5. Podprogramy

video prezentácia

funkcie

Funkcie

Doteraz sme pracovali so štandardnými funkciami, napríklad

- vstup a výstup `input()` a `print()`
- aritmetické funkcie `abs()` a `round()`
- generovanie postupnosti čísel pre for-cyklus `range()`

Všetky tieto funkcie niečo vykonali (vypísali, prečítali, vypočítali, …) a niektoré z nich vrátili nejakú hodnotu, ktorú sme mohli ďalej spracovať. Tiež sme videli, že niektoré majú rôzny počet parametrov, prípadne sú niekedy volané bez parametrov.

Okrem toho sme pracovali aj s funkciami, ktoré boli definované v iných moduloch:

- keď napíšeme `import random`, môžeme pracovať, napríklad s funkciami `random.randint()` a `random.randrange()`
- keď napíšeme `import math`, môžeme pracovať, napríklad s funkciami `math.sin()` a `math.cos()`
- keď napíšeme `import tkinter`, môžeme pracovať, napríklad s funkciami `tkinter.Canvas()` a `tkinter.mainloop()`

Všetky tieto a tisícky ďalších v Pythone naprogramovali programátori pred nejakým časom, aby nám neskôr zjednodušili samotné programovanie. Vytváranie vlastných funkcií pritom vôbec nie je komplikované a teraz sa to naučíme aj my.

Funkcie

Funkcia je pomenovaný blok príkazov (niekedy sa tomu hovorí aj podprogram). Popisujeme (**definujeme**) ju špeciálnou konštrukciou:

```
def meno_funkcie():      # zapamätaj si blok príkazov ako nový príkaz
    prikaz
    prikaz
    ...
```

Keď zapíšeme definíciu funkcie, zatiaľ sa z bloku príkazov (hovoríme tomu **telo funkcie**) nič nevykoná. Táto definícia sa „len“ zapamätá a jej **referencia** sa priradí k zadanému menu - vlastne sa do premennej `meno_funkcie` priradí referencia na telo funkcie. Je to podobné tomu, ako sa priraďovacím príkazom do premennej priradí hodnota z pravej strany príkazu.

Ako prvý príklad zapíšme takúto definíciu funkcie:

```
def vypis():
    print('**********')
    print('**********')
```

Zadefinovali sme funkciu s menom `vypis`, pričom telo funkcie obsahuje dva príkazy na výpis riadkov s hviezdičkami. Celý blok príkazov je odsunutý o 4 medzery rovnako ako sme odsúvali príkazy v cykloch a aj v podmienených príkazoch. Definícia tela funkcie končí vtedy, keď sa objaví riadok, ktorý už nie je

odsunutý. Touto definíciou sa ešte žiadne príkazy z tela funkcie nevykonávajú. Na to potrebujeme túto funkciu **zavolať**.

Volanie funkcie

Volanie funkcie je taký zápis, ktorým sa začnú vykonávať príkazy z definície funkcie. Stačí zapísať meno funkcie so zátvorkami a funkcie sa spustí:

```
meno_funkcie()
```

Samozrejme, že funkciu môžeme zavolať až vtedy, keď už Python pozná jej definíciu.

Zavolajme funkciu `vypis` v príkazovom režime:

```
>>> vypis()
**********
**********
>>>
```

Vidíme, že sa vykonali oba príkazy z tela funkcie a potom Python ďalej čaká na ďalšie príkazy. Zapíšme volanie funkcie aj s jej definíciou priamo do skriptu (teda v programovom režime):

```
def vypis():
    print('**********')
    print('**********')

print('hello')
vypis()
print('* Python *')
vypis()
```

Skôr, ako to spustíme, si uvedomme, čo sa udeje pri spustení:

- zapamätá sa definícia funkcie v premennej `vypis`
- vypíše sa slovo `'hello'`
- zavolá sa funkcia `vypis()`
- vypíše riadok s textom `'* Python *'`
- znovu sa zavolá funkcia `vypis()`

A teraz to spustíme:

```
hello
**********
**********
* Python *
**********
**********
```

Zapíšme teraz presné kroky, ktoré sa vykonajú pri volaní funkcie:

1. preruší sa vykonávanie práve bežiaceho programu (Python si presne zapamätá miesto, kde sa to stalo)
2. skočí sa na začiatok volanej funkcie
3. postupne sa vykonajú všetky príkazy
4. keď sa príde na koniec funkcie, zrealizuje sa **návrat** na zapamätané miesto, kde sa prerušilo vykonávanie programu a pokračuje sa vo vykonávaní ďalších príkazov za volaním funkcie

Pre volanie funkcie sú veľmi dôležité okrúhle zátvorky. Bez nich to už nie je volanie, ale len zisťovanie referencie na hodnotu, ktorá je priradená pre toto meno. Napríklad:

```
>>> vypis()
**********
```

```
**********
>>> vypis
<function vypis at 0x0205CB28>
```

Ak by sme namiesto volania funkcie takto zapísali len meno funkcie bez zátvoriek, ale v skripte (teda nie v interaktívnom režime), táto hodnota referencie by sa nevypísala, ale odignorovala. Toto býva dosť častá chyba začiatočníkov, ktorá sa ale ťažšie odhaľuje.

Ak zavoláme funkciu, ktorú sme ešte nedefinovali, Python vyhlási chybu, napríklad:

```
>>> vipis()
...
NameError: name 'vipis' is not defined
```

Samozrejme, že môžeme volať len definované funkcie.

```
>>> vypis()
**********
**********
>>> vypis = 'ahoj'      # tu sme zmenili obsah premennej vypis
>>> vypis
'ahoj'
>>> vypis()
...
TypeError: 'str' object is not callable
```

Hodnotou premennej `vypis` je už teraz znakový reťazec, a ten sa „nedá zavolať“, t.j. nie je „callable“ (tento objekt nie je zavolateľný ako funkcia).

Funkcie kreslia do grafickej plochy

Napíšme teraz funkciu, ktorá do grafickej plochy na náhodnú pozíciu napíše nejaký text, napríklad `'PYTHON'`:

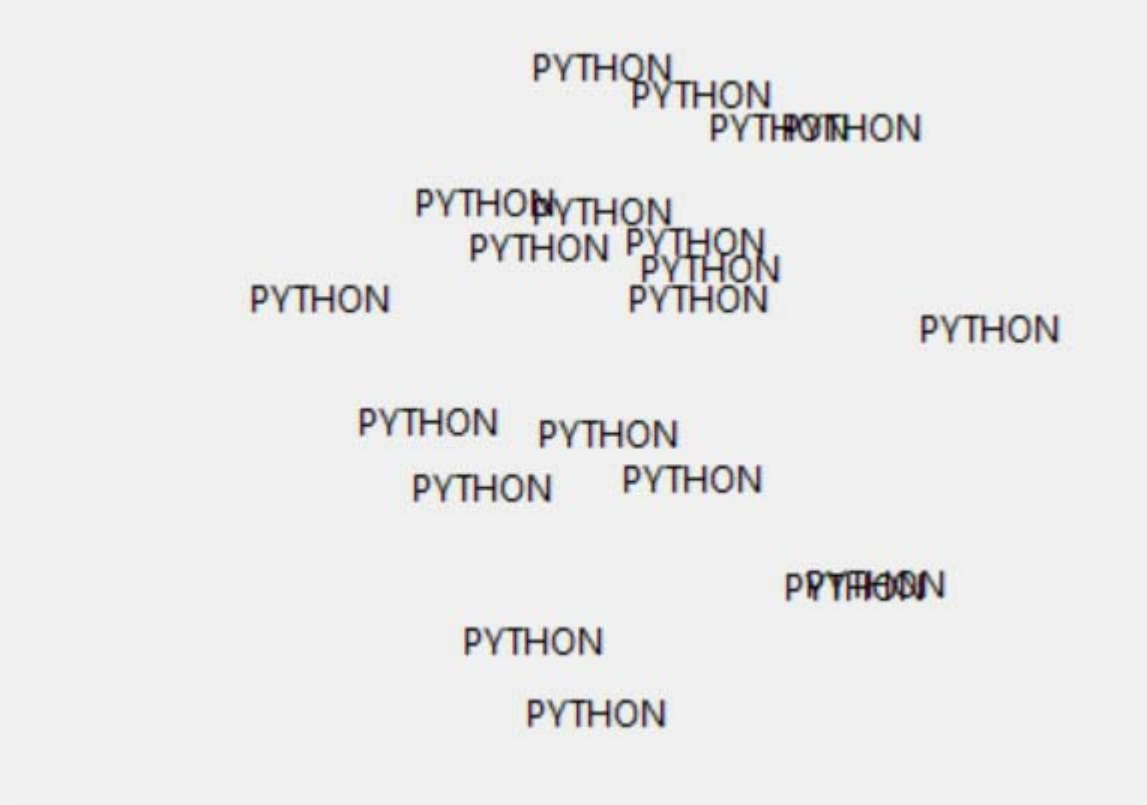
```
def kresli_text():
    x = random.randint(50, 330)
    y = random.randint(20, 240)
    canvas.create_text(x, y, text='PYTHON')
```

Aby sme túto funkciu mohli zavolať, musí už exitovať `randint` (zrejme z modulu `random`) aj `canvas` (vznikne pomocou modulu `tkinter`), teda

```
import tkinter
import random

def kresli_text():
    x = random.randint(50, 330)
    y = random.randint(20, 240)
    canvas.create_text(x, y, text='PYTHON')

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

for i in range(20):
    kresli_text()

tkinter.mainloop()
```

V tejto malej ukážke vidíme tieto novinky:

- v tele funkcie môžeme používať premenné, ktoré sú definované mimo tela funkcie (`random.randint` a `canvas`) - neskôr uvidíme, že im budeme hovoriť **globálne premenné**
- v tele funkcie sme do dvoch premenných `x` a `y` priradili nejaké hodnoty a ďalej sme ich používali v ďalšom príkaze funkcie - neskôr uvidíme, že im budeme hovoriť **lokálne premenné**



Parametre funkcií

Hotové funkcie, s ktorými sme doteraz pracovali, napríklad `print()` alebo `random.randint()`, mali aj parametre, vďaka čomu riešili rôzne úlohy. Parametre slúžia na to, aby sme mohli funkcii lepšie oznámiť, čo špecifické má urobiť: čo sa má vypísať, z akého intervalu má vygenerovať náhodné číslo, akú úsečku má nakresliť, prípadne akej farby, …

Parametre funkcie

Parametrom funkcie je **dočasná premenná**, ktorá vzniká pri volaní funkcie a prostredníctvom ktorej, môžeme do funkcie *poslať* nejakú hodnotu. Parametre funkcií definujeme počas definovania funkcie v **hlavičke funkcie** a ak ich je viac, oddeľujeme ich čiarkami:

```
def meno_funkcie(parameter):
    prikaz
    prikaz
    ...
```

Môžeme napríklad zapísať:

```
def vypis_hviezdiciek(pocet):
    print('*' * pocet)
```

V prvom riadku definície funkcie (hlavička funkcie) pribudla jedna premenná `pocet` - parameter. Táto premenná vznikne automaticky pri volaní funkcie, preto musíme pri volaní oznámiť hodnotu tohto parametra. Volanie zapíšeme:

```
>>> vypis_hviezdiciek(30)
******************************
>>> for i in range(1, 10):
        vypis_hviezdiciek(i)

*
**
***
****
*****
******
*******
********
*********
```

Pri volaní sa „skutočná hodnota“ **priradí** do parametra funkcie (premenná `pocet`).

Už predtým sme popísali mechanizmus volania funkcie, ale to sme ešte nepoznali parametre. Teraz doplníme tento postup o spracovanie parametrov. Najprv trochu terminológie:

- pri definovaní funkcie v hlavičke funkcie uvádzame tzv. **formálne parametre**: sú to nové premenné, ktoré vzniknú až pri volaní funkcie
- pri volaní funkcie musíme do zátvoriek zapísať hodnoty, ktoré sa stanú tzv. **skutočnými parametrami**: tieto hodnoty sa pri volaní priradia do formálnych parametrov

Mechanizmus volania vysvetlíme na volaní `vypis_hviezdiciek(30)`:

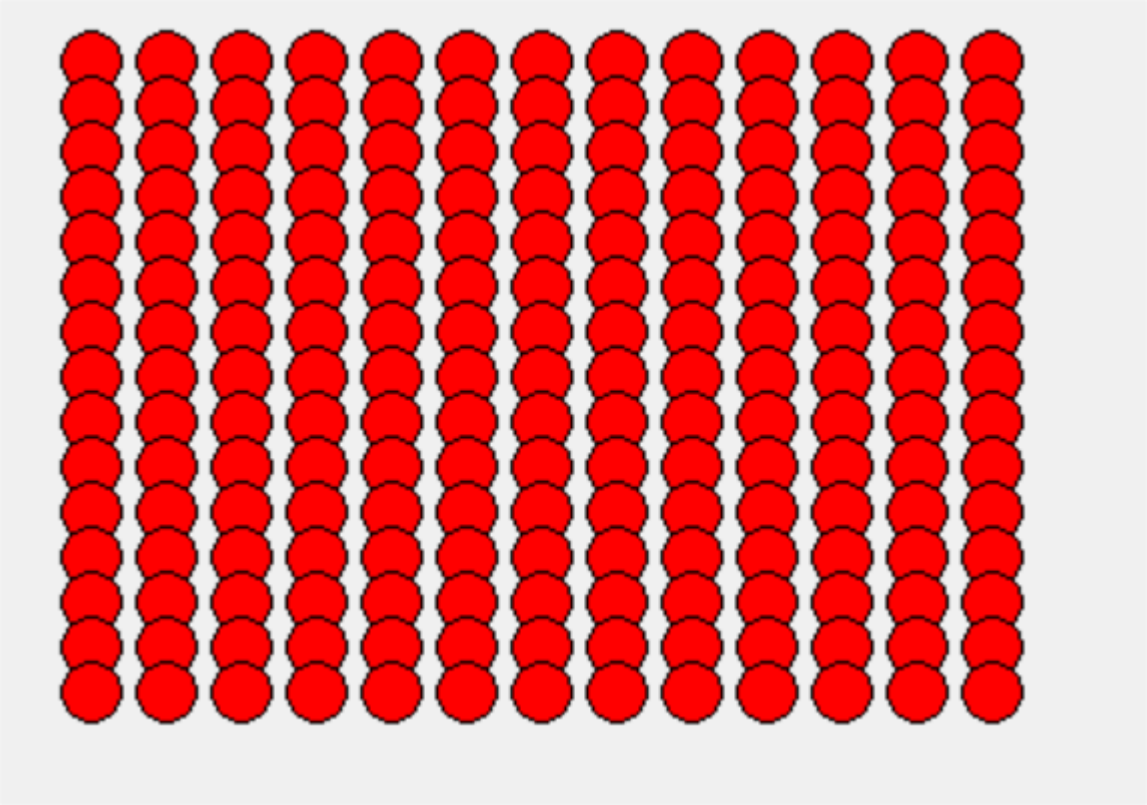
1. zapamätá sa návratová adresa volania
2. vytvorí sa **nová** premenná `pocet` (**formálny parameter**) a priradí sa do nej hodnota **skutočného parametra** `30`
3. vykonajú sa všetky príkazy v definícii funkcie (**telo funkcie**)
4. zrušia sa všetky premenné, ktoré vznikli počas behu funkcie
5. riadenie sa vráti na miesto, kde bolo volanie funkcie

Zapíšme novú funkciu `cerveny_kruh()`, ktorá bude mať dva parametre: súradnice stredu kruhu. Funkcia nakreslí kruh s polomerom 10 a s daným stredom:

```
def cerveny_kruh(x, y):
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
```

a môžeme ju zavolať napríklad takto:

```
import tkinter

def cerveny_kruh(x, y):
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

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

for x in range(30, 350, 25):
    for y in range(20, 240, 15):
        cerveny_kruh(x, y)

tkinter.mainloop()
```



Aj v tomto príklade si popíšme **Mechanizmus volania** funkcie:

1. zapamätá sa návratová adresa volania
2. vytvoria sa dve **nové** premenné `x` a `y` (**formálne parametre**) a priradia sa do nej hodnoty **skutočných parametrov** `30` a `20` (prvé volanie funkcie)
3. vykonajú sa všetky príkazy v definícii funkcie (**telo funkcie**)
4. zrušia sa všetky premenné, ktoré vznikli počas behu funkcie, teda `x` aj `y`
5. riadenie sa vráti na miesto, kde bolo volanie funkcie

Už vieme, že priraďovací príkaz vytvára premennú a referenciou ju spojí s nejakou hodnotou. Premenné, ktoré **vzniknú počas behu funkcie**, sa stanú **lokálnymi premennými**: budú existovať len počas tohto behu a po skončení funkcie, sa automaticky zrušia. Aj parametre vznikajú pri štarte funkcie a zanikajú pri jej skončení: tieto premenné sú pre funkciu tiež lokálnymi premennými.

V nasledovnom príklade funkcie `vypis_sucet()` počítame a vypisujeme súčet čísel od 1 po zadané `n`:

```
def vypis_sucet(n):
    sucet = 1
    print(1, end=' ')
    for i in range(2, n + 1):
        sucet = sucet + i
        print('+', i, end=' ')
    print('=', sucet)
```

Pri volaní funkcie sa pre parameter `n` = 10 vypíše:

```
>>> vypis_sucet(10)
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Počas behu vzniknú 2 lokálne premenné (`sucet` a `i`) a jeden parameter, ktorý je pre funkciu tiež lokálnou premennou:

- `n` vznikne pri štarte funkcie aj s hodnotou 10
- `sucet` vznikne pri prvom priradení `sucet = 1`
- `i` vznikne pri štarte for-cyklu

Po skončení behu funkcie sa všetky tieto premenné automaticky zrušia.

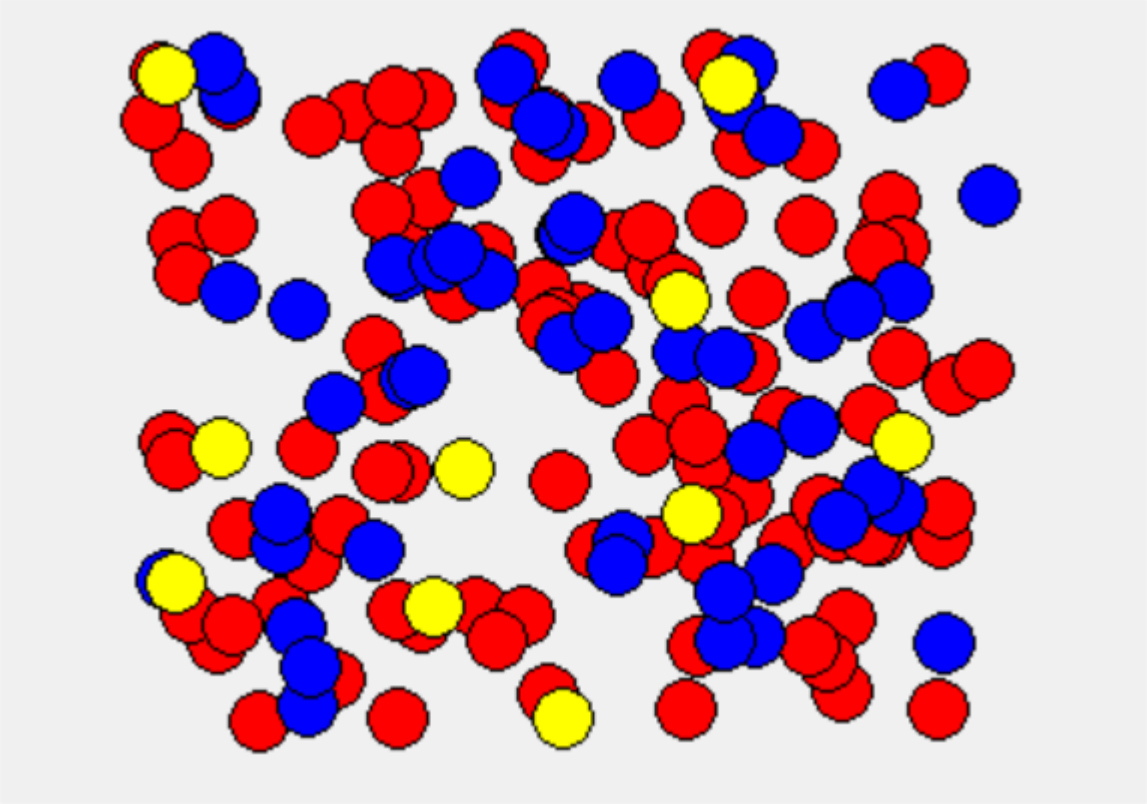
Pozrime sa na **lokálne premenné**, ktoré vznikajú vo funkcii `nahodne_kruhy(n, farba)`:

```
def nahodne_kruhy(n, farba):
    for i in range(n):
        x = random.randint(50, 330)
        y = random.randint(20, 240)
        canvas.create_oval(x-10, y-10, x+10, y+10, fill=farba)
```

Napríklad:

```
import tkinter
import random

def nahodne_kruhy(n, farba):
    for i in range(n):
        x = random.randint(50, 330)
        y = random.randint(20, 240)
        canvas.create_oval(x-10, y-10, x+10, y+10, fill=farba)

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

nahodne_kruhy(100, 'red')
nahodne_kruhy(50, 'blue')
nahodne_kruhy(10, 'yellow')

tkinter.mainloop()
```

Program postupne nakreslí na náhodné pozície 100 červených, 50 modrých a 10 žltých kruhov.



Všimnite si vo funkcii `nahodne_kruhy()` tri lokálne premenné (`i`, `x`, `y`) a dva parametre (`n`, `farba`), ktoré sú pre funkciu tiež lokálnymi premennými.

Menný priestor

Aby sme lepšie pochopili, ako naozaj fungujú **lokálne premenné**, musíme rozumieť, čo to je a ako funguje **menný priestor** (namespace). Najprv trochu ďalšej terminológie: všetky identifikátory v Pythone sú jedným z troch typov (Python má pre identifikátory 3 rôzne tabuľky mien):

- **štandardné**, napríklad `int`, `print`, …
  - hovorí sa tomu **builtins**
- **globálne** - definujeme ich na najvyššej úrovni mimo funkcií, napríklad funkcie `vypis_hviezdiciek`, `vypis_sucet`, `nahodne_kruhy`, alebo aj premenné `randrint`, `tkinter`, `canvas` (zrejme `tkinter` je referenciou na importovaný modul)
  - hovorí sa tomu **main**
- **lokálne** - vznikajú počas behu funkcie

Tabuľka štandardných mien (builtins) je pre celý program len jedna, tiež tabuľka globálnych mien (main) je len jedna, ale každá funkcia má svoju „súkromnú“ lokálnu tabuľku mien, ktorá vznikne pri štarte (zavolaní) funkcie a zruší sa pri konci vykonávania funkcie.

Keď na nejakom mieste použijeme identifikátor, Python ho najprv hľadá (v tzv. **menných priestoroch**):

- v lokálnej tabuľke mien, ak tam tento identifikátor nenájde, hľadá ho
- v globálnej tabuľke mien, ak tam tento identifikátor nenájde, hľadá ho
- v štandardnej tabuľke mien

Ak nenájde v žiadnej z týchto tabuliek, hlási chybu `NameError: name 'identifikátor' is not defined`.

Príkaz (štandardná funkcia) `dir()` vypíše tabuľku globálnych mien. Hoci pri štarte Pythonu by táto tabuľka mala byť prázdna, obsahuje niekoľko špeciálnych mien, ktoré začínajú aj končia znakmi `'__'`:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']
```

Keď teraz vytvoríme nejaké nové globálne mená, objavia sa aj v tejto globálnej tabuľke:

```
>>> premenna = 2015
>>> def funkcia():
        pass

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'funkcia', 'premenna']
```

Podobne sa vieme dostať aj k tabuľke štandardných mien (builtins):

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', ...
```

Takto sa vypíšu všetky preddefinované mená. Vidíme medzi nimi, napríklad `'int'`, `'print'`, `'range'`, `'str'`, …

S týmito tabuľkami súvisí aj príkaz na zrušenie premennej.

príkaz `del`

Príkazom `del` zrušíme identifikátor z tabuľky mien. Formát príkazu:

```
del premenná
```

Príkaz najprv zistí, v ktorej tabuľke sa identifikátor nachádza (najprv pozrie do lokálnej a keď tam nenájde, tak do globálnej tabuľky) a potom ho z tejto tabuľky vyhodí. Príkaz ale nefunguje pre štandardné mená.

Ukážme to na príklade: identifikátor `print` je menom štandardnej funkcie (v štandardnej tabuľke mien). Ak v priamom režime (čo je globálna úroveň mien) do premennej `print` priradíme nejakú hodnotu, toto meno vznikne v globálnej tabuľke:

```
>>> print('ahoj')
ahoj
>>> print=('ahoj')          # do print sme priradili nejakú hodnotu
>>> print
'ahoj'
>>> print('ahoj')
...
TypeError: 'str' object is not callable
```

Teraz už `print` nefunguje ako funkcia na výpis hodnôt, ale len ako obyčajná globálna premenná. Ale v štandardnej tabuľke mien `print` stále existuje, len je táto premenná **prekrytá** globálnym menom. Python predsa najprv prehľadáva globálnu tabuľku a až keď sa tam nenájde, hľadá sa v štandardnej tabuľke. A ako môžeme vrátiť funkčnosť štandardnej funkcie `print`? Stačí vymazať identifikátor z globálnej tabuľky:

```
>>> del print
>>> print('ahoj')
ahoj
```

Vymazaním globálneho mena `print` ostane definovaný len identifikátor v tabuľke štandardných mien, teda opäť začne fungovať funkcia na výpis hodnôt.

Pozrime sa teraz na prípad, keď sa v tele funkcie bude nachádzať volanie inej funkcie (tzv. **vnorené volanie**), napríklad:

```
def vypis_hviezdiciek(pocet):
    print('*' * pocet)

def trojuholnik(n):
    for i in range(1, n+1):
        vypis_hviezdiciek(i)
```

Pri ich definovaní v globálnom mennom priestore vznikli dva identifikátory: `vypis_hviezdiciek` a `trojuholnik`. Zavoláme funkciu `trojuholnik`:

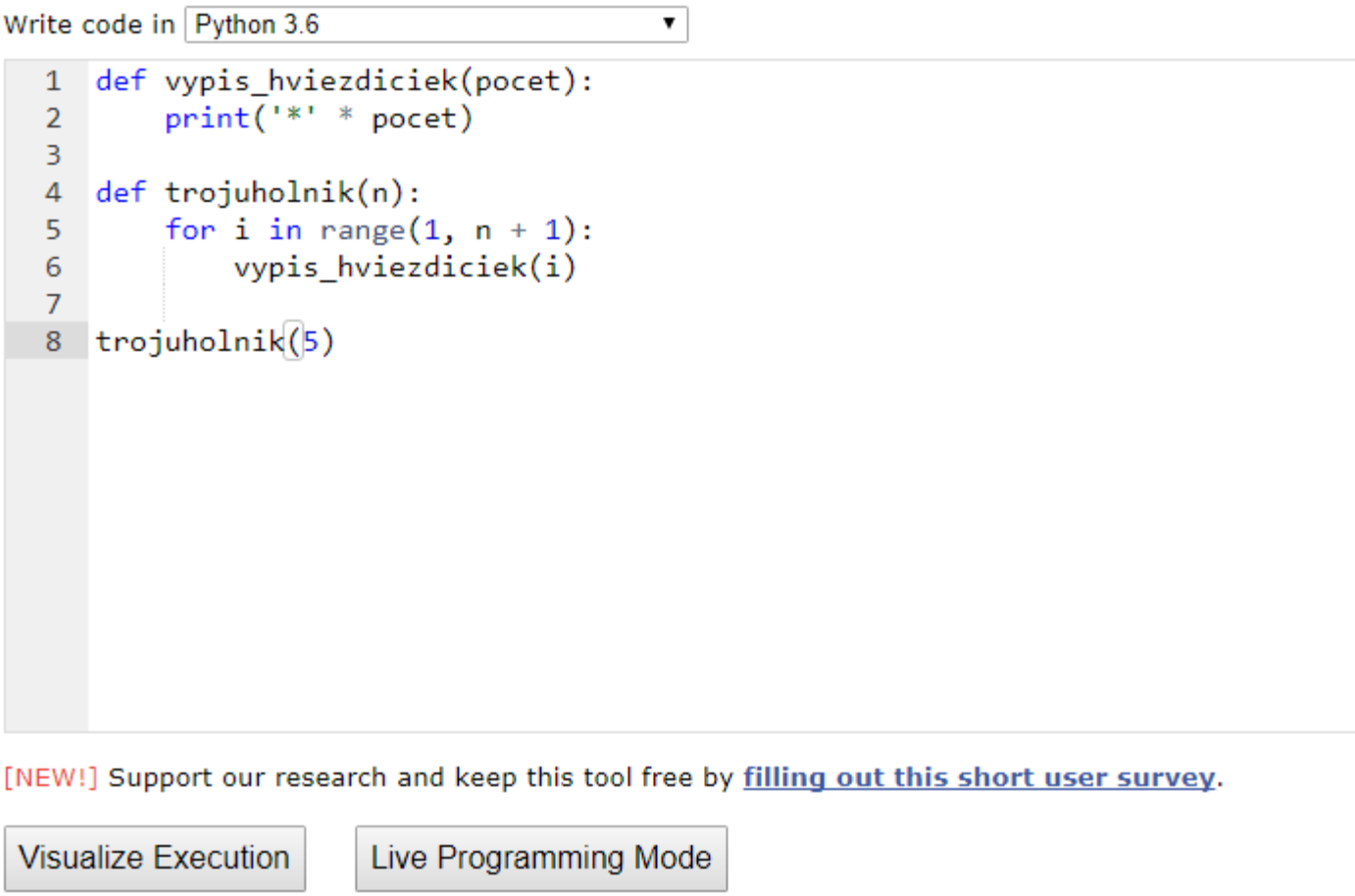
```
>>> trojuholnik(5)
```

Najprv sa pre túto funkciu vytvorí jej menný priestor (lokálna tabuľka mien) s dvomi lokálnymi premennými: `n` a `i`. Teraz **pri každom** (vnorenom) volaní `vypis_hviezdiciek(i)` sa pre túto funkciu:

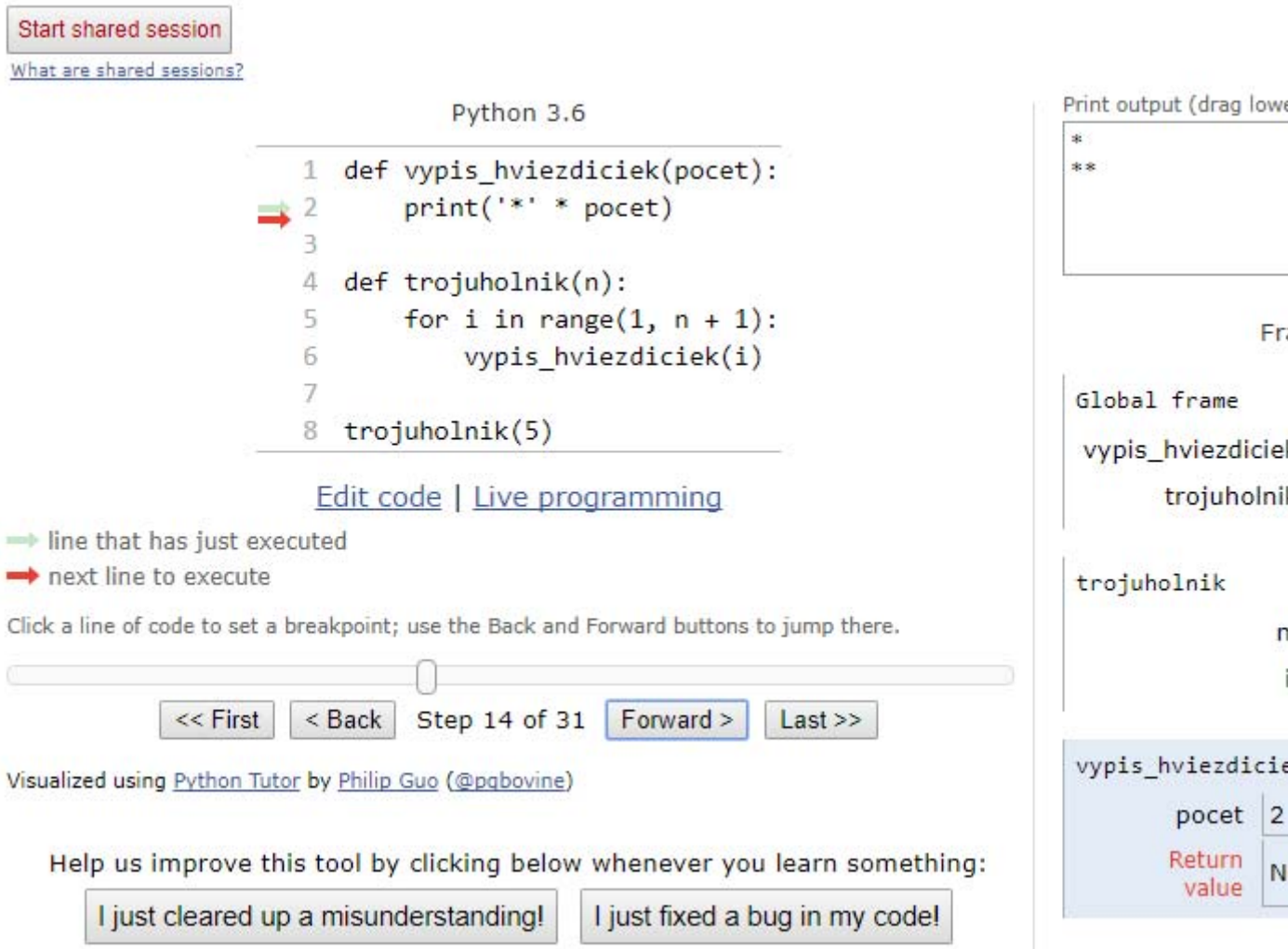
- vytvorí nový menný priestor s jedinou premennou `pocet`
- vykoná sa príkaz `print()`
- nakoniec sa zruší jej menný priestor, t.j. zanikne premenná `pocet`

Môžeme to odkrokovať pomocou http://www.pythontutor.com/visualize.html#mode=edit (zapneme voľbu `Python 3.6`):

- najprv do editovacieho okna zapíšeme nejaký program, napríklad:



- spustíme vizualizáciu pomocou tlačidla **Visualize Execution** a potom niekoľkokrát tlačíme tlačidlo **Forward >**



Všimnite si, že v pravej časti tejto stránky sa postupne zobrazujú menné priestory (tu sa nazývajú **frame**):

- najprv len globálny priestor s premennými `vypis_hviezdiciek` a `trojuholnik`
- potom sa postupne objavujú a aj miznú lokálne priestory týchto dvoch funkcií - na obrázku vidíme oba tieto menné priestory tesne pred ukončením vykonávania funkcie `trojuholnik` s parametrom `2`

Funkcie s návratovou hodnotou

Väčšina štandardných funkcií v Pythone na základe parametrov vráti nejakú hodnotu, napríklad:

```
>>> abs(-5.5)
5.5
>>> round(2.36, 1)
2.4
```

Funkcie, ktoré sme zatiaľ vytvárali my, takú možnosť nemali: niečo počítali, niečo vypisovali, niečo kreslili, ale žiadnu návratovú hodnotu nevytvárali. Aby funkcia mohla vrátiť nejakú hodnotu ako výsledok volania funkcie, musí sa v jej tele objaviť príkaz `return`, napríklad:

```
def meno(parametre):
    prikaz
    prikaz
    ...
    return hodnota                  # tato funkcia vráti výslednú hodnotu
```

Príkazom `return` sa ukončí výpočet funkcie (zruší sa jej menný priestor) a uvedená hodnota sa stáva výsledkom funkcie, napríklad:

```
def eura_na_koruny(eura):                   # prepočítanie na české koruny
    koruny = round(eura * 25.309, 2)
    return koruny
```

môžeme otestovať:

```
>>> print('máš', 123, 'euro, čo je', eura_na_koruny(123), 'českých korún')
máš 123 euro, čo je 3113.01 českých korún
```

Niekedy potrebujeme návratovú hodnotu počítať pomocou nejakého cyklu, napríklad nasledovná funkcia počíta súčet čísel od 1 do `n`:

```
def suma(n):
    vysledok = 0
    while n > 0:
        vysledok += n
        n -= 1
    return vysledok
```

Zároveň vidíme, že formálny parameter (je to predsa lokálna premenná) môžeme v tele funkcie modifikovať.

Už sme videli, že rozlišujeme dva typy funkcií:

- také, ktoré niečo robia (napríklad vypisujú, kreslia, …), ale nevracajú návratovú hodnotu (neobsahujú `return` s nejakou hodnotou)
- také, ktoré niečo vypočítajú a vrátia nejakú výslednú hodnotu - musia obsahovať `return` s návratovou hodnotou

Ďalej ukážeme, že rôzne funkcie môžu vracať hodnoty rôznych typov. Najprv číselné funkcie.

Výsledkom funkcie je číslo

Nasledovná funkcia počíta `n`-tú mocninu dvojky a tento výsledok ešte zníži o 1:

```
def pocitaj(n):
    return 2**n - 1
```

Zrejme výsledkom je vždy len číslo.

Ak chceme funkciu otestovať, buď ju spustíme s konkrétnym parametrom, alebo napíšeme cyklus, ktorý našu funkciu spustí s konkrétnymi hodnotami (niekedy na testovanie píšeme ďalšiu testovaciu funkciu, ktorá nerobí nič iné, „len“ testuje funkciu pre rôzne hodnoty a porovnáva ich s očakávanými výsledkami), napríklad:

```
>>> pocitaj(5)
31
>>> for i in 1, 2, 3, 8, 10, 16, 20, 32:
        print(f'pocitaj({i}) = {pocitaj(i)}')

pocitaj(1) = 1
pocitaj(2) = 3
pocitaj(3) = 7
pocitaj(8) = 255
pocitaj(10) = 1023
```

```
pocitaj(16) = 65535
pocitaj(20) = 1048575
pocitaj(32) = 4294967295
```

Ďalšia funkcia zisťuje dĺžku (počet znakov) zadaného reťazca. Využíva to, že for-cyklus vie prejsť všetky znaky reťazca a s každým môže niečo urobiť, napríklad zvýšiť počítadlo o 1:

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

Otestujeme:

```
>>> dlzka('Python')
6
>>> dlzka(10000 * 'ab')
20000
```

My sme už videli, že existuje štandardná funkcia `len`, ktorá robí toto isté (a zrejme efektívnejšie).

Výsledkom funkcie je logická hodnota

Funkcie môžu vracať aj hodnoty iných typov, napríklad:

```
def parne(n):
    return n % 2 == 0
```

vráti `True` alebo `False` podľa toho či je `n` párne (zvyšok po delení 2 bol 0), vtedy vráti `True`, alebo nepárne (zvyšok po delení 2 nebol 0) a vráti `False`. Túto istú funkciu môžeme zapísať aj tak, aby bolo lepšie vidieť tieto dve rôzne návratové hodnoty:

```
def parne(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Hoci táto verzia robí presne to isté ako predchádzajúca, skúsení programátori radšej používajú kratšiu prvú verziu. Keď chceme túto funkciu otestovať, môžeme zapísať:

```
>>> parne(10)
True
>>> parne(11)
False
>>> for i in range(20, 30):
        print(i, parne(i))

20 True
21 False
22 True
23 False
24 True
25 False
26 True
27 False
```

```
28 True
29 False
```

Výsledkom funkcie je reťazec

Napíšme funkciu, ktorá vráti nejaký reťazec v závislosti od hodnoty parametra:

```
def farba(ix):
    if ix == 0:
        return 'red'
    elif ix == 1:
        return 'blue'
    else:
        return 'yellow'
```

Funkcia vráti buď červenú, alebo modrú, alebo žltú farbu v závislosti od hodnoty parametra.

Opäť by ju bolo dobre najprv otestovať, napríklad:

```
>>> for i in range(6):
        print(i, farba(i))

0 red
1 blue
2 yellow
3 yellow
4 yellow
5 yellow
```

Uvedomte si, prečo ju môžeme zapísať aj takto bez `else` vetiev:

```
def farba(ix):
    if ix == 0:
        return 'red'
    if ix == 1:
        return 'blue'
    return 'yellow'
```

V takýchto prípadoch je na vás, ktorý zápis použijete, ktorý z nich sa vám zdá čitateľnejší. Zamyslite sa, čo bude výsledkom volania `farba(random.randrange(3))`.

Typy parametrov a typ výsledku

Python nekontroluje typy parametrov, ale kontroluje, čo sa s nimi robí vo funkcii. Napríklad funkcia:

```
def pocitaj(x):
    return 2*x + 1
```

bude fungovať pre čísla, ale pre reťazec spadne:

```
>>> pocitaj(5)
11
>>> pocitaj('a')
...
```

```
TypeError: Can't convert 'int' object to str implicitly
```

V tele funkcie ale môžeme kontrolovať typ parametra, napríklad takto

```
def pocitaj(x):
    if type(x) == str:
        return 2*x + '1'
    else:
        return 2*x + 1
```

a potom takéto volanie vráti:

```
>>> pocitaj(5)
11
>>> pocitaj('a')
'aa1'
```

Neskôr sa naučíme testovať typ nejakých hodnôt správnejším spôsobom, ale zatiaľ nám bude stačiť, keď to budeme riešiť takto jednoducho.

Napriek tomuto niektoré funkcie môžu fungovať rôzne pre rôzne typy, napríklad:

```
def urob(a, b):
    return 2*a + 3*b
```

niekedy funguje pre čísla aj pre reťazce. Otestujte.

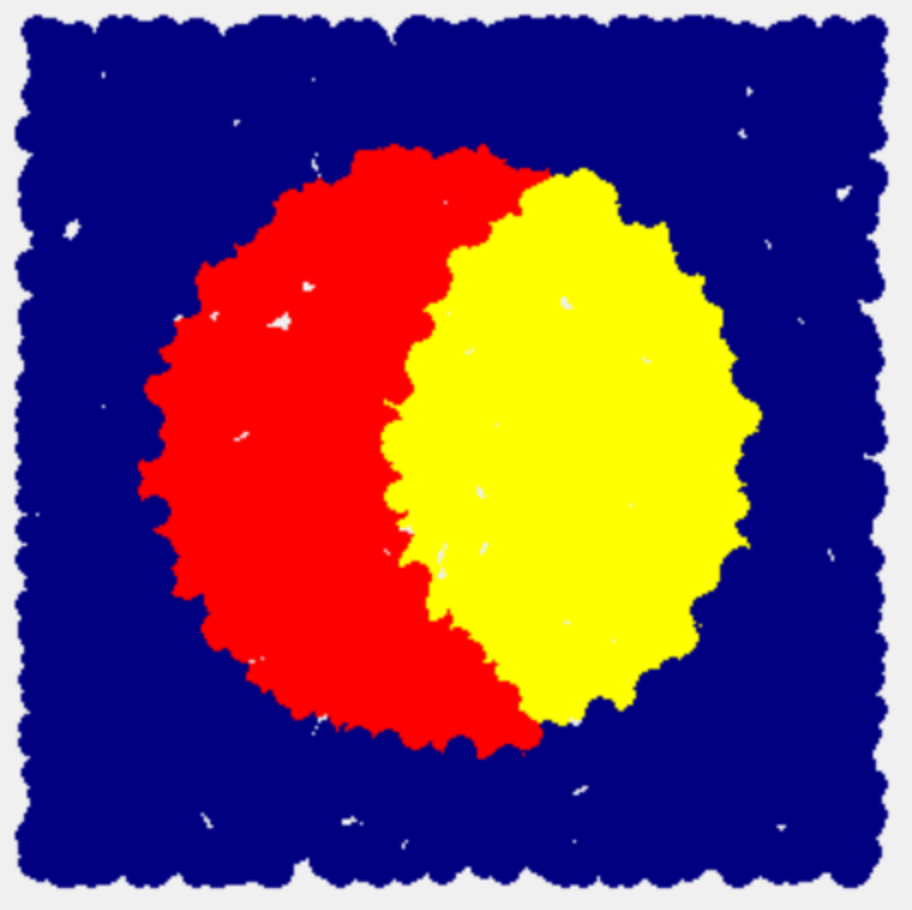
Grafické funkcie

Zadefinujeme funkcie, pomocou ktorých sa nakreslí 5000 náhodných farebných bodiek, ktoré budú zafarbené podľa nejakých pravidiel:

```
import tkinter
import random
import math

def vzd(x1, y1, x2, y2):
    return math.sqrt((x1-x2) ** 2 + (y1-y2) ** 2)

def kresli_bodku(x, y, farba):
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, width=0)

def farebne_bodky(pocet):
    for i in range(pocet):
        x = random.randint(10, 290)
        y = random.randint(10, 290)
        if vzd(x, y, 150, 150) > 100:
            kresli_bodku(x, y, 'navy')
        elif vzd(x, y, 230, 150) > 100:
            kresli_bodku(x, y, 'red')
        else:
            kresli_bodku(x, y, 'yellow')

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

farebne_bodky(5000)
```

```
tkinter.mainloop()
```

Funkcia `vzd()` počíta vzdialenosť dvoch bodov `(x1, y1)` a `(x2, y2)` v rovine - tu sa použil známy vzorec z matematiky. Táto funkcia nič nevypisuje, ale vracia číselnú hodnotu (desatinné číslo). Ďalšia funkcia `kresli_bodku()` nič nevracia, ale vykreslí v grafickej ploche malý kruh s polomerom 5, ktorý je zafarbený zadanou farbou. Tretia funkcia `farebne_bodky()` dostáva ako parameter počet bodiek, ktoré má nakresliť: funkcia na náhodné pozície nakreslí príslušný počet bodiek, pričom tie, ktoré sú od bodu `(150, 150)` vzdialené viac ako 100, budú tmavomodré (farba `'navy'`), tie, ktoré sú od bodu `(230, 150)` vzdialené viac ako 100, budú červené a všetku ostatné budú žlté. Všimnite si, že sme samotný program opäť zapísali až za definíciami všetkých funkcií. Po spustení dostávame približne takýto obrázok:



Náhradná hodnota parametra

Naučíme sa zadefinovať parametre funkcie tak, aby sme pri volaní nemuseli uviesť všetky hodnoty skutočných parametrov, ale niektoré sa automaticky dosadia, tzv. náhradnou hodnotou (default), napríklad:

```
def kresli_bodku(x, y, farba='red', r=5):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba, width=0)
```

V hlavičke funkcie môžeme k niektorým parametrom uviesť náhradnú hodnotu (vyzerá to ako priradenie). V tomto prípade to označuje, že ak tomuto formálnemu parametru nebude zodpovedať skutočný parameter, dosadí sa práve táto náhradná hodnota. Pritom musí platiť, že keď nejakému parametru v definícii funkcie určíte, že má náhradnú hodnotu, tak náhradnú hodnotu musíte zadať aj všetkým ďalším formálnym parametrom, ktoré sa nachádzajú v zozname parametrov za ním (ak sme zadefinovali náhradnú hodnotu pre parameter `farba`, musíme nejakú zadefinovať aj pre parameter `r`).

Teraz môžeme zapísať aj takéto volania tejto funkcie:

```
kresli_bodku(100, 200, 'blue', 3)    # farba bude 'blue' a r bude 3
kresli_bodku(150, 250, 'blue')       # farba bude 'blue' a r bude 5
kresli_bodku(200, 200)               # farba bude 'red'  a r bude 5
```

Parametre volané menom

Funkcia `kresli_bodku` má štyri parametre: `x`, `y`, `farba` a `r`.

Python umožňuje funkcie s parametrami volať tak, že skutočné parametre neurčujeme pozične (prvému skutočnému zodpovedá prvý formálny, druhému druhý, atď.) ale priamo pri volaní uvedieme meno parametra. Takto môžeme určiť hodnotu ľubovoľného parametra. Napríklad všetky tieto volania sú korektné:

```
kresli_bodku(10, 20, r=10)
kresli_bodku(farba='green', x=10, y=20)
kresli_bodku(r=7, farba='yellow', y=20, x=30)
```

Samozrejme aj pri takomto volaní môžeme vynechať len tie parametre, ktoré majú určenú náhradnú hodnotu, všetky ostatné parametre sa musia v nejakom poradí objaviť v zozname skutočných parametrov.

Farebný model RGB

Keďže už vieme vytvárať reťazce so šestnástkovým zápisom čísel (napríklad pomocou `'f{číslo:02x}'`), zapíšeme funkciu `rgb()`, ktorá bude vytvárať farby pomocou RGB-modelu:

```
def rgb(r, g, b):
    return f'#{r:02x}{g:02x}{b:02x}'
```

otestujme:

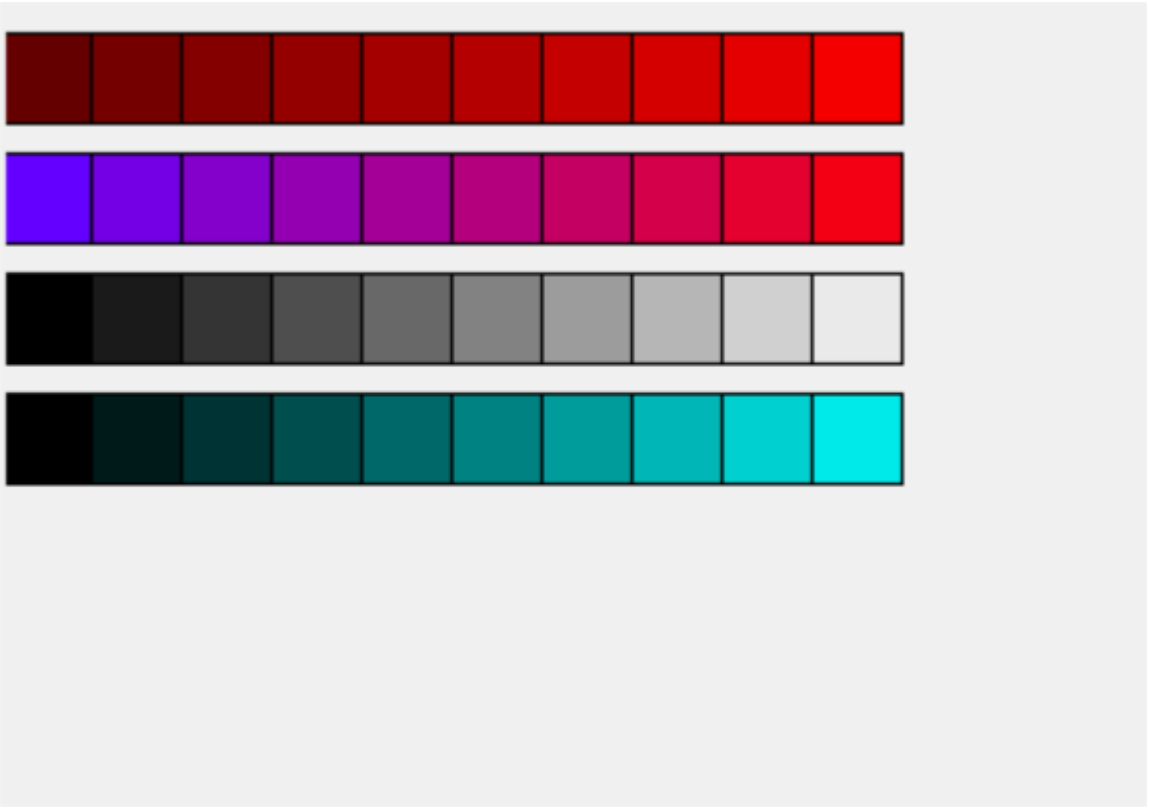
```
>>> rgb(255, 255, 0)
'#ffff00'
>>> rgb(0, 100, 0)
'#006400'
```

Funkciu `rgb()` môžeme využiť, napríklad na kreslenie farebných štvorcov:

```
import tkinter

def rgb(r, g, b):
    return f'#{r:02x}{g:02x}{b:02x}'

def stvorec(strana, x, y, farba=''):
    canvas.create_rectangle(x, y, x + strana, y + strana, fill=farba)

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

for i in range(10):
    stvorec(30, i*30, 10, rgb(100 + 16*i, 0, 0))
    stvorec(30, i*30, 50, rgb(100 + 16*i, 0, 255 - 26*i))
    stvorec(30, i*30, 90, rgb(26*i, 26*i, 26*i))
    stvorec(30, i*30, 130, rgb(0, 26*i, 26*i))
```

```
tkinter.mainloop()
```

Tento program nakreslí takýchto 40 zafarbených štvorcov:



Náhodné farby

Ak potrebujeme generovať náhodnú farbu, ale stačí nám iba jedna z dvoch možností, môžeme to urobiť, napríklad takto:

```
def nahodna2_farba():
    if random.randrange(2):
        return 'blue'
    return 'red'
```

Už ste sa stretli aj s tým, že by malo fungovať aj `random.choice(('blue', 'red'))`.

Podobne by sa zapísala funkcia, ktorá generuje náhodnú farbu jednu z troch a pod.

Ak ale chceme úplne náhodnú farbu z celej množiny všetkých farieb, využijeme RGB-model, napríklad takto:

```
def rgb(r, g, b):
    return f'#{r:02x}{g:02x}{b:02x}'

def nahodna_farba():
    return rgb(random.randrange(256), random.randrange(256), random.randrange(256))
```

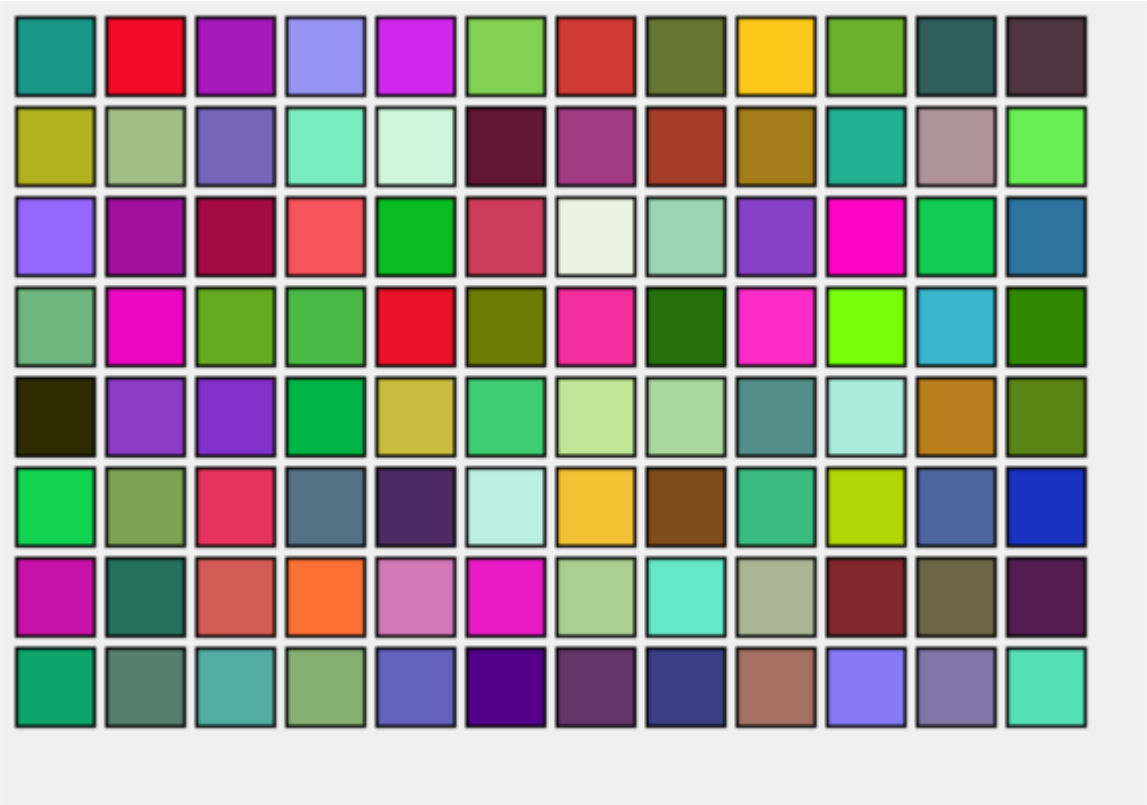
Už vieme, že sa to dá zapísať aj takto:

```
def nahodna_farba():
    return f'#{random.randrange(256**3):06x}'
```

Môžeme vygenerovať štvorcovú sieť náhodných farieb:

```
import tkinter
import random

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

def stvorec(strana, x, y, farba=''):
    canvas.create_rectangle(x, y, x + strana, y + strana, fill=farba)

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

for y in range(5, 235, 30):
    for x in range(5, 355, 30):
        stvorec(26, x, y, nahodna_farba())

tkinter.mainloop()
```

nejaký takýto obrázok:



Niekoľko užitočných matematických funkcií

Na záver ukážeme sériu zaujímavých matematických funkcií. Mnohé z nich sme už programovali predtým, ale teraz ich uvidíme v tvare funkcií. Začneme s `vypis_delitele(cislo)`, ktorá do jedného riadka vypíše všetky delitele daného čísla:

```
def vypis_delitele(cislo):
    for i in range(1, cislo+1):
        if cislo % i == 0:
            print(i, end=' ')
    print()

>>> vypis_delitele(24)
1 2 3 4 6 8 12 24
```

Ďalšia funkcia `sucet_delitelov(cislo)` tieto delitele nevypisuje, ale vráti ich súčet (pomocou `return`):

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

```
>>> sucet_delitelov(24)
60
>>> sucet_delitelov(11)
12
```

Funkcia `je_dokonale(cislo)` pomocou funkcie `sucet_delitelov()` zistí, či je dané číslo **dokonalé**, t.j. že súčet všetkých menších deliteľov ako samotné číslo sa rovná samotnému číslu. Napríklad delitele čísla `6` (menšie ako `6`) sú `1`, `2`, `3`. Ich súčet je `6`. Preto je číslo `6` dokonalé. Funkcia nič nevypisuje, ale vracia (pomocou `return`) `True` alebo `False`.

```
def je_dokonale(cislo)
    return sucet_delitelov(cislo) == 2*cislo
```

```
>>> je_dokonale(6)
True
>>> je_dokonale(24)
False
```

Ďalšia funkcia `vsetky_dokonale(od, do)` vypíše všetky dokonalé čísla v danom intervale `<od, do>`. Táto funkcia využije funkciu `je_dokonale()`:

```
def vsetky_dokonale(od, do):
    for cislo in range(od, do+1):
        if je_dokonale(cislo):
            print(cislo, 'je dokonalé')
```

```
>>> vsetky_dokonale(1, 30)
6 je dokonalé
28 je dokonalé
```

Zapíšeme funkciu `nsd(a, b)`, ktorá počíta najväčší spoločný deliteľ dvoch čísel. Použijeme tzv. Euklidov algoritmus, ktorý už pred vyše 2 tisíc rokmi popísal starogrécky matematik **Euklides**:

```
def nsd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

```
>>> nsd(60, 18)
6
>>> nsd(100000000, 1)
1
```

Iste ste si všimli, že volanie `nsd(100000000, 1)` trvá niekoľko sekúnd. Zrejme preto, lebo while-cyklus `100000000`-krát odpočíta `1`. Pritom si stačí uvedomiť, že takýto cyklus:

```
while a > b:
    a = a - b
```

pre kladné čísla `a` a `b` urobí to isté ako:

```
a = a % b
```

teda zvyšok po delení. Funkciu `nsd` môžeme teraz výrazne vylepšiť:

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

Odkrokujte tento algoritmus pomocou stránky http://www.pythontutor.com/visualize.html#mode=edit.

Ďalšia funkcia `pocet_delitelov(cislo)` pre dané číslo zistí počet všetkých deliteľov. Napríklad delitele čísla `6` sú `1`, `2`, `3`, `6`, preto funkcia vráti `4`. Funkcia nič nevypisuje, ale vracia (pomocou `return`) celé číslo:

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

```
>>> pocet_delitelov(6)
4
>>> pocet_delitelov(17)
2
```

Teraz funkcia `je_prvocislo(cislo)` pomocou funkcie `pocet_delitelov()` veľmi jednoducho zistí (vráti `True` alebo `False`), či je to prvočíslo (je deliteľné len `1` a samým sebou):

```
def je_prvocislo(cislo):
    return pocet_delitelov(cislo) == 2
```

```
>>> je_prvocislo(6)
False
>>> je_prvocislo(17)
True
```

Ďalšia funkcia `vsetky_prvocisla(od, do)` vypíše do jedného riadka všetky prvočísla v danom intervale:

```
def vsetky_prvocisla(od, do):
    for cislo in range(od, do+1):
        if je_prvocislo(cislo):
            print(cislo, end=' ')
    print()
```

```
>>> vsetky_prvocisla(1, 30)
2 3 5 7 11 13 17 19 23 29
```

Aj tento algoritmus odkrokujte pomocou stránky http://www.pythontutor.com/visualize.html#mode=edit.

Programátori veľmi obľubujú štandardné funkcie `min` a `max`, ktoré vrátia minimálny, resp. maximálnu hodnotu z dvoch daných hodnôt. Zapíšme tieto dve funkcie s použitím príkazu `if`:

```
def min(a, b):
    if a < b:
        return a
    return b

def max(a, b):
```

```
    if a > b:
        return a
    return b
```

Vďaka definícii týchto dvoch funkcií, nemôžeme ďalej v tomto programe používať štandardné funkcie `min` a `max`. Hovoríme, že sme ich prekryli novými definíciami. Ak by sme teraz chceli vytvoriť funkciu `min`, ktorá ale zistí minimum troch čísel, nebude fungovať:

```
def min(a, b, c):
    return min(min(a, b), c)
```

lebo sme opäť prekryli našu pôvodnú funkciu `min` novšou verziou. Mohli by sme to zapísať, napríklad takto:

```
def min2(a, b):
    if a < b:
        return a
    return b

def min(a, b, c):
    return min2(min2(a, b), c)
```

Už v druhej prednáške sme počítali faktoriál pomocou for-cyklu. Zapíšme to do funkcie:

```
def faktorial(n):
    vysl = 1
    for i in range(2, n+1):
        vysl *= i
    return vysl
```

Z tréningových dôvodov to môžeme zapísať aj pomocou while-cyklu:

```
def faktorial(n):
    vysl = 1
    while n > 1:
        vysl *= n
        n -= 1
    return vysl
```

Opäť vidíte, že sme tu použili parameter `n` ako lokálnu premennú. Funkciu `faktorial` by sme mohli využiť na výpočet kombinačného čísla:

```
def kombinacne_cislo(n, k):
    return faktorial(n) // (faktorial(n-k) * faktorial(k))
```

Hoci je toto správny zápis, matematici vedia, že sa to dá počítať aj výrazne efektívnejšie. Veď, napríklad pre `kombinacne_cislo(100, 2)` musí táto funkcia počítať dva veľké faktoriály `100!` a `98!` a potom ich medzi sebou deliť, pritom by stačilo vypočítať `100*99//2`. Kombinačné číslo sa dá vypočítať ako súčin `k` čísel od `n` smerom dole vydelené `k` faktoriálom. Zapíšme:

```
def kombinacne_cislo(n, k):
    vysl = 1
    for i in range(n+1-k, n+1):
        vysl *= i
    return vysl // faktorial(k)
```

Ďalším veľmi známym matematickým pojmom je Fibonacciho postupnosť. Táto postupnosť celých čísel začína dvomi číslami `0` a `1` a každé ďalšie sa počíta ako súčet dvoch predchádzajúcich členov postupnosti. Veľmi rýchlo by sme vedeli zapísať niekoľko prvých členov postupnosti:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Napíšme funkciu, ktorá vypočíta `n`-ty člen tejto postupnosti (prvky budeme počítať od nultého, ktorého hodnota je `0`):

```
def fibonacci(n):
    if n < 2:
        return n
    f1, f2 = 0, 1
    for i in range(n-1):
        f1, f2 = f2, f1+f2
    return f2
```

Niekedy môžete vidieť aj takýto zápis:

```
def fibonacci(n):
    a, b = 0, 1
    while n:
        a, b = b, a+b
        n -= 1
    return a
```

Cvičenia

L.I.S.T.

- riešenia **aspoň 12 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- používaj len konštrukcie z doterajších prednášok (žiadne zoznamy)
- pozri si Riešenie úloh 5. cvičenia

1. Napíš funkciu `obdlznik(sirka, znak='*')`, ktorá z daného znaku `znak` vypíše do troch riadkov výstupu obdĺžnik zadanej šírky. Napríklad pre volania:

```
2. obdlznik(30, '#')
3. obdlznik(6)
4. obdlznik(19, 'O')
```

   dostaneme výstup:

```
##############################
#                            #
##############################
******
*    *
******
OOOOOOOOOOOOOOOOOO
O                O
OOOOOOOOOOOOOOOOOO
```

2. Napíš funkciu `riadok(n, text='')`, ktorá vypíše `n` znakový reťazec hviezdičiek `'*'`, stred ktorého nahradí zadaným textom. Ak je tento zadaný `text` neprázdny, vloží na jeho začiatok aj koniec medzeru. Napríklad pre volania:

```
sir = 40
riadok(sir)
riadok(sir, 'Ján Botto')
riadok(sir, 'Žltá ľalija')
riadok(sir, '-')
riadok(sir, 'Stojí stojí mohyla')
riadok(sir, 'Na mohyle zlá chvíľa')
riadok(sir, 'na mohyle tŕnie chrastie')
riadok(sir, 'a v tom tŕní chrastí rastie')
riadok(sir)
```

dostaneme výstup:

```
****************************************
************** Ján Botto ***************
************* Žltá ľalija **************
****************** - ******************
********** Stojí stojí mohyla **********
********* Na mohyle zlá chvíľa *********
******* na mohyle tŕnie chrastie *******
***** a v tom tŕní chrastí rastie ******
****************************************
```

3. Napíš funkciu `priemer(a, b)`, ktorá vypočíta priemer dvoch zadaných čísel. Funkcia nič nevypisuje, ale pomocou `return` vráti vypočítanú hodnotu. Otestuj ju s rôznymi hodnotami parametrov. Napríklad:

```
>>> priemer(1, 4)
2.5
>>> priemer(3.14, 31.4)
17.27
```

4. Na prednáške si sa zoznámil s funkciou `nsd(a, b)`, ktorá počítala najväčší spoločný deliteľ dvoch čísel. Inšpiruj sa touto funkciou a napíš funkciu `nsn(a, b)`, ktorá vypočíta najmenší spoločný násobok dvoch čísel. Napríklad pre volania:

```
a, b = 129, 162
print(f'nsn({a}, {b}) =', nsn(a, b))
a, b = 60, 168
print(f'nsn({a}, {b}) =', nsn(a, b))
```

dostaneme výstup:

```
nsn(129, 162) = 6966
nsn(60, 168) = 840
```

5. Na prednáške si sa zoznámil s funkciou `fibonacci(n)`, ktorá počítala `n`-tý člen **fibonacciho postupnosti**. Napíš funkciu `fib_medzi(od, do)`, ktorá vypíše (pomocou `print`) všetky fibonacciho čísla z daného intervalu `<od, do>`. Táto funkcia by mala obsahovať len jeden while-cyklus (okrem priradení a `if`). Napríklad pre volania:

```
fib_medzi(10, 100)
fib_medzi(1000, 3000)
```

dostaneme výstup:

```
13 21 34 55 89
1597 2584
```

6. Na prednáške si sa zoznámil s funkciou `je_prvocislo(cislo)`, ktorá pomocou funkcie `pocet_delitelov(cislo)` zisťovala, či je dané `cislo` prvočíslo. Oprav túto funkciu `je_prvocislo(cislo)` tak, aby nevyužívala `pocet_delitelov(cislo)`, ale vo while-cykle zisťovala, či neexistuje aspoň jeden deliteľ v intervale `<2, odmocnina>`. Funkcia sa bude postupne snažiť nájsť takého deliteľa daného čísla, ktorého druhá mocnina nie je väčšia ako dané číslo. Napríklad, číslo `25` bude postupne deliť `2`, `3`, `4`, `5` (pre všetky ich druhá mocnina nie je väčšia ako `25`) a na `5` skončí, lebo delí `25`. Číslo `37` sa tiež pokúsi deliť `2`, `3`, `4`, `5`, `6` (žiadne z nich nie je deliteľom) a keďže pre všetky väčšie je ich druhá mocnina väčšia ako `37`, vyhlásime `37` za prvočíslo. Okrem funkcie `je_prvocislo(cislo)` napíš aj funkciu `dvojicky(od, do)`, ktorá v danom intervale `<od, do>` nájde všetky prvočíselné dvojičky (ich rozdiel je 2). Napríklad:

```
>>> dvojicky(3, 50)
3 5
5 7
11 13
17 19
29 31
41 43
>>> dvojicky(1000000, 1000300)
1000037 1000039
1000211 1000213
1000289 1000291
```

7. Napíš funkciu `vyhod_medzery(text)`, ktorá zo zadaného textu vyhodí všetky medzery. Nepoužívajte žiadne funkcie ani operácie s reťazcami, ktoré sme sa ešte neučili. Funkcia nič nevypisuje, ale pomocou `return` vráti nový reťazec. Otestuj ju s rôznymi hodnotami parametrov. Napríklad:

```
>>> vyhod_medzery('  mám   rád Python ')
'mámrádPython'
>>> vyhod_medzery('      ')
''
```

8. Napíš funkciu `hadanie(od, do)`, pomocou ktorej sa budeš vedieť zahrať s počítačom takúto hru: počítač si náhodne pomyslí číslo z intervalu `<od, do>` (neprezradí nám ho) a my sa ho budeme na maximálne 10 pokusov snažiť uhádnuť. Po každom pokuse nám oznámi, či náš typ je menší ako jeho číslo alebo väčší. Priebeh hry by mohol vyzerať, napríklad takto:

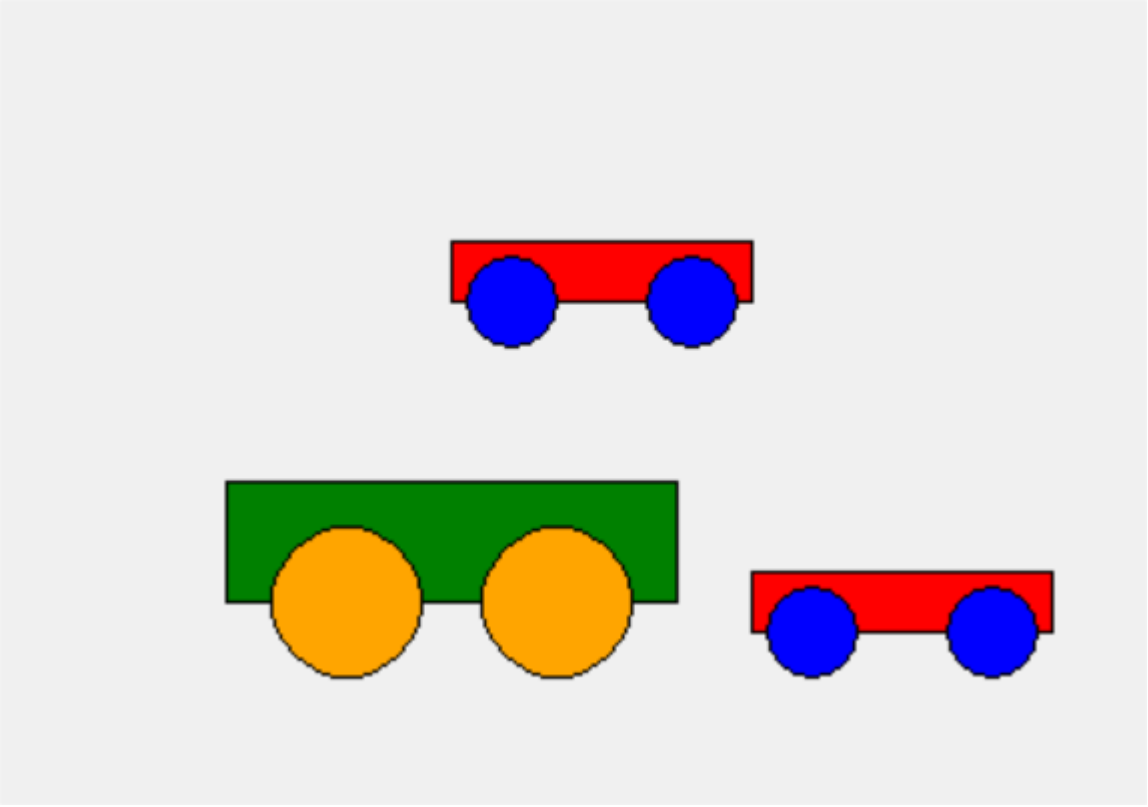
```
>>> hadanie(1, 100)
Myslím si číslo, uhádni ho!
tvoj tip: 50
*** pridaj
tvoj tip: 75
*** pridaj
tvoj tip: 88
*** uber
tvoj tip: 81
*** uber
tvoj tip: 78
*** uber
tvoj tip: 77
Uhádol si na 6. pokus. Gratulujem.

>>> hadanie(1, 100)
```

```
Myslím si číslo, uhádni ho!
tvoj tip: 10
*** pridaj
tvoj tip: 20
*** pridaj
tvoj tip: 30
*** pridaj
tvoj tip: 40
*** pridaj
tvoj tip: 50
*** pridaj
tvoj tip: 60
*** uber
tvoj tip: 59
*** uber
tvoj tip: 58
*** uber
tvoj tip: 57
*** uber
tvoj tip: 56
*** uber
Neuhádol si ani na 10 pokusov.
Myslel som si číslo 54.
```

Ak sa budeš hrať so svojim programom, mal by si vždy uhádnuť aj pre interval `hadanie(1, 500)`

9. Do daného programu dopíš dve chýbajúce funkcie `koleso` a `doska` tak, aby si dostal daný obrázok.

```
import tkinter

def koleso(...):
    ...

def doska(...):
    ...

def vozik(x, y):
    doska(x, y)
    koleso(x-30, y)
    koleso(x+30, y)

def velky_vozik(x, y):
    doska(x, y, 150, 40, 'green')
    koleso(x-35, y, 25, 'orange')
    koleso(x+35, y, 25, 'orange')

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

vozik(200, 100)
velky_vozik(150, 200)
vozik(300, 210)

tkinter.mainloop()
```



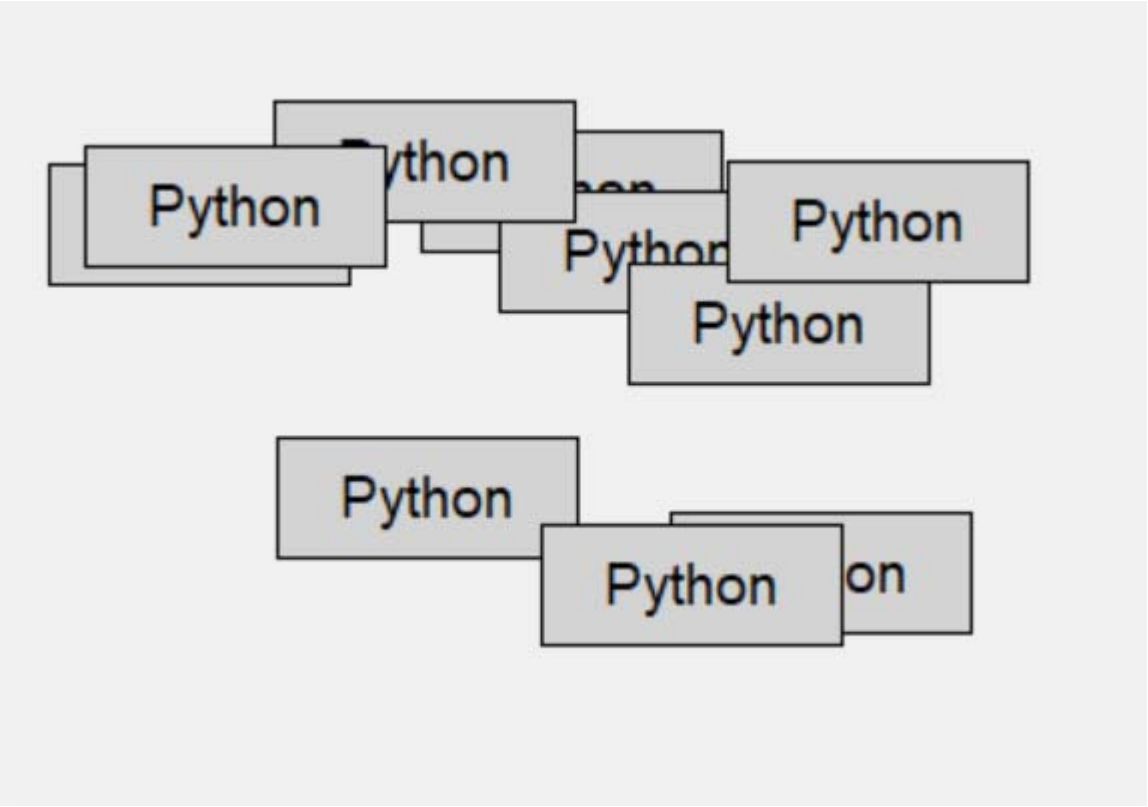
10. Napíš funkciu `kruhy(x, y)`, ktorá nakreslí 10 sústredných náhodne zafarbených kruhov, ich polomery budú `5`, `10`, `15`, … Napríklad pre volanie:

```
for i in range(10):
    kruhy(random.randint(50, 330), random.randint(50, 210))
```

by sa nakreslilo niečo takéto:

11. Napíš funkciu `karticka(x, y, text)`, ktorá nakreslí bledošedý obdĺžnik a do jeho stredu vypíše zadaný text. Stred kartičky má súradnice `(x, y)` a jej strany majú dĺžky `100` a `40`. Font písma môže byť, napríklad `'arial 14'`. Otestuj náhodným vygenerovaním 10 kartičiek, napríklad:

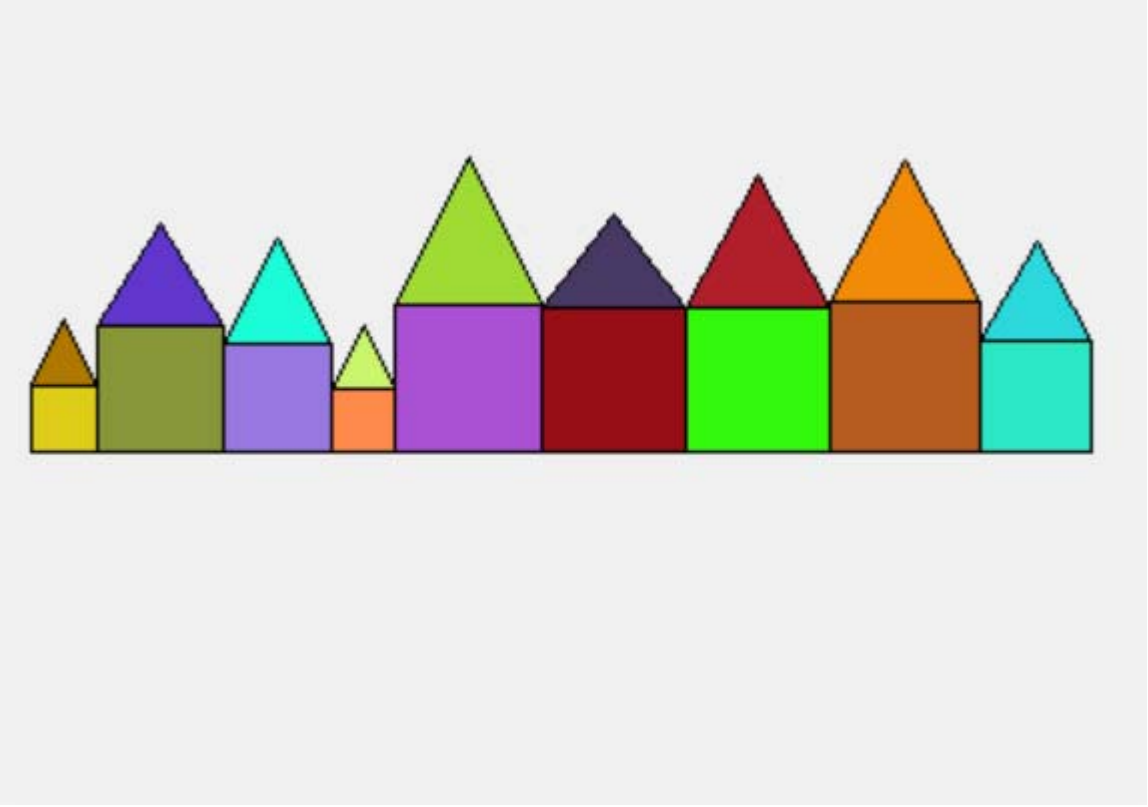
```
for i in range(10):
    karticka(random.randint(50, 300), random.randint(50, 200), 'Python')
```



12. Napíš funkciu `dom(x, y, vel1, vel2)`, ktorá nakreslí domček: štvorec má ľavý dolný roh `(x, y)` a veľkosť strany je `vel1`, trojuholník má výšku `vel2` a základňu `vel1`. Oba sú zafarbené rôznymi náhodnými farbami. Napríklad pre volanie:

```
x, y = 10, 150
while x < 330:
    v = random.randint(20, 50)
    dom(x, y, v, random.randint(v//2, v))
    x += v
```

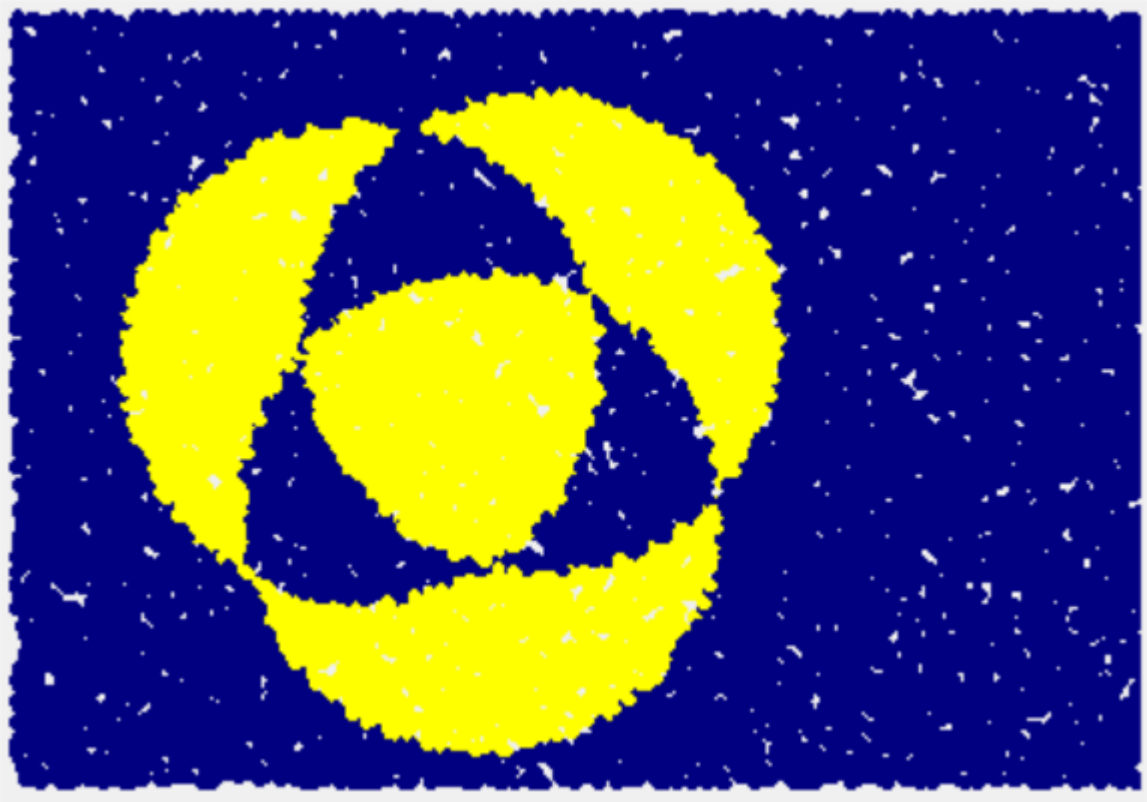
by sa nakreslilo:



13. Na prednáške sa pomocou farebných bodiek kreslil červený mesiac na modrom pozadí. Využívala sa pritom funkcia `vzd`. Napíš funkciu `farebne_bodky(r, x1, y1, x2, y2, x3, y3)`, ktorá na podobnom princípe grafickú plochu vybodkuje podľa týchto pravidiel: ak by sme nakreslili tri kruhy s polomerom `r` ale s rôznymi stredmi `(x1, y1)`, `(x2, y2)`, `(x3, y3)`, tieto by sa mohli čiastočne prekrývať. Bodky budeš farbiť tak, že tie oblasti, v ktorých nie je žiaden kruh alebo sa prekrývajú práve 2 kruhy zafarbíš na modro, ostatné oblasti budú žlté. Napríklad pre volanie:

```
14. farebne_bodky(80, 120, 120, 180, 110, 160, 170)
```

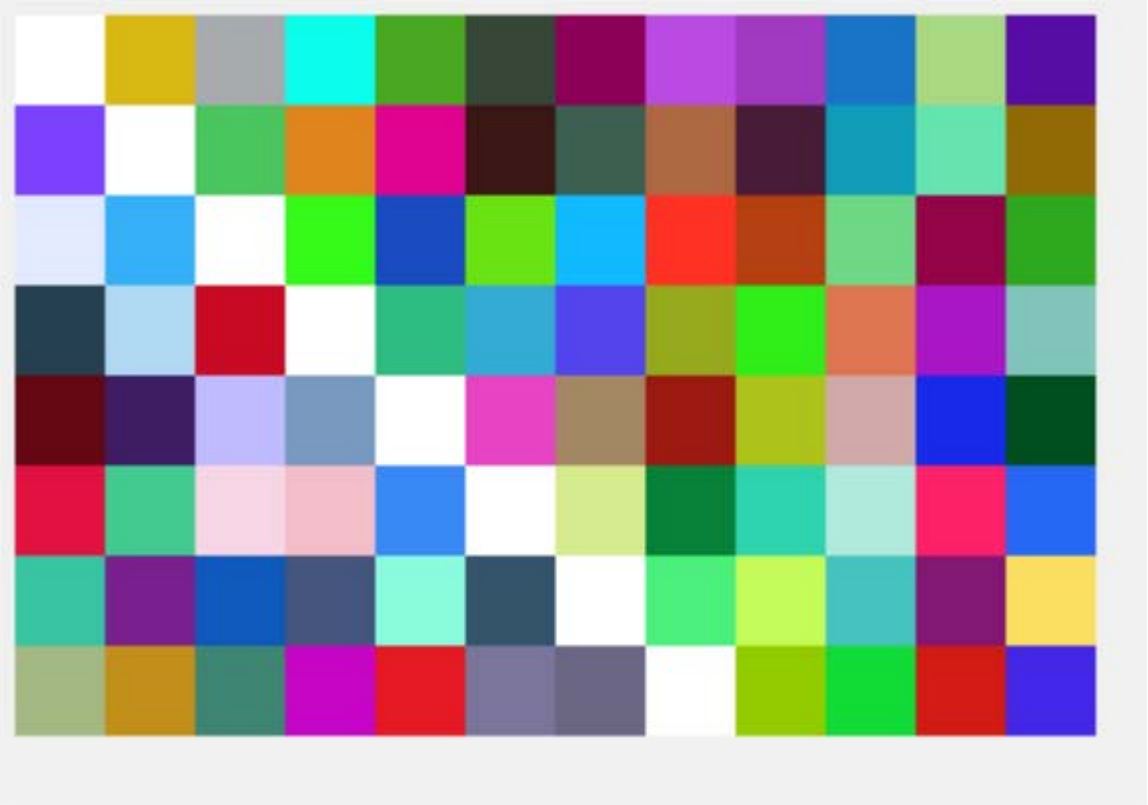
by sa nakreslilo:



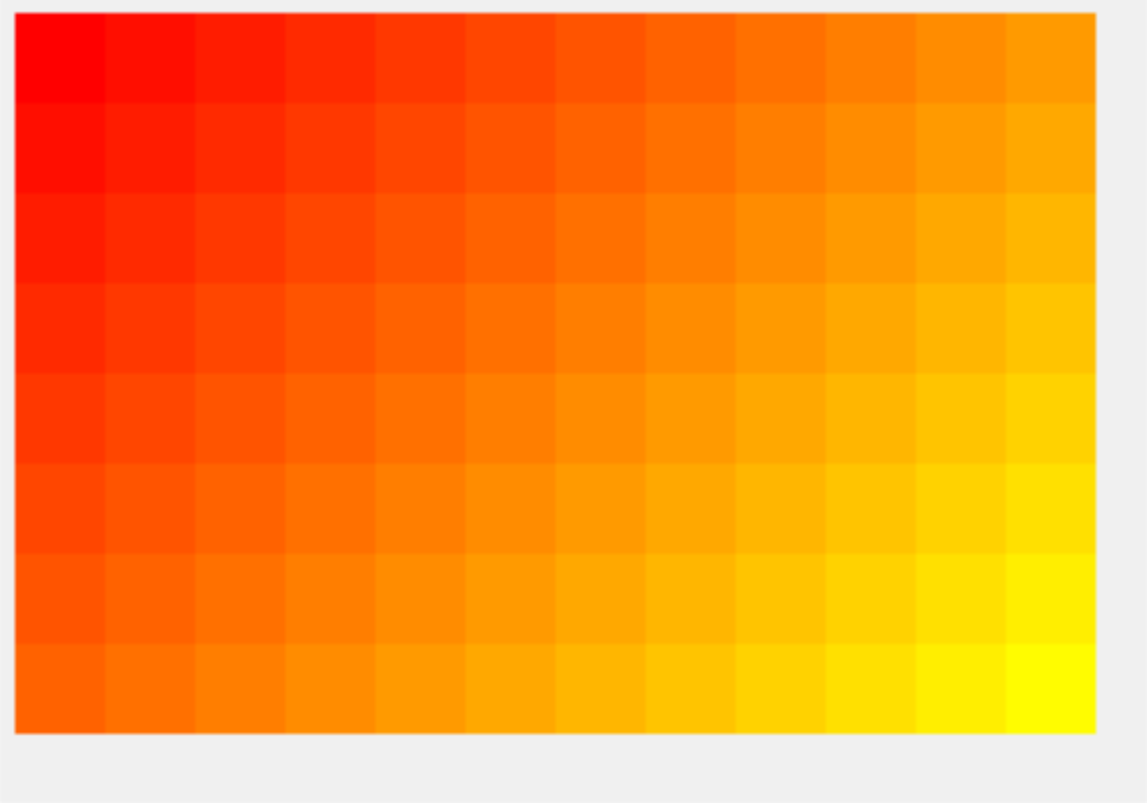
14. Napíš funkciu `stv(riadok, stlpec, farba='white')`, ktorá nakreslí farebný štvorec do myslenej štvorcovej siete, v ktorej je každé políčko veľké 30x30. Ľavý horný roh najľavejšieho horného štvorca má súradnice `(5, 5)`. Napríklad pre volanie:

```
for i in range(8):
    for j in range(12):
        if i == j:
            stv(i, j)
        else:
            stv(i, j, nahodna_farba())
```

by sa nakreslilo:



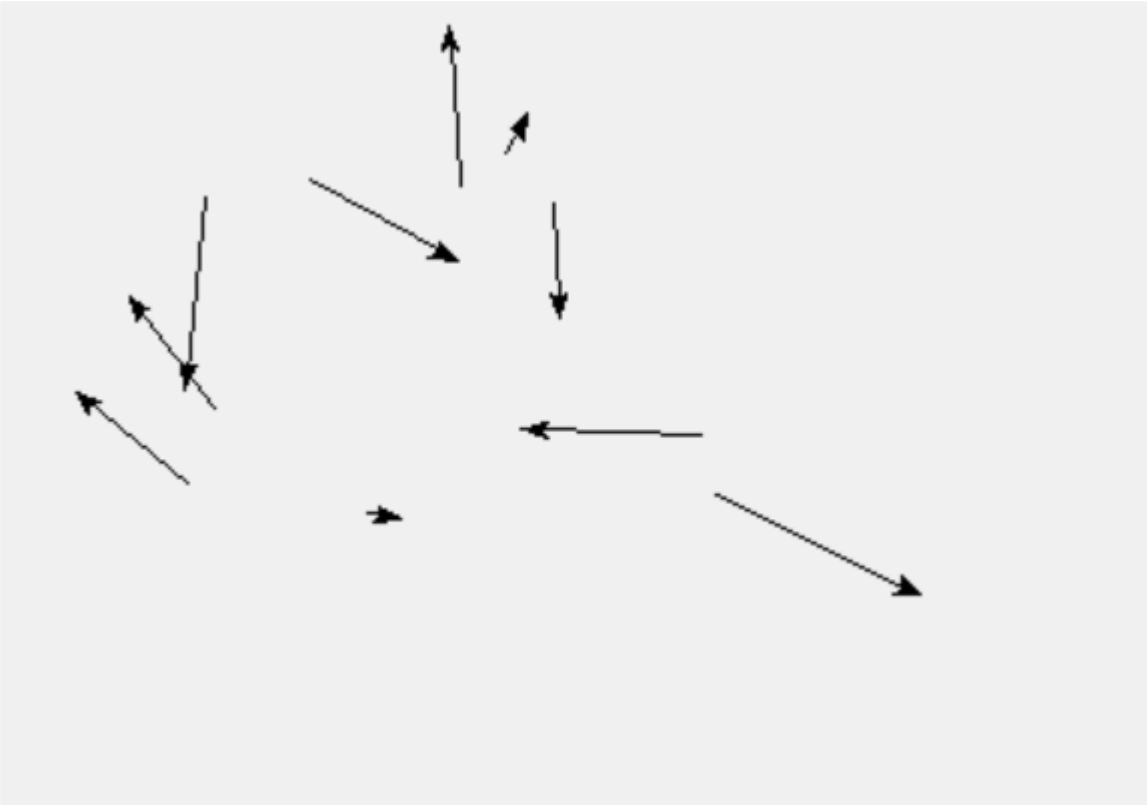
Napíš funkciu `rgb` (z prednášky) a pomocou nej zafarbi štvorce takto (ľavý horný štvorec má farbu `rgb(255, 0, 0)` a pravý dolný skoro `rgb(255, 255, 0)`, farba v ostatných štvorcoch plynulo prechádza - čím je štvorec bližšie k pravému dolnému rohu, tým je bližšie k žltej):



15. Vektor si môžeme predstaviť ako úsečku, ktorá je daná jedným vrcholom `(x, y)`, dĺžkou a uhlom. Uvedom si, že koncové body takéhoto vektora ležia na kružnici s polomerom dĺžka a daným stredom (bodom `(x, y)`). Úsečku nakreslíme tak, aby mala tvar šípky (do `create_line` pridáme pomenovaný parameter `arrow='last'`). Napíš funkciu `vektor(x, y, dlzka, uhol)`. Otestuj, napríklad takto:

```
for i in range(10):
    vektor(random.randint(50, 300), random.randint(50, 200),
           random.randint(10, 80), random.randint(0, 359))
```

a môžeš dostať:



16. Napíš funkciu `slnko(n, x, y)`, ktorá nakreslí slnko ako `n` lúčov (hrubšie žlté, resp. zlaté úsečky, ktoré vychádzajú zo stredu `(x, y)` a majú dĺžku `70`) a veľký žltý/zlatý kruh so stredom `(x, y)` a polomerom `40`. Otestuj, napríklad:

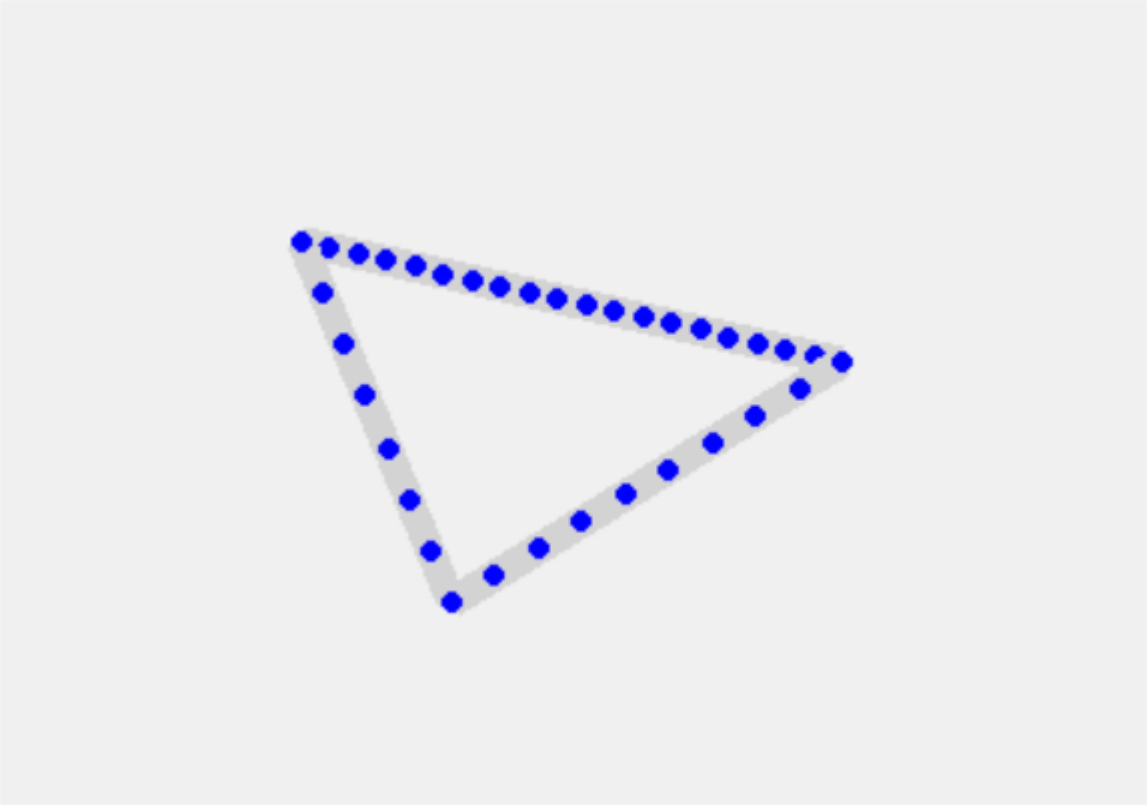
```
slnko(10, 100, 80)
slnko(20, 250, 120)
```

dostaneš:

17. Napíš funkciu `vybodkuj_usecku(x1, y1, x2, y2, n)`, ktorá nakreslí úsečku z bodu `(x1, y1)` do bodu `(x2, y2)`. Túto úsečku nekreslí pomocou `create_line`, ale pomocou `n` bodiek, t.j. malých modrých kruhov s polomerom `3`. Parameter `n` je minimálne `2` a vtedy sa nakreslia len dve bodky v koncových vrcholoch úsečky. Pre kontrolu najprv vykreslíme originálnu úsečku šedou farbou. Otestuj, napríklad:

```
canvas.create_line(100, 80, 280, 120, fill='lightgray', width=11)
vybodkuj_usecku(100, 80, 280, 120, 20)
canvas.create_line(280, 120, 150, 200, fill='lightgray', width=11)
vybodkuj_usecku(280, 120, 150, 200, 10)
canvas.create_line(150, 200, 100, 80, fill='lightgray', width=11)
vybodkuj_usecku(150, 200, 100, 80, 8)
```

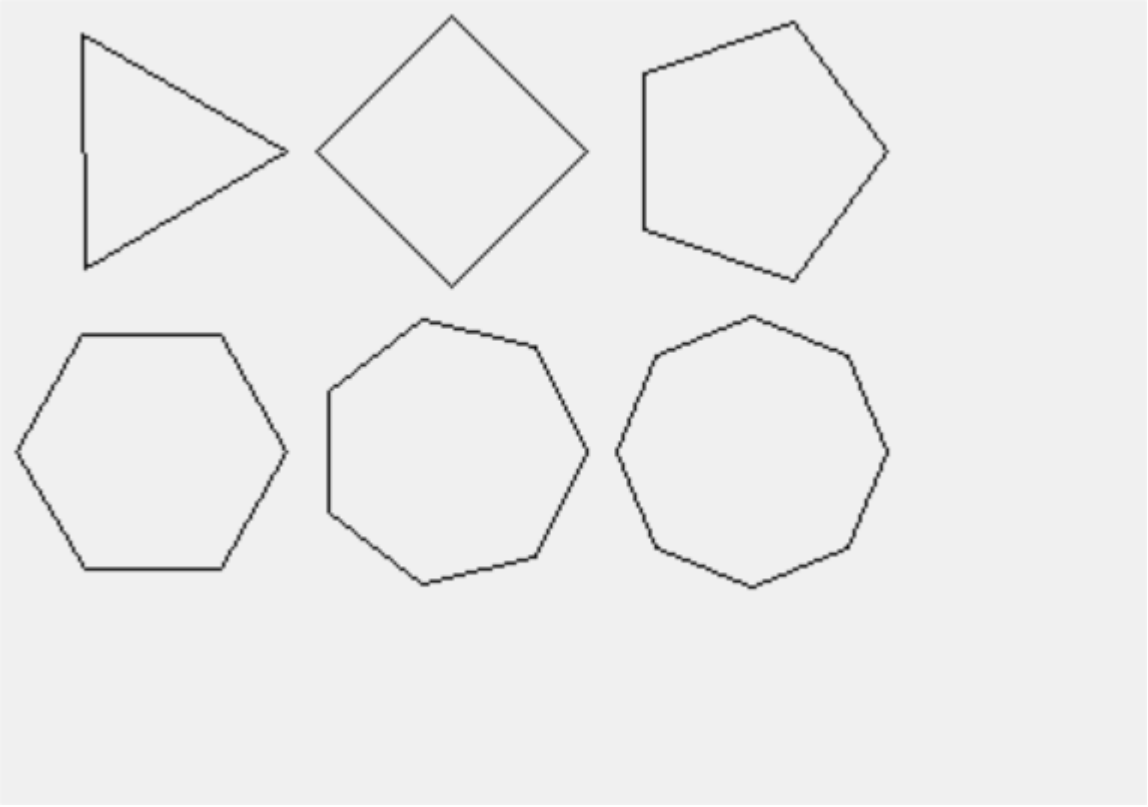
dostaneš:



18. Napíš funkciu `n_uholnik(n, x0, y0, r)`, ktorá nakreslí pravidelný `n`-uholník. Tento n-uholník bude vpísaný v myslenej kružnici so stredom `(x0, y0)` a s polomerom `r`. Napríklad pre volanie:

```
n_uholnik(3, 50, 50, 45)
n_uholnik(4, 150, 50, 45)
n_uholnik(5, 250, 50, 45)

n_uholnik(6, 50, 150, 45)
n_uholnik(7, 150, 150, 45)
n_uholnik(8, 250, 150, 45)
```

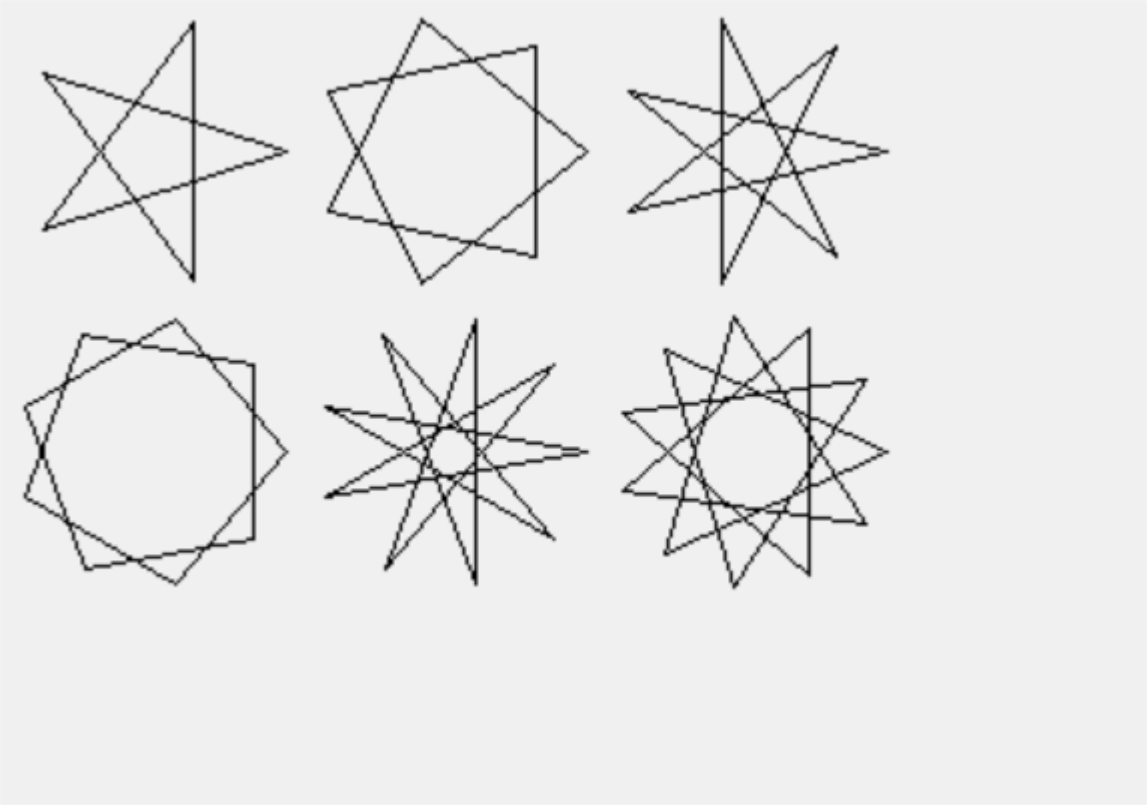
by sa malo nakresliť:



19. Napíš funkciu `n_hviezda(n, x0, y0, r, k=2)`, ktorá pracuje na rovnakom princípe ako `n_uholnik(n, x0, y0, r)` z predchádzajúcej úlohy. V tomto prípade sa ale nespájajú úsečkami najbližšie vrcholy `n`-uholníka, ale parameter `k` určuje, o koľko vrcholov sa presunieme pre každú úsečku. Napríklad `n_hviezda(5, 50, 50, 45, 2)` označuje, že sa budú spájať vrcholy `5`-uholníka takto: 0. vrchol s 2., potom 2. vrchol s 4., potom 4. s 1., 1. vrchol so 3. a na koniec (piata úsečka) 3. vrchol s 0. Zrejme pre `k=1` by sa kreslili pôvodné n-uholníky. Napríklad pre volanie:

```
n_hviezda(5, 50, 50, 45)
n_hviezda(7, 150, 50, 45)
n_hviezda(7, 250, 50, 45, 3)

n_hviezda(9, 50, 150, 45)
n_hviezda(9, 150, 150, 45, 4)
n_hviezda(11, 250, 150, 45, 4)
```

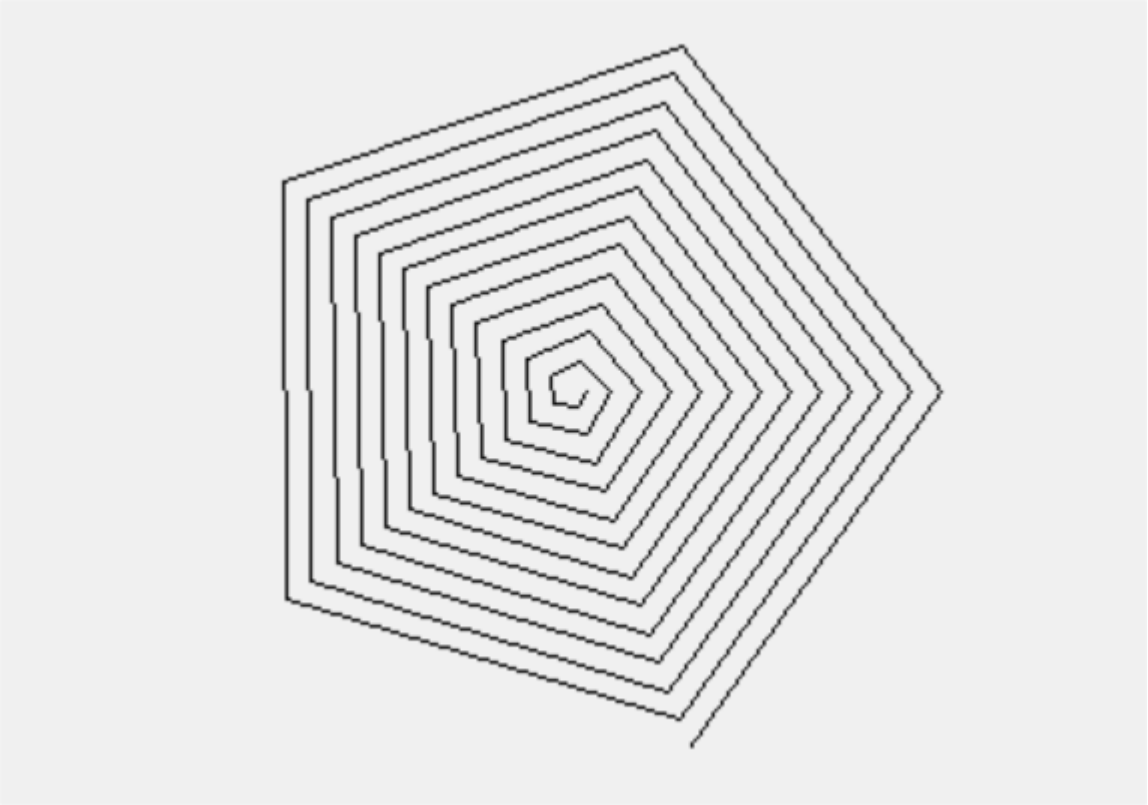
dostaneme:



20. Aj nasledovná funkcia `n_spirala(n, x0, y0, r)` vychádza z riešenia funkcie `n_uholnik(n, x0, y0, r)`. V tomto prípade sa nebude kresliť `n`-uholník, ale špirála. Každá úsečka tu bude spájať dva vrcholy lenže na stále sa zväčšujúcej kružnici so stredom `(x0, y0)`. Začína sa na kružnici s polomerom `5`. Prvý vrchol sa spojí s nasledujúcim ale na kružnici s polomerom o `2` väčším. Takto sa pokračuje, až kým by nebol polomer väčší ako `r`. Napríklad pre volanie:

```
21. n_spirala(5, 190, 130, 125)
```

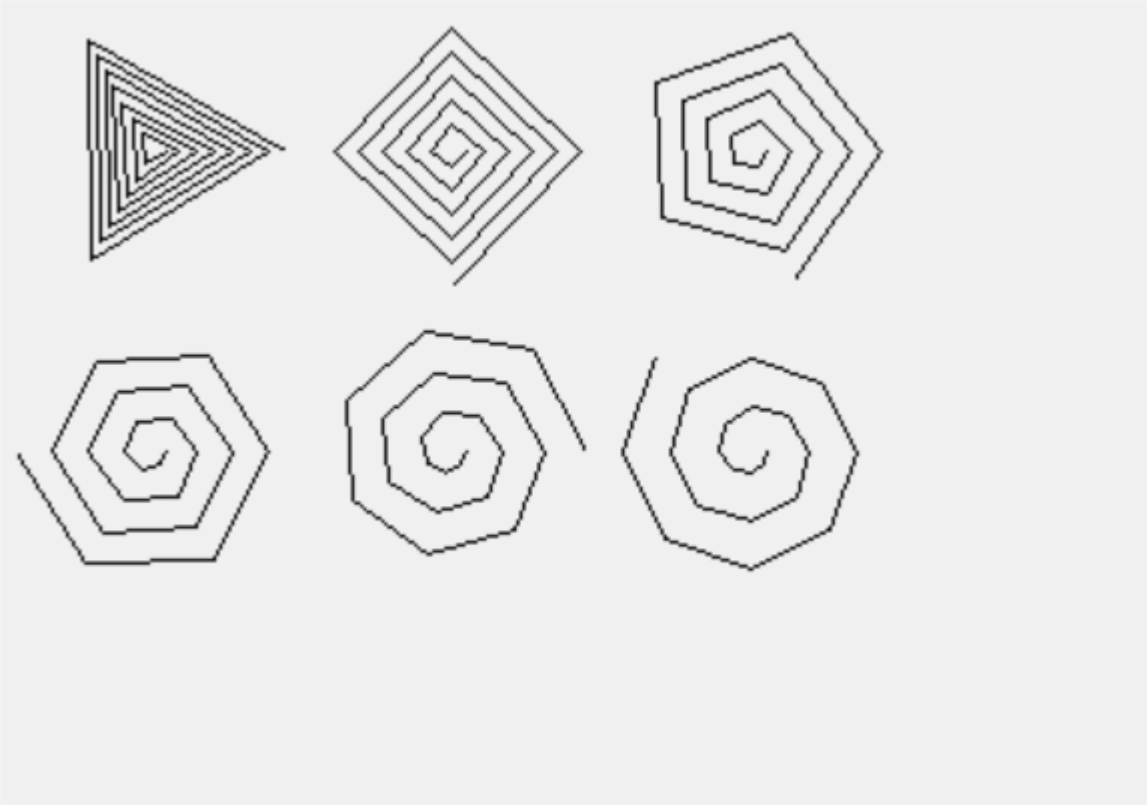
dostaneš:



A pre volania:

```
n_spirala(3, 50, 50, 45)
n_spirala(4, 150, 50, 45)
n_spirala(5, 250, 50, 45)

n_spirala(6, 50, 150, 45)
n_spirala(7, 150, 150, 45)
n_spirala(8, 250, 150, 45)
```

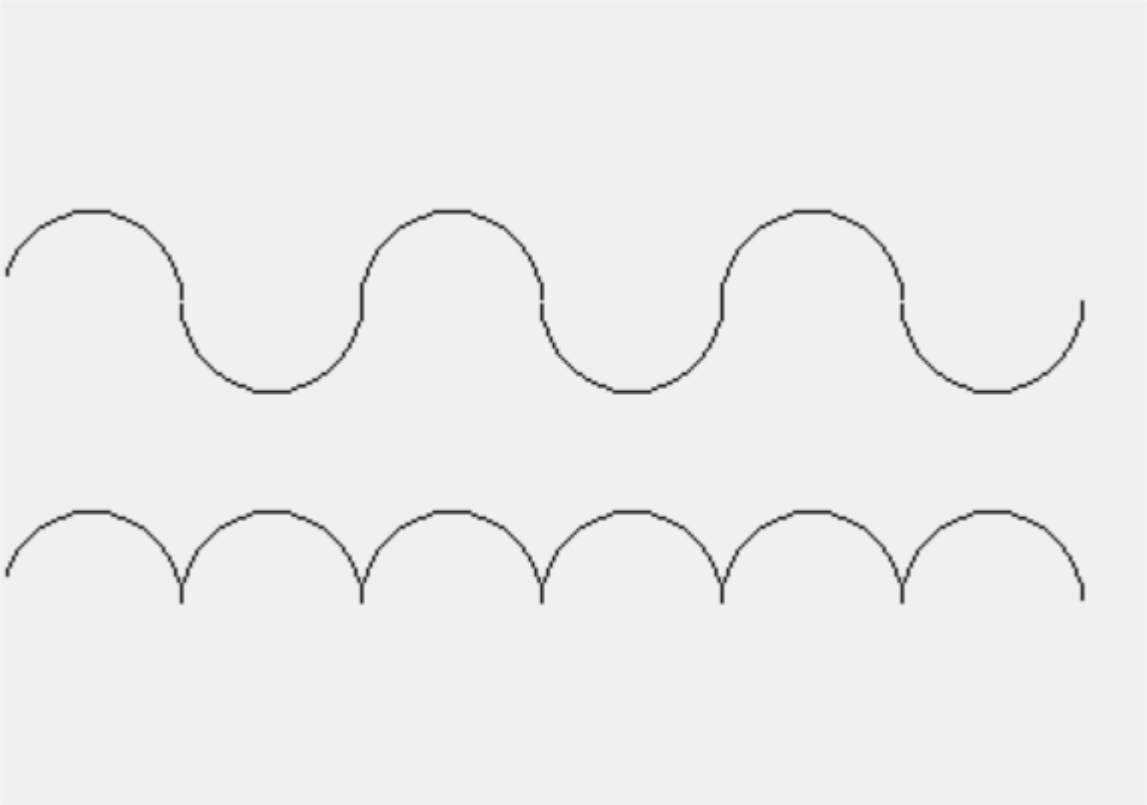
dostaneš:



21. Napíš dve funkcie `horna(x0, y0, r)` a `dolna(x0, y0, r)`, ktoré nakreslia len polovicu kružnice so stredom `(x0, y0)` a s polomerom `r`. Funkcia `horna` by mala nakresliť len hornú polovicu a `dolna` len dolnú. Kružnicu kresli ako 36-uholník, a teda polovica označuje 18 úsečiek. Napríklad volania:

```
horna(30, 100, 30)
dolna(90, 100, 30)
horna(150, 100, 30)
dolna(210, 100, 30)
horna(270, 100, 30)
dolna(330, 100, 30)

for i in range(6):
    horna(30+60*i, 200, 30)
```

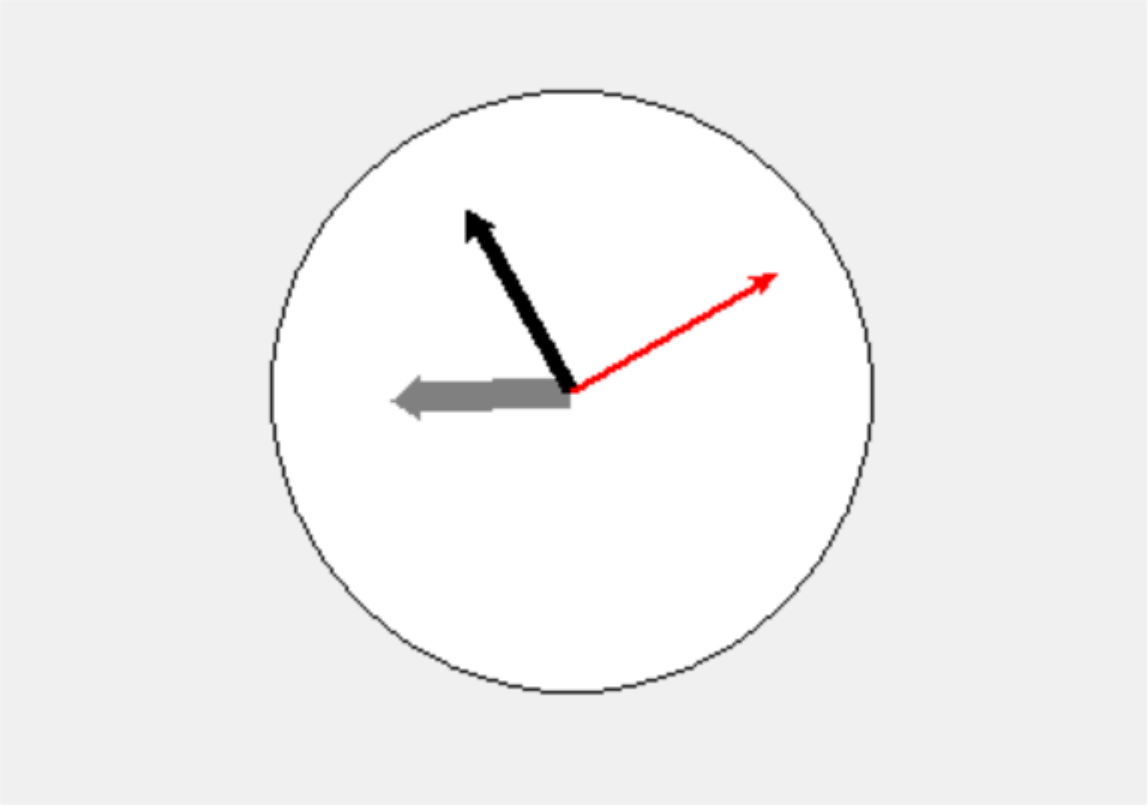
by nakreslili:



22. Napíš funkcie `rucicka(uhol, dlzka, hrubka, farba)` a hodinky(hod, min, sek), pomocou ktorých nakreslíme ručičkové hodinky. Funkcia `rucicka` nakreslí len jednu ručičku ako úsečku z bodu `(190, 130)` pod daným uhlom, danej farby a hrúbky (podobne ako sa kreslil vektor v 15. úlohe). Funkcia `hodinky` nakreslí ciferník (stačí kruh s polomerom 100) a tri ručičky pre hodiny (dĺžka `60`, hrúbka `10`, farba `'gray'`), pre minúty (dĺžka `70`, hrúbka `6`, farba `'black'`), pre sekundy (dĺžka `80`, hrúbka `2`, farba `'red'`). Napríklad volanie:

```
23. hodinky(8, 55, 10)
```

by nakreslilo:



Ak by si hodinky `zavolal` takto:

```
import time

while True:
    canvas.delete('all')
    h, m, s = time.localtime()[3:6]
    hodinky(h, m, s)
    canvas.update()
    canvas.after(1000)
```

ukazovali by aktuálny čas.