14. Triedy a objekty

video prezentácia

triedy a objekty

Čo už vieme:

- poznáme základné typy: `int`, `float`, `bool`, `str`, `list`, `tuple`
- niektoré ďalšie typy sme získali z iných modulov: `tkinter.Canvas`, `turtle.Turtle`
- premenné v Pythone sú vždy referencie na príslušné hodnoty
- pre rôzne typy máme v Pythone definované:
  - **operácie**: `7 * 8 + 9`, `'a' * 8 + 'b'`, `7 * [8] + [9]`
  - **funkcie**: `len('abc')`, `sum(zoznam)`, `min(ntica)`
  - **metódy**: `'11 7 234'.split()`, `zoznam.append('novy')`, `canvas.create_line(1,2,3,4)`, `t.fd(100)`
- funkcia `type(hodnota)` vráti **typ** hodnoty

Vlastný typ

V Pythone sú všetky typy objektové, t.j. popisujú objekty, a takýmto typom hovoríme **trieda** (po anglicky **class**). Všetky hodnoty (teda aj premenné) sú nejakého objektového typu, teda typu trieda, hovoríme im **inštancia triedy** (namiesto *hodnota alebo premenná typu trieda*).

Zadefinujme vlastný typ, teda novú triedu:

```
class Student:
    pass
```

Trochu sa to podobá definícii funkcie bez parametrov s prázdnym telom, napríklad:

```
def funkcia():
    pass
```

Pomocou konštrukcie `class Student:` sme vytvorili prázdnu triedu, t.j. nový typ `Student`, ktorý zatiaľ nič nevie. Keďže je to typ, môžeme vytvoriť premennú tohto typu (teda skôr hodnotu typu `Student`, na ktorú do premennej priradíme referenciu):

```
>>> fero = Student()       # inštancia triedy
>>> type(fero)
    <class '__main__.Student'>
>>> fero
    <__main__.Student object at 0x022C4FF0>
```

Objektovú premennú, teda **inštanciu triedy** vytvárame zápisom `MenoTriedy()` (neskôr budú v zátvorkách nejaké parametre). V našom prípade premenná `fero` obsahuje referenciu na objekt

nášho nového typu `Student`. Podobne to funguje aj s typmi, ktoré už poznáme, ale zatiaľ sme to takto často nerobili:

```
>>> i = int()
>>> type(i)
    <class 'int'>
>>> zoznam = list()
>>> type(zoznam)
    <class 'list'>
```

Všimnite si, že inštanciu sme tu vytvorili volaním `meno_typu()`. Všetky doterajšie štandardné typy majú svoj identifikátor zapísaný len malými písmenami: `int`, `float`, `bool`, `str`, `list`, `tuple`. Medzi pythonistami je ale dohoda, že nové typy, ktoré budeme v našich programoch definovať, budeme zapisovať s prvým písmenom veľkým. Preto sme zapísali, napríklad typ `Student`.

Spomeňte si, ako sme definovali korytnačku:

```
>>> import turtle
>>> t = turtle.Turtle()
```

Premenná `t` je referenciou na objekt triedy `Turtle`, ktorej definícia sa nachádza v module `turtle` (preto sme museli najprv urobiť `import turtle`, aby sme dostali prístup k obsahu tohto modulu). Už vieme, že `t` je inštanciou triedy `Turtle`.

Atribúty

O objektoch hovoríme, že sú to **kontajnery na dáta**. V našom prípade premenná `fero` je referenciou na prázdny kontajner. Pomocou priradenia môžeme objektu vytvárať nové súkromné premenné, tzv. **atribúty**. Takéto súkromné premenné nejakého objektu sa správajú presne rovnako ako bežné premenné, ktoré sme používali doteraz, len sa nenachádzajú v hlavnej pamäti (v globálnom mennom priestore) ale v „pamäti objektu“. Atribút vytvoríme tak, že za meno objektu `fero` zapíšeme meno tejto súkromnej premennej, pričom medzi nimi musíme zapísať bodku. Ak takýto atribút ešte neexistoval, vytvoríme ho priradením:

```
>>> fero.meno = 'Frantisek'
```

Týmto zápisom sme vytvorili novú premennú (atribút objektu) a priradili sme jej hodnotu reťazec `'Frantisek'`. Ak ale chceme zistiť, čo sa zmenilo v objekte `fero`, nestačí zapísať:

```
>>> print(fero)
    <__main__.Student object at 0x022C4FF0>
```

Totiž `fero` je stále referenciou na objekt typu `Student` a Python zatiaľ netuší, čo znamená, že takýto objekt chceme nejako slušne vypísať. Musíme zadať:

```
>>> fero.meno
    'Frantisek'
```

Pridajme do objektu `fero` ďalší atribút:

```
>>> fero.priezvisko = 'Fyzik'
```

Tento objekt teraz obsahuje dve súkromné premenné `meno` a `priezvisko`. Aby sme ich vedeli slušne vypísať, môžeme vytvoriť pomocnú funkciu `vypis`:

```
def vypis(st):
    print('volam sa', st.meno, st.priezvisko)
```

Funkcia má jeden parameter `st` a ona z tohto objektu (všetko v Pythone sú objekty) vyberie dve súkromné premenné (atribúty `meno` a `priezvisko`) a tieto vypíše:

```
>>> vypis(fero)
    volam sa Frantisek Fyzik
```

Do tejto funkcie by sme mohli poslať ako parameter hodnotu ľubovoľného typu nielen `Student`: táto hodnota ale musí byť objektom s atribútmi `meno` a `priezvisko`, inak dostávame takúto chybu:

```
>>> i = 123
>>> vypis(i)
   ...
   AttributeError: 'int' object has no attribute 'meno'
```

Teda chyba oznamuje, že celé čísla nemajú atribút `meno`. Vytvorme ďalšiu inštanciu triedy `Student`:

```
>>> zuzka = Student()
>>> type(zuzka)
    <class '__main__.Student'>
```

Aj `zuzka` je objekt typu `Student` - je to zatiaľ prázdny kontajner atribútov. Ak zavoláme:

```
>>> vypis(zuka)
    ...
    AttributeError: 'Student' object has no attribute 'meno'
```

dostali sme rovnakú správu, ako keď sme tam poslali celé číslo. Ak chceme, aby to fungovalo aj s týmto novým objektom, musíme tieto dve súkromné premenné vytvoriť, napríklad:

```
>>> zuzka.meno = 'Zuzana'
>>> zuzka.priezvisko = 'Matikova'
>>> vypis(zuzka)
    volam sa Zuzana Matikova
```

Objekty sú meniteľné (mutable)

Atribúty objektu sú súkromné premenné, ktoré sa správajú presne rovnako ako „obyčajné“ premenné. Premenným môžeme meniť obsah, napríklad:

```
>>> fero.meno = 'Ferdinand'
>>> vypis(fero)
```

```
    volam sa Ferdinand Fyzik
```

Premenná `fero` stále obsahuje referenciu na rovnaký objekt (kontajner), len sa trochu zmenil jeden z atribútov. Takejto vlastnosti objektov sme doteraz hovorili **meniteľné** (**mutable**):

- napríklad zoznamy sú **mutable**, lebo niektoré operácie zmenia obsah zoznamu ale nie referenciu na objekt (`zoznam.append('abc')` pridá do zoznamu nový prvok)
- ak dve premenné referencujú ten istý objekt (napríklad priradili sme `zoznam2 = zoznam`), tak takáto **mutable** zmena jedného z nich zmení obe premenné
- väčšina doterajších typov `int`, `float`, `bool`, `str` a `tuple` sú **immutable** teda nemenné, s nimi tento problém nenastáva
- nami definované nové typy (triedy) sú vo všeobecnosti **mutable** - ak by sme chceli vytvoriť novú **immutable** triedu, treba ju definovať veľmi špeciálnym spôsobom a tiež s ňou potom treba pracovať veľmi opatrne

Ukážme si to na príklade:

```
>>> mato = fero
>>> vypis(mato)
    volam sa Ferdinand Fyzik
```

Objekt `mato` nie je novým objektom ale referenciou na ten istý objekt ako `fero`. Zmenou niektorého atribútu sa zmení obsah oboch premenných:

```
>>> mato.meno = 'Martin'
>>> vypis(mato)
    volam sa Martin Fyzik
>>> vypis(fero)
    volam sa Martin Fyzik
```

Preto si treba dávať naozaj veľký pozor na priradenie **mutable** objektov.

Funkcie

Už sme definovali funkciu `vypis()`, ktorá vypisovala dva konkrétne atribúty parametra (objektu). Táto funkcia nemodifikovala žiaden atribút, ani žiadnu doteraz existujúcu premennú. Zapíšme funkciu `urob()`, ktorá dostane dva znakové reťazce a vytvorí z nich nový objekt typu `Student`, pričom tieto dva reťazce budú obsahom dvoch atribútov `meno` a `priezvisko`:

```
def urob(m, p):
    novy = Student()
    novy.meno = m
    novy.priezvisko = p
    return novy
```

Pomocou tejto funkcie vieme definovať nové objekty typu `Student`, ktoré už budú mať vytvorené oba atribúty `meno` a `priezvisko`, napríklad:

```
>>> fero = urob('Ferdinand', 'Fyzik')
>>> zuzka = urob('Zuzana', 'Matikova')
>>> mato = urob('Martin', 'Fyzik')
```

```
>>> vypis(fero)
    volam sa Ferdinand Fyzik
>>> vypis(zuzka)
    volam sa Zuzana Matikova
>>> vypis(mato)
    volam sa Martin Fyzik
```

Funkcia `urob()` nemodifikuje žiaden svoj parameter ani žiadne globálne premenné, len vytvára novú inštanciu (a modifikuje atribúty svojej lokálnej premennej) a tú potom vracia ako výsledok funkcie. Funkcie, ktoré majú túto vlastnosť (nič nemodifikujú, len vytvárajú niečo nové) nazývame **pravé funkcie** (po anglicky **pure function**). Pravou funkciou bude aj funkcia `kopia`, ktorá na základe jedného objektu vyrobí nový, ktorý je jeho kópiou. Predpokladáme, že robíme kópiu inštancie `Student`, ktorá má atribúty `meno` a `priezvisko`:

```
def kopia(iny):
    novy = Student()
    novy.meno = iny.meno
    novy.priezvisko = iny.priezvisko
    return novy
```

Ak má `zuzka` sestru Evu, môžeme ju vytvoriť takto:

```
>>> evka = kopia(zuzka)
>>> evka.meno = 'Eva'
>>> vypis(evka)
    volam sa Eva Matikova
>>> vypis(zuzka)
    volam sa Zuzana Matikova
```

Obe inštancie sú teraz dva rôzne kontajnery, teda obe majú svoje vlastné súkromné premenné `meno` a `priezvisko`.

Okrem pravých funkcií existujú tzv. **modifikátory** (po anglicky **modifier**). Je to funkcia, ktorá niečo zmení, najčastejšie atribút nejakého objektu. Funkcia `nastav_hoby()` nastaví danému objektu atribút `hoby` a vypíše o tom text:

```
def nastav_hoby(st, text):
    st.hoby = text
    print(st.meno, st. priezvisko, 'ma hoby', st.hoby)

>>> nastav_hoby(fero, 'gitara')
    Ferdinand Fyzik ma hoby gitara
>>> nastav_hoby(evka, 'cyklistika')
    Eva Matikova ma hoby cyklistika
```

Oba objekty `fero` aj `evka` majú teraz už 3 atribúty, pričom `mato` a `zuzka` majú len po dvoch.

Keďže vlastnosť funkcie **modifikátor** je pre všetky **mutable** objekty veľmi dôležitá, pri písaní nových funkcií si vždy musíme uvedomiť, či je to modifikátor alebo pravá funkcia a často túto informáciu zapisujeme aj do dokumentácie.

Všimnite si, že

```
def zmen(st):
    meno = st.meno
```

```
    meno = meno[::-1]
    print(meno)
```

nie je modifikátor, lebo hoci funkcia mení obsah premennej `meno`, táto je len lokálnou premennou funkcie `zmen` a nemá žiaden vplyv ani na parameter `st` ani na žiadnu inú premennú.

Metódy

Všetky doteraz vytvárané funkcie dostávali ako jeden z parametrov objekt typu `Student` (inštanciu triedy) alebo takýto objekt vracali ako výsledok funkcie. Lenže v objektovom programovaní platí:

- **objekt** je kontajner údajov, ktoré sú vlastne súkromnými premennými objektu (**atribúty**)
- **trieda** je kontajner funkcií, ktoré vedia pracovať s objektmi (aj týmto funkciám niekedy hovoríme **atribúty**, ale častejšie ich voláme **metódy**)

Takže funkcie nemusíme vytvárať tak ako doteraz globálne v hlavnom mennom priestore (tzv. `'__main__'`), ale priamo ich môžeme definovať v triede. Pripomeňme si, ako vyzerá definícia triedy:

```
class Student:
    pass
```

Príkaz `pass` sme tu uviedli preto, lebo sme chceli vytvoriť prázdne telo triedy (podobne ako pre `def` ale aj `while` a `if`). Namiesto `pass` ale môžeme zadefinovať funkcie, ktoré sa stanú súkromné pre túto triedu. Takýmto funkciám hovoríme **metóda**. Platí tu ale jedno veľmi dôležité pravidlo: prvý parameter metódy musí byť premenná, v ktorej metóda dostane inštanciu tejto triedy a s ňou sa bude ďalej pracovať. Zapíšme funkcie `vypis()` a `nastav_hoby()` ako **metódy** (t.j. funkcie definované vo vnútri triedy, teda sú to atribúty triedy):

```
class Student:

    def vypis(self):
        print('volam sa', self.meno, self. priezvisko)

    def nastav_hoby(self, text):
        self.hoby = text
        print(self.meno, self. priezvisko, 'ma hoby', self.hoby)
```

Čo sa zmenilo:

- obe funkcie sú **vnorené** do definície triedy a preto sú odsunuté vpravo
- obom funkciám sme zmenili prvý parameter `st` na `self` - toto sme robiť nemuseli, ale je to **dohoda** medzi pythonistami, že prvý parameter metódy sa bude vždy volať **self** bez ohľadu pre akú triedu túto metódu definujeme (obe funkcie by fungovali korektne aj bez premenovania tohto parametra)

Keďže `vypis()` už teraz nie je globálna funkcia ale metóda, nemôžeme ju volať tak ako doteraz `vypis(fero)`, ale k menu uvedieme aj meno kontajnera (meno triedy), kde sa táto funkcia nachádza, teda `Student.vypis(fero)`:

```
>>> fero = urob('Ferdinand', 'Fyzik')
>>> zuzka = urob('Zuzana', 'Matikova')
>>> mato = urob('Martin', 'Fyzik')
>>> Student.vypis(fero)
    volam sa Ferdinand Fyzik
>>> Student.vypis(zuzka)
    volam sa Zuzana Matikova
>>> Student.vypis(mato)
    volam sa Martin Fyzik
```

Takýto spôsob volania metód však nie je bežný:

```
Trieda.metoda(instancia, parametre)
```

Namiesto neho sa používa trochu pozmenený, pričom sa vynecháva meno triedy. Budeme používať takéto poradie zápisu volania metódy:

```
instancia.metoda(parametre)
```

čo znamená:

```
>>> fero.vypis()
    volam sa Ferdinand Fyzik
>>> zuzka.vypis()
    volam sa Zuzana Matikova
>>> mato.vypis()
    volam sa Martin Fyzik
```

Podobne zapíšeme priradenie hoby dvom študentom. Namiesto zápisu:

```
>>> Student.nastav_hoby(fero, 'gitara')
    Ferdinand Fyzik ma hoby gitara
>>> Student.nastav_hoby(evka, 'cyklistika')
    Eva Matikova ma hoby cyklistika
```

si radšej zvykneme na:

```
>>> fero.nastav_hoby('gitara')
    Ferdinand Fyzik ma hoby gitara
>>> evka.nastav_hoby('cyklistika')
    Eva Matikova ma hoby cyklistika
```

S takýmto zápisom volania funkcií (teda metód) sme sa už stretli skôr, ale asi to bola pre nás doteraz veľká záhada, napríklad:

```
>>> zoznam = [2, 5, 7]
>>> zoznam.append(11)
>>> zoznam.pop(0)
    2
```

```
>>> a = '12-34-56'.split('-')
```

znamená:

```
>>> zoznam = [2, 5, 7]
>>> list.append(zoznam, 11)
>>> list.pop(zoznam, 0)
    2
>>> a = str.split('12-34-56', '-')
```

Teda `append()` je metóda triedy `list` (pythonovský zoznam), ktorá má dva parametre: `self` (samotný zoznam, ktorý sa bude modifikovať) a `hodnota`, ktorá sa bude do zoznamu pridávať na jeho koniec. Táto metóda je zrejme definovaná niekde v triede `list` a samotná jej deklarácia by mohla vyzerať nejako takto:

```
class list:
    ...
    def append(self, hodnota):
        ...
```

Magické metódy

Do novo vytváranej triedy môžeme pridávať ľubovoľné množstvo metód (súkromných funkcií), pričom majú jediné obmedzenie: prvý parameter by mal mať meno `self`. Už vieme, že takúto metódu môžeme volať nielen:

```
Trieda.metoda(instancia, parametre)
```

ale radšej ako:

```
instancia.metoda(parametre)
```

Okrem tohto štandardného mechanizmu volania metód, existuje ešte niekoľko **špeciálnych metód**, pre ktoré má Python aj iné využitie. Pre tieto špeciálne (tzv. **magické**) metódy má Python aj špeciálne pravidlá. My sa s niektorými z týchto magických metód budeme zoznamovať priebežne na rôznych prednáškach, podľa toho, ako ich budeme potrebovať.

Magické metódy majú definíciu úplne rovnakú ako bežné metódy. Python ich rozpozná podľa ich mena: ich meno začína aj končí dvojicou podčiarkovníkov `__`. Pre Python je tento znak bežná súčasť identifikátorov, ale využíva ich aj na tento špeciálny účel. Ako prvé sa zoznámime s magickou metódou `__init__()`, ktorá je jednou z najužitočnejších a najčastejšie definovaných magických metód.

metóda `__init__()`

Je magická metóda, ktorá slúži na **inicializovanie atribútov** daného objektu. Má tvar:

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

Metóda môže (ale nemusí) mať ďalšie parametre za `self`. Metóda nič nevracia, ale najčastejšie obsahuje len niekoľko priradení.

Túto metódu (ak existuje) Python **automaticky** zavolá, v tom momente, keď sa vytvára nová inštancia.

Keď zapíšeme `instancia = Trieda(parametre)`, tak Python postupne:

1. vytvorí nový objekt typu `Trieda` - zatiaľ je to **prázdny kontajner**
   - vytvorí sa pomocná referencia na tento nový `objekt`
2. ak existuje metóda `__init__()`, zavolá ju s príslušnými parametrami: `Trieda.__init__(objekt, parametre)`
   - keď Python zavolá našu metódu `__init__()`, znamená to, že samotný objekt už existuje (dostaneme ho v parametri `self`), ale zatiaľ je to prázdny kontajner bez atribútov premenných - tie vzniknú až priraďovacím príkazom do týchto atribútov
3. do premennej `instancia` priradí práve vytvorený `objekt`
   - v tejto premennej už máme hotový objekt, ktorý prešiel inicializáciou v `__init__()`

Hovoríme, že metóda `__init__()` **inicializuje** objekt (niekedy sa hovorí aj, že **konštruuje**, resp. že je to **konštruktor**). Najčastejšie sa v tejto metóde priraďujú hodnoty do atribútov, napríklad:

```
class Student:

    def __init__(self, meno, priezvisko, hoby=''):
        self.meno = meno
        self.priezvisko = priezvisko
        self.hoby = hoby

    def vypis(self):
        print('volam sa', self.meno, self. priezvisko)

    def nastav_hoby(self, text):
        self.hoby = text
        print(self.meno, self. priezvisko, 'ma hoby', self.hoby)
```

Vďaka tomu už nepotrebujeme funkciu `urob()`, ale inštanciu aj s atribútmi vyrobíme pomocou konštruktora:

```
>>> fero = Student('Ferdinand', 'Fyzik')
>>> fero.nastav_hoby('gitara')
    Ferdinand Fyzik ma hoby gitara
>>> evka = Student('Eva', 'Matikova, 'cyklistika')
```

Štandardná funkcia dir()

Funkcia `dir()` vráti postupnosť (zoznam) všetkých atribútov triedy alebo inštancie. Pozrime najprv nejakú prázdnu triedu:

```
>>> class Test: pass
>>> dir(Test)
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
    '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
```

```
    '__subclasshook__', '__weakref__']
```

Vidíme, že napriek tomu, že sme zatiaľ pre túto triedu nič nedefinovali, v triede sa nachádza veľa rôznych atribútov. Jeden z nich už poznáme: `__init__` je magická metóda.

Vždy keď zadefinujeme nový atribút alebo metódu, objaví sa aj v tomto zozname `dir()`:

```
>>> t = Test()
>>> t.x = 100
>>> t.y = 200
>>> dir(t)
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
    '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
    '__subclasshook__', '__weakref__', 'x', 'y']
```

Na konci tohto zoznamu sú dva nové atribúty `x` a `y`.

Príklad s grafikou

Zadefinujeme novú triedu `Kruh(r, x, y)`, ktorá bude mať 3 atribúty pre kruh v grafickej ploche: polomer a súradnice stredu:

```
class Kruh:

    def __init__(self, r, x, y):
        self.r = r
        self.x = x
        self.y = y
```

Teraz, keď máme triedu, môžeme vytvárať nové inštancie (objekty), napríklad:

```
>>> a = Kruh(70, 200, 100)
>>> b = Kruh(10, 180, 80)
>>> c = Kruh(10, 220, 80)
```

Tieto objekty sú zatiaľ len „kontajnery“ pre atribúty.

Do takejto triedy môžeme v inicializácii pridať aj ďalšie atribúty, ktoré nie sú v parametroch inicializácie, napríklad:

```
class Kruh:

    def __init__(self, r, x, y):
        self.r = r
        self.x = x
        self.y = y
        self.farba = 'blue'
```

Znamená, že vždy keď vytvoríme nový objekt, okrem 3 atribútov `r`, `x` a `y` sa vytvorí aj atribút `farba` s hodnotou `'blue'`.

Teraz zadefinujeme pomocnú funkciu `kresli_kruh(kruh)`, ktorá očakáva parameter typu `Kruh` a tento kruh potom nakreslí do grafickej plochy (predpokladáme, že grafická plocha je už vytvorená a prístupná pomocou premennej `canvas`):

```
def kresli_kruh(kruh):
    canvas.create_oval(kruh.x-kruh.r, kruh.y-kruh.r,
                       kruh.x+kruh.r, kruh.y+kruh.r,
                       fill=kruh.farba)
```

Otestujeme:

```
import tkinter

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

a = Kruh(70, 200, 100)
a.farba = 'yellow'
b = Kruh(10, 180, 80)
c = Kruh(10, 220, 80)
kresli_kruh(a)
kresli_kruh(b)
kresli_kruh(c)

tkinter.mainloop()
```

Takéto objekty kruhy môžeme uložiť aj do zoznamu a potom aj ich nakreslenie môže vyzerať takto:

```
...
zoznam = [a, b, c]
for k in zoznam:
    kresli_kruh(k)
```

Ak teraz zadáme:

```
>>> zoznam
    [<__main__.Kruh object>, <__main__.Kruh object>, <__main__.Kruh object>]
```

vidíme len to, že zoznam obsahuje nejaké tri objekty (inštancie) typu `Kruh`. Zadefinujme preto metódu `výpis()`, ktorá vypíše detaily konkrétneho objektu. Do triedy `Kruh` dopíšeme túto metódu a do konštruktora pridáme aj štvrtý parameter `farba`. Tiež funkciu `kresli_kruh()` prepíšeme na metódu `kresli()`:

```
import tkinter

class Kruh:

    def __init__(self, r, x, y, farba='blue'):
        self.r = r
        self.x = x
        self.y = y
        self.farba = farba
```

```
    def vypis(self):
        print(f'Kruh({self.r}, {self.x}, {self.y}, {self.farba!r})')

    def kresli(self):
        canvas.create_oval(self.x-self.r, self.y-self.r,
                           self.x+self.r, self.y+self.r,
                           fill=self.farba)

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

a = Kruh(70, 200, 100, 'yellow')
b = Kruh(10, 180, 80)
c = Kruh(10, 220, 80)

zoznam = [a, b, c]
for k in zoznam:
    k.kresli()
for k in zoznam:
    k.vypis()

tkinter.mainloop()
```

Tento program teraz vypíše:

```
Kruh(70, 200, 100, 'yellow')
Kruh(10, 180, 80, 'blue')
Kruh(10, 220, 80, 'blue')
```

Pozrime ešte, čo nám vrátia volania funkcie `dir()` pre triedu `Kruh` aj inštanciu `a`:

```
>>> dir(Kruh)
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
    '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
    '__subclasshook__', '__weakref__', 'kresli', 'vypis']
>>> dir(a)
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
    '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
    '__subclasshook__', '__weakref__', 'farba', 'kresli', 'vypis', 'r', 'x', 'y']
```

Všimnite si, že v triede `Kruh` pribudli dva atribúty, ktoré nie sú magickými metódami: `kresli` a `vypis`, v inštancii `a` okrem týchto metód pribudli 4 atribúty: `farba`, `r`, `x` a `y`.

Cvičenia

L.I.S.T.

- riešenia **aspoň 8 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- pozri si Riešenie úloh 14. cvičenia

1. Zadefinuj triedu `Cas`, ktorá bude mať dva celočíselné atribúty `hodiny` a `minuty`.
   Aj **inicializácia** (metóda `__init__()`) bude mať dva
   parametre `hodiny` a `minuty`. **Metóda** `vypis()` vypíše nastavený čas v tvare `čas je 9:17`.
   Trieda `Cas`:

```
class Cas:
    ...
```

   Otestuj, napríklad:

```
>>> c = Cas(9, 17)
>>> c.vypis()
    čas je 9:17
>>> d = Cas(10, 5)
>>> d.vypis()
    čas je 10:05
```

   Zamysli sa, čo sa stane pre volanie `Cas.vypis(c)`, čím sa to líši od `c.vypis()`?

2. Do triedy `Cas` z úlohy (1) pridaj **metódu** `str()`, ktorá nič nevypisuje, ale namiesto toho vráti (`return`) znakový reťazec s hodinami a minútami v tvare `'9:17'`. Napríklad:

```
>>> c = Cas(9, 1)
>>> print('teraz je', c.str())
    teraz je 9:01
```

3. Do triedy `Cas` z (2) úlohy dopíš **metódu** `pridaj()`, ktorá bude mať 2
   parametre `hodiny` a `minuty`. Metóda pridá k uloženému času zadané hodiny a minúty.
   Napríklad:

```
>>> cas = Cas(17, 40)
>>> print('teraz je', cas.str())
    teraz je 17:40
>>> cas.pridaj(1, 35)
>>> print('neskôr', cas.str())
    neskôr 19:15
```

4. Napíš **globálnu funkciu** (nie metódu) `neskor(cas, hodiny, minuty)`, ktorá vytvorí
   (`return`) novú inštanciu triedy `Cas`. Táto nová inštancia bude od zadaného času
   (parameter `cas` je tiež inštancia triedy `Cas`) posunutá o zadaný počet hodín a minút.
   Využi metódu `pridaj()`. Napríklad:

```
>>> c = Cas(17, 40)
>>> d = neskor(c, 2, 55)
>>> print(c.str())
    17:40
>>> print(d.str())
    20:35
```

5. Vytvor pätnásť-prvkový zoznam inštancií triedy `Cas`, v ktorom prvý prvok reprezentuje `8:10` a každý ďalší je posunutý o 50 minút. Ďalšie časy v zozname vytváraj v cykle, využi funkciu `neskor()`. Napríklad:

```
zoznam = [Cas(8, 10)]
for i in range(14):       # vyrobí 15-prvkový zoznam časov
    zoznam ...
for c in zoznam:
    print(c.str(), end=' ')
```

vypíše:

```
8:10 9:00 9:50 ... 19:50
```

6. Zapíš definíciu triedy `Zlomok`, ktorá v **inicializácii** vytvorí dva atribúty `citatel` a `menovatel`. **Metóda** `vypis()` vypíše pomocou `print()` tento zlomok v tvare `zlomok je 3/8`. Napríklad:

```
>>> z1 = Zlomok(3, 8)
>>> z2 = Zlomok(2, 4)
>>> z1.vypis()
    zlomok je 3/8
>>> z2.vypis()
    zlomok je 2/4
```

7. Pridaj do triedy `Zlomok` z úlohy (6) dve metódy:
   - **metóda** `str()` vráti (nič nevypisuje) reťazec v tvare `3/8`
   - **metóda** `float()` vráti (nič nevypisuje) desatinné číslo, ktoré reprezentuje daný zlomok

   Napríklad:

```
>>> z = Zlomok(3, 8)
>>> print('z je', z.str())
    z je 3/8
>>> print('z je', z.float())
    z je 0.375
>>> w = Zlomok(2, 4)
>>> print('w je', w.str())
    w je 2/4
>>> print('w je', w.float())
    w je 0.5
```

8. Zadefinuj triedu `Body`, ktorá si bude uchovávať momentálny stav bodov (napríklad získané body v nejakej hre). Trieda bude mať tieto metódy:
   - **metóda** `pridaj()` k momentálnemu stavu pridá 1 bod
   - **metóda** `uber()` od momentálneho stavu odoberie 1 bod;
   - **metóda** `kolko()` vráti celé číslo - momentálny bodový stav

Napríklad:

```
>>> b = Body()
>>> for i in range(10):
        b.pridaj()
>>> b.uber()
>>> b.uber()
>>> print('body =', b.kolko())
    body = 8
```

9. Zadefinuj triedu `Subor` s metódami:
   - **inicializácia** `__init__(meno_suboru)` vytvorí nový prázdny súbor
   - **metóda** `pripis(text)` na koniec súboru pridá nový riadok so zadaným textom; použi `open(..., 'a')`
   - **metóda** `vypis()` vypíše (`print`) momentálny obsah súboru

   Napríklad:

```
>>> s = Subor('text.txt')
>>> s.pripis('prvy riadok')
>>> s.pripis('druhy riadok')
>>> s.vypis()
    prvy riadok
    druhy riadok
>>> s.pripis('posledny riadok')
>>> s.vypis()
    prvy riadok
    druhy riadok
    posledny riadok
```

10. Zadefinuj triedu `Kniha`, ktorá bude udržiavať informácie o jednej knihe. Trieda bude mať tieto metódy:
    - **inicializácia** `__init__(autor, titul)`
    - **metóda** `nastav_vydavatela(vydavatel)` pridá informáciu o vydavateľovi
    - **metóda** `nnastav_rok(rok)` pridá informáciu o roku vydania
    - **metóda** `vypis()` vypíše informácie o knihe

    Napríklad:

```
k1 = Kniha('Dobsinsky', 'Rozpravky')
k1.nastav_vydavatela('Mlade Leta')
k2 = Kniha('Lasica', 'Bodka')
k2.nastav_rok(2007)
k1.vypis()
k2.vypis()
```

vypíše:

```
Kniha: Dobsinsky: Rozpravky,  Mlade Leta
Kniha: Lasica: Bodka, 2007
```

11. Zadefinuj triedu `Zoznam`, pomocou ktorej si budeme vedieť udržiavať zoznam svojich záväzkov (sľubov, povinností, …). Tieto budeš uchovávať v atribúte `zoznam` typu `list`. Trieda obsahuje tieto metódy:
    - **metóda** `pridaj(prvok)`, ak sa tam takýto záväzok ešte nenachádza, pridá ho na koniec
    - **metóda** `vyhod(prvok)`, ak sa tam takýto záväzok nachádzal, vyhodí ho
    - **metóda** `je_v_zozname(prvok)` vráti `True` alebo `False` podľa toho, či sa tam tento záväzok nachádza
    - **metóda** `vypis()` vypíše všetky záväzky v tvare `zoznam: záväzok, záväzok, záväzok`

    Napríklad:

```
moj = Zoznam()
moj.pridaj('upratat')
moj.pridaj('behat')
moj.pridaj('ucit sa')
if moj.je_v_zozname('behat'):
    print('musis behat')
else:
    print('nebehaj')
moj.pridaj('upratat')
moj.vyhod('spievat')
moj.vypis()
```

    vypíše:

```
musis behat
zoznam: upratat, behat, ucit sa
```

12. Zadefinuj triedu `TelefonnyZoznam`, ktorá bude udržiavať informácie o telefónnych číslach (ako zoznam `list` dvojíc `tuple`). Trieda bude mať tieto metódy:
    - **metóda** `pridaj(meno, telefon)` pridá do zoznamu dvojicu `(meno, telefon)`
        - ak takéto `meno` v zozname už existovalo, nepridáva novú dvojicu, ale nahradí len telefónne číslo
    - **metóda** `vypis()` vypíše celý telefónny zoznam.

    Napríklad:

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

    vypíše:

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

13. Zadefinuj triedu `Okno`, ktorá otvorí grafické okno a do stredu vypíše zadaný text. Výška otvoreného okna nech je 100. Vypísaný text nech je v strede okna fontom veľkosti `50`. **Inicializácia** (metóda `__init__()`) vytvorí nový `canvas` (výšky 100) a do jeho stredu vypíše zadaný text. V svojich atribútoch si zapamätá `canvas` aj identifikačný kód pre `create_text()`. Ďalšie dve metódy menia vypísaný text:
    - **metóda** `zmen(text)` zmení vypísaný text (zrejme na to použije `itemconfig()`)
    - **metóda** `farba(farba)` zmení farbu vypísaného textu (zrejme na to použije `itemconfig()`)

    Napríklad:

```
import tkinter
okno = Okno('ahoj')
okno.farba('red')
okno.zmen('Python')
```

    Vyskúšaj vytvoriť dve inštancie `Okno`.