou k funkcii `ntica(retazec)` zo (4) príkladu, napr.

```
>>> ntica_str((15, -7, 104))
'15 -7 104'
>>> ntica_str((42,))
'42'
>>> ntica_str(())
''
```

riešenie:

```
def ntica_str(param):
    vysl = ''
    for prvok in param:
        vysl += str(prvok) + ' '
```

```
    return vysl.strip()

print(repr(ntica_str((15, -7, 104))))
print(repr(ntica_str((42,))))
print(repr(ntica_str(())))
```

7. Napíšte funkciu `citaj(subor)`, ktorá prečíta textový súbor s menom `subor` a vytvorí z neho n-ticu, ktorá pre každý riadok obsahuje n-ticu čísel. Predpokladajte, že súbor má štruktúru z príkladu (2), t.j. obsahuje riadky čísel oddelené medzerou. Využite funkciu `ntica(retazec)` zo (4) príkladu. Pre súbory `'cisla.txt'` a `'cisla2.txt'` z (3) príkladu funkcia vráti:

```
>>> citaj('cisla.txt')
((1, 2), (), (15, -7, 104))
>>> citaj('cisla2.txt')
((1, 2, 3), (0,), (15, -7, 104, 112))
```

riešenie:

```
def citaj(subor):
    vysl = ()
    with open(subor) as vstup:
        for riadok in vstup:
            vysl += (ntica(riadok.strip()),)
    return vysl

print(citaj('cisla.txt'))
print(citaj('cisla2.txt'))
```

8. Napíšte funkciu `spoj(ntica,retazec)`, ktorá prvky danej n-tice (sú to reťazce) spojí do jedného výsledného reťazca pričom medzi ne vkladá zadaný reťazec. Napr.

```
>>> c = ('pisem', 'program', 'v', 'pythone')
>>> spoj(c, ' ')
'pisem program v pythone'
>>> spoj(c, '<=>')
'pisem<=>program<=>v<=>pythone'
>>> spoj(('hello',), '???')
'hello'
```

Najprv úlohu riešte for-cyklom a prechádzaním prvkov n-tice, potom to vyriešte pomocou metódy `join()` pre znakové reťazce. Volanie tejto metódy má tvar:

```
ret'azec.join(ntica)
```

riešenie:

```
def spoj(ntica, retazec):
    vysl = ''
    for prvok in ntica:
        if vysl:
            vysl += retazec + prvok
        else:
            vysl = prvok
    return vysl

c = ('pisem', 'program', 'v', 'pythone')
print(spoj(c, ' '))
```

```
print(spoj(c, '<=>'))
print(spoj(('hello',), '???'))

# riesenie pomocou join()

def spoj(ntica, retazec):
    return retazec.join(ntica)

c = ('pisem', 'program', 'v', 'pythone')
print(spoj(c, ' '))
print(spoj(c, '<=>'))
print(spoj(('hello',), '???'))
```

9. Napíšte funkciu `zisti(subor)`, ktorá prečíta textový súbor, o každom riadku zistí jeho dĺžku (bez koncového `'\n'`) a vráti priemernú dĺžku riadkov súboru. Napr. pre súbor `'subor.txt'`:

```
prvy
druhy riadok
stredny
end
```

dostávame:

```
>>> zisti('subor.txt')
6.5
```

riešenie:

```
def zisti(subor):
    sucet = pocet = 0
    with open(subor) as vstup:
        for riadok in vstup:
            sucet += len(riadok[:-1])
            pocet += 1
    return sucet / pocet

print(zisti('subor.txt'))
```

10. Napíšte funkciu `len_dlhe(subor1,subor2)`, ktorá číta riadky vstupného súboru s menom `subor1` a niektoré z nich kopíruje do výstupného súboru s menom `subor2`: kopíruje len tie, ktoré nie sú kratšie ako priemerná dĺžka všetkých riadkov súboru. Priemernú dĺžku riadkov počítajte ako funkcia `zisti()` v príklade (10), túto funkciu nevolajte, ale kód zapíšte do funkcie: cieľom je, aby ste v jednej funkcii dvakrát prešli celý obsah súboru.

Napr. pre súbor z (9) príkladu volanie `len_dlhe('subor.txt','subor2.txt')` vyrobí súbor `'subor2.txt'` s týmito riadkami:

```
druhy riadok
stredny
```

riešenie:

```
def len_dlhe(subor1, subor2):
    sucet = pocet = 0
    with open(subor1) as vstup:
        for riadok in vstup:
            sucet += len(riadok[:-1])
            pocet += 1
```

```
    priemer = sucet / pocet

    with open(subor1) as vstup, open(subor2, 'w') as vystup:
            for riadok in vstup:
                if len(riadok[:-1]) >= priemer:
                    vystup.write(riadok)

len_dlhe('subor.txt', 'subor2.txt')
```

11. Napíšte funkciu `vyrob(subor,retazec)`, ktorá vytvorí súbor s meno `subor`:
    - do prvého riadka zapíše zadaný reťazec
    - do druhého riadka zapíše ten istý reťazec, v ktorom sa všetky samohlásky nahradia písmenom `'a'`
    - do tretieho riadka opäť to isté ale samohlásky sa nahradia písmenom `'e'`
    - atď. sa postupne vygenerujú reťazce s nahrádzajúcimi znakmi `'i'`, `'o'`, `'u'`
    - inšpiráciou nech sú slová pesničky 'sedi mucha na stene, sedi a spi'

    riešenie:

```
def vyrob(subor, retazec):
    with open(subor, 'w') as vystup:
        print(retazec, file=vystup)
        for znak in 'eiouy':
            retazec = retazec.replace(znak, 'a')
        print(retazec, file=vystup)
        znak0 = 'a'
        for znak in 'eiou':
            retazec = retazec.replace(znak0, znak)
            print(retazec, file=vystup)
            znak0 = znak

vyrob('subor3.txt', 'sedi mucha na stene, sedi a spi')
```

12. Napíšte funkciu `utriedene(ntica)`, ktorá dostáva parameter nejakú n-ticu a zistí, či jej prvky sú utriedené vzostupne (od najmenšieho po najväčšie), napr.

```
>>> a = (-22, 4, 7, 7, 13, 22)
>>> utriedene(a)
True
>>> utriedene(a + a)
False
>>> utriedene(())
True
```

    riešenie:

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

a = (-22, 4, 7, 7, 13, 22)
print(utriedene(a))
print(utriedene(a+a))
print(utriedene(()))
```

13. Napíšte funkciu `kresli_a_pocitaj(ntica)`, ktorá dostáva ako parameter n-ticu bodov nejakého n-uholníka (napr. štvorca), nakreslí tento n-uholník s náhodnou farbou výplne. Funkcia zároveň vypočíta a vráti novú n-ticu bodov:

    - prvý prvok je bod v strede prvej strany n-uholníka (stred úsečky 1. a 2. vrcholu), atď.
    - posledný prvok je bod v strede n-tej strany, t.j. stred medzi posledným a prvým vrcholom

```
>>> body = ((50, 50), (150, 50), (150, 150), (50, 150))
>>> kresli_a_pocitaj(body)
((100.0, 50.0), (150.0, 100.0), (100.0, 150.0), (50.0, 100.0))
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=300, height=300, bg='white')
canvas.pack()

def kresli_a_pocitaj(ntica):
    farba = '#{:06x}'.format(random.randrange(256**3))
    canvas.create_polygon(ntica, fill=farba)
    nova = ()
    ntica += (ntica[0],)
    for i in range(len(ntica)-1):
        a, b = ntica[i:i+2]
        bod = (a[0]+b[0])/2, (a[1]+b[1])/2
        nova += (bod,)
    return nova

body = ((50, 50), (150, 50), (150, 150), (50, 150))
print(kresli_a_pocitaj(body))
```

14. Napíšte funkciu `n_krat(n,ntica)`, ktorá `n`-krát zavolá funkciu `kresli_a_pocitaj()` z (13) príkladu ale po každom volaní nahradí hodnotu `ntica` výsledkom zavolanej funkcie (n-ticou so stredmi strán).

```
>>> n_krat(5, ((50, 50), (150, 50), (150, 150), (50, 150)))
# nakreslí 5 vpísaných štvorcov
```

riešenie:

```
def n_krat(n, ntica):
    for i in range(n):
        ntica = kresli_a_pocitaj(ntica)

n_krat(5, ((50, 50), (150, 50), (150, 150), (50, 150)))
n_krat(4, ((100, 300), (300, 300), (200, 127)))
```

KAPITOLA 5

Cvičenie

5.1 Polia (list)

Riešte najmä úlohy na prácu s poľami, keďže v pondelok bude priebežný test, na ktorom sa objavia úlohy na prácu s poľami, ale nebudú tam žiadne úlohy na udalosti.

1. funkcia `generuj(n,do)` vygeneruje `n`-prvkové pole náhodných celých čísel, všetky sú z intervalu `<0,do-1>`

```
>>> a = generuj(6, 10)
>>> print(a)
[9, 1, 0, 9, 5, 7]
```

riešenie:

```
import random

def generuj(n, do):
    pole = []
    for i in range(n):
        pole.append(random.randrange(do))
    return pole

a = generuj(6, 10)
print(a)
```

2. funkcia `pole_cisel(retazec)` dostáva ako parameter znakový reťazec, ktorý obsahuje celé čísla oddelené medzerou; funkcia z neho vytvorí pole čísel (využite metódu `split()` a funkciu `int()`); funkcia nič nevypisuje, ale vráti toto pole ako výsledok funkcie

```
>>> a = pole_cisel('19 173 7 97 1001')
>>> print(a)
[19, 173, 7, 97, 1001]
```

riešenie:

```
def pole_cisel(retazec):
    pole = retazec.split()
    for i in range(len(pole)):
        pole[i] = int(pole[i])
    return pole
```

```
# alebo aj inak

def pole_cisel(retazec):
    pole = []
    for prvok in retazec.split():
        pole.append(int(prvok))
    return pole

a = pole_cisel('19 173 7 97 1001')
print(a)
```

3. funkcia `z_pola_retazec(pole)` z poľa čísel vyrobí znakový reťazec, v ktorom sú čísla oddelené medzerou

```
>>> pole = [19, 173, 7, 97, 1001]
>>> r = z_pola_retazec(pole)
>>> r
'19 173 7 97 1001'
```

riešenie:

```
def z_pola_retazec(pole):
    pom = pole[:]
    for i in range(len(pom)):
        pom[i] = str(pom[i])
    return ' '.join(pom)

pole = [19, 173, 7, 97, 1001]
r = z_pola_retazec(pole)
print(repr(r))
```

4. funkcia `pridaj(pole,postupnost)` pridá prvky postupnosti (ľubovoľný iterovateľný objekt) na koiec poľa, funkcia nič nevracia

```
>>> pole = [19, 173, 7, 97, 1001]
>>> pridaj(pole, (37, -6))
>>> pole
[19, 173, 7, 97, 1001, 37, -6]
```

riešenie:

```
def pridaj(pole, postupnost):
    for prvok in postupnost:
        pole.append(prvok)

# alebo pomocou standardnej metody extend()

def pridaj(pole, postupnost):
    pole.extend(postupnost)

pole = [19, 173, 7, 97, 1001]
pridaj(pole, (37, -6))
print(pole)
```

5. funkcia `sucet(pole1,pole2)` vráti nové pole, ktoré vzikne tak, že sa sčítavajú zodpovedajúce prvky dvoch vstupných polí, ak je jedno zo vstupných polí kratšie ako druhé, tak zvyšok výsledného poľa bude obsahovať prvky väčšieho poľa bez zmeny

```
>>> pole = sucet([5, 'a', 3.14], [-1, 'b'])
>>> pole
[4, 'ab', 3.14]
```

riešenie:

```
def sucet(pole1, pole2):
    vysl = []
    dlzka1, dlzka2 = len(pole1), len(pole2)
    dlzka = min(dlzka1, dlzka2)
    for i in range(dlzka):
        vysl.append(pole1[i]+pole2[i])
    if dlzka1 > dlzka:
        vysl += pole1[dlzka:]
    elif dlzka2 > dlzka:
        vysl += pole2[dlzka:]
    return vysl

# to iste sa da zapisat zhustenejsie

def sucet(pole1, pole2):
    vysl = []
    for i in range(min(len(pole1), len(pole2))):
        vysl.append(pole1[i]+pole2[i])
    return vysl + pole1[len(vysl):] + pole2[len(vysl):]

pole = sucet([5, 'a', 3.14], [-1, 'b'])
print(pole)
```

6. funkcia `len_cisla(pole)` ponechá v poli len čísla (`int` alebo `float`), ostatné prvky z poľa vyhodí, vytvorte 2 verzie
   1. zmení obsah vstupného poľa a nič nevracia
   2. nemení obsah poľa ale vráti nové pole s číselnými prvkami

   pre 1. verziu:

```
>>> pole = [(1, 2), 5, 'a', 3.14]
>>> len_cisla(pole)
>>> pole
[5, 3.14]
```

   pre 2. verziu:

```
>>> pole = [(1, 2), 5, 'a', 3.14]
>>> len_cisla(pole)
[5, 3.14]
>>> pole
[(1, 2), 5, 'a', 3.14]
```

   riešenie:

   tri riešenia 1. verzie:

```
def len_cisla(pole):
    vysl = []
    for prvok in pole:
        if type(prvok) == int or type(prvok) == float:
```

```
            vysl.append(prvok)
    pole[:] = vysl

def len_cisla(pole):
    i = 0
    while i < len(pole):
        if type(pole[i]) != int and type(pole[i]) != float:
            pole.pop(i)
        else:
            i += 1

def len_cisla(pole):
    for i in range(len(pole)-1, -1, -1):
        if type(pole[i]) != int and type(pole[i]) != float:
            pole.pop(i)

pole = [(1, 2), 5, 'a', 3.14]
len_cisla(pole)
print(pole)
```

riešenie 2. verzie:

```
def len_cisla(pole):
    vysl = []
    for prvok in pole:
        if type(prvok) == int or type(prvok) == float:
            vysl.append(prvok)
    return vysl

pole = [(1, 2), 5, 'a', 3.14]
print(len_cisla(pole))
print(pole)
```

7. funkcia `pridaj_sucty(meno_suboru)` číta daný súbor, pričom v každom riadku sú len celé čísla, funkcia na koniec každého riadka pridá súčet všetkých prvkov daného riadka. Postupujte takto:
   - najprv vytvorí pole riadkov (reťazcov)
   - potom pre každý riadok vyrobí pole čísel (funkcia z 2. príkladu) a vypočíta ich súčet, tento súčet pridá ho na koniec reťazca (v poli reťazcov)
   - zapíše všetky riadky z poľa riadkov späť do toho istého súboru

   riešenie:

```
def pridaj_sucty(meno_suboru):
    with open(meno_suboru) as vstup:
        pole = list(vstup)
    with open(meno_suboru, 'w') as vystup:
        for riadok in pole:
            sucet = 0
            for prvok in riadok.split():
                sucet += int(prvok)
            print(riadok.strip(), sucet, file=vystup)

print('*** subor.txt ***\n' + open('subor.txt').read())
pridaj_sucty('subor.txt')
print('*** subor.txt ***\n' + open('subor.txt').read())
```

8. funkcia `miesaj(pole)` pomieša poradie prvkov poľa, funkcia nič nevracia, ale zmení obsah poľa; môžete postupovať takto:
   - vyrobí sa kópia poľa a samotné pole sa vyprázdni (možno na to využijete metódu `pole.clear()`)
   - potom sa postupne do výsledného poľa: z kópie vyberie náhodný prvok, zaradí ho do výsledného na koniec (metóda `append()`) a z kópie ho pritom vyhodí (metóda `pop()`)

```
>>> pole = ['jeden', 2, 'tri', 4]
>>> miesaj(pole)
>>> pole
['jeden', 4, 2, 'tri']
```

   - rovnakú činnosť robí aj funkcia `random.shuffle()`, vy to ale riešte bez volania tejto funkcie

   riešenie:

```
import random

def miesaj(pole):
    kopia = pole[:]
    pole.clear()
    while kopia:
        i = random.randrange(len(kopia))
        pole.append(kopia.pop(i))

pole = ['jeden', 2, 'tri', 4]
miesaj(pole)
print(pole)
```

9. zistite čo vygeneruje táto funkcia:

```
def urob(n, d):
    pole = [x, y] = [200, 150]
    dx, dy = 1, 1                    # dx, dy = 1, 0
    for i in range(d, d*n+1, d):
        pole.extend([x+i*dx, y+i*dy])
        dx, dy = dy, -dx
 return pole
```

   - zistite, čo sa zmení, ak druhý riadok funkcie zmeníme tak, ako je to v komentári tohot riadka
   - zamyslite sa, čomu v matematike zodpovedá `dx,dy = dy,-dx`
   - môže vám pomôcť, keď výsledok funkcie, napr. `urob(30,4)`, vykreslíte pomocou `create_line()`

   riešenie:

   - funkcia generuje body štvorcovej špirály so stredom (200, 150): vytvorí `n` úsečiek s dlžkami: `d`, `2*d`, `3*d`, `4*d`, ...
   - ak `dx,dy` je vektor v rovine, potom `dx,dy = dy,-dx` je jeho otočenie o 90 stupňov
   - funkcia `urob()` pre `dx,dy = 1,0`, vytvorí kosoštvorcovú špirálu: jej vrcholy ležia buď na x-ovej alebo y-ovej osi

   môžete otestovať napr.:

```
import tkinter

canvas = tkinter.Canvas()
```

```
canvas.pack()
canvas.create_line(urob(20, 5))
```

10. funkcia `kresli(pole)` dostáva ako parameter pole čísel z <0,250>. Funkcia z týchto hodnôt nakreslí histogram (stĺpcový diagram): obdĺžniky, ktoré budú ležať tesne vedľa seba so šírkou 10 a výškou podľa príslušného prvku poľa (prvý obdĺžnik má šírku 10 a výšku podľa prvého prvku poľa, umiestnite ho na ľavý okraj grafickej plochy; druhý bude tesne vedľa neho s výškou podľa druhého prvku poľa).

```
>>> kresli([100,150,170,50,100,200,250,150])
```

riešenie:

```
def kresli(pole, sirka=10):
    x = 5
    for vyska in pole:
        farba = '#{:06x}'.format(random.randrange(256**3))
        canvas.create_rectangle(x, 255, x+sirka, 255-vyska,␣
↪fill=farba)
        x += sirka

import tkinter, random
canvas = tkinter.Canvas()
canvas.pack()
kresli([100,150,170,50,100,200,250,150], 30)
```

11. funkcia `rozdel(pole)` rozdelí pole celých čísel na dve nové polia: všetky nezáporné prvky budú v prvom výslednom poli a všetky zvyšné v druhom; funkcia vráti výsledok ako dvojicu (typ `tuple`) práve vytvorených polí

```
>>> a, b = rozdel([1, -2, 3, 4, 0, -5])
>>> a
[1, 3, 4, 0]
>>> b
[-2, -5]
```

riešenie:

```
def rozdel(pole):
    pole1, pole2 = [], []
    for prvok in pole:
        if prvok >= 0:
            pole1.append(prvok)
        else:
            pole2.append(prvok)
    return pole1, pole2

a, b = rozdel([1, -2, 3, 4, 0, -5])
print(a)
print(b)
```

12. funkcia `uprac(pole)` presťahuje všetky prvky so zápornou hodnotou na začiatok poľa (v tom poradí, ako boli v poli), ostatné prvky posunie vpravo; funkcia nevracia žiadnu hodnotu, len nejako modifikuje vstupné pole

```
>>> a = [0, 5, -3, -1, 2, -2, 7, -4]
>>> uprac(a)
>>> a
[-3, -1, -2, -4, 0, 5, 2, 7]
```

riešenie:

```
def uprac(pole):
    i = 0
    for j in range(len(pole)):
        if pole[j] < 0:
            pole.insert(i, pole.pop(j))
            i += 1

# alebo

def uprac(pole):
    i = 0
    kladne = []
    while i < len(pole):
        if pole[i] >= 0:
            kladne.append(pole.pop(i))
        else:
            i += 1
    pole.extend(kladne)    # prilepi na koniec pola

a = [0, 5, -3, -1, 2, -2, 7, -4]
uprac(a)
print(a)
```

5.2 Udalosti v grafickej ploche

13. kliknutie do plochy na mieste kliknutia vypíše celé číslo: najprv 1, potom postupne po každom kliknutí o jedna vyššie

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    global cislo
    cislo += 1
    canvas.create_text(event.x, event.y, text=cislo)

cislo = 0
canvas.bind('<Button-1>', klik)
```

14. podobne ako (13), ale od druhého kliknutia okrem vypísaného čísla spojí úsečkou tento kliknutý bod s predchádzajúcim - treba si pamätať aj pozíciu predchádzajúceho kliknutia, všetky takto kreslené úsečky majú inú náhodnú farbu

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    global cislo, pred
    cislo += 1
    xy = event.x, event.y
    canvas.create_text(xy, text=cislo)
    if pred:
        farba = '#{:06x}'.format(random.randrange(256**3))
        canvas.create_line(pred, xy, fill=farba)
    pred = xy

cislo = 0
pred = ()
canvas.bind('<Button-1>', klik)
```

15. kliknutie do plochy na tomto mieste nakreslí hviezdičku:

    - skladá sa z 50 náhodne vygenerovaných úsečiek, ktoré všetky začínajú z kliknutého bodu `(x,y)` a ich koncový bod má `x` aj `y` zmenené o `random.randint(-30,30)`
    - všetkých týchto 50 úsečiek má rovnakú náhodne vybranú farbu - každé ďalšie kliknutie zvolí novú farbu pre 50 úsečiek

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    x0, y0 = event.x, event.y
    farba = '#{:06x}'.format(random.randrange(256**3))
    for i in range(50):
        x = x0 + random.randint(-30,30)
        y = y0 + random.randint(-30,30)
        canvas.create_line(x0, y0, x, y, fill=farba)

canvas.bind('<Button-1>', klik)
```

16. písanie znakov do grafickej plochy (použite udalosť `'<Key>'`, ktorá v parametri `event.char` vráti stlačený znak):

    - prvý znak sa vypíše na súradnice (10,50), každý ďalší má x posunuté o 16
    - zvoľte font napr. `'consolas 20'`

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klaves(event):
    global x
    canvas.create_text(x, y, text=event.char, font='consolas 20')
    x += 16

x, y = 10, 50
canvas.bind_all('<Key>', klaves)
```

17. podobne ako (16) len sa nebude pre každý znak vytvárať nový grafický objekt (`create_text()`), ale na začiatku sa vyrobí jeden grafický objekt (s prázdnym textom) v strede plochy a po každom zatlačení znaku sa tento pridá do momentálneho reťazca a grafický objekt sa opraví pomocou `itemconfig()`

    - ak je to naprogramované správne, mal by fungovať aj kláves `Enter` a ďalšie znaky by mali prejsť už na nový riadok
    - doprogramujte kláves `BackSpace` (`chr(8)`) tak, že sa zo zobrazeného textu odstráni posledný znak

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klaves(event):
    global t
    if event.char == chr(8):
        t = t[:-1]
    else:
        t += event.char
    canvas.itemconfig(text, text=t)

text = canvas.create_text(300, 225, text='', font='consolas 20')
```

```
t = ''
canvas.bind_all('<Key>', klaves)
```

18. kliknutie do plochy sem nakreslí štvorček (jeho stred je kliknutý bod) so stranou 20 náhodnej farby
    - ak ešte nepustíme kliknutie, ale myš ťaháme, posúva sa aj nakreslený štvorček
    - každé ďalšie kliknutie začne nový štvorček s náhodnou farbou

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    global stv
    farba = '#{:06x}'.format(random.randrange(256**3))
    x, y = event.x, event.y
    stv = canvas.create_rectangle(x-10, y-10, x+10, y+10, fill=farba)

def tahanie(event):
    x, y = event.x, event.y
    canvas.coords(stv, x-10, y-10, x+10, y+10)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

19. kliknutie do plochy nakreslí na tomto mieste kružnicu s nulovým polomerom (zatiaľ ju nebude vidieť)
    - počas ťahania myši sa táto kružnica prekresľuje (`coords()`) tak, že stred sa nemení, ale mení sa jej polomer, aby pozícia myši bola na obvode kružnice
    - kružnicu môžete vyplniť náhodnou farbou

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    global kruh, x, y
    farba = '#{:06x}'.format(random.randrange(256**3))
    x, y = event.x, event.y
```

```
    kruh = canvas.create_oval(0, 0, 0, 0, fill=farba)

def tahanie(event):
    r = ((event.x-x)**2 + (event.y-y)**2)**.5
    canvas.coords(kruh, x-r, y-r, x+r, y+r)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

20. pri kliknutí do plochy sa naštartuje časovač, ktorý o 1 zvýši veľkým fontom vypísané celé číslo - pri štarte je tam 0, časovač toto číslo každých 0.1 sekundy zvyšuje o 1; snažte sa nemodifikovať žiadnu globálnu premennú (len grafický objekt text), pomôže vám funkcia `canvas.itemcget(id,'text')`, ktorá vráti text textového objektu s identifikačným číslom `id`

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    canvas.itemconfig(cislo, text=0)
    casovac()

def casovac():
    x = int(canvas.itemcget(cislo, 'text'))
    canvas.itemconfig(cislo, text=x+1)
    canvas.after(100, casovac)

cislo = canvas.create_text(300, 200, text='', font='consolas 100')
canvas.bind('<Button-1>', klik)
```

21. program z predchádzajúcej (20) úlohy zmeňte tak, že kliknutie počas behu časovača, tento časovač zastaví a znovu kliknutie pokračuje vo zvyšovaní čísla

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    global bezi
    bezi = not bezi
```

```
    if bezi:
        casovac()

def casovac():
    x = int(canvas.itemcget(cislo, 'text'))
    canvas.itemconfig(cislo, text=x+1)
    if bezi:
        canvas.after(100, casovac)

bezi = False
cislo = canvas.create_text(300, 200, text='0', font='consolas 100')
canvas.bind('<Button-1>', klik)
```

22. naprogramujte pohyb sekundovej ručičky: úsečka dĺžky 150 s jedným vrcholom v (300, 225), ktorá sa každú sekundu posunie o 6 stupňov (resp. 10-krát za sekundu o 0.6 stupna)

```
import tkinter

sirka, vyska = 600, 450
canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter, math

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def casovac():
    global uhol
    x0, y0 = 300, 225
    x = x0 + 150*math.sin(uhol/180*math.pi)
    y = y0 - 150*math.cos(uhol/180*math.pi)
    canvas.coords(rucicka, x0, y0, x, y)
    uhol += 6
    canvas.after(1000, casovac)

uhol = 0
rucicka = canvas.create_line(0, 0, 0, 0, width=5)
casovac()
```

23. naprogramujte odrážanie gulečníkovej gule od okrajov grafickej plochy: na začiatku sa nachádza v strede a pohybuje sa napr. o `(dx,dy)=(5,3)`, keď narazí na jednu zo 4 stien, príslušná zložka vektora sa zneguje, napr. ak narazí na pravú stenu (x-ová súradnica > šírka plochy), tak vektor posunu sa zmení `(dx,dy)=(-5,3)`

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()
...
```

riešenie:

```
import tkinter

sir, vys = 400, 300
```

```
canvas = tkinter.Canvas(width=sir, height=vys, bg='white')
canvas.pack()

x, y, r = sir//2, vys//2, 15
lopta = canvas.create_oval(x-r, y-r, x+r, y+r, fill='red')

dx, dy = 5, 3
cas = 50

def start(e):
    pohyb()

def pohyb():
    global dx, dy
    canvas.move(lopta, dx, dy)
    x, y = canvas.coords(lopta)[:2]
    x, y = x+r, y+r
    if x < 10:
        dx = abs(dx)
    if x > sir-10:
        dx = -abs(dx)
    if y < 10:
        dy = abs(dy)
    if y > vys-10:
        dy = -abs(dy)
    canvas.update()
    canvas.after(cas, pohyb)

canvas.bind('<Button-1>', start)
```

KAPITOLA 6

Cvičenie

6.1 Korytnačky (turtle)

1. zadefinujte funkcie `stvorec()` a `trojuholnik()` s parametrami: dĺžka strany, farba pera, hrúbka pera, ktoré nakreslia štvorec a rovnostranný trojuholník. Obe funkcie budú mať definované pre parametre farba a hrúbka pera náhradné hodnoty, napr.

```
def stvorec(strana, farba='black', hrubka=1):
    ...
```

riešenie:

```
def stvorec(strana, farba='black', hrubka=1):
    t.pencolor(farba)
    t.width(hrubka)
    for i in range(4):
        t.fd(strana)
        t.rt(90)

def trojuholnik(strana, farba='black', hrubka=1):
    t.pencolor(farba)
    t.width(hrubka)
    for i in range(3):
        t.fd(strana)
        t.rt(120)
```

2. pomocou funkcií pre kreslenie štvorca nakreslite domček: modrý štvorec (s hrúbkou čiar 20) a na ňom červený trojuholník (s hrúbkou čiar 10)

```
def domcek(strana):
    ...
```

riešenie:

```
def domcek(strana):
    stvorec(strana, 'blue', 20)
    t.pu()
    t.fd(strana)
    t.rt(30)
    t.pd()
    trojuholnik(strana, 'red', 10)
```

```
import turtle, random
t = turtle.Turtle()
t.lt(90)
domcek(100)
```

3. funkcia `domcek2()` nakreslí taký istý domček ako predchádzajúca úloha (celý čiernou tenkou čiarou), ale pri kreslení neprejde po žiadnej čiare viac krát (zrejme nevyužijete funkcie `stvorec()` a `trojuholnik()` a samotné kreslenie skončíte v inom vrchole ako ste začali)

```
def domcek2(strana):
    ...
```

riešenie:

```
def domcek2(strana):
    for i in range(4):
        t.fd(strana)
        t.rt(90)
    t.lt(60)
    for i in range(2):
        t.fd(strana)
        t.rt(120)

import turtle, random
t = turtle.Turtle()
domcek2(100)
```

4. funkcia `rad_domcekov(n,strana)` nakreslí vedľa seba rad `n` domčekov (využite funkciu `domcek`), ale tak, aby bola medzi nimi medzera 10.

```
rad_domcekov(n, strana):
    ...
```

riešenie:

```
def rad_domcekov(n, strana):
    for i in range(n):
        domcek(strana)
        t.pu()
        t.lt(30)
        t.bk(strana)
        t.rt(90)
        t.fd(strana+10)
        t.lt(90)
        t.pd()

import turtle, random
t = turtle.Turtle()
t.lt(90)
rad_domcekov(10, 50)
```

5. funkcia `rad_domcekov2(pole)` podobne ako (4) kreslí rad domčekov, ale veľkosti strán domčekov sú zadané v poli (`pole` je pole celých čísel), napr.

```
>>> rad_domcekov2([50,60,70,60,50])
```

nakreslí vedľa seba 5 domčekov so stranami postupne: 50,60,70,60,50; medzi domčekmi je medzera 10

riešenie:

```
def rad_domcekov2(pole):
    for strana in pole:
        domcek(strana)
        t.pu()
        t.lt(30)
        t.bk(strana)
        t.rt(90)
        t.fd(strana+10)
        t.lt(90)
        t.pd()

import turtle, random
t = turtle.Turtle()
t.lt(90)
rad_domcekov2([50,60,70,60,50])
```

6. opravte nasledovný program tak, aby nakreslil veľké číslice ciferníka hodín, t.j. čísla od 1 do 12 na správnych miestach:

```
t.pu()
for i in range(30):
    t.fd(100)
    t.write(i)
    t.fd(-100)
    t.lt(12)
```

- korytnačia metóda `t.write(text)` vypíše na mieste, kde sa nachádza korytnačka, zadaný text (alebo hodnotu ľubovoľného typu)
- táto metóda môže mať parameter `font` podobne ako `create_text` v `tkinter` (napr `t.write('ahoj',font='arial 20')`)

riešenie:

```
import turtle
t = turtle.Turtle()
t.pu()
t.lt(60)
for i in range(1, 13):
    t.fd(100)
    t.write(i, font='arial 20')
    t.fd(-100)
    t.rt(30)
```

7. korytnačía metóda `t.stamp()` opečiatkuje tvar (`shape`) korytnačky do plochy; využite ideu z (6) a namiesto čísel na obvode kruhu opečiatkujte korytnačku v tvare `'turtle'`, takto by vzniklo napr. 30 čiernych tvarov korytnačiek, pred príkaz `t.stamp()` zmeňte farbu pera a výplne tak, aby mali tieto obrázky náhodné farby

```
t.pu()
for i in range(30):
    t.fd(100)
    t.stamp()
    t.fd(-100)
    t.lt(12)
```

riešenie:

```
import turtle
t = turtle.Turtle()
t.pu()
t.lt(60)
for i in range(1, 13):
    t.fd(100)
    t.write(i, font='arial 20')
    t.fd(-100)
    t.rt(30)
```

8. nasledovné príkazy kreslia 5-cípu hviezdu:

```
def hviezda(vel):
    for i in range(5):
        t.fd(vel)
        t.rt(144)
```

   - pozmeňte tento program tak, aby sa nakreslila 5-cípa hviezda ale bez vnútorných čiar, t.j. len obrys takejto hviezdy (novej hviezde nemusíte zachovať tú istú veľkosť) - budete kresliť nie 5 ale 10 čiar a po každej sa o nejaký uhol otočíte

   riešenie:

```
def hviezda(vel):
    for i in range(5):
        t.fd(vel)
        t.rt(144)
        t.fd(vel)
        t.lt(72)

import turtle, random
t = turtle.Turtle()
hviezda(50)
```

9. pridajte funkcii hviezda ďalší parameter `farba`, vďaka ktorej sa vnútro hviezdy zafarbí danou farbou

```
def hviezda(vel, farba=''):
    ...
```

   - farba prázdny reťazec znamená, že vnútro hviezdy nechceme zafarbovať (toto už zabezpečí vyfarbovanie pomocou `begin_fill()` a `end_fill()`)

   riešenie:

```
def hviezda(vel, farba=''):
    t.fillcolor(farba)
    t.begin_fill()
    for i in range(5):
        t.fd(vel)
        t.rt(144)
        t.fd(vel)
        t.lt(72)
    t.end_fill()

import turtle, random
t = turtle.Turtle()
hviezda(100, 'red')
```

10. funkcia `hviezdy()` nakreslí na náhodné pozície `n` malých žltých hviezd (náhodných veľkostí medzi 10 a 30) - využite funkciu `hviezda` z (9)

```
def hviezdy(n):
    ...

turtle.bgcolor('navy')
hviezdy(100)
```

- funkcia `bgcolor()` zafarbí celú grafickú plochu na zadanú farbu

riešenie:

```
def hviezdy(n):
    for i in range(n):
        t.pu()
        t.setpos(random.randint(-300, 300), random.randint(-300,
↪300))
        t.pd()
        hviezda(random.randint(10, 30), 'yellow')

import turtle, random
turtle.bgcolor('navy')
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
hviezdy(100)
```

11. funkcia `prechadzka(velkost,posun,pocet)` rozbehne pre korytnačku náhodné prechádzanie sa, pričom stráži oblasť, ktorá je zjednotením dvoch kruhov s polomerom `velkost` a so stredmi `(0,0)` a `(posun,0)`; keď korytnačka vystúpi z týchto kruhov, cúvne späť. Kým sa korytnačka hýbe v prvom kruhu, čiary kreslí červenou, inak keď sa pohybuje v druhom kruhu, kreslí modrou

```
def prechadzka(velkost, posun, pocet=1000):
    ...

turtle.delay(0)
t.speed(0)
t.ht()
t.pensize(5)
prechadzka(50, 50)
```

- parameter `pocet` určuje počet krokov prechádzky (počet prechodov cyklu)

riešenie:

```
def prechadzka(velkost, posun, pocet=1000):
    for i in range(pocet):
        t.rt(random.randrange(360))
        t.fd(5)
        if t.distance(0, 0) <= velkost:
            t.pencolor('red')
        elif t.distance(posun, 0) <= velkost:
            t.pencolor('blue')
        else:
            t.bk(5)


import turtle, random
```

```
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
t.pensize(5)
prechadzka(50, 50, 10000)
```

6.2 Rekurzia

12. zadefinujte funkciu `vpisane(n,strana)`, ktorá nakreslí rekurzívnu krivku vpísané trojuholniky:
    - pre `n=0` funkcia nerobí nič
    - pre `n=1` funkcia nakreslí rovnostranný trojuholník so zadanou stranou
    - pre `n=2` funkcia nakreslí rovnostranný trojuholník so zadanou stranou a do neho "vpíše" polovičný trojuholník (jeho vrcholy budú v stredoch strán)
    - pre každé ďalšie `n` budú mať ja vpísané trojuholníky ďalšie vpísané, t.j. funkcia nakreslí `n` trojuholníkov, každý ďalší má o polovicu menšie strany ako predchádzajúci
    - snažte sa obrázok nakresliť **jedným ťahom**, t.j. korytnačka neprejde po žiadnej čiare viackrát a ani nezdvihne pero

    riešenie:

```
def vpisane(n, strana):
    if n > 0:
        t.fd(strana/2)
        t.lt(60)
        vpisane(n-1, strana/2)
        t.rt(60)
        t.fd(strana/2)
        t.lt(120)
        for i in range(2):
            t.fd(strana)
            t.lt(120)

import turtle
turtle.delay(0)
t = turtle.Turtle()
#t.speed(0)
vpisane(7, 400)
```

13. zadefinujte funkciu `kriziky(n,d)`, ktorá nakreslí rekurzívnu krivku krížiky:
    - pre `n=0` funkcia nerobí nič
    - pre `n=1` funkcia nakreslí kríž (4 na seba kolmé čiary zadanej dĺžky `d`) - kreslenie skončí tam kde začala
    - pre `n=2` funkcia opäť nakreslí kríž zadanej veľkosti, ale konci všetkých 4 ramien kríža nakreslí tiež kríž ale s ramenami tretinovej veľkosti, pričom tieto menšie krížiky budú oproti ramenu otočené o 45 stupňov (nakreslí sa 1 kríž veľkosti `d` a 4 veľkosti `d/3`)
    - každá ďalšia úroveň krivky nakreslí menšie a menšie krížiky na konci ramien vnorených krížov (pre `n=3` sa nakreslí 1 kríž veľkosti `d`, 4 veľkosti `d/3` a 16 veľkosti `d/9`)

    riešenie:

```
def kriziky(n, d):
    if n > 0:
        for i in range(4):
            t.fd(d)
            t.rt(45)
            kriziky(n-1, d/3)
            t.lt(45)
            t.bk(d)
            t.rt(90)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
kriziky(5, 150)
```

14. predchádzajúcu funkciu (13) opravte tak, aby kreslené krížiky nemuseli mať 4 ramená, ale ľubovoľný iný počet:

```
def kriziky(n, d, pocet=4):
    ...

kriziky(4, 150, 5)      # nakreslí 1 velký, 5 mensich, 25 este mensich a 125␣
↪najmensich krizikov
```

riešenie:

```
def kriziky(n, d, pocet=4):
    if n > 0:
        for i in range(pocet):
            t.fd(d)
            t.rt(45)
            kriziky(n-1, d/3, pocet)
            t.lt(45)
            t.bk(d)
            t.rt(360/pocet)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
kriziky(4, 150, 7)
```

15. zadefinujte rekurzívnu funkciu `mocnina(n,k)`, ktorá vypočíta `n**k` pre celé nezáporné `k` len pomocou násobenia:

    - `mocnina(n,0)` = 1
    - `mocnina(n,k)` = `mocnina(n,k-1) * n`

    funkciu otestujte napr. pre `mocnina(2,900)` a porovnajte s `2**900`

    riešenie:

```
def mocnina(n, k):
    if k == 0:
        return 1
    return n * mocnina(n, k-1)
```

```
x = mocnina(2, 900)
print(x == 2**900)
```

16. rekurzívne riešenie predchádzajúcej funkcie `mocnina(n,k)` môžeme urýchliť, ak môžeme okrem násobenia použiť aj umocňovanie na 2 (čo je opäť len náseobením so samým sebou):

    - `mocnina(n,0)` = 1
    - `mocnina(n,k)` = `mocnina(n,k//2) ** 2` ... pre párne `k`
    - `mocnina(n,k)` = `mocnina(n,k-1) * n` ... pre nepárne `k`

    funkciu otestujte napr. pre `mocnina(2,10000)` a porovnajte s `2**10000`

    riešenie:

```
def mocnina(n, k):
    if k == 0:
        return 1
    if k%2:        # pre neparne
        return n * mocnina(n, k-1)
    m = mocnina(n, k//2)
    return m * m

x = mocnina(2, 10000)
print(x == 2**10000)
```

17. zadefinujte rekurzívnu funkciu `palindrom(retazec)`, ktorá zistí (vráti `True` alebo `False`), či je zadaný reťazec palindrom, t.j. či sa číta rovnako od začiatku ako od konca; pri tomto zisťovaní sa ignorujú medzery a nerozlišujú sa malé a veľké písmená, napr.

```
>>> palindrom('Jelenovi Pivo Nelej')
True
```

    - rekurzia by mala pracovať na takomto princípe: porovná prvé a posledné písmeno a ak sú zhodné ešte zistí, či aj reťazec bez prvého a posledného písmena je palindrom

    riešenie:

```
def pom_palindrom(r):       # pomocna funkcia
    if len(r) <= 1:
        return True
    if r[0] != r[-1]:
        return False
    return pom_palindrom(r[1:-1])

def palindrom(retazec):
    return pom_palindrom(retazec.lower().replace(' ',''))

print(palindrom('Jelenovi Pivo Nelej'))
```

    to isté zapísané kompaktnejšie:

```
def palindrom(retazec):
    def pal(r):
        return len(r) <= 1 or r[0]==r[-1] and pal(r[1:-1])
    return pal(retazec.lower().replace(' ',''))

print(palindrom('Jelenovi Pivo Nelej'))
```

18. na prednáške bola rekurzívna funkcia pre výpočet členov fibonacciho postupnosti:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

- tento algoritmus je tak pomalý, že pomocou neho nevypočítame ani `fib(100)`
- opravte ho tak, aby namiesto jedného člena postupnosti vracal dvojicu: (i-ty člen, i+1-ty člen) - vaša rekurzia by toto mala využiť pre zisťovanie ďalšieho člena postupnosti (stačí zistiť `fib(n-1)` a z neho vieme vypočítať `fib(n)` aj `fib(n+1)`

```
>>> for i in range(10):
        print(i, fib(i))

0 (0, 1)
1 (1, 1)
2 (1, 2)
3 (2, 3)
4 (3, 5)
5 (5, 8)
6 (8, 13)
7 (13, 21)
8 (21, 34)
9 (34, 55)
```

riešenie:

```
def fib(n):
    if n == 0:
        return 0,1
    f = fib(n-1)
    return f[1], f[0]+f[1]

for i in range(10):
    print(i, fib(i))
```

KAPITOLA 7

Cvičenie

7.1 Dvojrozmerné polia

1. zadefinujte funkciu `vypis(pole,sirka=4)`, ktorá vypíše dvojrozmerné pole do riadkov, pričom každý prvok je formátovaný na zadanú šírku, napr. `'{:4}'.format(hodnota)` - funkcia musí nastaviť zadanú šírku:

```
>>> vypis([[1, 6, 3.14], [0.5, 1.5, 2.5]], 5)
  1     6  3.14
0.5   1.5   2.5
```

riešenie:

```
def vypis(pole, sirka=4):
    for r in pole:
        for p in r:
            print('{:{}}'.format(p, sirka), end=' ')
        print()
```

2. zadefinujte funkcie `max2(pole)`, `min2(pole)` a `sum2(pole)`, ktoré zistia najväčší prvok, najmenší prvok a súčet všetkých prvkov poľa - využite štandardné funkcie `max()`, `min()`, `sum()`

```
>>> pole = [[1, 6, 3.14], [0.5, 1.5, 2.5]]
>>> max2(pole)
6
>>> min2(pole)
0.5
>>> sum2(pole)
14.64
```

riešenie:

```
def max2(pole):
    vysl = max(pole[0])
    for riadok in pole[1:]:
        vysl = max(vysl, max(riadok))
    return vysl

def min2(pole):
    vysl = min(pole[0])
    for riadok in pole[1:]:
        vysl = min(vysl, min(riadok))
```

```
    return vysl

def sum2(pole):
    vysl = 0
    for riadok in pole:
        vysl += sum(riadok)
    return vysl
```

3. zadefinujte funkciu `ocisluj(pole)`, ktorá zmení všetky prvky dvojrozmerného poľa tak, že ich postupne prechádza po stĺpcoch a čísluje ich celými číslami od 0, riadky poľa nemusia mať rovnakú dĺžku

```
>>> ab = [[1, 1], [1, 1, 1], [1], [1, 1, 1, 1]]
>>> ocisluj(ab)
>>> vypis(ab, 2)
 0  4
 1  5  7
 2
 3  6  8  9
```

riešenie:

```
def ocisluj(pole):
    j, poc, b = 0, 0, True
    while b:
        b = False
        for i in range(len(pole)):
            if j < len(pole[i]):
                pole[i][j] = poc
                poc += 1
                b = True
        j += 1

ab = [[1,1],[1,1,1],[1],[1,1,1,1]]
ocisluj(ab)
vypis(ab, 2)
```

4. zadefinujte funkciu `nafukni(pole,priadkov,pstlpcov,hodn=None)`, ktorá doplní riadky dvojrozmerného poľa tak, aby mal každý `pstlpcov` prvkov (riadok doplní hodnotami `hodn`), pričom dlhšie riadky skráti na príslušnú šírku; ak je riadkov viac ako `priadkov`, tak zvyšné riadky vyhodí, ak ich je menej ako `priadkov`, tak ich doplní chýbajúcimi s hodnotami `hodn`

```
>>> pole = [[1, 1], [1, 1, 1], [1], [1, 1, 1, 1]]
>>> nafukni(pole, 5, 3, 2)
>>> vypis(pole, 2)
 1 1 2
 1 1 1
 1 2 2
 1 1 1
 2 2 2
```

riešenie:

```
def nafukni(pole, priadkov, pstlpcov, hodn=None):
    del pole[priadkov:]
    for i in range(len(pole)):
        del pole[i][pstlpcov:]
        pole[i].extend([hodn] * (pstlpcov-len(pole[i])))
```

```
    while len(pole) < priadkov:
        pole.append([hodn] * pstlpcov)
```

alebo

```
def nafukni(pole, priadkov, pstlpcov, hodn=None):
    p = []
    for i in range(priadkov):
        p.append([hodn] * pstlpcov)
    for i in range(priadkov):
        for j in range(pstlpcov):
            if i < len(pole) and j < len(pole[i]):
                p[i][j] = pole[i][j]
    pole[:] = p
```

5. zadefinujte funkciu `sucet(pole1,pole2)`, kde obe polia majú rovnaké rozmery, vytvorí nové pole, v ktorom má každý prvok súčet dvoch zodpovedajúcich prvkov v dvoch poliach

```
>>> s = sucet([[1,2],[3,4]], [[0,5],[-4,0]])
>>> vypis(s, 2)
 1 7
-1 4
```

riešenie:

```
def sucet(pole1, pole2):
    vysl = []
    for i in range(len(pole1)):
        riadok = []
        for j in range(len(pole1[i])):
            riadok.append(pole1[i][j] + pole2[i][j])
        vysl.append(riadok)
    return vysl
```

6. zadefinujte funkciu `preklop(pole)`, ktorá vráti nové pole, ktoré je preklopením pôvodného podľa hlavnej uhlopriečky; predpokladajte, že riadky poľa majú rovnakú dĺžku

```
>>> x = preklop([[1,2,3],[4,5,6]])
>>> vypis(x, 2)
 1 4
 2 5
 3 6
```

riešenie:

```
def preklop(pole):
    vysl = []
    for j in range(len(pole[0])):
        riadok = []
        for i in range(len(pole)):
            riadok.append(pole[i][j])
        vysl.append(riadok)
    return vysl
```

7. zadefinujte funkciu `posun(pole)`, ktorá posunie riadky poľa o jeden nahor, pričom prvý riadok sa stane posledným; funkcia nič nevracia ani nevypisuje, funkcia len modifikuje parameter `pole`

```
>>> y = [[1,2,3],[4,5,6],[7,8,9]]
>>> posun(y)
>>> vypis(y, 2)
 4 5 6
 7 8 9
 1 2 3
```

riešenie:

```
def posun(pole):
    r = pole[0]
    pole[:-1] = pole[1:]
    pole[-1] = r
```

čo je to isté ako

```
def posun(pole):
    pole[:-1], pole[-1] = pole[1:], pole[0]
```

alebo inak

```
def posun(pole):
    pole[:] = pole[1:] + pole[0:1]
```

alebo ešte inak

```
def posun(pole):
    pole.append(pole.pop(0))
```

8. zadefinujte funkciu `pascalov_trojuholnik(n)`, ktorá vygeneruje prvých `n` riadkov pascalovho trojuholníka a uloží ich do dvojrozmerného poľa

```
>>> pt = pascalov_trojuholnik(5)
>>> pt
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

riešenie:

```
def pascalov_trojuholnik(n):
    vysl = []
    for i in range(1, n+1):
        riadok = [1]
        for j in range(1, i):
            riadok.append(riadok[-1] * (i-j) // j)
        vysl.append(riadok)
    return vysl
```

9. zadefinujte funkciu `zapis(pole,subor)`, ktorá zapíše dvojrozmerné pole do súboru so zadaným menom (môžete využiť modul `json`)

```
>>> zapis([[1,2,3],[4,5,6]], 'pole.txt')
>>> open('pole.txt').read()
'[[1, 2, 3], [4, 5, 6]]'
```

riešenie:

```
def zapis(pole, subor):
    import json
    with open(subor, 'w') as f:
        json.dump(pole, f)
```

10. zadefinujte funkciu `citaj(subor)`, ktorá vráti dvojrozmerné pole prečítané zo súboru vo formáte z (9) úlohy

```
>>> z = citaj('pole.txt')
>>> z
[[1, 2, 3], [4, 5, 6]]
```

riešenie:

```
def citaj(subor):
    import json
    with open(subor) as f:
        return json.load(f)
```

7.2 Asociatívne polia (dict)

11. zadefinujte asociatívne pole `vaha` s aspoň 10 prvkami, v ktorom sú váhy zvierat v zoo, (napr. slon 5000, opica 18, žirafa 900, lev 200, ...)

```
>>> vaha = {...}
```

riešenie:

```
vaha = {'opica': 18, 'slon': 5000, 'zirafa': 900, 'papagaj': 1,
↪'antilopa': 400,
        'lev': 200, 'dikobraz': 20, 'pyton': 200, 'tava': 600, 'vlk':
↪ 50}
```

12. zadefinujte funkciu `len_prvky(apole,viac=True,hodnota=100)` s parametrom asociatívne pole `apole`, ktorá z neho vypíše tie kľúče (zvieratá) aj s hodnotami (váhami), ktorých asociovaná hodnota je väčšia (pre `viac=True` alebo menšia pre `viac=False`) ako zadaná `hodnota`

```
>>> vaha = {...}
>>> len_prvky(vaha, False, 20)    # len s váhou menšou ako 20
```

riešenie:

```
def len_prvky(apole, viac=True, hodnota=100):
    for k, h in apole.items():
        if viac and h > hodnota or not viac and h < hodnota:
            print(k, h)
```

13. zadefinujte funkciu `vyber(...)` podobná predchádzajúcej, ktorá nič nevypisuje, ale vytvorí nové asociatívne pole s dvojicami, ktoré spĺňajú danú podmienku; pôvodné asociatívne pole ostane bez zmeny

```
>>> vaha = {...}
>>> vaha1 = vyber(vaha, False, 20)    # len s váhou menšou ako 20
>>> vaha1
{...}
```

riešenie:

```
def vyber(apole, viac=True, hodnota=100):
    vysl = {}
    for k, h in apole.items():
        if viac and h > hodnota or not viac and h < hodnota:
            vysl[k] = h
    return vysl
```

14. zadefinujte funkciu `kresli(apole)`, ktorá nakreslí útvary zadané v asociatívnom poli (chýbajúce parametre pošlite do príslušnej funkcie s hodnotou `None`)

```
utvary = [
    {'tvar':'rectangle', 'xy':[100, 100, 200, 150], 'fill':'red'},
    {'tvar':'oval', 'xy':[100, 100, 200, 150], 'fill':'yellow'},
    {'tvar':'text', 'xy':[150, 50], 'fill':'red', 'text':'Python', 'font':
↪'arial 30'},
    {'tvar':'oval', 'xy':[125, 150, 175, 200]},
    {'tvar':'text', 'xy':[150, 175], 'text':'HURA'}]

kresli(utvary)
```

riešenie:

```
def kresli(utvary):
    for u in utvary:
        if u['tvar'] == 'rectangle':
            canvas.create_rectangle(u['xy'],fill=u.get('fill'))
        elif u['tvar'] == 'oval':
            canvas.create_oval(u['xy'],fill=u.get('fill'))
        elif u['tvar'] == 'text':
            canvas.create_text(u['xy'],fill=u.get('fill'),text=u[
↪'text'],font=u.get('font'))

import tkinter
canvas = tkinter.Canvas()
canvas.pack()
kresli(utvary)
```

15. zadefinujte funkciu `otoc(apole)`, ktorá vytvorí nové asociatívne pole, v ktorom prevráti každú dvojicu `kluc<->hodnota`; predpokladáme, že každá hodnota je len pri jednom kľúči

```
>>> otoc({'prvy':1, 'druhy':2})
{1: 'prvy', 2: 'druhy'}
```

riešenie:

```
def otoc(apole):
    vysl = {}
    for k, h in apole.items():
        vysl[h] = k
    return vysl
```

16. zadefinujte funkciu `otoc2(apole)`, ktorá robí to isté ako (15), ale novovytvorené hodnoty sú n-tice (`tuple`) a ak dva (a viac) kľúčov majú rovnakú hodnotu, pri otočení sú všetky kľúče v jednej n-tici

```
>>> otoc2({'prvy':1, 'druhy':2, 'treti':1})
{1: ('prvy', 'treti'), 2: ('druhy',)}
```

riešenie:

```
def otoc2(apole):
    vysl = {}
    for k, h in apole.items():
        vysl[h] = vysl.get(h, ()) + (k,)
    return vysl
```

17. máme dané dvojrozmerné pole celých čísel, funkcia `vsetky_rozne(pole)` zistí, či sú všetky riadky tohto poľa navzájom rôzne; využite asociatívne pole (do asociatívneho poľa môžeme ukladať napr. počty výskytov jednotlivých riadkov)

```
>>> a = [[1,2,3],[4,5],[4,5,6]]
>>> b = [[1,2,3],[4,5],[1,2,3],[4,5,6]]
>>> vsetky_rozne(a)
True
>>> vsetky_rozne(b)
False
```

riešenie:

```
def vsetky_rozne(pole):
    pom = {}
    for riadok in pole:
        if tuple(riadok) in pom:
            return False
        pom[tuple(riadok)] = 1
    return True
```

18. dané je asociatívne pole `sifra[písmeno]=písmeno`, v ktorom každému písmenu (od 'a' po 'z') zodpovedá nejaké iné písmeno; zadefinujte funkciu `zasifruj(apole,text)`, ktorá pomocou šifry v prvom parametri `apole` zašifruje zadaný `text`;nepísmenové znaky v tomto text nemení; šifrovanie textu znamená, že každé písmeno v texte sa nahradí písmenom z asociatívneho poľa

```
>>> sifra = {'a':'b', 'b':'c', 'c':'d', 'd':'a', 'e':'f', 'f':'g', 'g':'h', .
 ↪..}
>>> zasifruj(sifra, 'beda gaga')
'cfab hbhb'
```

riešenie:

```
def zasifruj(apole, text):
    vysl = ''
    for znak in text:
        vysl += apole.get(znak, znak)
    return vysl

sifra = {'a':'b', 'b':'c', 'c':'d', 'd':'a', 'e':'f', 'f':'g', 'g':'h
 ↪', 'h':'e',
         'i':'j', 'j':'k', 'k':'l', 'l':'i', 'm':'n', 'n':'o', 'o':'p
 ↪', 'p':'m',
         'q':'r', 'r':'s', 's':'t', 't':'q', 'u':'v', 'v':'w', 'w':'x
 ↪', 'x':'u',
         'y':'z', 'z':'y'}

print(zasifruj(sifra, 'programujem v pythone'))
```

19. zadefinujte funkciu `rozsifruj(apole,text)`, ktorá dostáva šifrovacie asociatívne pole `apole` a zašifrovaný text z úlohy (18); funkcia tento text rozšifruje, využije funkciu `otoc()` z úlohy (15)

```
>>> rozsifruj(sifra, 'cfad hbhb')
'beda gaga'
```

riešenie:

```
def rozsifruj(apole, text):
    vysl = ''
    novy = otoc(apole)
    for znak in text:
        vysl += novy.get(znak, znak)
    return vysl
```

čo je to isté ako:

```
def rozsifruj(apole, text):
    return zasifruj(otoc(apole), text)
```

20. vytvorte frekvenčnú tabuľku výskytov za sebou idúcich dvojíc písmen: funkcia `pocty_dvojic(text)` vráti asociatívne pole počítadiel výskytov, napr. pre text `'jazyk python'` zaeviduje 9 dvojíc písmen: `'ja'`, `'az'`, `'zy'`, `'yk'`, `'py'`, `'yt'`, ... ; otestujte aj pre nejaký veľký súbor a tiež zistite 10 najčastejších prípadov

```
>>> p = pocty_dvojic(open('twain.txt').read())
>>> p
{'hr': 150, 'uj': 1, 'sh': 809, 'gn': 39, 'ek': 24,  ...}
>>> # 10 najcastejsich
[(6697, 'he'), (6169, 'th'), (4804, 'an'), (4149, 'nd'), (3540, 'in'),
 (3229, 'ou'), (3005, 'er'), (2288, 'to'), (2199, 'wa'), (2125, 're')]
```

riešenie:

```
def pocty_dvojic(text):
    vysl = {}
    pred = ' '
    for znak in text:
        if 'a'<=pred<='z' and 'a'<=znak<='z':
            vysl[pred+znak] = vysl.get(pred+znak,0)+1
        pred = znak
    return vysl

pocet = pocty_dvojic(open('twain.txt').read())
p = []
for k, h in pocet.items():
    p.append((h, k))
p.sort(reverse=True)
print(p[:10])
```

21. hádžeme dvoma kockami (s číslami od 1 do 6): evidujeme počty výskytov dvojíc čísel, pričom pre hod dvoch kociek `(a,b)`, ak `a>b` sa zaeviduje ako `(b,a)`; hodíme `n`-krát a funkcia `pocty_kociek(n)` vráti asociatívne pole, v ktorom pre všetky hody spočíta počet prípadov; hádžeme náhodným generátorom

```
>>> pocet = pocty_kociek(100)
>>> pocet
{(1, 2): 3, (2, 6): 5, (5, 5): 4, (4, 5): 7, (1, 5): 6, ...}
```

riešenie:

```
import random

def pocty_kociek(n):
    vysl = {}
    for i in range(n):
        a,b = random.randint(1,6),random.randint(1,6)
        if a>b:
            a,b = b,a
        vysl[a,b] = vysl.get((a,b),0)+1
    return vysl

pocet = pocty_kociek(100)
print(pocet)
```

KAPITOLA 8

Cvičenie

8.1 Triedy a objekty

8.1.1 trieda Zlomok

1. zadefinujte triedu `Zlomok`, s dvoma atribútmi `citatel` a `menovatel`

```
class Zlomok:
    def __init__(self, ...):
        ...
```

aby fungovalo, napr.

```
>>> a = Zlomok(3, 4)
>>> a.citatel
3
>>> a.menovatel
4
>>> b = Zlomok()
>>> b.citatel
0
>>> b.menovatel
1
```

riešenie:

```
class Zlomok:
    def __init__(self, c, m):
        self.citatel = c
        self.menovatel = m
```

8.2 Triedy a metódy

2. do triedy `Zlomok` dopíšte metódu `__repr__()`, vďaka ktorej sa zlomok vypíše v peknom tvare, napr.

```
>>> a
3÷4
>>> repr(a)
'3÷4'
```

namiesto znaku '÷' si môžete zvoliť ľubovoľný vhodný symbol

riešenie:

```
class Zlomok:
    def __init__(self, c, m):
        self.citatel = c
        self.menovatel = m

    def __repr__(self):
        return '{}÷{}'.format(self.citatel, self.menovatel)
```

3. upravte inicializáciu `__init__()` triedy `Zlomok` tak, aby sa zlomok uložil v základnom tvare, napr.

```
>>> b = Zlomok(24, 36)
>>> b
2÷3
```

Môžete využiť Euklidov algoritmus z niektorých predchádzajúcich cvičení.

riešenie:

```
class Zlomok:
    def __init__(self, c, m):

        def nsd(a, b):
            while b > 0:
                a, b = b, a%b
            return a

        delitel = nsd(c, m)
        self.citatel = c // delitel
        self.menovatel = m // delitel
```

4. do triedy `Zlomok` dopíšte metódu `sucet()`, ktorá vráti (`return`) novú inštanciu triedy `Zlomok` a táto bude súčtom samotného zlomku (`self`) a nejakého iného zlomku, napr.

```
>>> z1 = Zlomok(1, 2)
>>> z2 = Zlomok(1, 3)
>>> z3 = z1.sucet(z2)
>>> z3
5÷6
```

riešenie:

```
class Zlomok:
    ...

    def sucet(self, iny):
        a, b = self.citatel, self.menovatel
        c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d + b*c, b*d)
```

5. metódu `sucet()` triedy `Zlomok` upravte tak, aby fungovala nielen s parametrom typu `Zlomok`, ale aj s celým číslom, napr.

```
>>> z2 = Zlomok(1, 3)
>>> z2.sucet(2)
7÷3
```

vo funkcii môžete testovať typ parametra, napr. `if type(param) == int:`

riešenie:

```
class Zlomok:
    ...

    def sucet(self, iny):
        a, b = self.citatel, self.menovatel
        if type(iny) == int:
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d + b*c, b*d)
```

6. do triedy `Zlomok` dopíšte ďalšie metódy (zrejme všetky sú pravé funkcie):
   - metóda `rozdiel()` vytvorí nový zlomok s rozdielom dvoch zlomkov
   - metóda `sucin()` vytvorí nový zlomok so sucinom dvoch zlomkov
   - metóda `podiel()` vytvorí nový zlomok s podielom dvoch zlomkov
   - metóda `int()` vráti celé číslo, ktoré je celou časťou zlomku
   - metóda `float()` vráti desatinné číslo, ktoré je hodnotou zlomku

```
>>> z3 = Zlomok(3, 2)
>>> z3.int()
1
>>> z3.float()
1.5
>>> z3.sucin(Zlomok(4, 9))
2÷3
>>> Zlomok(3, 5).podiel(Zlomok(2, 3))
9÷10
```

riešenie:

```
class Zlomok:
    def __init__(self, c, m):

        def nsd(a, b):
            while b > 0:
                a, b = b, a%b
            return a

        delitel = nsd(c, m)
        self.citatel = c // delitel
        self.menovatel = m // delitel

    def __repr__(self):
        return '{}÷{}'.format(self.citatel, self.menovatel)

    def sucet(self, iny):
        a, b = self.citatel, self.menovatel
        if type(iny) == int:
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d + b*c, b*d)
```

```
    def rozdiel(self, iny):
        a, b = self.citatel, self.menovatel
        if type(iny) == int:
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d - b*c, b*d)

    def sucin(self, iny):
        a, b = self.citatel, self.menovatel
        if type(iny) == int:
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*c, b*d)

    def podiel(self, iny):
        a, b = self.citatel, self.menovatel
        if type(iny) == int:
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d, b*c)

    def int(self):
        return self.citatel // self.menovatel

    def float(self):
        return self.citatel / self.menovatel
```

8.2.1 trieda Karticka

7. zadefinujte triedu `Karticka`, ktorá v grafickej ploche vykreslí farebný obdĺžnik s nejakým textom v strede, použite deklarácie:

```
class Karticka:
    font = 'arial 20 bold'
    canvas = None

    def __init__(self, text, x, y, sirka=100, vyska=30, farba='beige'):
        ...

import tkinter

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

k1 = Karticka('Python', 150, 100)
```

- vďaka tomuto zápisu sa do triedy dostáva **triedny** atribút `canvas` aj s priradenou grafickou plochou, každá inštancia tento atribút vidí ako `self.canvas` a preto sa kartička vykreslí už pri inicializácii
- tiež využite triedny atribút `font` pre font vypisovaného textu
- parametre `x` a `y` sú ľavým hodným vrcholom kartičky, pričom text bude na kartičke vycentrovaný

- parameter `farba` určuje farebné pozadie kreslenej kartičky (obdĺžnika)

riešenie:

```
class Karticka:
    font = 'arial 20 bold'
    canvas = None

    def __init__(self, text, x, y, sirka=100, vyska=30, farba='beige
↪'):
        self.canvas.create_rectangle(x, y, x+sirka, y+vyska,␣
↪fill=farba)
        self.canvas.create_text(x+sirka//2, y+vyska//2, text=text,␣
↪font=self.font)
```

8. zmeňte metódu `__init__()` tak, aby fungovala aj pre neurčené súradnice `x` a `y`:

```
class Karticka:
    font = 'arial 20 bold'

    def __init__(self, text, x=None, y=None, sirka=100, vyska=30, farba=
↪'beige'):
        ...

...
k1 = Karticka('Python')
k2 = Karticka('Pascal', farba='gray')
```

- metóda v prípade, že zistí, že `x` alebo `y` majú hodnotu `None`, vygeneruje náhodnú pozíciu kartičky, napr. `random.randrange(300)`

riešenie:

```
class Karticka:
    font = 'arial 20 bold'
    canvas = None

    def __init__(self, text, x=None, y=None, sirka=100, vyska=30,␣
↪farba='beige'):
        if x is None:
            x = random.randrange(300)
        if y is None:
            y = random.randrange(300)
        self.canvas.create_rectangle(x, y, x+sirka, y+vyska,␣
↪fill=farba)
        self.canvas.create_text(x+sirka//2, y+vyska//2, text=text,␣
↪font=self.font)
```

9. dopíšte metódy `zmen_text()` a `zmen_poz()`:

```
class Karticka:
    ...

    def zmen_text(self, text):
        ...

    def zmen_poz(self, x, y, sirka, vyska):
        ...
```

- pomocou metódy `zmen_text()` zmeníme text na kartičke
- pomocou metódy `zmen_poz()` presunieme kartičku (aj s textom) na novú pozíciu a meníme jej pri tom aj veľkosť

riešenie:

```
class Karticka:
    font = 'arial 20 bold'
    canvas = None

    def __init__(self, text, x=None, y=None, sirka=100, vyska=30,␣
↪farba='beige'):
        if x is None:
            x = random.randrange(300)
        if y is None:
            y = random.randrange(300)
        self.i1 = self.canvas.create_rectangle(x, y, x+sirka,␣
↪y+vyska, fill=farba)
        self.i2 = self.canvas.create_text(x+sirka//2, y+vyska//2,␣
↪text=text, font=self.font)

    def zmen_text(self, text):
        self.canvas.itemconfig(self.i2, text=text)

    def zmen_poz(self, x, y, sirka, vyska):
        self.canvas.coords(self.i1, x, y, x+sirka, y+vyska)
        self.canvas.coords(self.i2, x+sirka//2, y+vyska//2)
```

10. zmeňte metódu `zmen_poz()` tak, aby fungovala aj iba s dvoma parametrami: keď jej neurčíme šírku a výšku, tak mení iba `x` a `y`:

```
class Karticka:
    ...

    def zmen_poz(self, x, y, sirka=None, vyska=None):
        ...
```

riešenie:

```
class Karticka:
    ...

    def __init__(self, text, x=None, y=None, sirka=100, vyska=30,␣
↪farba='beige'):
        if x is None:
            x = random.randrange(300)
        if y is None:
            y = random.randrange(300)
        self.sirka, self.vyska = sirka, vyska
        self.i1 = self.canvas.create_rectangle(x, y, x+sirka,␣
↪y+vyska, fill=farba)
        self.i2 = self.canvas.create_text(x+sirka//2, y+vyska//2,␣
↪text=text, font=self.font)

    def zmen_poz(self, x, y, sirka=None, vyska=None):
        if sirka is None:
            sirka = self.sirka
        else:
```

```
            self.sirka = sirka
        if vyska is None:
            vyska = self.vyska
        else:
            self.vyska = vyska
        self.canvas.coords(self.i1, x, y, x+sirka, y+vyska)
        self.canvas.coords(self.i2, x+sirka//2, y+vyska//2)
```

11. dopíšte metódu `__repr__()` tak, aby sa takýto zodpovedajúci reťazec dal použiť na opätovné vygenerovanie kartičky, napr.

```
>>> k = Karticka('hello', 200, 100, 150, 50, 'yellow')
>>> repr(k)
"Karticka('hello', 200, 100, 150, 50, 'yellow')"
```

riešenie:

```
class Karticka:
    font = 'arial 20 bold'
    canvas = None

    def __init__(self, text, x=None, y=None, sirka=100, vyska=30,␣
↪farba='beige'):
        if x is None:
            x = random.randrange(300)
        if y is None:
            y = random.randrange(300)
        self.text = text
        self.x, self.y = x, y
        self.sirka, self.vyska = sirka, vyska
        self.farba = farba
        self.i1 = self.canvas.create_rectangle(x, y, x+sirka,␣
↪y+vyska, fill=farba)
        self.i2 = self.canvas.create_text(x+sirka//2, y+vyska//2,␣
↪text=text, font=self.font)

    def __repr__(self):
        return 'Karticka({}, {}, {}, {}, {}, {})'.format(
            repr(self.text), self.x, self.y, self.sirka, self.vyska,
            repr(self.farba))

    def zmen_text(self, text):
        self.text = text
        self.canvas.itemconfig(self.i2, text=text)

    def zmen_poz(self, x, y, sirka=None, vyska=None):
        self.x, self.y = x, y
        if sirka is None:
            sirka = self.sirka
        else:
            self.sirka = sirka
        if vyska is None:
            vyska = self.vyska
        else:
            self.vyska = vyska
        self.canvas.coords(self.i1, x, y, x+sirka, y+vyska)
        self.canvas.coords(self.i2, x+sirka//2, y+vyska//2)
```

12. vygenerujte niekoľko rôznych kartičiek (napr. `for i in range(10): Karticka('Python')`) a potom jednej z nich zmeňte font (zmeníte jej atribút `font` a znovu ju prekreslíte); zrejme pri generovaní kartičiek si musíte tieto kartičky niekam ukladať (napr. do poľa kartičiek)

    riešenie:

```
class Karticka:
    ...

    def zmen_font(self, font):
        self.font = font
        self.canvas.itemconfig(self.i2, font=font)

pole = []
for i in range(10):
    pole.append(Karticka('Python'))

pole[9].zmen_font('consolas 16')
```

8.2.2 trieda Kniha

13. zadefinujte triedu `Kniha` s atribútmi: `autori`, `titul`, `vydavatel`, `rok`, `cena`

```
class Kniha:

    def __init__(self, ...):
        ...

    def __repr__(self):
        ...
```

metódu `__repr__()` zadefinujte tak, aby sa z reťazca dal zrekonštruovať objekt Kniha, napr. v tvare:

```
>>> k = Kniha(['Lasica', 'Satinský'], 'Dialógy', 'Forza Music', 2008, 12.58)
>>> k
Kniha(['Lasica', 'Satinský'], 'Dialógy', 'Forza Music', 2008, 12.58)
```

riešenie:

```
class Kniha:

    def __init__(self, autori, titul, vydavatel, rok, cena):
        self.autori = autori
        self.titul = titul
        self.vydavatel = vydavatel
        self.rok = rok
        self.cena = cena

    def __repr__(self):
        return 'Kniha({}, {}, {}, {}, {})'.format(
            repr(self.autori),
            repr(self.titul),
            repr(self.vydavatel),
            self.rok,
            self.cena)
```

14. zadefinujte triedu `Obchod`, v ktorej sa bude pracovať s poľom inštancií triedy `Kniha`:

```
class Obchod:

    def __init__(self):
        self.pole = []

    def pridaj(self, kniha):          # pridá knihu do pol'a
        ...

    def citaj(self, meno_suboru):     # prečíta knihy zo súboru a pridá ich␣
↪do svojho pol'a
        ...

    def zapis(self, meno_suboru):     # zapíše všetky knihy do suboru (ich␣
↪``repr()``)
        ...

    def hladaj(self, polozka, hodnota):     # vypíše všetky knihy, ktoré␣
↪majú zadanú položku s danou hodnotou
        ...                                 # položka je jedna z 'autor',
↪'titul', 'vydavatel', 'rok', 'cena'
```

napr.

```
>>> obch = Obchod()
>>> obch.citaj('knihy.txt')
>>> obch.hladaj('autor', 'Lasica')
Kniha(['Lasica', 'Satinský'], 'Dialógy', 'Forza Music', 2008, 12.58)
>>> obch.hladaj('rok', 2015)      # takú knihu nemá, nevypíše nič
>>>
```

riešenie:

```
class Obchod:

    def __init__(self):
        self.pole = []

    def pridaj(self, kniha):          # pridá knihu do pol'a
        self.pole.append(kniha)

    def citaj(self, meno_suboru):     # prečíta knihy zo súboru a␣
↪pridá ich do svojho pol'a
        with open(meno_suboru, encoding='utf-8') as f:
            self.pole = []
            r = f.readline()
            while r:
                a = r.strip().split(';')
                t = f.readline().strip()
                v = f.readline().strip()
                r = int(f.readline())
                c = float(f.readline())
                self.pridaj(Kniha(a, t, v, r, c))
                r = f.readline()

    def zapis(self, meno_suboru):     # zapíše všetky knihy do␣
↪suboru (ich ``repr()``)
        with open(meno_suboru, 'w', encoding='utf-8') as f:
            for k in self.pole:
```

```
                print(';'.join(k.autori), file=f)
                print(k.titul, file=f)
                print(k.vydavatel, file=f)
                print(k.rok, file=f)
                print(k.cena, file=f)

    def hladaj(self, polozka, hodnota):     # vypíše všetky knihy,␣
↪ktoré majú zadanú položku s danou hodnotou
        for kniha in self.pole:             # položka je jedna z
↪'autor', 'titul', 'vydavatel', 'rok', 'cena'
            if (polozka=='autor'     and hodnota in kniha.autori or
                polozka=='titul'     and kniha.titul==hodnota or
                polozka=='vydavatel' and kniha.vydavatel==hodnota or
                polozka=='rok'       and kniha.rok==hodnota or
                polozka=='cena'      and kniha.cena==hodnota):
                print(kniha)
```

alebo s využitím knižnice json

```
import json

class Obchod:

    def __init__(self):
        self.pole = []

    def pridaj(self, kniha):            # pridá knihu do pol'a
        self.pole.append(kniha)

    def citaj(self, meno_suboru):       # prečíta knihy zo súboru a␣
↪pridá ich do svojho pol'a
        with open(meno_suboru, encoding='utf-8') as f:
            self.pole = []
            for k in json.load(f):
                a, t, v, r, c = k
                self.pridaj(Kniha(a, t, v, r, c))

    def zapis(self, meno_suboru):       # zapíše všetky knihy do␣
↪suboru (ich ``repr()``)
        with open(meno_suboru, 'w', encoding='utf-8') as f:
            p = []
            for k in self.pole:
                p.append([k.autori, k.titul, k.vydavatel, k.rok, k.
↪cena])
            json.dump(p, f)

    def hladaj(self, polozka, hodnota):     # vypíše všetky knihy,␣
↪ktoré majú zadanú položku s danou hodnotou
        for kniha in self.pole:             # položka je jedna z
↪'autor', 'titul', 'vydavatel', 'rok', 'cena'
            if (polozka=='autor'     and hodnota in kniha.autori or
                polozka=='titul'     and kniha.titul==hodnota or
                polozka=='vydavatel' and kniha.vydavatel==hodnota or
                polozka=='rok'       and kniha.rok==hodnota or
                polozka=='cena'      and kniha.cena==hodnota):
                print(kniha)
```

testujeme:

```
obch = Obchod()
obch.pridaj(Kniha(['Lasica', 'Satinský'], 'Dialógy', 'Forza Music',␣
↪2008, 12.58))
obch.pridaj(Kniha(['Satinský'], 'Chlapci z Dunajskej',
↪'Vydavatel'stvo PT', 2012, 7.90))
obch.pridaj(Kniha(['Lasica'], 'Bodka', 'Forza Music', 2007, 9.42))
obch.zapis('knihy.txt')
obch.citaj('knihy.txt')
obch.hladaj('autor', 'Lasica')
obch.hladaj('cena', 7.9)
```

8.2.3 trieda Mnozina

15. zadefinujte triedu `Mnozina`, ktorá bude realizovať množinu celých čísel **pomocou asociatívneho poľa**:

```
class Mnozina:

    def __init__(self, inic=None):
        self.pole = {}
        ...

    def __repr__(self):
        ...

    def pridaj(self, cislo):
        ...
```

- ak metóda `__init__()` dostane ako parameter nejakú postupnosť celých čísel (napr. `list`, `range()`, ...), tak táto sú počiatočnými hodnotami množiny - postupne tieto čísla pridá do množiny (pomocou metódy `pridaj()`)
- metóda `__repr__()` vráti množinu v čitateľnom tvare, napr. ako `'{2,3,5,7,11}'` - na poradí vypísaných prvkov nezáleží
- metóda `pridaj()` pridá do poľa ďalší prvok (ako kľúč), pričom hodnota môže byť ľubovoľná napr. `None`
- Python obsahuje štandardný typ `set`, ktorý realizuje typ množina - v tejto triede by ste ho nemali používať

riešenie:

```
class Mnozina:

    def __init__(self, inic=None):
        self.pole = {}
        if inic:
            for i in inic:
                self.pridaj(i)

    def __repr__(self):
        return repr(tuple(self.pole))

    def pridaj(self, cislo):
        self.pole[cislo] = None

import random
```

```
m = Mnozina(range(7,25,3))
for i in range(20):
    m.pridaj(random.randint(10, 29))
print(m)
```

15. do triedy `Mnozina` dodefinujte ďalšie metódy:
    - metóda `vyhod()` vyhodí z množiny dané číslo
    - metóda `zisti()` zistí, či sa dané číslo nachádza v množine (vráti `True` alebo `False`)
    - metóda `pocet()` vráti počet prvkov v množine
    - metóda `min()` vráti najmenšie číslo v množine
    - metóda `max()` vráti najväčšie číslo v množine
    - metóda `zjednotenie()` pridá prvky z inej množiny do samotnej množiny
    - metóda `prienik()` nechá v množine len tie prvky, ktoré sa **nachádzajú** aj v inej množine
    - metóda `rozdiel()` nechá v množine len tie prvky, ktoré sa **nenachádzajú** v inej množine

```
>>> m1 = Mnozina((2, 3, 5, 11, 13))
>>> m1.pridaj(7)
>>> m1
{2, 3, 5, 7, 11, 13}
>>> m1.zjednotenie(Mnozina(range(8, 15, 3)))
>>> m1
{2, 3, 5, 7, 8, 11, 13, 14}
>>> m1.prienik(Mnozina(range(1, 20, 2)))
>>> m1
{3, 5, 7, 11, 13}
```

riešenie:

```
class Mnozina:

    def __init__(self, inic=None):
        self.pole = {}
        if inic:
            for i in inic:
                self.pridaj(i)

    def __repr__(self):
        return repr(tuple(sorted(self.pole)))

    def pridaj(self, cislo):
        self.pole[cislo] = None

    def vyhod(self, cislo):
        del self.pole[cislo]

    def zisti(self, cislo):
        return cislo in self.pole

    def pocet(self):
        return len(self.pole)

    def min(self):
        return min(self.pole)
```

```
    def max(self):
        return max(self.pole)

    def zjednotenie(self, ina):
        for i in ina.pole:
            self.pridaj(i)

    def prienik(self, ina):
        pom, self.pole = self.pole, {}
        for i in ina.pole:
            if i in pom:
                self.pridaj(i)

    def rozdiel(self, ina):
        for i in ina.pole:
            if i in self.pole:
                self.vyhod(i)


m1 = Mnozina((2, 3, 5, 11, 13))
m1.pridaj(7)
print(m1)
m1.zjednotenie(Mnozina(range(8, 15, 3)))
print(m1)
m1.prienik(Mnozina(range(1, 20, 2)))
print(m1)
```

KAPITOLA 9

Cvičenie

9.1 Triedy a dedičnosť

1. zadefinujte triedu `MojaTurtle`, ktorá bude mať novú metódu `stvorec(velkost,farba=None)`; táto metóda nakreslí štvorec so stranou `velkost`, pričom počiatočná pozícia korytnačky je v strede štvorca (berie sa do úvahy aj natočenie korytnačky); nakreslený štvorec bude vyplnený zadanou farbou, resp. pre nezadanú farbu (t.j. `None`) použije náhodnú farbu; korytnačka po dokreslení štvorca ostane v jeho strede s rovnakým natočením

```
class MojaTurtle(...):
```

   - otestujte nakreslením 20 štvorcov na náhodných pozíciách

   riešenie:

```
import turtle
import random

class MojaTurtle(turtle.Turtle):
    def stvorec(self, velkost, farba=None):
        if farba is None:
            farba = '#{:06x}'.format(random.randrange(256**3))
        self.pu()
        self.fd(velkost/2)
        self.lt(90)
        self.pd()
        self.fillcolor(farba)
        self.begin_fill()
        for i in range(4):
            self.fd(velkost/2)
            self.lt(90)
            self.fd(velkost/2)
        self.end_fill()
        self.pu()
        self.rt(90)
        self.bk(velkost/2)
        self.pd()


t = MojaTurtle()
t.speed(0)
for i in range(20):
```

```
    t.pu()
    t.setpos(random.randint(-300, 300), random.randint(-300, 300))
    t.seth(random.randrange(360))
    t.pd()
    t.stvorec(40)
```

2. do triedy z úlohy (1) dodefinujte metódu `stvorce(pocet)`, ktorá nakreslí zadaný počet štvorcov: všetky s rovnakým stredom a ich veľkosti strán budú: 10, 20, 30, ... (strany štvorcov budú navzájom rovnobežné); využite metódu `stvorec()` z (1)

```
class MojaTurtle(...):
```

   - otestujte `stvorce(2)`, `stvorce(5)` aj `stvorce(10)`
   - zrejme štvorce kreslite tak, aby aj menšie boli vidieť po nakreslení väčších

   riešenie:

```
import turtle
import random

class MojaTurtle(turtle.Turtle):
    ...

    def stvorce(self, pocet):
        for velkost in range(pocet*10, 0, -10):
            self.stvorec(velkost)
            #self.lt(10)

t = MojaTurtle()
t.speed(0)
t.lt(20)
t.stvorce(20)
```

3. do triedy z úlohy (1) dodefinujte metódu `stvorce2(velkost)`, ktorá pomocou metódy `stvorec()` nakreslí dva vpísané štvorce: väčší so stranou `velkost` a ten druhý tak, že jeho vrcholy budú ležať v stredoch strán väčšieho

```
class MojaTurtle(...):
```

   - metódu otestujte

   riešenie:

```
import turtle
import random
from math import sqrt

class MojaTurtle(turtle.Turtle):
    ...

    def stvorce2(self, velkost, pocet=2):
        for i in range(pocet):
            self.stvorec(velkost)
            self.lt(45)
            velkost *= sqrt(2) / 2

t = MojaTurtle()
t.speed(0)
```

```
t.lt(20)
t.stvorce2(200, 5)
```

4. do triedy `MojaTurtle` dopíšte metódu `strom()` (môžete použiť strom z prednášky (17), ktorá nakreslí príslušný strom, ale v každom liste stromu (triviálny prípad rekurzie) nakreslí malý farebný štvorec - použite metódu `stvorec(10)`

```
class MojaTurtle(...):
```

   - metódu otestujte

   riešenie:

```
class MojaTurtle(turtle.Turtle):
    ...

    def strom(self, n, d):
        self.fd(d)
        if n > 0:
            self.lt(40)
            self.strom(n-1, d*0.6)
            self.rt(90)
            self.strom(n-1, d*0.7)
            self.lt(50)
        else:
            self.stvorec(10)
        self.bk(d)

turtle.delay(0)
t = MojaTurtle()
t.speed(0)
t.lt(90)
t.pu()
t.bk(200)
t.pd()
t.strom(6, 200)
```

5. z triedy `MojaTurtle` odvoďte novú triedu `MojaTurtle1`, ktorá bude rovné čiary (korytnačia metóda `fd()`) kresliť dvojnásobne dlhé

```
class MojaTurtle1(...):
```

   - napr. `fd(10)` takejto korytnačky nakreslí čiaru dvojnásobne dlhú, t.j. 20
   - otestujte kreslenie rekurzívneho stromu (metóda `strom()`) korytnačkou tohto typu (zrejme metóda `bk()` bude robiť malé problémy: buď prerobte aj tú, alebo ju nahraďte volaním `fd(-dlzka)`)

   riešenie:

```
class MojaTurtle(turtle.Turtle):
    ...

class MojaTurtle1(MojaTurtle):
    def fd(self, d):
        super().fd(2*d)

    def bk(self, d):
        super().bk(2*d)
```

```
turtle.delay(0)
t = MojaTurtle1()
t.speed(0)
t.lt(90)
t.pu()
t.bk(100)
t.pd()
t.strom(6, 100)
```

6. z triedy `MojaTurtle` odvoďte novú triedu `MojaTurtle2`, ktorá bude rovné čiary (korytnačia metóda `fd()`) kresliť cikcakom (prekrytá metóda `fd()` z prednášky (17)

```
class MojaTurtle2(...):
```

   - otestujte kreslenie farebného štvorca (metóda `stvorec()`) korytnačkou tohto typu
   - otestujte kreslenie rekurzívneho stromu s cikcak čiarami

   riešenie:

```
class MojaTurtle(turtle.Turtle):
    ...

class MojaTurtle1(MojaTurtle):
    ...

class MojaTurtle2(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)
            self.rt(120)
            super().fd(5)
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)

turtle.delay(0)
t = MojaTurtle2()
t.speed(0)
t.lt(90)
t.pu()
t.bk(100)
t.pd()
#t.stvorec(200)
t.strom(6, 200)
```

7. pozmeňte definíciu triedy `MojaTurtle2` tak, aby bola odvodená z `MojaTurtle1`

```
class MojaTurtle2(...):
```

   - teraz to otestujte podobne ako v úlohe (6)

   riešenie:

```
class MojaTurtle(turtle.Turtle):
    ...

class MojaTurtle1(MojaTurtle):
```

```
    ...

class MojaTurtle2(MojaTurtle1):
    ...

turtle.delay(0)
t = MojaTurtle2()
t.speed(0)
t.lt(90)
t.pu()
t.bk(100)
t.pd()
#t.stvorec(100)
t.strom(6, 100)
```

8. z triedy `MojaTurtle` odvoďte novú triedu `MojaTurtle3`, ktorá bude mať vymenené otáčanie vľavo a vpravo (opravíte korytnačie metódy `rt()` a `lt()`)

```
class MojaTurtle3(...):
```

   - otestujte kreslením rekurzívneho stromu, prípadne aj s cikcak čiarami (trieda bude odvodená od `MojaTurtle2`)

riešenie:

```
class MojaTurtle(turtle.Turtle):
    ...

class MojaTurtle1(MojaTurtle):
    ...

class MojaTurtle2(MojaTurtle1):
    ...

class MojaTurtle3(MojaTurtle2):
    def rt(self, uhol):
        super().lt(uhol)

    def lt(self, uhol):
        super().rt(uhol)

turtle.delay(0)
t = MojaTurtle3()
t.speed(0)
t.rt(90)
t.pu()
t.bk(100)
t.pd()
#t.stvorec(100)
t.strom(6, 100)
```

Cvičenie

10.1 Výnimky

1. funkcia `sucet_int(pole)` vypočíta súčet prvkov poľa, pričom predpokladá, že všetky prvky sú celé čísla; ak niektorý prvok nie je celé číslo, tak ho odignoruje (využite funkciu `isinstance()`)

```
>>> sucet_int([2,'3',4.0])
2
>>> sucet_int(['1','2','3'])
0
```

riešenie:

```
def sucet_int(pole):
    vysl = 0
    for prvok in pole:
        if isinstance(prvok, int):
            vysl += prvok
    return vysl
```

2. funkcia `cele(hodnota)` prevedie danú hodnotu na celé číslo, ak sa to nedá, funkcia vráti 0 (využite funkciu `int()` a odchytávanie výnimiek)

```
>>> cele(12.3)
12
>>> cele('13')
13
>>> cele(42)
42
>>> cele('12.3')
0
```

riešenie:

```
def cele(hodnota):
    try:
        return int(hodnota)
    except (TypeError, ValueError):
        return 0
```

3. funkcia `sucet_int2(pole)` vypočíta súčet prvkov poľa, pričom predpokladá, že všetky prvky sú celé čísla; ak niektorý prvok nie je celé číslo, tak sa ho snaží najprv prekonvertovať na celé, ak sa ani to nedá, tak tento

jeden prvok odignoruje (využite funkciu `cele(hodnota)` z predchádzajúceho príkladu)

```
>>> sucet_int2([2,'3',4.0,'päť'])
9
>>> sucet_int2(['1','2','3','1a','2b','3c'])
6
```

riešenie:

```
def sucet_int2(pole):
    vysl = 0
    for prvok in pole:
        vysl += cele(prvok)
    return vysl
```

4. funkcia `sucet(pole)` vypočíta súčet prvkov poľa, pričom predpokladá, že všetky prvky sú rovnakého typu ako prvý prvok poľa; ak je niektorý prvok iného typu ako prvý prvok, tak sa ho snaží najprv prekonvertovať na tento typ, ak sa ani to nedá, tak tento jeden prvok odignoruje

```
>>> sucet([2,'3',4.0,'päť'])
9
>>> sucet(['1',2,0.3,'abc'])
'120.3abc'
>>> sucet([[1,2],3,'4x'])
[1, 2, '4', 'x']
>>> sucet([(1,2),(3,4)])
(1, 2, 3, 4)
>>> sucet([])                # vráti None
>>>
```

riešenie:

```
def sucet(pole):
    if len(pole) == 0:
        return
    vysl = pole[0]
    for prvok in pole[1:]:
        try:
            vysl += type(pole[0])(prvok)
        except (TypeError, ValueError):
            pass
    return vysl
```

5. funkcia `desatinne(retazec)` zistí, či je daný reťazec desatinným číslom (`float`), funkcia vráti `True` alebo `False`

```
>>> desatinne('123')
True
>>> desatinne('22.7')
True
>>> desatinne('22/7')
False
```

riešenie:

```
def desatinne(retazec):
    try:
        float(retazec)
```

```
        return True
    except (TypeError, ValueError):
        return False
```

6. funkcia `existuje(meno_suboru)` zistí, či existuje súbor s daným menom, funkcia vráti ‘‘ True‘‘ alebo `False`, použite `try-except`

```
>>> existuje('abc.txt')
False
>>> existuje('C:/Python35/python.exe')
True
```

riešenie:

```
def existuje(meno_suboru):
    try:
        with open(meno_suboru):
            return True
    except (TypeError, OSError, FileNotFoundError):
        return False
```

7. v súbore sa nachádza nejaký text (slová oddelené medzerami), ktorý môže obsahovať aj čísla; funkcia `iba_cisla(meno_suboru)` vráti pole celých čísel, ktoré sa nachádzajú v tomto súbore, použite `try-except`

   - ak súbor obsahuje

```
farmar ma 15 ovci a 127 sliepok
pricom 18. februara bolo od -15 do +2 stupnov
```

   - volanie funkcie vráti

```
>>> print(iba_cisla('subor.txt'))
[15, 127, -15, 2]
```

riešenie:

```
def iba_cisla(meno_suboru):
    vysl = []
    with open(meno_suboru) as subor:
        for slovo in subor.read().split():
            try:
                cislo = int(slovo)
                vysl.append(cislo)
            except (TypeError, ValueError):
                pass
    return vysl
```

8. funkcia `hladaj(pole,hodnota)` hľadá prvý výskyt danej hodnoty v dvojrozmernom poli: vráti dvojicu (riadok, stĺpec) alebo spadne na chybe `ValueError`, ak sa tam táto hodnota nevyskytuje; riešte tak, aby sa využila metódu `index()` pre polia

```
def hladaj(pole, hodnota):
    for riadok in ... :
        ... pole[riadok].index(...) ...
```

napr.

```
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 4)
(1, 3)
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 0)
...
ValueError: 0 is not in list
```

riešenie:

```
def hladaj(pole, hodnota):
    for riadok in range(len(pole)):
        try:
            stlpec = pole[riadok].index(hodnota)
            return riadok, stlpec
        except ValueError:
            pass
    raise ValueError(repr(hodnota) + ' is not in list')
```

9. (*podobné zadanie ako úloha (8)*) funkcia `hladaj(pole,hodnota)` hľadá prvý výskyt danej hodnoty v dvojrozmernom poli: vráti dvojicu (riadok, stĺpec) alebo spadne na chybe `ValueError`, ak sa tam táto hodnota nevyskytuje; riešte tak, aby sa hľadalo dvomi vnorenými for-cyklami (pre každý riadok a pre každý stĺpec) - keď nájde prvý výskyt, vráti jej pozíciu

```
def hladaj(pole, hodnota):
    for riadok in ... :
        for stlpec in ... :
            if ... :
                return riadok, stlpec
    ...
```

napr.

```
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 4)
(1, 3)
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 0)
...
ValueError: 0 is not in list
```

riešenie:

```
def hladaj(pole, hodnota):
    for riadok in range(len(pole)):
        for stlpec in range(len(pole[riadok])):
            if pole[riadok][stlpec] == hodnota:
                return riadok, stlpec
    raise ValueError(repr(hodnota) + ' is not in list')
```

10. (*podobné zadanie ako úloha (9)*) funkcia `hladaj(pole,hodnota)` hľadá prvý výskyt danej hodnoty v dvojrozmernom poli: vráti dvojicu (riadok, stĺpec) alebo spadne na chybe `ValueError`, ak sa tam táto hodnota nevyskytuje; riešte tak, aby sa hľadalo dvomi vnorenými for-cyklami (pre každý riadok a pre každý stĺpec) - keď nájde prvý výskyt vyskočí z oboch for-cyklov a až potom vráti pozíciu - na vyskočenie využite vlastnú výnimku, ktorú zachytíte na konci funkcie

```
class Vyskoc(Exception): pass

def hladaj(pole, hodnota):
    for riadok in ... :
        for stlpec in ... :
```

```
            if ... :
                '''vyskoc z oboch cyklov'''
    ...
    return riadok, stlpec
```

napr.

```
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 4)
(1, 3)
>>> hladaj([[1,2,3],[1,2,3,4],[4,5]], 0)
...
ValueError: 0 is not in list
```

riešenie:

```
class Vyskoc(Exception): pass

def hladaj(pole, hodnota):
    try:
        for riadok in range(len(pole)):
            for stlpec in range(len(pole[riadok])):
                if pole[riadok][stlpec] == hodnota:
                    raise Vyskoc
        raise ValueError(repr(hodnota) + ' is not in list')
    except Vyskoc:
        pass
    return riadok, stlpec
```

10.2 Triedy a operácie 1

10.2.1 Operátory indexovania

11. zadefinujte vlastnú triedu `Pole` (nebude odvodená od `list`) s metódami tak, aby fungovalo:

```
>>> pole = Pole(5)
>>> pole
[None, None, None, None, None]
>>> for i in range(len(pole)):
        pole[i] = (i+3)**2

>>> for i in range(len(pole)):
        print(pole[i], end=' ')

9 16 25 36 49
>>> pole.append(99)
>>> pole
[9, 16, 25, 36, 49, 99]
>>> pole[2.1]
...
TypeError: index musi byt cele cislo
>>> pole[7] = 100
...
IndexError: index je mimo rozsahu pola
```

Zrejme budete musieť definovať niektoré magické metódy.

riešenie:

```
class Pole:
    def __init__(self, n):
        self.pole = [None]*n

    def __len__(self):
        return len(self.pole)

    def __getitem__(self, index):
        try:
            return self.pole[index]
        except TypeError:
            raise TypeError('index musi byt cele cislo')
        except IndexError:
            raise IndexError('index je mimo rozsahu pola')

    def __setitem__(self, index, hodnota):
        try:
            self.pole[index] = hodnota
        except TypeError:
            raise TypeError('index musi byt cele cislo')
        except IndexError:
            raise IndexError('index je mimo rozsahu pola')

    def append(self, hodnota):
        self.pole.append(hodnota)

    def __repr__(self):
        return repr(self.pole)
```

12. do triedy `Pole` doplňte dva atribúty `pocet_get` a `pocet_set`, ktoré budú mať pri inicializácii hodnotu 0; doplňte niektoré metódy triedy tak, aby každé zisťovanie hodnoty prvku poľa zvýšilo počítadlo `pocet_get` o 1 a každé nové priradenie do prvku poľa zvýšilo počítadlo `pocet_set` o 1, napr.

```
>>> pole = Pole(5)
>>> pole[0] = 1
>>> for i in range(1, len(pole)):
        pole[i] = 2*pole[i-1] + 1

>>> pole
[1, 3, 7, 15, 31]
>>> pole.pocet_get, pole.pocet_set
(4, 5)
```

riešenie:

```
class Pole:
    def __init__(self, n):
        self.pole = [None]*n
        self.pocet_get = self.pocet_set = 0

    def __len__(self):
        return len(self.pole)

    def __getitem__(self, index):
        try:
            self.pocet_get += 1
```

```
            return self.pole[index]
        except TypeError:
            raise TypeError('index musi byt cele cislo')
        except IndexError:
            raise IndexError('index je mimo rozsahu pola')

    def __setitem__(self, index, hodnota):
        try:
            self.pocet_set += 1
            self.pole[index] = hodnota
        except TypeError:
            raise TypeError('index musi byt cele cislo')
        except IndexError:
            raise IndexError('index je mimo rozsahu pola')

    def append(self, hodnota):
        self.pole.append(hodnota)

    def __repr__(self):
        return repr(self.pole)
```

13. do triedy `Pole` doplňte novú metódu tak, aby fungoval operátor `in`: tento operátor bude zisťovať prítomnosť hodnoty v poli pomocou algoritmu binárneho vyhľadávania (predpokladáme, že pole je usporiadané), napr.

```
>>> pole = Pole(0)
>>> for i in range(10):
        pole.append(i**2)

>>> pole
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> pole.pocet_get, pole.pocet_set
(0, 0)
>>> 49 in pole
True
>>> pole.pocet_get, pole.pocet_set
(4, 0)
```

všimnite si, že pri zisťovaní prítomnosti prvku v poli (operácia `in`) sa využila metóda na zisťovanie hodnoty, ktorá zvyšuje počítadlo `pocet_get`

riešenie:

```
class Pole:
    ...
    def __contains__(self, hodnota):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self[stred] > hodnota:
                kon = stred - 1
            elif self[stred] < hodnota:
                zac = stred + 1
            else:
                return True
        return False
```

14. do triedy `Pole` doplňte metódu `vloz(hodnota)`, ktorá predpokladá, že pole je utriedené a na správne miesto

vloží novú hodnotu, aby ostalo pole utriedené; metóda využije vlastné metódy na zisťovanie hodnoty prvku poľa, na zmenu hodnoty prvku poľa a na pridanie novej hodnoty na koniec poľa

```
>>> pole = Pole(0)
>>> for i in range(10):
        pole.append(i**2)

>>> pole
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> pole.vloz(20)
>>> pole
[0, 1, 4, 9, 16, 20, 25, 36, 49, 64, 81]
>>> pole.pocet_get, pole.pocet_set
(9, 6)
>>> pole.vloz(100)
>>> pole
[0, 1, 4, 9, 16, 20, 25, 36, 49, 64, 81, 100]
>>> pole.pocet_get, pole.pocet_set
(17, 7)
```

riešenie:

```
class Pole:
    ...
    def vloz(self, hodnota):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self[stred] > hodnota:
                kon = stred - 1
            elif self[stred] < hodnota:
                zac = stred + 1
            else:
                zac = stred
                break
        self.append(None)
        i = len(self)-1
        while i > zac:
            self[i] = self[i-1]
            i -= 1
        self[zac] = hodnota
```

Cvičenie

11.1 Zásobníky a rady

1. na začiatku máme prázdny zásobník, zistite, čo sa vypíše (skúste to ručne):

```
>>> push(2); push(3); push(5); pop(); top(); push(7); push(11); pop(); pop();␣
↪pop(); is_empty()
```

riešenie:

zapíšme pre kontrolu príkazov:

```
import stack

s = stack.Stack()
push = s.push
pop = s.pop
top = s.top
is_empty = s.is_empty
```

potom:

```
>>> push(2); push(3); push(5); pop(); top(); push(7); push(11); pop();␣
↪pop(); pop(); is_empty()
5
3
11
7
3
False
>>>
```

2. zistite, čo robí tento program - skúste to ručne bez spúšťania na počítači (používame modul `stack` z prednášky):

```
import stack
s = stack.Stack()
for i in range(20):
    if i<12 and i%3!=0:
        s.push(i)
    elif not s.is_empty():
        print(s.pop())
```

riešenie:

```
=
2
5
8
11
10
7
4
1
```

3. napíšte funkciu `vypis(zasobnik)`, ktorá vypíše obsah zásobníka (po jej skončení bude zásobník prázdny), napr.

```
>>> s = stack.Stack()
>>> # vlož do zásobníka nejaké hodnoty, napr. prvky range(1,8,2)
>>> vypis(s)
stack: 7 5 3 1
>>> s.is_empty()
True
```

riešenie:

```
import stack

def vypis(s):
    print('stack:', end=' ')
    while not s.is_empty():
        print(s.pop(), end=' ')
    print()
```

```
>>> s = stack.Stack()
>>> for i in range(1, 8, 2):
        s.push(i)

>>> vypis(s)
stack: 7 5 3 1
>>> s.is_empty()
True
```

4. využite pomocný zásobník a opravte funkciu `vypis()` tak, aby obsah zásobníka po výpise ostal nezmenený (nesmiete pritom pracovať s atribútom `self.pole`)

```
>>> vypis(s)
stack: 7 5 3 1
>>> s.push(9)
>>> vypis(s)
stack: 9 7 5 3 1
```

riešenie:

```
def vypis(s):
    pom = stack.Stack()     # alebo pom = type(s)()
    print('stack:', end=' ')
    while not s.is_empty():
        hodnota = s.pop()
        pom.push(hodnota)
```

```
        print(hodnota, end=' ')
    print()
    while not pom.is_empty():
        s.push(pom.pop())
```

5. napíšte tieto funkcie, ktoré pracujú so zásobníkom a využívajú pomocný zásobník (môžete používať len základné metódy zásobníka):

   - `pocet(zasobnik)` vráti počet prvkov v zásobníku, zásobník ostane bez zmeny
   - `daj_z_dna(zasobnik)` vyberie a vráti (ako metóda `pop()`) najspodnejší prvok zásobníka, ostatné prvky nemení
   - `daj_na_dno(zasobnik,data)` vloží (ako metóda `push()`) novú hodnotu na úplné dno zásobníka, ostatné prvky nemení

```
>>> s = stack.Stack()
>>> s.push(5); s.push(7); s.push(9)
>>> daj_na_dno(s,3)
>>> pocet(s)
4
>>> vypis(s)
stack: 9 7 5 3
>>> daj_z_dna(s)
3
>>> vypis(s)
stack: 9 7 5
```

riešenie:

```
def pocet(s):
    pom = stack.Stack()     # alebo pom = type(s)()
    vysl = 0
    while not s.is_empty():
        pom.push(s.pop())
        vysl += 1
    while not pom.is_empty():
        s.push(pom.pop())
    return vysl

def daj_z_dna(s):
    pom = stack.Stack()     # alebo pom = type(s)()
    while not s.is_empty():
        pom.push(s.pop())
    vysl = pom.pop()
    while not pom.is_empty():
        s.push(pom.pop())
    return vysl

def daj_na_dno(s, data):
    pom = stack.Stack()     # alebo pom = type(s)()
    while not s.is_empty():
        pom.push(s.pop())
    s.push(data)
    while not pom.is_empty():
        s.push(pom.pop())
```

6. dopíšte chýbajúce časti tak, aby sa do zásobníka dostali len konkrétne prvočísla menšie ako 20, pričom od vrchu zásobníka smerom ku dnu budú od najmenšieho po najväčšie (neriešte pre všeobecné prvočísla, ale len pre 8

konkrétnych):

```
s = stack.Stack()
for i in range(20):
    if ...:
        s.push(...)
```

```
>>> vypis(s)
stack: 2 3 5 7 11 13 17 19
```

riešenie:

```
s = stack.Stack()
for i in range(20):
    if 19-i in (2, 3, 5, 7, 11, 13, 17, 19):
        s.push(19-i)
```

7. v znakovom reťazci je text, z ktorého nás zaujímajú len znaky zátvoriek: '()[]{}' (napr. je to pythonovský výraz `[{1:[],2:()}[2]][0]`); napíšte funkciu `zisti(retazec)`, ktorá zistí (vráti `True`/`False`), či sú zátvorky správne popárované

   - úlohu riešte tak, že ignorujete nezátvorkové znaky, ľavé zátvorky dávate do zásobníka a pri pravých zisťujete, či zodpovedá zátvorke na vrchu zásobníka

```
>>> zisti('pole[{1:[x],2:()}[2]][0]')
True
>>> zisti('(}{)')
False
```

riešenie:

```
def zisti(retazec):
    s = stack.Stack()
    for znak in retazec:
        if znak in '([{':
            s.push(znak)
        elif znak in ')]}':
            if {')':'(',']':'[','}':'{'}[znak] != s.pop():
                return False
    return s.is_empty()
```

8. nasledujúce aritmetické výrazy zapíšte v postfixovom aj v prefixovom tvare:

```
1 + 2 * 3 - 4 * 5 + 6 * 7

10 - (11 - 2 * 4) / (1 + 2 * 3) + 5
```

riešenie:

```
1 + 2 * 3 - 4 * 5 + 6 * 7
postfix: 1 2 3 * + 4 5 * - 6 7 * +
prefix:  + - + 1 * 2 3 * 4 5 * 6 7

10 - (11 - 2 * 4) / (1 + 2 * 3) + 5
postfix: 10 11 2 4 * - 1 2 3 * + / - 5 +
prefix:  + - 10 / - 11 * 2 4 + 1 * 2 3 5
```

9. nasledujúce výrazy preveďte do prefixu a tiež ich ručne vyhodnoďte:

```
9 3 1 / - 2 +

9 3 - 1 2 + /
```

riešenie:

```
9 3 1 / - 2 +
prefix:  + - 9 / 3 1 2
hodnota: 8

9 3 - 1 2 + /
prefix:  / - 9 3 + 1 2
hodnota: 2
```

10. funkcia `pocitaj()` z prednášky vyhodnocuje celočíselné aritmetické výrazy zapísané v postfixe; prepíšte ju tak, aby pracovala s desatinnými číslami a fungovali aj operácie `'-'`, `'/'` a `'**'`:

```
def pocitaj(retazec):
    s = Stack()
    for p in retazec.split():
        if p == '+':
            s.push(s.pop()+s.pop())
        elif p == '*':
            s.push(s.pop()*s.pop())
        else:
            s.push(int(p))
    return s.pop()
```

napr.

```
>>> pocitaj('5 1 7 / + 2.0 -')
3.1428571428571432
```

riešenie:

```
def pocitaj(retazec):
    s = stack.Stack()
    for p in retazec.split():
        if p == '+':
            s.push(s.pop()+s.pop())
        elif p == '-':
            v2 = s.pop()
            v1 = s.pop()
            s.push(v1 - v2)
        elif p == '*':
            s.push(s.pop()*s.pop())
        elif p == '/':
            v2 = s.pop()
            v1 = s.pop()
            s.push(v1 / v2)
        elif p == '**':
            v2 = s.pop()
            v1 = s.pop()
            s.push(v1 ** v2)
        else:
            s.push(float(p))
    return s.pop()
```

11. funkcia `dosad(retazec,pole)` dostáva ako reťazec logický výraz v postfixovom tvare: tento môže obsahovať premenné 'a', 'b', 'c', ... a logické operátory 'and', 'or', '==', '!=' a 'not' ('not' je unárny); funkcia vyhodnotí tento výraz, pričom za premenné dosadí hodnoty z asociatívneho poľa (v poli premenným zodpovedajú logické hodnoty `True` a `False`), premennej 'a' zodpovedá hodnota `pole['a']`, premennej 'b' hodnota `pole['b']`, atď.

```
>>> dosad('a b or a b and ==', {'a':True,'b':False})
False
>>> dosad('x not', {'x':False})
True
```

riešenie:

```
def dosad(retazec, pole):

    def hodnota(p):
        if isinstance(p, bool):
            return p
        return pole[p]

    s = stack.Stack()
    for p in retazec.split():
        if p == 'and':
            v2 = hodnota(s.pop())
            v1 = hodnota(s.pop())
            s.push(v1 and v2)
        elif p == 'or':
            v2 = hodnota(s.pop())
            v1 = hodnota(s.pop())
            s.push(v1 or v2)
        elif p == '==':
            v2 = hodnota(s.pop())
            v1 = hodnota(s.pop())
            s.push(v1 == v2)
        elif p == '!=':
            v2 = hodnota(s.pop())
            v1 = hodnota(s.pop())
            s.push(v1 != v2)
        elif p == 'not':
            v1 = hodnota(s.pop())
            s.push(not v1)
        else:
            s.push(p)
    return s.pop()
```

12. ručne zistite, čo sa vypíše operáciami s radom, ak bol na začiatku prázdny:

```
enqueue(3); enqueue(7); front(); enqueue(9); dequeue(); enqueue(11);␣
↪dequeue(); dequeue(); is_empty()
```

riešenie:

```
3
3
7
9
False
```

11.2 Triedy a operácie 2.

13. upravte triedu `Cas` (zo 16. prednášky):

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __repr__(self):
        return 'Cas {}:{:02}:{:02}'.format(self.sek//3600, self.sek//60%60,␣
↪self.sek%60)

    def sucet(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy=self.sek-iny.sek)

    def vacsi(self, iny):
        return self.sek > iny.sek
```

tak, aby fungovalo:

```
>>> pole = [Cas(10,1,3),Cas(0,30,20),Cas(0,45)]
>>> sum(pole)
Cas 11:16:23
>>> sorted(pole)
[Cas 0:30:20, Cas 0:45:00, Cas 10:01:03]
>>> min(pole)
Cas 0:30:20
>>> pole[1] == Cas(0,29,80)
True
```

Uvedomte si, že musíte definovať nielen `__add__()`, ale aj `__radd__()`

riešenie:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __repr__(self):
        return 'Cas {}:{:02}:{:02}'.format(self.sek//3600, self.sek//
↪60%60, self.sek%60)

    def __add__(self, iny):
        if isinstance(iny, int):
            return Cas(sekundy=self.sek+iny)
        return Cas(sekundy=self.sek+iny.sek)

    __radd__ = __add__

    def rozdiel(self, iny):
        return Cas(sekundy=self.sek-iny.sek)

    def vacsi(self, iny):
```

```
        return self.sek > iny.sek

    def __lt__(self, iny):
        return self.sek < iny.sek

    def __eq__(self, iny):
        return self.sek == iny.sek
```

14. funkcia `pocet_roznych(meno_suboru)` vráti počet rôznych riadkov nejakého súboru, napr. ak súbor obsahuje iba 5 rovnakých riadkov, funkcia vráti 1, prázdny súbor vráti 0

```
>>> pocet_roznych('subor.txt')
1
```

riešenie:

```
def pocet_roznych(meno_suboru):
    with open(meno_suboru) as subor:
        mnoz =set()
        for riadok in subor:
            mnoz.add(riadok)
    return len(mnoz)
```

alebo to isté kompaktnejšie:

```
def pocet_roznych(meno_suboru):
    with open(meno_suboru) as subor:
        return len(set(subor))
```

15. funkcia `iba_int(mnozina)` z danej množiny vráti podmnožinu celých čísel

```
>>> iba_int({1.,'2',3,(4,5),.6,7})
{3, 7}
```

riešenie:

```
def iba_int(mnozina):
    vysl = set()
    for prvok in mnozina:
        if isinstance(prvok, int):
            vysl.add(prvok)
    return vysl
```

16. funkcia `stvorpismenove(pole)` vráti množinu štvorpísmenových slov, ktoré sa nachádzajú vo vstupnom poli (predpokladáme, že vstupom je pole reťazcov)

```
>>> stvorpismenove(['boy', 'girl', 'house', 'doll'])
{'girl', 'doll'}
```

riešenie:

```
def stvorpismenove(pole):
    vysl = set()
    for slovo in pole:
        if len(slovo) == 4:
            vysl.add(slovo)
    return vysl
```

17. funkcia `vsetky_mena(zoo)`, ktorá vráti množinu všetkých mien zvierat v zoo; parametrom funkcie je pole asociatívnych polí informácií o zvieratách, napr.

```
>>> zoo = [{'druh':'slon','meno':'Bimbo'},{'druh':'opica','meno':'Milica'},{
↪'meno':'Ferdo','druh':'mravec'}]
>>> vsetky_mena(zoo)
{'Bimbo', 'Ferdo', 'Milica'}
```

riešenie:

```
def vsetky_mena(zoo):
    vysl = set()
    for zviera in zoo:
        vysl.add(zviera['meno'])
    return vysl
```

18. funkcia `urob(m1,m2,m3)` dostáva ako parametre 3 množiny: výsledkom funkcie bude nová množina, ktorá obsahuje všetky také prvky, ktoré nie sú v `m1`, ale sú buď v `m2` alebo v `m3` (ale nie naraz v oboch); skúste to vyriešiť bez cyklov, iba pomocou priradení

```
>>> urob({1,3,5,7},{1,2,3,4,5},{4,5,6,7})
{2, 6}
>>> urob(set(),{1,2,3,4,5},{4,5,6,7})
{1, 2, 3, 6, 7}
```

riešenie:

```
def urob(m1, m2, m3):
    return (m2|m3) - (m2&m3) - m1
```

alebo:

```
def urob(m1, m2, m3):
    return (m2-m3 | m3-m2) - m1
```

alebo:

```
def urob(m1, m2, m3):
    return (m2 ^ m3) - m1
```

Cvičenie

12.1 Animovaný obrázok

Použite záverečnú verziu projektu z 22. prednášky.

1. otestujte funkčnosť projektu (stiahnite si aj súbory s obrázkami, vytvorte aj modul `mojequeue`)
   - doplňte ďalší typ obrázku (nemusí byť animovaný), napr. tento obrázok môže chodiť len vodorovne a odrážať sa od okrajov plochy:

riešenie:

```
class Plocha:
    def __init__(self):
        ...
        self.obr4 = [tkinter.PhotoImage(file='hroch.png')]

    def klik(self, event):
        t = ri(0, 3)
        if t == 0:
            a = Anim(self.obr1, event.x, event.y,
                     ri(0, 6), ri(-3, 3), ri(30, 130))
        elif t == 1:
            a = Anim(self.obr2, event.x, event.y,
                     ri(-6, -1), ri(-1, 1), ri(100, 160))
        elif t == 2:
            a = AnimOdraz(self.obr3, event.x, event.y,
                          ri(-6, 6), ri(-6, 6), ri(10, 20))
        else:
            a = AnimOdraz(self.obr4, event.x, event.y,
                          ri(-2, 2) or 3, 0, 50)
        self.queue.insert(a.tik, a)

    ...
```

2. opravte metódu `klik()` tak, aby sa nové objekty vytvárali pri kliknutí len vtedy, keď sa pod kurzorom myši nenachádza žiaden objekt (typu `Anim`)

```
class Anim:
    ...
    def klik(self, x, y):
        ...
        return True

class Plocha:
    ...
    def klik(self, event):
        ...
```

- treba si všetky objekty pamätať v nejakom poli (napr. `self.pole`) a pri vytvorení objektu si odložiť do poľa jeho referenciu (na konci `klik()` pridať `self.pole.append(a)`)
- do triedy `Anim` pridať metódu `klik(self,x,y)`, ktorá vráti `True` vtedy, keď je bod `(x,y)` vo vnútri obdĺžnika obrázku (poznáme stred obrázku `(self.x,self.y)` a vieme zistiť jeho veľkosť, napr. `self.obr[0].width()` je šírka)
- metóda `klik()` v triede `Plocha` najprv zistí, či kliknutý bod nie je vo vnútri niektorého z obrázkov (napr. `self.pole[i].klik(event.x,event.y)`) a ak áno, skončí, inak vyrobí na tomto mieste náhodný animovaný objekt

riešenie:

```
class Anim:
    ...
    def klik(self, x, y):
        s, v = self.obr[0].width(), self.obr[0].height()
        xx, yy = self.x-s//2, self.y-v//2
        return xx <= x < xx+s and yy <= y < yy+v

class Plocha:
    def __init__(self):
        ...
        self.pole = []
        self.timer()

    def klik(self, event):
        for a in self.pole:
            if a.klik(event.x, event.y):
                return
        ...
        self.pole.append(a)

    ...
```

3. opravte projekt tak, že počas zatlačenia myši nad niektorým objektom (prišla udalosť `'<Button-1>'` a ešte nenastalo pustenie, teda udalosť `'<ButtonRelease-1>'`) sa tento objekt nepohybuje iba animuje; pri pustení myši pokračuje vo svojej dráhe

```
class Anim:
    ...
    def posun(self):
        ...

class Plocha:
```

```
    ...
    def klik(self, event):
        ...
    def pusti(self, event):       # udalosť pre '<ButtonRelease-1>'
        ...
```

- nejako oznámite samotnému objektu, že jeho metóda `posun()` iba animuje fázy, ale nemení `self.x` ani `self.y` (buď pridáte nejakú metódu na nastavenie nového atribútu, napr. `stop`, alebo priamo tento atribút nastavíte z metód `klik()` a `pusti()`)
- metóda `posun()` nebude hýbať s objektom, keď je nastavený atribút `stop`

riešenie:

```
class Anim:
    def __init__(self, obr, x, y, dx=0, dy=0, tik=100):
        ...
        self.stop = False

    ...

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        if self.stop:
            return
        ...

class AnimOdraz(Anim):
    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        if self.stop:
            return
        ...

class Plocha:
    def __init__(self):
        ...
        self.ktory = None
        self.canvas.bind('<ButtonRelease-1>', self.pusti)

    def pusti(self, event):
        if self.ktory:
            self.ktory.stop = False
            self.ktory = None

    def klik(self, event):
        for a in self.pole:
            if a.klik(event.x, event.y):
                a.stop = True
                self.ktory = a
                return
        ...

    ...
```

4. dopíšte metódu `tahaj()`, ktorá spracuje udalosť ťahania myši `'<B1-Motion>'`

```
class Plocha:
    ...
    def tahaj(self, event):      # udalost' pre '<B1-Motion>'
        ...
```

- počas ťahania myši sa ťahaný objekt animuje ale nerobí svoje posuny - len reaguje na pohyby myši

riešenie:

```
import tkinter
from random import randint as ri
import time
import mojequeue

class Anim:
    canvas = None
    sirka = None
    vyska = None

    def __init__(self, obr, x, y, dx=0, dy=0, tik=100):
        self.x, self.y = x, y
        self.dx, self.dy = dx, dy
        self.tik = tik
        self.obr = obr
        self.faza = 0
        self.stop = False
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        if self.stop:
            return
        self.x += self.dx
        if self.x < 0:
            self.x += self.sirka
        if self.x >= self.sirka:
            self.x -= self.sirka
        self.y += self.dy
        if self.y < 0:
            self.y += self.vyska
        if self.y >= self.vyska:
            self.y -= self.vyska
        self.canvas.coords(self.id, self.x, self.y)

    def klik(self, x, y):
        s, v = self.obr[0].width(), self.obr[0].height()
        xx, yy = self.x-s//2, self.y-v//2
        return xx <= x < xx+s and yy <= y < yy+v

    def move(self, x, y):
        self.x, self.y = x, y
        self.canvas.coords(self.id, self.x, self.y)

class AnimOdraz(Anim):
    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
```

```
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        if self.stop:
            return
        self.x += self.dx
        if self.x < 50:
            self.dx = abs(self.dx)
        if self.x >= self.sirka-50:
            self.dx = -abs(self.dx)
        self.y += self.dy
        if self.y < 50:
            self.dy = abs(self.dy)
        if self.y >= self.vyska-50:
            self.dy = -abs(self.dy)
        self.canvas.coords(self.id, self.x, self.y)

class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        Anim.sirka = self.canvas['width'] = self.im.width()
        Anim.vyska = self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.obr1 = []
        for i in range(8):
            self.obr1.append(tkinter.PhotoImage(file='a1/vtak'+str(i)+'.
↪png'))

        self.obr2 = []
        for i in range(8):
            self.obr2.append(tkinter.PhotoImage(file='a2/zajo'+str(i)+'.
↪png'))

        self.obr3 = []
        for i in range(21):
            self.obr3.append(tkinter.PhotoImage(file='a3/z'+str(i)+'.png
↪'))

        self.obr4 = [tkinter.PhotoImage(file='hroch.png')]

        self.pole = []
        self.ktory = None
        self.queue = mojequeue.PriorityQueue()
        self.canvas.bind('<Button-1>', self.klik)
        self.canvas.bind('<ButtonRelease-1>', self.pusti)
        self.canvas.bind('<B1-Motion>', self.tahaj)
        self.timer()

    def tahaj(self, event):
        if self.ktory:
            self.ktory.move(event.x, event.y)

    def pusti(self, event):
        if self.ktory:
            self.ktory.stop = False
            self.ktory = None
```

```
    def klik(self, event):
        for a in self.pole:
            if a.klik(event.x, event.y):
                a.stop = True
                self.ktory = a
                return
        t = ri(0, 3)
        if t == 0:
            a = Anim(self.obr1, event.x, event.y,
                     ri(0, 6), ri(-3, 3), ri(30, 130))
        elif t == 1:
            a = Anim(self.obr2, event.x, event.y,
                     ri(-6, -1), ri(-1, 1), ri(100, 160))
        elif t == 2:
            a = AnimOdraz(self.obr3, event.x, event.y,
                          ri(-6, 6), ri(-6, 6), ri(10, 20))
        else:
            a = AnimOdraz(self.obr4, event.x, event.y,
                          ri(-2, 2) or 3, 0, 50)
        self.queue.insert(a.tik, a)
        self.pole.append(a)

    def timer(self):
        while not self.queue.is_empty() and self.queue.front()[0] <= time.
↪time():
            a = self.queue.dequeue()[1]
            a.posun()
            self.queue.insert(a.tik, a)
        self.canvas.after(10, self.timer)

p = Plocha()
```

12.2 Turingov stroj

5. zapíšte a otestujte program pre Turingov stroj, ktorý akceptuje len vstupný reťazec (vstupné slovo) zo znakov `'a'` a `'b'`: na začiatku sú len znaky `'a'` a za tým len znaky `'b'`, napr. by mal akceptovať tieto vstupy:

```
'aaab'
''
'bbbb'
'a'
```

riešenie:

```
print(Turing('''
0 a a > 0
0 b b > 1
1 b b > 1
1 _ _ = k
0 _ _ = k
''', 'aabb', konc={'k'}).rob())
```

6. zapíšte a otestujte program pre Turingov stroj, ktorý akceptuje len vstup zložený zo znakov `'a'` a `'b'`, pričom musí byť párnej dĺžky a ak by sme ho rozdelili na dvojice za sebou idúcich znakov, tak každá je buď `'ab'`

alebo `'ba'`, napr. program by mal akceptovať tieto vstupy:

```
'abab'
''
'baab'
'ba'
'abababababbababababa'
```

riešenie:

```
print(Turing('''
0 a a > 1a
0 b b > 1b
0 _ _ = k
1a b b > 0
1b a a > 0
''', 'abababababbababababa', konc={'k'}).rob())
```

7. do triedy `Turing` dopíšte metódy:
   - `pocet()` vráti počet krokov práve zbehnutého programu
   - `znovu(paska)` nastaví nový obsah pásky a aj ďalšie atribúty tak, aby nasledujúce volanie `rob()` spustilo turingov stroj od začiatku (zrejme nastaví `poz` a `stav`)
   - `__repr__()` vráti obsah pásky ale bez úvodných a záverečných prázdnych znakov

```
class Turing:
    ...
```

riešenie:

```
class Turing:
    def __init__(self, program='', paska='', stav='0', konc={'1'}):
        self.program = {}
        for riadok in program.split('\n'):
            if riadok:
                a,b,c,d,e = riadok.split()
                self.program[a, b] = (c, d, e)
        self.paska = list(paska or '_')
        self.stav0 = self.stav = stav
        self.konc = konc
        self.poz = 0
        self.poc = 0

    def pocet(self):
        return self.poc

    def __repr__(self):
        return ''.join(self.paska).strip('_')

    def znovu(self, paska):
        self.poz = 0
        self.stav = self.stav0
        self.paska = list(paska or '_')

    def rob(self, vypis=True):
        self.poc = 0
        while self.stav not in self.konc:
            if vypis:
```

```
                print(''.join(self.paska))
                print(' '*self.poz+'^', self.stav)
            try:
                symb, smer, stav = self.program[self.stav, self.
↪paska[self.poz]]
            except:
                return False
            self.paska[self.poz] = symb
            if smer == '>':
                self.poz += 1
                if self.poz >= len(self.paska):
                    self.paska.append('_')
            elif smer == '<':
                if self.poz > 0:
                    self.poz -= 1
                else:
                    self.paska.insert(0, '_')
            self.stav = stav
            self.poc += 1
        if vypis:
            print(''.join(self.paska))
            print(' '*self.poz+'^', self.stav)
        return True
```

8. zapíšte a otestujte program pre Turingov stroj: na páske sú dve slová zložené len z `'a'` a `'b'` oddelené jedným znakom `'_'`, program akceptuje tento vstup, len keď sú obe slová rovnaké, napr. by mal akceptovať

```
'ab_ab'
'a_a'
'bbbbb_bbbbb'
'bbbbba_bbbbba'
```

riešenie:

```
t = Turing('''
0 a x > sa
0 b x > sb
0 _ _ > kontrola

sa a a > sa
sa b b > sa
sa _ _ > sa1
sa1 x x > sa1
sa1 a x < spat

sb a a > sb
sb b b > sb
sb _ _ > sb1
sb1 x x > sb1
sb1 b x < spat

spat x x < spat
spat _ _ < spat1
spat1 a a < spat1
spat1 b b < spat1
spat1 x x > 0

kontrola x x > kontrola
```

```
kontrola _ _ = k
''', 'abba_abba', konc={'k'})
print(t.rob())
t.znovu('bbbbba_bbbbba')
print(t.rob(False))
t.znovu('bbbbbba_bbbbba')
print(t.rob(False))
```

9. zapíšte a otestujte program pre Turingov stroj, ktorý celé vstupné slovo zo znakov `'x'` a `'y'` skopíruje za seba
   - napr. pre `'xyxx'` bude po skončení na páske `'xyxx_xyxx'`

   riešenie:

```
t = Turing('''
s1 x X > sx
s1 y Y > sy
s1 _ _ = stop

sx x x > sx
sx y y > sx
sx _ _ > sx1
sx1 x x > sx1
sx1 y y > sx1
sx1 _ x < spat

sy x x > sy
sy y y > sy
sy _ _ > sy1
sy1 x x > sy1
sy1 y y > sy1
sy1 _ y < spat

spat x x < spat
spat y y < spat
spat _ _ < spat
spat X x > s1
spat Y y > s1
''', 'xyxx', 's1', {'stop'})
print(t.rob(False))
print(t)
t.znovu('yyyyyyx')
print(t.rob(False))
print(t)
t.znovu('xxxxxyxxxxz')
print(t.rob(False))
print(t)
```

10. zapíšte a otestujte program pre Turingov stroj, ktorý akceptuje len slová, ktoré obsahujú rovnaký počet písmen `'x'` a `'y'`
    - napr. akceptuje `'xyxyyx'` a neakceptuje `'xxxyy'` ani `'xxzyy'`

    riešenie:

```
t = Turing('''
s1 x x > s1
s1 y y > s1
s1 _ _ < s2
```

```
s2 x x < s2
s2 y y < s2
s2 _ _ > s3

s3 x _ > sx
s3 y _ > sy
s3 X _ > s3
s3 Y _ > s3
s3 _ _ = stop

sx x x > sx
sx X X > sx
sx Y Y > sx
sx y Y < spat

sy y y > sy
sy X X > sy
sy Y Y > sy
sy x X < spat

spat X X < spat
spat Y Y < spat
spat x x < spat
spat y y < spat
spat _ _ > s3

''', '', 's1', {'stop'})
for slovo in '', 'xyxyyx', 'xyxyyxa', 'yyyxxx', 'xy'*10, 'xyx':
    t.znovu(slovo)
    print(repr(slovo), t.rob(False))
```

Cvičenie

13.1 Spájané štruktúry

V úlohách použite deklarácie z prednášky:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

def vypis(zoznam):
    while zoznam is not None:
        print(zoznam.data, end=' -> ')
        zoznam = zoznam.next
    print(None)
```

Riešenia aspoň niektorých úloh odkrokujte vo vizualizácii http://pythontutor.com/visualize.html.

1. Vytvorte dva zoznamy `z1` a `z2`, ktoré budú obsahovať písmená reťazcov `'Pyt'` a `'hon'`. Oba tieto najprv vypíšte, potom spojte do jedného a znovu vypíšte:

```
>>> z1 = ...
>>> z2 = ...
>>> vypis(z1)
>>> vypis(z2)
>>> # spoj
>>> vypis(z1)
```

   riešenie:

```
z1 = Vrchol('P', Vrchol('y', Vrchol('t')))
z2 = Vrchol('h', Vrchol('o', Vrchol('n')))
vypis(z1)
vypis(z2)
z1.next.next.next = z2
vypis(z1)
```

2. Z daného zoznamu dĺžky 6 postupne vyhoďte: druhý vrchol, prvý vrchol, predposledný vrchol, posledný vrchol.

```
>>> zoz = ...      # napr. písmená ret'azca 'Python'
>>> vypis(zoz)
P -> y -> t -> h -> o -> n -> None
>>> # vyhod' druhý
>>> vypis(zoz)
```

```
P -> t -> h -> o -> n -> None
>>> # vyhoď prvý
>>> vypis(zoz)
t -> h -> o -> n -> None
>>> # vyhoď predposledný
>>> vypis(zoz)
t -> h -> n -> None
>>> # vyhoď posledný
>>> vypis(zoz)
t -> h -> None
```

riešenie:

```
zoz = Vrchol('P', Vrchol('y', Vrchol('t', Vrchol('h', Vrchol('o', Vrchol(
↪'n'))))))
vypis(zoz)
# vyhoď druhý
zoz.next = zoz.next.next
vypis(zoz)
# vyhoď prvý
zoz = zoz.next
vypis(zoz)
# vyhoď predposledný
zoz.next.next = zoz.next.next.next
vypis(zoz)
# vyhoď posledný
zoz.next.next = None
vypis(zoz)
```

3. Napíšte funkciu `vyrob(post)`, ktorej parameter `post` je ľubovoľné postupnosť hodnôt (napr. pole, reťazec, `range()`, ...). Funkcia z týchto hodnôt vytvorí spájaný zoznam, v ktorom sú vrcholy v poradí, v akom boli v postupnosti (uvedomte si, že postupnosť prvkov môžeme prechádzať aj v opačnom poradí).

```
>>> z = vyrob(['prvy', 'druhy', 'treti'])
>>> vypis(z)
prvy -> druhy -> treti -> None
>>> zz = vyrob(range(2, 12, 3))
>>> vypis(zz)
2 -> 5 -> 8 -> 11 -> None
```

riešenie:

```
def vyrob(post):
    zoz = None
    for i in reversed(post):
        zoz = Vrchol(i, zoz)
    return zoz
```

4. Napíšte funkciu `maxim(zoz)`, ktorá vráti najväčší prvok v zozname. Predpokladáme, že prvky zoznamu sa dajú navzájom porovnávať (napr. zoznam neobsahuje naraz čísla aj reťazce). Pre prázdny zoznam funkcia vyvolá výnimku `EmptyError` (z 20. prednášky v zimnom semestri).

```
>>> z = vyrob(((2, 3), (3, 1), (1, 4)))
>>> vypis(z)
(2, 3) -> (3, 1) -> (1, 4) -> None
>>> m = maxim(z)
>>> print(m)
(3, 1)
```

```
>>> zz = None     # prazdny zoznam
>>> maxim(zz)
...
EmptyError
```

riešenie:

```
class EmptyError(Exception): pass

def maxim(zoz):
    if zoz is None:
        raise EmptyError
    vysl = zoz.data
    zoz = zoz.next
    while zoz is not None:
        if vysl < zoz.data:
            vysl = zoz.data
        zoz = zoz.next
    return vysl
```

5. Napíšte funkciu `sucet(zoz)`, ktorá vráti súčet všetkých prvkov v zozname. Predpokladáme, že prvky zoznamu sú rovnakého typu (napr. zoznam čísel, zoznam reťazcov, zoznam n-tíc, ...). Pre prázdny zoznam funkcia vyvolá výnimku `EmptyError`.

```
>>> z = vyrob(('Py', 't', 'hon'))
>>> vypis(z)
Py -> t -> hon -> None
>>> m = sucet(z)
>>> m
'Python'
>>> zz = vyrob(range(1, 11))
>>> sucet(zz)
55
>>> sucet(vyrob([(1, 2), (3,), (4, 5, 6)]))
(1, 2, 3, 4, 5, 6)
```

riešenie:

```
def sucet(zoz):
    if zoz is None:
        raise EmptyError
    vysl = zoz.data
    zoz = zoz.next
    while zoz is not None:
        vysl += zoz.data
        zoz = zoz.next
    return vysl
```

6. Napíšte funkcie `pridaj_zac(zoz,hodnota)` a `pridaj_kon(zoz,hodnota)`, ktoré pridajú danú hodnotu na začiatok, resp. koniec spájaného zoznamu. Obe funkcie vrátia ako svoju hodnotu referenciu na začiatok zoznamu.

```
>>> z = None
>>> z = pridaj_zac(z, 'A')
>>> z = pridaj_kon(z, 'B')
>>> z = pridaj_zac(z, 'C')
>>> z = pridaj_kon(z, 'D')
```

```
>>> vypis(z)
C -> A -> B -> D -> None
```

riešenie:

```
def pridaj_zac(zoz, data):
    return Vrchol(data, zoz)

def pridaj_kon(zoz, data):
    if zoz is None:
         return Vrchol(data)
    kon = zoz
    while kon.next is not None:
        kon = kon.next
    kon.next = Vrchol(data)
    return zoz
```

7. Napíšte funkciu `pripocitaj(zoz,hodnota)`, ktorá ku každému prvku zoznamu pripočíta zadanú číselnu hodnotu.

```
>>> zo = vyrob([2, 3, 5, 7, 11])
>>> pripocitaj(zo, 7)
>>> vypis(zo)
9 -> 10 -> 12 -> 14 -> 18 -> None
```

riešenie:

```
def  pripocitaj(zoz, hodnota):
    while zoz is not None:
        zoz.data += hodnota
        zoz = zoz.next
```

8. Napíšte funkciu `kopia_rev(zoz)`, ktorá z daného zoznamu vyrobí nový, v ktorom budú rovnaké prvky ale v opačnom poradí ako v pôvodnom zozname.

```
>>> z1 = vyrob('Python')
>>> z2 = kopia_rev(z1)
>>> vypis(z1)
P -> y -> t -> h -> o -> n -> None
>>> vypis(z2)
n -> o -> h -> t -> y -> P -> None
>>> z3 = kopia_rev(z2)
>>> vypis(z3)
P -> y -> t -> h -> o -> n -> None
```

riešenie:

```
def kopia_rev(zoz):
    novy = None
    while zoz is not None:
        novy = Vrchol(zoz.data, novy)
        zoz = zoz.next
    return novy
```

9. Napíšte funkciu `daj_pole(zoz)`, ktorá zo zadaného zoznamu vyrobí pole hodnôt.

```
>>> z100 = vyrob((3, 'ahoj', (100, 200), 3.14))
>>> pole = daj_pole(z100)
>>> pole
[3, 'ahoj', (100, 200), 3.14]
>>> vypis(z100)
3 -> ahoj -> (100, 200) -> 3.14 -> None
```

riešenie:

```
def daj_pole(zoz):
    vysl = []
    while zoz is not None:
        vysl.append(zoz.data)
        zoz = zoz.next
    return vysl
```

10. Napíšte funkciu `rovnake(zoz1,zoz2)`, ktorá zistí, či majú dva zoznamy rovnaké prvky (v rovnakom poradí).

```
>>> rovnake(vyrob([1, 2, 3, 4]), vyrob(range(1, 5)))
True
>>> rovnake(vyrob('abc'), vyrob('cba'))
False
```

riešenie:

```
def rovnake(zoz1, zoz2):
    while zoz1 is not None and zoz2 is not None:
        if zoz1.data != zoz2.data:
            return False
        zoz1 = zoz1.next
        zoz2 = zoz2.next
    return zoz1 is None and zoz2 is None
```

11. Napíšte funkciu `ocisluj(zoz)`, ktorá vráti nový spájaný zoznam (pôvodný nechá bez zmeny), v ktorom sú všetky prvky dvojicami, pričom prvou hodnotou je poradové číslo a druhou je hodnota z pôvodného zoznamu.

```
>>> a = vyrob('abeceda')
>>> b = ocisluj(a)
>>> vypis(a)
a -> b -> e -> c -> e -> d -> a -> None
>>> vypis(b)
(0, 'a') -> (1, 'b') -> (2, 'e') -> (3, 'c') -> (4, 'e') -> (5, 'd') -> (6,
↪'a') -> None
```

riešenie:

```
def ocisluj(zoz):
    novy, index = None, 0
    while zoz is not None:
        novy = pridaj_kon(novy, (index, zoz.data))
        zoz = zoz.next
        index += 1
    return novy
```

12. Napíšte funkciu `spoj(zoz1,zoz2)`, ktorá na koniec neprázdneho zoznamu `zoz1` pripojí zoznam `zoz2`.

```
>>> x = vyrob(range(5))
>>> y = vyrob('abc')
>>> spoj(x, y)
>>> vypis(x)
0 -> 1 -> 2 -> 3 -> 4 -> a -> b -> c -> None
```

riešenie:

```
def spoj(zoz1, zoz2):
    while zoz1.next is not None:
        zoz1 = zoz1.next
    zoz1.next = zoz2
```

Cvičenie

14.1 Spájané zoznamy

Pracujte s touto verziou triedy spájaný zoznam:

```
class Zoznam:
    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self, pole=None):
        self.zac = self.kon = None
        if pole is not None:
            for prvok in pole:
                self.pridaj_kon(prvok)

    def __repr__(self):
        zoz = self.zac
        vysl = '('
        while zoz is not None:
            vysl += repr(zoz.data) + '->'
            zoz = zoz.next
        return vysl + ')'

    def __len__(self):
        zoz = self.zac
        vysl = 0
        while zoz is not None:
            vysl += 1
            zoz = zoz.next
        return vysl

    def __contains__(self, data):
        zoz = self.zac
        while zoz is not None and zoz.data != data:
            zoz = zoz.next
        return zoz is not None

    def pridaj_zac(self, data):
        self.zac = self.Vrchol(data, self.zac)
        if self.kon is None:
            self.kon = self.zac
```

```
    def pridaj_kon(self, data):
        if self.zac is None:
            self.zac = self.kon = self.Vrchol(data, self.zac)
        else:
            self.kon.next = self.Vrchol(data)
            self.kon = self.kon.next

    def map(self, fun):
        zoz = self.zac
        while zoz is not None:
            zoz.data = fun(zoz.data)
            zoz = zoz.next
```

1. Do triedy `Zoznam` dodefinujte metódu `ity()` a otestujte ju:
   - metóda vráti referenciu na príslušný vrchol alebo `None`, ak neexistuje
   - vrcholy číslujeme od 0 po `len(zoznam)-1`

   ```
   class Zoznam:
       ...
       def ity(self, index):
           ...
   ```

   napr.

   ```
   >>> z = Zoznam(['prvy', 'druhy', 'treti', 'stvrty'])
   >>> print(z.ity(2).data)
   treti
   >>> print(z.ity(-1))
   None
   ```

   riešenie:

   ```
   class Zoznam:
       ...
       def ity(self, index):
           p = self.zac
           while p is not None and index > 0:
               p = p.next
               index -= 1
           if index != 0:
               return None
           return p
   ```

2. Pomocou metódy `ity()` dodefinujte tieto ďalšie metódy a otestujte ich:
   - tieto metódy umožnia indexovať prvky zoznamu, t.j. pomocou indexu zistíme hodnotu prvku (`__getitem__()`), alebo zmeníme hodnotu prvku s daným indexom (`__setitem__()`), alebo vyhodíme zo zoznamu prvok na danom indexe
   - ak prvok s daným indexom neexistuje, metóda vyvolá výnimku `IndexError` (zrejme vtedy, keď `ity()` vráti `None`)

   ```
   class Zoznam:
       ...
       def __getitem__(self, index):
           ...
       def __setitem__(self, index, novy):
   ```

```
    ...
def __delitem__(self, index):
    ...
```

napr.

```
>>> z = Zoznam(range(100, 1001, 100))
>>> z[4]                                    # automacky sa zavolá z.__
↪getitem__(4)
500
>>> z[2] = -1                               # automacky sa zavolá z.__
↪setitem__(2, -1)
>>> del z[9]                                # automacky sa zavolá z.__
↪delitem__(9)
>>> z
...
```

riešenie:

```
class Zoznam:
    ...
    def __getitem__(self, index):
        p = self.ity(index)
        if p is None:
            raise IndexError()
        return p.data

    def __setitem__(self, index, novy):
        p = self.ity(index)
        if p is None:
            raise IndexError()
        p.data = novy

    def __delitem__(self, index):
        if index == 0 and self.zac is not None:
            self.zac = self.zac.next
            if self.zac is None:
                self.kon = None
        else:
            p = self.ity(index-1)
            if p is None or p.next is None:
                raise IndexError()
            if p.next == self.kon:
                self.kon = p
            p.next = p.next.next
```

3. Dodefinujte metódu `reverse()`, ktorá prevráti poradie prvkov v zozname, metódu otestujte:

   - pokúste sa nevytvárať nové vrcholy zoznamu, ale využiť pôvodné, pričom "iba" prehádžete referencie medzi vrcholmi

```
class Zoznam:
    ...
    def reverse(self):
        ...
```

napr.

```
>>> z = Zoznam(range(1, 30, 3))
>>> z.reverse()
>>> z
...
```

riešenie:

```
class Zoznam:
    ...
    def reverse(self):
        p = self.kon = self.zac
        if p is None:
            return
        p = p.next
        self.zac.next = None
        while p is not None:
            pp = p.next
            p.next = self.zac
            self.zac = p
            p = pp
```

to isté sa dá zapísať aj kompaktnejšie:

```
class Zoznam:
    ...
    def reverse(self):
        p = self.kon = self.zac
        if p is None:
            return
        p, self.zac.next = p.next, None
        while p is not None:
            p.next, self.zac, p = self.zac, p, p.next
```

4. Dodefinujte metódu `copy()`, ktorej výsledkom je úplne nový zoznam, ktorý je kópiou samotného zoznamu (vyrobí sa kópia každého vrcholu), metódu otestujte:

   - uvedomte si, že táto metóda vráti objekt typu `Zoznam`

```
class Zoznam:
    ...
    def copy(self):
        ...
```

napr.

```
>>> z = Zoznam(range(1, 30, 3))
>>> z1 = z.copy()
>>> z1[5] = 'ahoj'
>>> z                    # jeho obsah by sa nemal zmenit'
...
```

riešenie:

```
class Zoznam:
    ...
    def copy(self):
        res = Zoznam()
        p = self.zac
```

```
        while p is not None:
            res.pridaj_kon(p.data)
            p = p.next
        return res
```

neskôr sa naučíme, prečo bude fungovať aj:

```
class Zoznam:
    ...
    def copy(self):
        return Zoznam(z)
```

14.2 Spájané zoznamy, funkcie

5. Napíšte funkciu `sucet()` s premenlivým počtom parametrov, ktorá vypočíta súčet/zreťazenie svojich parametrov.
   - ak je funkcia zavolaná bez parametrov vráti `None`
   - funkcia s parametrami aplikuje operáciu plus + - k hodnote prvého parametra sa postupne pripočítavajú/prireťazujú ďalšie

```
def sucet(...):
    ...
```

napr.

```
>>> sucet(1, 2, 3, 4)
10
>>> sucet('Py', 't', 'hon')
'Python'
>>> sucet(*range(1000))
499500
>>> sucet([0], [1], [2], [3], [4], [5])
[0, 1, 2, 3, 4, 5]
>>> sucet((1, 2), (3, 4))
(1, 2, 3, 4)
>>> print(sucet())
None
```

riešenie:

```
def sucet(*pole):
    if len(pole) == 0:
        return None
    res = pole[0]
    for prvok in pole[1:]:
        res += prvok
    return res
```

6. Funkcia `mocniny(n,k)` vráti pole `k`-tych mocnín čísel od 1 do `n`. Použite generátorovú notáciu:

```
def mocniny(n, k):
    return [...]
```

napr.

```
>>> mocniny(4, 3)
[1, 8, 27, 64]
```

riešenie:

```
def mocniny(n, k):
    return [i**k for i in range(1, n+1)]
```

7. Funkcia `ocisluj(pole)` vráti pole (`list`) dvojíc (`tuple`), v ktorých prvým prvkom je poradové číslo (číslujeme od 0) a druhým je príslušný prvok poľa. Funkcia robí to isté ako štandardné funkcia `list(enumerate(pole))`:

   - riešte generátorovou notáciou, ale nepoužite `enumerate()`

```
def ocisluj(pole):
    return [...]
```

napr.

```
>>> ocisluj([2, 3, 5, 7, 11, 13, 17])
[(0, 2), (1, 3), (2, 5), (3, 7), (4, 11), (5, 13), (6, 17)]
>>> ocisluj('python')
[(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
>>> list(enumerate('python'))
[(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

riešenie:

```
def ocisluj(pole):
    return [(i, pole[i]) for i in range(len(pole))]
```

8. Funkcia `pole2(m,n,data=0)` vygeneruje dvojrozmerné pole s `m` riadkami a `n` stĺpcami, v ktorom majú všetky prvky zadanú hodnotu `data`.

   - použite generátorovú notáciu

```
def pole2(m, n, data=0):
    ...
```

napr.

```
>>> pole2(3, 4)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> pole2(5, 2, 'a')
[['a', 'a'], ['a', 'a'], ['a', 'a'], ['a', 'a'], ['a', 'a']]
```

riešenie:

```
def pole2(m, n, data=0):
    return [[data]*n for i in range(m)]
```

9. Funkcia `apole(mnozina)` dostáva ako parameter nejakú množinu slov a vytvorí z nej asociatívne pole (`dict`), ktorého kľúčmi sú slová z množiny a zodpovedajúcimi hodnotami sú dĺžky týchto slov (znakových reťazcov).

   - použite generátorovú notáciu

```
def apole(mnozina):
    ...
```

napr.

```
>>> apole({'prvy', 'druhy', 'sty', ''})
{'': 0, 'sty': 3, 'prvy': 4, 'druhy': 5}
>>> apole({a+b for a in ('a', 'bc') for b in ('', 'x', 'yz')})
{'ayz': 3, 'ax': 2, 'a': 1, 'bc': 2, 'bcyz': 4, 'bcx': 3}
```

riešenie:

```
def apole(mnozina):
    return {slovo:len(slovo) for slovo in mnozina}
```

10. Do triedy `Zoznam` dodefinujte metódy, ktorých parametrami sú funkcie:

    - metóda `zmen()` prejde všetky prvky zoznamu a tie, pre ktoré platí `fun1` na ne aplikuje funkciu `fun2` (zmení im hodnotu)
    - metóda `nechaj()` v zozname nechá len tie prvky, pre ktoré platí podmienka `fun`, ostatné vyhodí (je to analogické funkcii `filter()`)

```
class Zoznam:
    ...
    def zmen(self, fun1, fun2):
        ...
    def nechaj(self, fun):
        ...
```

napr.

```
>>> z = Zoznam([1, 2, 3, 4, 5, 6, 7])
>>> z.zmen(lambda x: x%2==1, lambda x: x*x+1) # nepárne umocní na 2␣
↪a pripočíta 1
>>> z.nechaj(lambda x: x%2==0)                # vyhodí nepárne
>>> z
...
```

riešenie:

```
class Zoznam:
    ...
    def zmen(self, fun1, fun2):
        p = self.zac
        while p is not None:
            if fun1(p.data):
                p.data = fun2(p.data)
            p = p.next

    def nechaj(self, fun):
        p = self.zac
        if p is not None:
            while p.next is not None:
                if not fun(p.next.data):
                    p.next = p.next.next
                else:
                    p = p.next
```

```
            self.kon = p
            if not fun(self.zac.data):
                self.zac = self.zac.next
            if self.zac is None:
                self.kon = None
```

Cvičenie

15.1 Zásobníky a rady

1. Otestujte algoritmus vypĺňania z prednášky (pre plochu aspoň 200x200), použite zásobník a aj rad - oba **realizované pomocou spájaných zoznamov**: každú realizáciu zásobníka alebo radu uložte do samostatného súboru a vo vašom testovacom programe urobíte `import`
   - napr. v súbore `stack1.py` bude:

```
class Stack:
    ...
```

   - v súbore `queue1.py` bude:

```
class Queue:
    ...
```

2. Do oboch tried `Stack` aj `Queue` doplňte udržiavanie informácie o maximálnom počte prvkov v štruktúre
   - treba pridať 2 atribúty: `vel` - momentálna velkosť a `max` - doteraz maximálna veľkosť
   - v metóde, ktorá pridáva prvok sa otestuje, či momentálna veľkosť nie je väčšia ako doterajšie maximum
   - v metóde, ktorá odoberá prvok, sa iba zmenší momentálna veľkosť štruktúry

```
class Stack:
    ...
    def __init__(self):
        ...
        self.vel = 0
        self.max = 0
```

   - po skončení vypĺňacieho algoritmu vypíšte z použitej štruktúry (zásobník alebo rad) tento maximálny počet
3. Porovnajte približnú rýchlosť vypĺňania rovnakoveľkej plochy napr. 200x200 (pomocou `time.time()`) pre všetky tieto verzie (každú verziu štruktúry treba mať v samostatnom súbore):
   - zásobník pomocou poľa
   - zásobník pomocou spájaného zoznamu
   - rad pomocou poľa a operácií `pole.pop(0)` a `pole.append`
   - rad pomocou spájaného zoznamu

- rad pomocou cyklického poľa

4. Sledujte priebeh vypĺňacieho algoritmu pre veľké pole, ktoré na začiatku obsahuje niekoľko vodorovných čiar (aspoň 3) s rozostupmi, pričom na každej tejto čiare je niekde jednopixlová medzera (tieto čiary nekreslite pomocou `create_line()` ale bodkovaním v cykle)

   - zjednodušene by to mohlo vyzerať asi takto (ale pre pole aspoň 200x200):

```
..........
..........
***.******
..........
..........
*******.**
..........
..........
****.*****
..........
......+...
```

   - znakom `'+'` je označené, kde by sa mohlo naštartovať vypĺňanie

5. Ideu vypĺňania oblasti môžeme využiť aj na trochu iné úlohy: v dvojrozmernom poli znakov sú niektoré znaky prekážkou (napr. `'*'`) iné nie (napr. `'.'`). Úlohou bude niečo o takomto poli zistiť. Napíšte funkciu `zisti(pole)`, ktorá dostáva parameter dvojrozmerné pole znakov (vytvorte ho napr. prečítaním textového súboru) a zistí, či zvyšok poľa tvorí jedinú oblasť, t.j. dá sa dostať z ľubovoľného miesta na iné obídením prekážok. Budete to riešiť tak, že začnete v ľubovoľnom políčku bez prekážky, spustíte vypĺňací algoritmus a po ňom skontrolujete, či v poli neostalo nejaké nevyplnené miesto. Vstupné pole môžete pri tom modifikovať.

   - môžete to otestovať napr. na takýchto poliach:

```
..........
..........
***.******
..........
..........
*******.**
..........
..........
****.*****
..........
..........
```

   alebo:

```
..........
..........
..********
..*.......
.....*....
******....
..........
..........
******....
....*.....
....*.....
```

6. Napíšte funkciu `kolko_oblasti(pole)`, ktorá funguje na podobnom princípe ako `zisti(pole)` z predchádzajúcej úlohy: funkcia zistí, koľko oddelených oblastí obsahuje dané dvojrozmerné pole znakov.

Zrejme, ak o nejakom poli funkcia `zisti(pole)` vyhlási `True`, toto pole obsahuje jedinú oblasť a teda `kolko_oblasti(pole)` vráti hodnotu 1.

- napr. pre pole:

```
......*....
......*....
......*....
********....
......*....
....*******
....*......
....*......
```

- funkcia vráti hodnotu 4

KAPITOLA 16

Cvičenie

16.1 Binárne stromy

Na riešenie úloh si z prednášky okopírujte definíciu triedy `Vrchol` a tiež niektoré funkcie, ktoré môžete využiť.

1. Do premennej `strom` sme priradili binárny strom
   - nakreslite ho na papier (bez počítača)

```
>>> strom =␣
↪Vrchol(7,Vrchol(13,None,Vrchol(5,Vrchol(8))),Vrchol(2,Vrchol(11,Vrchol(19)),Vrchol(3)))
```

   - skontrolujte vašu kresbu s vykreslením pomocou `kresli()`
2. Na papier nakreslite taký binárny strom, ktorý má 6 listov a v týchto listoch sú postupne z ľava doprava písmená zo slova 'Python', zvyšné vnútorné vrcholy stromu obsahujú nepísmenové znaky.
   - vytvorte tento strom pomocou priradenia (alebo aj viacerých priradení)

```
strom = Vrchol(...)
```

   - nakreslite tento strom pomocou `kresli()`
3. Zapíšte také priradenie, aby sa v strome z úlohy (2) vrchol s písmenom 'y' nahradil písmenom *i*
   - otestujte vykreslením tohto stromu

```
strom.... = 'i'
kresli(...)
```

4. Ručne zistite, čo sa vypíše
   - upravená funkcia `vypis`:

```
def vypis(vrch, k):
    if vrch is None:
        print('.'*k, None)
        return
    print('.'*k, vrch.data)
    vypis(vrch.left, k+2)
    vypis(vrch.right, k+2)

vypis(Vrchol('koren',Vrchol('lavy'),Vrchol('pravy')), 0)
```

5. Napíšte funkciu `pocet_listov(vrch)`, ktorá vráti počet listov v strome

- funkcia bude rekurzívna a bude riešiť tieto prípady:
  - pre prázdny strom bude výsledkom 0
  - pre vrchol, ktorý je listom (ľavý aj pravý podstrom je `None`), bude výsledkom 1
  - inak zistíme počet listov v ľavom podstrome a aj počet listov v pravom podstrome, tieto dva výsledky spočítame a tento súčet je výsledným počtom listov v celom strome
- funkciu otestujte, napr.

```
def pocet_listov(vrch):
    ...
```

```
>>> strom = Vrchol('X',Vrchol('Y'),Vrchol('Z'))
>>> pocet_listov(strom)
2
>>> pocet_listov(Vrchol('X',Vrchol('Y',Vrchol('Z',Vrchol('U')))))
1
```

6. Napíšte funkciu `pocet2(vrch)`, ktorá vráti počet vrcholov v strome, ktoré majú práve 2 synov
   - funkciu otestujte aj na väčšom strome, ktorý vygenerujete náhodne

```
def pocet2(vrch):
    ...
```

7. Pozmeňte funkciu `kresli(...)` tak, aby bol koreň pri vykreslení zafarbený na červeno a všetky listy v strome na modro
   - do funkcie môžete pridať ďalší parameter, napr.

```
def kresli(vrch, sir, x, y, koren=True):
    ...
```

8. Napíšte funkciu `strom1(n)`, ktorá vygeneruje strom s `n` vrcholmi, pričom každý z nich, okrem posledného, má práve jedného syna, hodnoty vo vrcholoch sú postupne 0, 1, 2, ..., `n`-1. Koreň stromu má iba ľavého syna, tento má iba pravého, ďalší má opäť iba ľavého, atď.
   - riešte najprv pomocou rekurzie - môžete použiť aj ďalšie parametre s náhradnou hodnotou, napr.

```
def strom1(n, vlavo=True):    # alebo def strom1(n, hodnota=0):
    ...
```

   - riešte aj bez rekurzie

```
def strom1(n):
    ...
```

9. Napíšte funkciu `strom2(n)`, ktorá vygeneruje strom s výškou `n`, pričom koreň má 2 synov (alebo žiadneho pre `n` = 0), ľavý podstrom má synov iba vľavo a pravý podstrom má všetky vrcholy iba vpravo. Vrcholy majú hodnoty podľa úrovne: koreň 0, jeho synovia 1, ich synovia 2, atď.

- riešte bez rekurzie

```
def strom2(n):
    ...
```

10. Napíšte funkciu `vytvor_uplny(n)`, ktorá vráti vygenerovaný úplný strom: jeho najvyššia úroveň je `n` a hodnoty vo všetkých vrcholoch nech sú 0:

- zrejme použijete rekurziu: úplý strom úrovne `n` sa skladá z ľavého úplného stromu úrovne `n-1`, z pravého úplného strému tiež úrovne `n-1` a z koreňa, ktorý má tieto dva podstromy:

```
def vytvor_uplny(n):
    ...
```

11. Napíšte funkciu `nachadza_sa(vrch,hodnota)`, ktorá zistí (vráti, `True` alebo `False`), či sa v strome nachádza daná hodnota
    - riešite zrejme rekurzívne:

```
def nachadza_sa(vrch, hodnota):
    ...
```

12. Napíšte funkciu `pocet_vyskytov(vrch,hodnota)`, ktorá zistí počet výskytov danej hodnoty v strome
    - riešite zrejme rekurzívne:

```
def pocet_vyskytov(vrch, hodnota):
    ...
```

13. Napíšte funkciu `mapuj(vrch,funkcia)`, ktorá zmení hodnoty vo všetkých vrcholoch stromu aplikovaním danej funkcie, t.j. pre každý vrchol zoberie hodnotu, zavolá s ňou danú funkciu a túto novú vypočítanú hodnotu vráti do vrcholu

```
def mapuj(vrch, funkcia):
    ...
```

14. Napíšte funkciu `min_max(vrch)`, ktorá vráti dvojicu (`tuple`) s minimálnou a maximálnou hodnotou v strome
    - napr.

```
def min_max(vrch):
    ...

print(min_max(Vrchol(5, Vrchol(7), Vrchol(2))))

(2, 7)
```

15. Napíšte funkcie `daj_pole(vrch)` a `daj_mnozinu(vrch)`, ktoré vrátia buď pole, alebo množinu všetkých hodnôt v strome

```
def daj_pole(vrch):
    ...

def daj_mnozinu(vrch):
    ...
```

KAPITOLA 17

Cvičenie

17.1 Trieda BinarnyStrom

1. Navrhnite na papieri binárny strom s 10 vrcholmi (s číslami od 0 do 9), v ktorom väčšina vnútorných vrcholov má oboch synov.

- pre tento váš strom ručne vypíšte postupnosti:
  - preorder
  - inorder
  - postorder
  - po úrovniach

2. Nevieme ako vyzerá nejaký binárny strom, ale poznáme jeho inorder a postorder.
   - dané poradia:

```
inorder =  2 8 3 5 9 1 7 4 10 6
postorder =  8 2 5 3 7 1 10 6 4 9
```

   - ručne zostavte preorder

3. Ručne nakreslite všetky binárne stromy, ktoré majú 3 vrcholy a sú očíslované hodnotami 1, 2, 3 a to po úrovniach.

- ku každému nakreslenému stromu vypíšte jeho inorder

4. Zostavte triedu `BinarnyStrom` z prednášky aj so všetkými tam uvedenými metódami.

- otestujte náhodné generovanie stromu, aj všetky typy výpisov

5. Do triedy `BinarnyStrom` dopíšte metódy:
   - `mapuj(funkcia)` - zmení hodnotu v každom vrchole aplikovaním danej funkcie

```
def mapuj(self, funkcia):
    ...
    vrch.data = funkcia(vrch.data)
    ...

strom = BinarnyStrom()
...
strom.mapuj(lambda x: x*11)
```

- `ocisluj_inorder(start=0,krok=1)` - očísluje všetky vrcholy celými číslami od hodnoty `start` s krokom `krok`

```
def ocisluj_inorder(self, start, krok):
    ...

strom = BinarnyStrom()
...
strom.ocisluj_inorder(10, 10)
print('inorder = ', end='')
strom.vypis_inorder()
```

- `prirad(pole)` - postupne vyberá prvky poľa a priraďuje ich do vrcholov poľa v poradí preorder; ak je prvkov v poli menej ako vrcholov stromu, zvyšné vrcholy ostanú bezo zmeny

```
def prirad(self, start, krok):
    ...

strom = BinarnyStrom()
...
strom.prirad([2, 3, 5, 7, 11, 13])
print('preorder = ', end='')
strom.vypis_preorder()
```

- `brat(vrchol)` - funkcia vráti referenciu na brata, resp. `None`, ak neexistuje

```
def brat(self, vrchol):
    ...

strom = BinarnyStrom()
...
print(strom.brat(strom.root.right))
```

- `ity(i)` - vráti hodnotu v `i`-tom vrchole (ak by sme ho prechádzali preorderom)

```
def ity(self, i):
    ...

strom = BinarnyStrom()
...
for i in range(len(strom)):
    print(i, strom.ity(i))
```

- `kopia()` - vráti kópiu celého stromu

```
def kopia(self):
    ...

strom = BinarnyStrom()
...
kopia = strom.kopia()
kopia.ocisluj_inorder(10, 10)
print('povodny = ', end='')
strom.vypis_preorder()
print('kopia = ', end='')
kopia.vypis_preorder()
```

6. Využitím algoritmu prechádzania po úrovniach (levelorder) naprogramujte metódy:

- `ocisluj_levelorder(start=0,krok=1)` - očísluje všetky vrcholy celými číslami od hodnoty `start` s krokom `krok`

```
def ocisluj_levelorder(self, start=0, krok=1):
    ...

strom = BinarnyStrom()
...
strom.ocisluj_levelorder(1)
strom.kresli()
```

- `v_urovni(k)` - vráti zoznam (`list`) vrcholov (ich hodnôt) v danej úrovni, pre `k=0` zoznam obsahuje hodnotu v koreni

```
def v_urovni(self, k):
    ...

strom = BinarnyStrom()
...
for k in range(strom.vyska()):
    print('v urovni', k, '=', strom.v_urovni(k))
```

- `sirka()` - zistí šírku stromu

```
def sirka(self):
    ...

strom = BinarnyStrom()
...
print('sirka =', strom.sirka())
```

- `zisti_urovne()` - do každého vrcholu pridá nový atribút `vrch.level`, ktorý bude obsahovať úroveň vrcholu

```
def zisti_urovne(self):
    ...

strom = BinarnyStrom()
...
strom.zisti_urovne()
print(strom.root.right.level)
```

7. Nasledovné rekurzívne metódy prepíšte pomocou algoritmu po úrovniach na ich nerekurzívne verzie:

- `vyska()`

```
def vyska_nerek(self):
    ...

strom = BinarnyStrom()
...
print('vyska =', strom.vyska(), strom.vyska_nerek())
```

- `pocet()`

```
def pocet_nerek(self):
    ...

strom = BinarnyStrom()
```

```
...
print('pocet =', strom.pocet(), strom.pocet_nerek())
```

- `sucet()`

```
def sucet_nerek(self):
    ...

strom = BinarnyStrom()
...
print('sucet =', strom.sucet(), strom.sucet_nerek())
```

Cvičenie

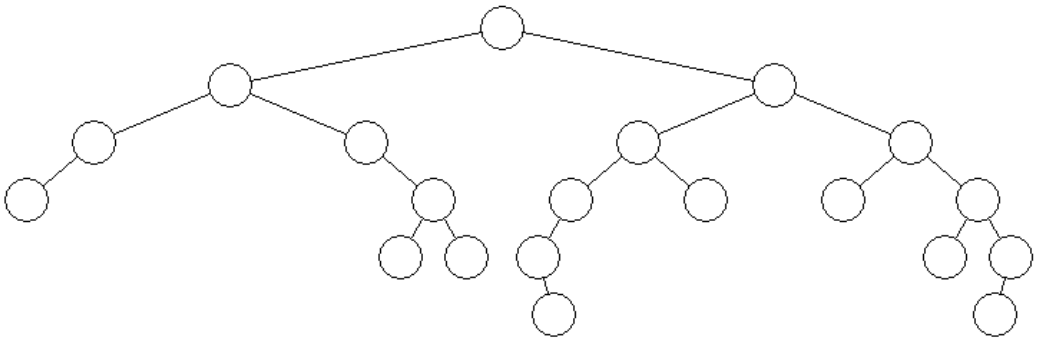
18.1 Použitie stromov

18.1.1 binárne vyhľadávacie stromy

1. Ručne (na papieri, do súboru) vytvorte BVS postupným pridávaním (6, 12, 2, 4, 10, 7, 3, 9, 5, 11, 8, 1, 0) do prázdneho stromu
   - postupne pre každý prvok postupnosti "ručne vykonáte" operáciu `pridaj()`
2. Ručne vypíšte `preorder()` a `inorder()` postupnosti z tohto stromu

```
preorder:
inorder:
```

3. Dopíšte do stromu na obrázku čísla z postupnosti `range(20)` tak, aby vznikol BVS:



4. Do triedy `BinarnyVyhladavaciStrom` dopíšte metódy `min()` a `max()`, ktoré vrátia **referenciu** na minimálny, resp. maximálny vrchol v **podstrome** od zadaného vrcholu
   - parametrom oboch týchto metód je referencia na niektorý konkrétny vrchol v strome (napr. koreň)
   - minimum, resp. maximum sa hľadá od tohto prvku smerom k listom

```
class BinarnyVyhladavaciStrom:
    ...
    def min(self, vrchol):
        ...
    def max(self, vrchol):
        ...
```

5. Napíšte metódu `pridaj_pole(pole)`, ktorá postupne do stromu pridá (metódou `pridaj()`) všetky prvky zadaného poľa
   - pomocou tejto metódy vytvorte BVS z (1) príkladu a skontrolujte `preorder()` a `inorder()` z (2) príkladu

```
class BinarnyVyhladavaciStrom:
    ...
    def pridaj_pole(self, pole):
        ...
```

6. Napíšte metódu `inorder_pole()`, ktorá prejde vrcholy v strome v poradí **inorder** a vráti pole s hodnotami vo vrcholoch:

```
class BinarnyVyhladavaciStrom:
    ...
    def inorder_pole(self):
        ...
```

7. Napíšte metódu `zrus()`, ktorá vyhodí zo stromu zadaný prvok (ak sa v strome nachádza), pričom zachová **BVS** vlastnosť

```
class BinarnyVyhladavaciStrom:
    ...
    def zrus(self, hodnota):
        ...
```

- uvažujte pritom tieto prípady:
  - vrchol sa tam nenachádza a teda nie je čo robiť
  - vyhadzovaný vrchol je list (nemá žiadnych synov), tak ho stačí zrušiť a jeho otcovi nastaviť `None`
  - vyhadzovaný vrchol má len jedného syna: vrchol vyhodíme a v strome ho nahradíme jeho jediným synom
  - ak má rušený vrchol oboch synov:
    - najprv sa nájde minimum v pravom podstrome vyhadzovaného vrcholu
    - toto minimum vyhodí (tento vrchol má max. jedného syna) a jeho hodnotu dá namiesto rušeného vrcholu
    - (fungovalo by to aj s maximom v ľavom podstrome):
- tento algoritmus si môžete najprv vyskúšať ručne na nejakom nakreslenom BVS

18.1.2 Aritmetické stromy

8. Ručne (na papieri) nakresliť strom pre aritmetický výraz `6*5+7/(4-2*5)`
9. Pomocou pomocnej funkcie `v()` z prednášky zadefinujte strom zo (8) úlohy a pomocou `kresli()` ho nakreslite

```
v = AritmetickyStrom.Vrchol
strom = ...
```

10. Napíšte metódu `__repr__`, ktorá z aritmetického stromu vráti presne taký istý reťazec, akým sme pomocou funkcie `v()` definovali samotný strom (napr. v 9. úlohe). Metóda by mala vrátiť reťazec v tvare `"v('+',v('*',v(3),v(4)),v('/',v(10),v(2)))"`

- uvedomte si, že tento reťazec je len trochu upravený **prefixový** tvar výrazu

```
class AritmetickyStrom:
    ...
    def __repr__(self):
        ...
```

11. Pre aritmetický strom prepíšte metódu *__init__* tak, aby, ak je parametrom pole reťazcov a čísel, predpokladáme, že je to preorderový zoznam aritmetického stromu. Metóda z tohto poľa vytvorí celý aritmetický strom
    - fungovať by malo napr. `s = AritmetickyStrom(prefix=['*','+',4,5,'-',7,9])`

```
class AritmetickyStrom:
    ...
    def __init__(self, prefix=None):
        ...
```

18.1.3 Všeobecné stromy

12. pre všeobecný strom napíšte metódy:
    - `preorder()` - vráti postupnosť hodnôt v poli (typu `list`)
    - `vyska()` - zistí výšku stromu
    - `pocet_listov()` - zistí počet listov stromu
    - `stupen()` - zistí stupeň stromu (t.j. maximum z počtov synov všetkých vrcholov stromu)
    - `level_order()` - vráti všetky prvky po úrovniach (výsledok bude typu `list`)

```
class VseobecnyStrom:
    ...
    def preorder(self):
        ...
    def vyska(self):
        ...
    def pocet_listov(self):
        ...
    def stupen(self):
        ...
    def level_order(self):
        ...
```

- metódy otestujte na náhodne vygenerovanom strome tak, že strom vykreslíte a skontrolujete s výsledkami volaní týchto metód

KAPITOLA 19

Cvičenie

19.1 Triedenia

1. Zapíšte tri funkcie `utried1(veta)`, `utried2(veta)` a `utried3(veta)`, ktoré nejako usporiadajú slová v danej vete. Využijete štandardnú funkciu `sorted()`.
   - `utried1()` usporiada slová vo vete podľa abecedy
   - `utried2()` usporiada slová vo vete podľa abecedy ale zostupne
   - `utried3()` usporiada slová vo vete podľa dĺžky slov (najprv najkratšie) a až keď sú rovnako dlhé podľa abecedy

```
def utried1(veta):
    return ...

def utried2(veta):
    return ...

def utried3(veta):
    return ...
```

```
>>> utried1('kohutik jaraby nechod do zahrady')
'do jaraby kohutik nechod zahrady'
>>> utried2('kohutik jaraby nechod do zahrady')
'zahrady nechod kohutik jaraby do'
>>> utried3('jano ide z blavy do brna')
'z do ide brna jano blavy'
```

2. Napíšte funkciu `najcastejsie()`, ktorá z nejakého textu (znakový reťazec s medzerami a koncami riadkov ‘n’) vypíše 5 najčastejšie sa vyskytujúcich slov spolu aj s ich počtami výskytov (napr. súbor `dobs.txt`):

```
>>> text = open('dobs.txt').read()
>>> najcastejsie(text)
abcd 123
xy 55
...
```

3. Spojazdnite vizualizáciu testov z prednášky a pozorujte, ako sa triedi 300-prvkové náhodné pole pomocou insert_sortu a quick_sortu.
4. Doplňte funkciu `min_sort()` tak, aby jej výsledkom bola dvojica (počet porovnaní, počet výmen), funkciu otestujte s 1000 prvkovým poľom:

- náhodných hodnôt
- vzostupne utriedenými hodnotami
- zostupne utriedenými hodnotami

```
def min_sort(pole):
    for i in range(len(pole)-1):
        for j in range(i+1,len(pole)):
            if pole[i] > pole[j]:                    # porovnanie
                pole[i],pole[j] = pole[j],pole[i]    # vymena
    return ...
```

5. Funkcia `bubble_sort()` z prednášky vykoná vnútorný cyklus n-krát aj vtedy, keď sa už žiadna výmena nerobí, napr. pre utriedené pole sa neurobí žiadna výmena; upravte túto funkciu tak, aby sa vnútorný for-cyklus vykonával len vtedy, keď sa v predchádzajúcom prechode vykonala aspoň jedna výmena; inak, triedenie skončí

```
def bubble_sort(pole):
    ...
    return ...
```

6. Podobne ako v (4) upravte aj `bubble_sort()` z úlohy (5) tak, aby jej výsledkom bola dvojica (počet porovnaní, počet výmen):

```
def bubble_sort(pole):
    ...
    return ...
```

7. Upravte `insert_sort()` a `quick_sort()` tak, aby tiež vrátili dvojice ako v (4) a v (6)

```
def insert_sort(pole):
    ...
    return ...
```

```
def quick_sort(pole):
    ...
    return ...
```

8. Funkcia `test(pole)` odmeria rýchlosť vykonávania všetkých 4 triedení pre rovnaké pole, funkcia pre každé triedenie vypíše čas, počet porovnaní a počet výmen:
   - `min_sort()`
   - `bubble_sort()`
   - `insert_sort()`
   - `quick_sort()`
9. zapíšte a otestujte ďalší algoritmus triedenia `merge_sort()`, ktorý je podobne ako `quick_sort()` rekurzívny; funkcia nie je **in-place**, teda nemení obsah parametra, ale vráti utriedené pole:
   - triviálny prípad: ak počet prvkov triedeného poľa nie je väčší ako 2, pole je už utriedené, teda vráti ho ako výsledok funkcie
   - inak rozdelí pole na dve polia polovičnej dĺžky (pre nepárnu dĺžku je jedna z polovíc o 1 dlhšia)
   - každú z polovíc rekurzívne utriedi
   - obe utriedené polovice zlúči do utriedeného výsledku:
     - postupne prechádza prvky oboch polí a do výsledného poľa vyberá ten z nich, ktorý je menší

KAPITOLA 20

Cvičenie

20.1 Grafy

1. Graf je zadaný dvojicami vrcholov, ktoré sú spojené **neorientovanými** hranami
   - nakreslite ako ohodnotený **neorientovaný** graf (šípky vedú oboma smermi)

```
Bratislava Trnava 50
Nitra Trnava 45
Nitra Komarno 130
Bratislava Komarno 100
Trnava Trencin 75
BBystrica Nitra 115
Partizanske Nitra 50
Trencin Partizanske 45
BBystrica Partizanske 100
```

2. Zapíšte graf z (1) v týchto reprezentáciách (v niektorých z nich neberieme do úvahy hodnoty na hranách):
   (a) pole množín susedností (vrcholy grafu očíslujte číslami od 0 do 6)
   (b) pole asociatívnych polí susedností (opäť len čísla vrcholov)
   (c) asociatívne pole množín susedností (pracujete s reťazcami a nie číslami vrcholov)
   (d) matica susedností
   (e) matica susedností s váhami

```
graf1 = [...]
graf2 = [...]
graf3 = {...}
graf4 = [[...], ...]
graf5 = [[...], ...]
```

3. Reprezentáciu grafu **asociatívne pole množín susedností** z prednášky upravte tak, aby sa namiesto množín susedia uchovávali v **asociatívnych poliach** (`dict`), t.j. v každom vrchole bude jedno asociatívne pole, v ktorom bude pre každého suseda hodnota príslušnej hrany
   - keďže túto reprezentáciu budeme potrebovať pre neorientovaný graf, príslušne upravte aj metódu `pridaj_hranu()` triedy `Graf` (aby sa hrana automaticky definovala oboma smermi)
   - takto zadefinujete reprezentáciu **asociatívne pole asociatívnych polí**

```
class Vrchol:
    def __init__(self, meno):
        self.meno = meno
        self.sus = dict()

    def pridaj_hranu(self, v2, vaha):
        ...

    ...

class Graf:
    def __init__(self):
        self.pole = {}

    def pridaj_vrchol(self, meno):
        ...

    ...
```

- otestujte, napr.

```
>>> g = Graf()
>>> g.pridaj_hranu('BA', 'TT', 50)
>>> g.pridaj_hranu('NR', 'TT', 45)
>>> print(g)
```

4. Pre triedu `Graf` z (3) príkladu zmeňte inicializáciu `__init__(self,meno_suboru='')` tak, aby v prípade, že zadáme meno súboru s textom z (1) príkladu, vytvorí sa príslušný graf.
   - vytvorí sa neorientovaný ohodnotený graf v reprezentácii asociatívne pole asociatívnych polí:

```
class Graf:
    def __init__(self, meno_suboru=''):
        ...
```

   - riadky popisujúce graf z príkladu (1) zapíšte do súboru, napr. `'subor.txt'`, potom otestujte

```
graf = Graf('subor.txt')
print(graf)
```

5. Do triedy `Graf` z (4) príkladu pridajte metódu `kresli()`, ktorá nakreslí graf do grafickej plochy.
   - predpokladáme, že súbor, z ktorého sa číta graf, je doplnený o súradnice vrcholov:
     - na začiatku súboru budú definované súradnice všetkých vrcholov (napr. v tvare: `Bratislava 50 200`) a až za tým (za jedným prázdnym riadkom) budú popísané hrany, t.j. dvojice miest a ich vzdialenosť
   - napr.

```
Bratislava 30 260
Trnava 130 200
Nitra 230 230
Trencin 230 50
Komarno 230 390
BBystrica 440 100
Partizanske 300 130

Bratislava Trnava 50
```

```
Nitra Trnava 45
Nitra Komarno 130
Bratislava Komarno 100
Trnava Trencin 75
BBystrica Nitra 115
Partizanske Nitra 50
Trencin Partizanske 45
BBystrica Partizanske 100
```

- vrcholy grafu sa budú kresliť ako krúžky s menami vrcholov (mená môžete skrátiť napr. na nejaké dve písmená)
- hrany budú nakreslené ako úsečky medzi vrcholmi, pričom v strede hrán sa vypíše váha hrany
- zrejme do triedy `Vrchol` bude treba pridať nové atribúty pre súradnice a tiež pozmeniť inicializáciu s čítaním súboru:

```
class Vrchol:
    def __init__(self, meno, x, y):
        self.meno = meno
        self.sus = dict()
        self.x, self.y = x, y

    ...

class Graf:
    canvas = None

    def __init__(self, meno_suboru=''):
        ...

    def kresli(self):
        ...
```

6. Do triedy `Graf` z (5) príkladu pridajte metódu `zapis(meno_suboru)`, ktorá uloží momentálny graf do textového súboru tak, aby fungovalo spätné prečítanie metódou `__init__(meno_suboru)`; metódu otestujte tak, že do prečítaného grafu pridáte nový vrchol a aj nejaké hrany, napr.

```
graf = Graf('subor.txt')
graf.pridaj_vrchol('Levice', 330, 250)        # poloha v grafickej ploche
graf.pridaj_hranu('Nitra','Levice',45)
graf.pridaj_hranu('Levice','Komarno',75)
graf.pridaj_hranu('Levice','BBystrica',100)
graf.kresli()
graf.zapis('novy.txt')
```

7. Zapíšte funkciu `generuj(n)`, ktorá zadefinuje graf (typu `Graf` z príkladu (6)) a vráti ho ako výsledok funkcie; tento graf bude mať `n` vrcholov:
   - vrcholy budú v grafickej ploche umiestnené na náhodných pozíciách
   - ich mená vygenerujte náhodne 3-písmenové reťazce (zložené len z veľkých písmen)
   - hranami ich pospájajte tak, že prvý vygenerovaný vrchol je spojený s druhým, druhý s tretím, ... posledný s prvým (keďže graf bude neorientovaný, pre `n`>2 bude mať každý vrchol dvoch susedov)
   - hodnoty na hranách budú: 1 pre prvú hranu, 2 pre druhú, ... `n` pre poslednú

```
def generuj(n):
    ...
```

```
graf = generuj(10)
graf.kresli()
graf.zapis('generuj10.txt')
```

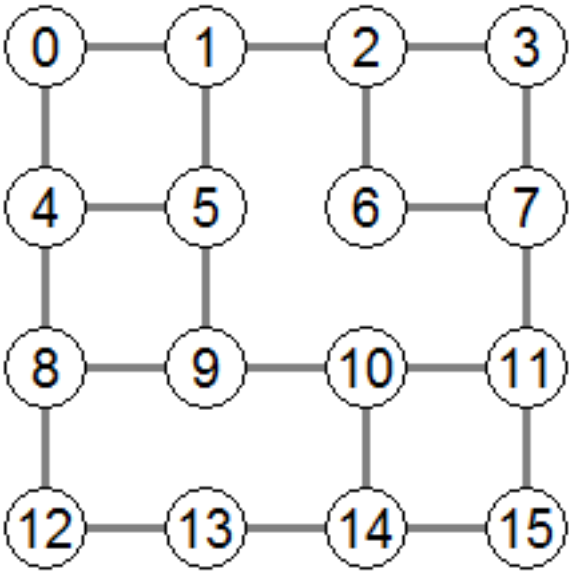
8. Do triedy `Graf` z príkladu (6) dodefinujte metódu `cesta(pole)`, ktorá zafarbí červenou farbou hrany zadanej cesty v nakreslenom grafe
   - cesta je popísaná poľom, ktoré obsahuje postupnosť mien vrcholov
   - metóda vráti `False`, ak zadaná cesta obsahuje neexistujúcu hranu, inak vráti `True`

```
graf = Graf()
graf.citaj('subor.txt')
graf.kresli()
print(graf.cesta(['Trnava','Nitra','Partizanske','Komarno']))
```

Cvičenie

21.1 Prehľadávanie grafu

1. Pre daný graf zostavte (ručne na papieri) poradie navštívených vrcholov rôznymi algoritmami, pričom štartujeme vo vrchole č. 10:



   - tento graf bol vytvorený rovnakou inicializáciou ako bola v prednáške
   - rekurzívny algoritmus do dohĺbky:

```
def vypis1(self, v1):
    def dohlbky(v1):
        visited.add(v1)
        print(v1, end=' ')
        for v2 in sorted(self.pole[v1].sus):
            if v2 not in visited:
                dohlbky(v2)

    visited = set()
    dohlbky(v1)
    print()
```

2. Pre daný graf z úlohy (1) zostavte (ručne na papieri) poradie navštívených vrcholov - opäť začíname vo vrchole 10

   - nerekurzívny algoritmus dohĺbky:

```
def vypis2(self, v1):
    visited = set()
    stack = [v1]
    while stack:
        v1 = stack.pop()
        if v1 not in visited:
          visited.add(v1)
          print(v1, end=' ')
          for v2 in sorted(self.pole[v1].sus):    # sorted(self.pole[v1].sus,␣
↪reverse=True):
              if v2 not in visited:
                  stack.append(v2)
    print()
```

   - zmení sa niečo, keď utriedenie vrcholov vo for-cykle bude v opačnom poradí (`reverse=True`)?

3. Pre daný graf z úlohy (1) zostavte (ručne na papieri) poradie navštívených vrcholov - opäť začíname vo vrchole 10

- algoritmus došírky:

```
def vypis3(self, v1):
    visited = set()
    queue = [v1]
    while queue:
        v1 = queue.pop(0)
        if v1 not in visited:
            visited.add(v1)
            print(v1, end=' ')
            for v2 in sorted(self.pole[v1].sus):
                if v2 not in visited:
                    queue.append(v2)
    print()
```

4. Rovnaké zadanie ako v úlohe (3), len algoritmus je doplnený o evidovanie úrovne (vzdialenosti) vrcholov

   - algoritmus došírky aj s úrovňami:

```
def vypis4(self, v1):
    visited = set()
    queue = [(v1, 0)]
    while queue:
        v1, u = queue.pop(0)             # z frontu sme vybrali vrchol a jeho␣
↪uroven
        if v1 not in visited:
            visited.add(v1)
            print((v1, u), end=' ')
            for v2 in sorted(self.pole[v1].sus):
                if v2 not in visited:
                    queue.append((v2, u+1))       # vlozili sme vrchol a jeho␣
↪uroven
    print()
```

5. Aby sme mohli preveriť ručné výpisy z úloh (1) - (4), umožníme pri inicializácii grafu namiesto náhodných hrán zadanať nami špecifikovanú množinu hrán.

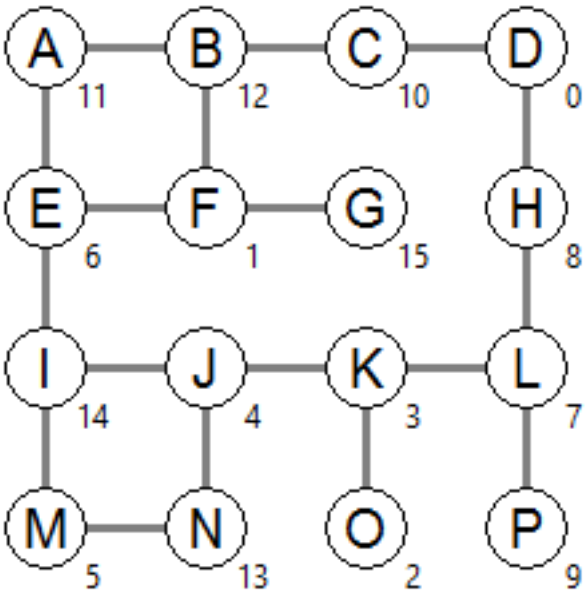
- inicializácia `__init__()` bude mať pridaný parameter, v ktorom vymenujeme množinu hrán grafu:
  - v prípade, že parameter `hrany` bude mať hodnotu `None`, metóda bude pracovať ako doteraz - generovať náhodné hrany
  - inak, parameter `hrany` bude obsahovať dvojice vrcholov - namiesto náhodných hrán sa teraz zadefinujú práve tieto hrany

```
def __init__(self, n, hrany=None):
    ...
```

- a spustite, napr.

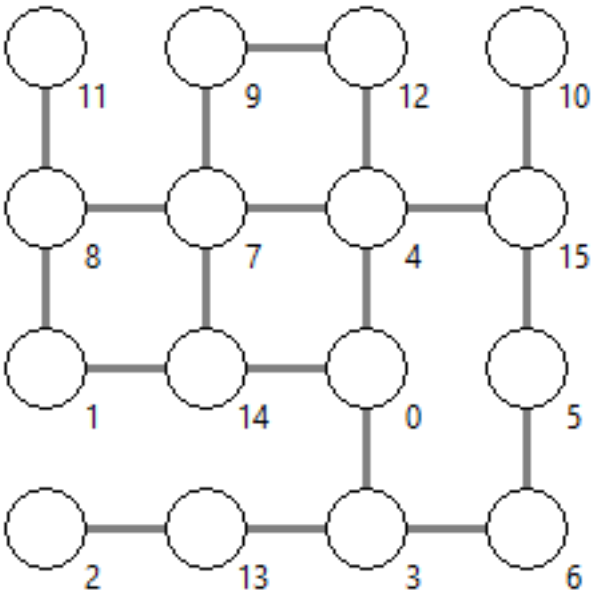
```
g = Graf(4, [(0,1), (1,2), (0,4), ...])
g.vypis1(10)
g.vypis2(10)
g.vypis3(10)
g.vypis4(10)
```

6. Vo vrcholoch grafu sa nachádza atribút `meno`, v ktorom sme doteraz uchovávali poradové číslo vrcholu. Teraz sme sem vložili velké písmená od 'A' do 'P':



- čísla pri každom vrchole vyjadrujú ich indexy v `self.pole` (ich pôvodné mená)
- uvedomte si, že tento graf musel byť skonštruovaný inak, ako sme to doteraz robili v `__init__()`
- metódy `vypis1()` až `vypis4()` z úloh (1) až (4) sme upravili tak, že sa nevypisuje index (`print(v1,end=' ')`), ale tento atribút `meno`
- zistite, čo sa vypíše, keď všetky tieto výpisy začneme vo vrchole s indexom 0

7. Budete riešiť opačnú úlohu k (6): vieme, že výpis atribútu `meno` vrcholov bol v poradí 'A B C D E F G H I J K L M N O P'



- zistite, ako vyzeral graf (teda písmená v atribúte `meno`), ak sa toto poradie vytvorilo pre tieto algoritmy:
  - `vypis1()` - rekurzívny "dohlbky"
  - `vypis1()` - nerekurzívny "dohlbky"
  - `vypis1()` - "dosirky"

8. Do triedy `Graf` dopíšte metódy, ktoré zisťujú najväčší komponent v grafe:
   - triedu `Graf` použite z prednášky

```
class Graf:

    ...

    def max(self):          # počet vrcholov v najväčšom komponente
        ...
    def max_set(self):      # množina vrcholov (indexov do ``self.pole``) v␣
↪najväčšom komponente
        ...
```

9. Do triedy `Graf` dopíšte metódy
   - `presne(v1,d)` - pre daný vrchol `v1` a vzdialenosť `d` vráti množinu všetkých vrcholov, ktorých najkratšia vzdialenosť od `v1` je presne `d`
   - `zafarbi_presne(v1,d,farba)` - pre daný vrchol `v1` a vzdialenosť `d` zafarbí všetky vrcholy, ktorých najkratšia vzdialenosť od `v1` je presne `d`
   - `zafarbi_blizko(v1,d,farba)` - pre daný vrchol `v1` a vzdialenosť `d` zafarbí všetky vrcholy, ktorých najkratšia vzdialenosť od `v1` nie je väčšia `d`

```
class Graf:

    ...

    def presne(self, v1, d):
        ...
    def zafarbi_presne(self, v1, d, farba):
```

```
        ...
    def zafarbi_blizko(self, v1, d, farba):
        ...
```

KAPITOLA 22

Cvičenie

22.1 Backtracking

0. Stiahnite si aplikáciu `VizTree` z adresy http://edi.fmph.uniba.sk/~tomcsanyiova/VizTree.zip
   - rozzipujte ju a spustite `VizTree.jar`
   - vyberte si ľubovoľné dve zadania z menu **Assignment** a skúste ich vyriešiť
1. Napíšte funkciu `generuj(n)`, ktorá vygeneruje a vypíše všetky `n`-tice čísel z `<0,n-1>`, v ktorých i-ty prvok nie je väčší ako (i+1)-ty
   - funkcia by nemala používať žiadne globálne premenné (napr. `pole`)
   - napr.

```
>>> generuj(2)
0 0
0 1
1 1
>>> generuj(3)
0 0 0
0 0 1
0 0 2
0 1 1
0 1 2
0 2 2
1 1 1
1 1 2
1 2 2
2 2 2
```

   - **pomôcka**: môžete využiť generovanie n-tíc z prednášky, pričom samotnú rekurzívnu funkciu vnoríte do funkcie `generuj()`:

```
def generuj(n):
    def generuj_rek(i):
        ...
            generuj_rek(i+1)

    pole = [0]*n
    generuj_rek(0)
```

```
generuj(2)
generuj(3)
```

2. Napíšte funkciu `vsetky(mnozina)`, ktorá pre danú množinu písmen (kde počet písmen v množine je `n`) vygeneruje a vypíše všetky `n`-znakové slová, ktoré obsahujú len písmená z danej množiny

   - napr. (výpis v ľubovoľnom poradí)

```
>>> vsetky({'x','y'})
xx
xy
yx
yy
```

   - **pomôcka**: opäť využijete vnorenú rekurzívnu funkciu:

```
def vsetky(mnozina):
    def generuj_rek():
        ...
            generuj_rek()

    slovo = []                  # slovo = ''
    generuj_rek()

vsetky({'x','y'})
vsetky(set('ahoj'))
```

   - uvedomte si, že ak by bolo `slovo = ''`, úloha by sa riešila trochu komplikovanejšie

3. Napíšte funkciu `vsetky_len_raz(mnozina)`, ktorá pre danú množinu písmen (kde počet písmen v množine je `n`) vygeneruje a vypíše všetky `n`-znakové slová, ktoré obsahujú všetky písmená z množiny (každé práve raz)

   - napr.

```
>>> vsetky_len_raz({'a','k','m'})
akm
amk
kam
kma
mak
mka
```

   - **pomôcka**: do riešenia z úlohy (2) pridajte test, ktorým zistíte, či sa pridávaný znak už nenachádza vo vytváranom slove

```
def vsetky_len_raz(mnozina):
    ...
```

4. Úlohy (2) a (3) riešte tak, aby funkcie nič nevypisovali, ale namiesto toho vrátili pole vygenerovaných slov

   - napr.

```
>>> vsetky_pole({'x','y'})
['xx', 'xy', 'yx', 'yy']
>>> vsetky_len_raz_pole(set('akm'))
['akm', 'amk', 'kam', 'kma', 'mak', 'mka']
```

5. Upravte program n-dám na šachovnici n x n z prednášky tak, aby našiel všetky riešenia pre n-dám na šachovnici n x m, kde n < m

   - napr.

```
>>> Damy(2, 3).ries()            # šachovnica vel'kosti 2x3
0 2
2 0
pocet rieseni: 2
```

   - **pomôcka**:

```
class Damy:
    def __init__(self, n, m):
        self.n, self.m = n, m
        ...

Damy(2, 3).ries()
Damy(3, 4).ries()
```

6. Riešte takýto problém:

   - n detí sa príhlásilo na k rôznych krúžkov (každé dieťa si zvolilo svoju množinu krúžkov)
   - keďže kapacita každého krúžku je obmedzená (hodnotou limit), vedenie školy rozhodlo, že každé dieťa bude navštevovať len jeden z krúžkov, do ktorých sa prihlásilo
   - priraďte každé dieťa do jedného z krúžkov tak, aby sa nepresiahol zadaný limit a pritom každé z detí bude chodiť do jedného zo svojich zvolených krúžkov
   - napr. takto vyzerá textový súbor s menami detí a ich vybraných krúžkov:

```
ana kreslenie sach foto
boris futbal kreslenie
cyril futbal foto
dasa foto
eva sach kreslenie
fero futbal sach
gusto futbal kreslenie
hana kreslenie foto sach
ivan kreslenie futbal
jana foto sach
karol kreslenie foto
lucia sach futbal
```

   - vypíšte ku každému krúžku zoznam detí, napr. pre limit=3 môže byť takýto výsledok:

```
kreslenie: eva ivan karol
sach: jana lucia fero
futbal: cyril boris gusto
foto: hana dasa ana
```

   - **pomôcka**:
     - prečítajte súbor a nejako rozumne to uchovajte v nejakej dátovej štruktúre (napr. asociatívne pole množín krúžkov - každé z detí ma priradenú svoju množinu)
     - pre každý krúžok pripravíte prázdnu množinu (alebo pole)
     - rekurzívny backtracking spracuje jedno meno dieťaťa (postupne ho vkladá do príslušných množín krúžkov) a zakaždým sa rekurzívne zavolá s nasledovným menom dieťaťa

  - keď už sa spracovali všetky deti, vypíšu sa množiny krúžkov a prehľadávanie môže skončiť (stačí nám jedno riešenie, lebo môže ich byť veľmi veľa)

7. **ručne** zistite, čo vygeneruje tento backtracking

   - všimnite si, že `pole` je dvojrozmerným poľom znakov:

```
def backtracking(r, s):
    if 0 <= r < len(pole) and 0 <= s < len(pole[r]):
        if pole[r][s] == 'x':
            print('\n'.join(' '.join(r) for r in pole), '\n')
        elif pole[r][s] == '.':
            pole[r][s] = '-'; backtracking(r, s+1)
            pole[r][s] = '|'; backtracking(r+1, s)
            pole[r][s] = '\\'; backtracking(r+1, s+1)
            pole[r][s] = '.'

pole = [list(r) for r in ('...','..x')]
backtracking(0, 0)
```

8. Zmenou poľa v predchádzajúcom príklade sa mení aj zoznam vygenerovaných riešení

   - zistite, **koľko rôznych riešení** sa vygeneruje pre takto definované `pole`:

```
pole = [list(r) for r in ('...','.m.','..x')]
```

   alebo:

```
pole = list(map(list, ('...m','..m.','.m..','m..x')))
```

9. Upravte funkciu `backtracking()` z úlohy (7) tak, aby nevypisovala všetky riešenia, len vrátila počet všetkých riešení

   - funkciu môžete zapúzdriť do nejakej triedy

Cvičenie

23.1 Backtracking na grafoch

Na riešenie úloh si z prednášky okopírujte definíciu triedy `Graf` a tiež niektoré ďalšie metódy.

1. Do triedy `Graf` dopíšte generovanie náhodných hodnôt na hranách grafu z intervalu `<1,4>`, túto hodnotu vypíšte do stredu príslušnej hrany

```
class Graf:
    ...
    def __init__(self, n):
        ...
    def kresli_hranu(self, i, j, farba='gray'):
        ...
```

2. Otestujte backtracking z prednášky, ktorý hľadá najkratšiu (resp. najdlhšiu) cestu medzi dvoma kliknutými vrcholmi.
   - počas hľadania cesty sa farebne vyfarbujú hrany (na červeno), cez ktoré práve prechádza
   - na záver sa nájdená cesta zafarbí na modro (alebo vypíše správa, že sa nenašla)
   - testujte s ohodnotením hrán z úlohy (1)

```
class Graf:
    ...
def udalost_klik(self, event):
    ...
def start(self, v1, v2):
    ...
def backtracking(self, v1, v2):
    ...
```

   - môžete atribúty `hodnota` a `cesta` nahradiť parametrami metódy `backtracking()`

3. Do náhodného generovania hrán grafu v inicializácii `__init__()` pridajte generovanie aj šikmých hrán
   - s nejakou pravdepodobnosťou spojí momentálny vrchol `v1` s vrcholom, ktorý je o riadok aj o stĺpec o 1 menší (zrejme to nerobí v prvom riadku ani stĺpci)

```
class Graf:
    ...
    def __init__(self, n):
        ...
```

- otestujte hľadanie ciest z (2) aj na takomto grafe

4. Zmeňte metódu `backtracking()` tak, aby našla najkratší (resp. najdlhší) cyklus, ktorý začína (a teda aj končí) vo vrchole `v1` a prechádza cez `v2`

- ```
  class Graf:
      ...
  def backtracking(self, v1, v2):
      ...
  ```

5. Napíšte metódu `zisti5()`, ktorá pomocou algoritmu prehľadávania s návratom zistí, koľko existuje v danom grafe rôznych cyklov dĺžky presne 5 (zaujíma nás počet vrcholov a nie váhy na hranách)

   - zrejme v pôvodnom grafe, v ktorom boli hrany náhodne generované len vodorovne a zvislo, nemôže existovať žiaden cyklus dĺžky 5
   - metóda na záver vypíše tento zistený počet

   ```
   class Graf:
       ...
   def zisti5(self):
       ...
   ```

6. Pre danú konštantu `D` a kliknutím na jeden vrchol zafarbite (napr. žltou farbou) všetky také vrcholy, pre ktoré existuje cesta s hodnotou presne `D`, ktorá vychádza so zadaného (kliknutého) vrcholu

- ```
  D = 10

  class Graf:
      ...
  def udalost_klik(self, event):
      ...
  def start(self, v1):
      ...
  def backtracking(self, ...):
      ...
  ```

7. Zmeňte metódu `backtracking()` tak, aby najkratšia (resp. najdlhšia) cesta mohla prechádzať cez tie isté vrcholy aj viackrát, len treba strážiť, aby nešla viackrát po tej istej hrane

- ```
  class Graf:
      ...
  def start(self, v1, v2):
      ...
  def backtracking(self, v1, v2, ...):
      ...
  ```

8. Ohodnotený neorientovaný graf je definovaný takto:

   - v každom riadku je popísaná jedna hrana ako čísla dvoch vrcholov a hodnota (reťazec) na hrane

   ```
   1 2 a
   1 4 ma
   1 5 a
   3 4 m
   4 2 am
   6 4 a
   5 6 u
   5 3 am
   4 5 em
   ```
   ```

- zistite, koľko existuje rôznych ciest (cez vrcholy aj hrany môžeme prechádzať ľubovoľný počet krát), ktorých hodnota je reťazec `'mamamaemuaemamamamu'`
- úlohu riešte prečítaním grafu zo súboru a potom pomocou backtrackingu
- program vypíše všetky cesty ako postupnosti navštívených vrcholov