15. Triedy a metódy

video prezentácia

triedy a metódy

Zhrňme, čo už vieme o triedach a objektoch

Novú triedu najčastejšie definujeme takto:

```
class Meno_triedy:

    def __init__(self, parametre):
        ...

    def metoda1(self, parametre):
        ...

    def metoda2(self, parametre):
        ...

    def metoda3(self, parametre):
        ...
```

Objekty (inštancie triedy) vytvárame a používame takto:

```
>>> premenna = Meno_triedy(...)   # skonštruovanie objektu
>>> premenna.atribut = hodnota    # vytvorenie nového atribútu/zmena hodnoty atribútu
>>> premenna.metoda(parametre)    # zavolanie metódy
```

Metóda musí mať pri definovaní prvý parameter `self`, ktorý reprezentuje samotnú inštanciu: Python sem pri volaní metódy automaticky dosadí samotný objekt, nasledovné dvojice volaní robia to isté:

```
>>> premenna.metoda(parametre)
>>> Meno_triedy.metoda(premenna, parametre)

>>> 'mama ma emu'.count('ma')
    3
>>> str.count('mama ma emu', 'ma')
    3
```

Metódy sú súkromné funkcie definované v triede. Pozrime sa na príklad z cvičení, v ktorom sme definovali triedu `Cas` s dvoma atribútmi `hodiny` a `minuty`:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def vypis(self):
        print(f'cas je {self.hodiny}:{self.minuty:02}')

    def str(self):
```

```
        return f'{self.hodiny}:{self.minuty:02}'
```

Spomínali sme, že v Pythone všetky funkcie (a teda aj metódy) môžeme rozdeliť do dvoch základných typov:

- **modifikátor** - mení niektorú hodnotu atribútu (alebo aj viacerých), napríklad:

```
- class Cas:
-     ...
-     def pridaj(self, hodiny, minuty):
-         self.hodiny += hodiny + (self.minuty+minuty) // 60
-         self.minuty = (self.minuty+minuty) % 60
```

- **pravá funkcia** - nemení atribúty a nemá ani žiadne vedľajšie účinky (nemení globálne premenné); najčastejšie vráti nejakú hodnotu - môže to byť aj nový objekt, napríklad:

```
- class Cas:
-     ...
-     def kopia(self):
-         return Cas(self.hodiny, self.minuty)
```

- uvedomte si, že nemeniteľné typy (**immutable**) obsahujú iba pravé funkcie (zrejme okrem inicializácie `__init__()`, ktorá bude veľmi často modifikátor)

Magická metóda __str__

Niektoré metódy sú špeciálne (tzv. **magické**) - nimi definujeme špeciálne správanie (nová trieda sa lepšie prispôsobí filozofii Pythonu). Zatiaľ sme sa zoznámili len s jednou magickou metódou `__init__()`, ktorá inicializuje atribúty - automaticky sa vyvolá hneď po vytvorení (skonštruovaní) objektu.

Ďalšou veľmi často definovanou magickou metódou je `__str__()`. Jej využitie ukážeme na príklade triedy `Cas`, ktorú sme doteraz používali takto:

```
>>> c = Cas(9, 17)
>>> c.vypis()
    cas je 9:17
>>> print('teraz je', c.str())
    teraz je 9:17
```

Už sme si trochu zvykli, že keď priamo vypisujeme inštanciu `c`, dostaneme:

```
>>> c
    <__main__.Cas object at 0x03220F10>
>>> print(c)
    <__main__.Cas object at 0x03220F10>
>>> str(c)
    '<__main__.Cas object at 0x03220F10>'
```

Čo je ale dosť nezaujímavá informácia. Asi by bolo užitočné, keby Python nejako pochopil, že reťazcová reprezentácia nášho objektu by mohla byť výsledkom nejakej našej metódy a automaticky by ju použil, napríklad v `print()` alebo aj v `str()`.

Naozaj presne toto aj funguje: keď zadefinujeme magickú metódu `__str__()` a Python na niektorých miestach bude potrebovať reťazcovú reprezentáciu objektu, tak zavolá túto našu vlastnú metódu. Zrejme výsledkom tejto metódy **musí byť** znakový reťazec, inak by Python protestoval. Opravme triedu `Cas` so zadefinovaním magickej metódy:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def __str__(self):
        return f'{self.hodiny}:{self.minuty:02}'

    def vypis(self):
        print('cas je', self)     # Python tu za nas urobil self.__str__()
```

Všimnite si, ako sme mohli vďaka tomuto opraviť metódu `vypis()`: keďže `print()` po vypísaní `'cas je'` chce vypísať aj `self` a zrejme táto inštancia nie je znakový reťazec (je to predsa inštancia triedy `Cas`), Python pohľadá, či táto trieda nemá náhodou definovanú metódu na reťazcovú reprezentáciu tohto objektu (teda `__str__()`) a keďže takú nájde, zavolá ju a má znakový reťazec z premennej `self`. Otestujeme:

```
>>> c = Cas(9, 17)
>>> c.vypis()
    cas je 9:17
>>> print('teraz je', c.__str__())
    teraz je 9:17
>>> print('teraz je', c)
    teraz je 9:17
>>> c
    <__main__.Cas object at 0x03220F10>
>>> print(c)
    9:17
>>> str(c)
    '9:17'
```

Všimnite si nie veľmi pekný zápis volania `c.__str__()`. Teraz už vieme, že to takto zapisovať nemusíme: v príkaze `print()` stačí písať len meno premennej `c`, prípadne, ak potrebujeme naozaj reťazec, krajší je zápis `str(c)`. Štandardnú funkciu `str()` by ste si mohli predstaviť, že je definovaná takto:

```
def str(objekt=''):
    return objekt.__str__()
```

Pričom predpokladáme, že v každej triede Pythonu je zadefinovaná magická metóda `__str__()`. Ak sa o tom chcete presvedčiť, vyskúšajte:

```
>>> x = 3.14
>>> str(x)
    '3.14'
>>> str(123)
    '123'
>>> str([123, x])
    '[123, 3.14]'
```

teraz už vieme, že v skutočnosti sa zavolá (`inštancia.metóda()`):

```
>>> x.__str__()
    '3.14'
>>> (123).__str__()
    '123'
>>> [123, x].__str__()
    '[123, 3.14]'
```

ale aj toto je len „skrátený tvar“ takýchto volaní (`trieda.metóda(inštancia)`):

```
>>> float.__str__(x)
    '3.14'
>>> int.__str__(123)
    '123'
>>> list.__str__([123, x])
    '[123, 3.14]'
```

Volanie metódy z inej metódy

Zatiaľ sme v našich jednoduchých príkladoch, v ktorých sme definovali nejaké triedy, neriešili situácie, v ktorých v jednej metóde voláme nejakú inú metódu tej istej triedy. Doplňme do triedy `Cas` aj metódu `pridaj()`:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def __str__(self):
        return f'{self.hodiny}:{self.minuty:02}'

    def vypis(self):
        print('cas je', self)

    def kopia(self):
        return Cas(self.hodiny, self.minuty)

    def pridaj(self, hodiny, minuty):
        self.hodiny += hodiny + (self.minuty+minuty) // 60
        self.minuty = (self.minuty+minuty) % 60
```

Pridajme teraz ďalšiu metódu `kopia_a_pridaj()`, ktorá vyrobí kópiu objektu a zároveň v tejto kópii posunie hodiny aj minúty:

```
class Cas:
    ...
    def kopia_a_pridaj(self, hodiny, minuty):
        novy = Cas(self.hodiny, self.minuty)
        novy.pridaj(hodiny, minuty)
        return novy
```

Vidíme, že novovytvorený objekt `novy` zavolal svoju metódu `pridaj()`, preto sme to museli zapísať `novy.pridaj(...)`. Prvý riadok tejto metódy je priradenie `novy = Cas(self.hodiny, self.minuty)`, ktoré vytvorí kópiu objektu `self`. Ale na toto už máme hotovú metódu `kopia()`, takže môžeme to zapísať aj takto:

```
class Cas:
    ...
    def kopia_a_pridaj(self, hodiny, minuty):
        novy = self.kopia()
        novy.pridaj(hodiny, minuty)
        return novy
```

Tu vidíme, že keď potrebujeme, aby objekt `self` zavolal niektorú svoju metódu, musíme pred meno metódy pripísať `self` aj s bodkou, tak ako je to v tejto metóde, teda `self.kopia()`. Uvedomte si, že bez tohto `self.` by toto označovalo volanie obyčajnej funkcie (nie metódy), ktorá je buď globálna alebo niekde

lokálne zadefinovaná. Metódy teda voláme buď `self.metóda()` alebo pre inú ako `self` inštanciu `novy.metóda()`

Ďalej uvedieme niekoľko príkladov, v ktorých sa stretneme s doteraz vysvetľovanými pojmami.

Príklad s nemeniteľnou triedou čas

Vylepšíme triedu `Cas`: bude mať 3 atribúty: `hod`, `min`, `sek` (pre hodiny, minúty, sekundy). Všetky metódy vytvoríme ako **pravé funkcie**, vďaka čomu sa bude táto trieda správať ako **immutable** (nemeniteľný typ):

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.hod = hodiny
        self.min = minuty
        self.sek = sekundy

    def __str__(self):
        return f'{self.hod}:{self.min:02}:{self.sek:02}'

    def sucet(self, iny):
        return Cas(self.hod + iny.hod, self.min + iny.min, self.sek + iny.sek)

    def vacsi(self, iny):
        return (self.hod > iny.hod or
                self.hod == iny.hod and self.min > iny.min or
                self.hod == iny.hod and self.min == iny.min and self.sek > iny.sek)
```

Zadefinovali sme dve nové metódy `sucet()` a `vacsi()`. Metóda `sucet()` vytvorí novú inštanciu a `vacsi()` zistí, či je čas väčší ako nejaký iný.

Otestujme:

```
cas1 = Cas(10, 22, 30)
cas2 = Cas(10, 8)
print('cas1 =', cas1)
print('cas2 =', cas2)
print('sucet =', cas1.sucet(cas2))
print('cas1 > cas2 =', cas1.vacsi(cas2))
print('cas2 > cas1 =', cas2.vacsi(cas1))
```

Vypíše:

```
cas1 = 10:22:30
cas2 = 10:08:00
sucet = 20:30:30
cas1 > cas2 = True
cas2 > cas1 = False
```

Vidíme, že metóda `vacsi()`, ktorá porovnáva dva časy, je dosť prekomplikovaná, lebo treba porovnávať tri atribúty v jednom aj druhom objekte. Hoci pri triede `Cas` by sme mohli metódu `vacsi` trochu zjednodusit aj tekto:

```
class Cas:

    ...

    def vacsi(self, iny):
```

```
        return (self.hod, self.min, self.sek) > (iny.hod, iny.min, iny.sek)
```

Pomocná metóda

Predchádzajúce riešenie má viac problémov:

- pomocou metódy `sucet()` môžeme vytvoriť čas, v ktorej minúty alebo sekundy majú hodnotu väčšiu ako 59
- dva časy sa porovnávajú dosť komplikovane

Vytvorme pomocnú funkciu (teda metódu), ktorá z daného času vypočíta celkový počet sekúnd. Zároveň opravíme aj inicializáciu `__init__()`:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        cas = abs(3600*hodiny + 60*minuty + sekundy)
        self.hod = cas // 3600
        self.min = cas // 60 % 60
        self.sek = cas % 60

    def __str__(self):
        return f'{self.hod}:{self.min:02}:{self.sek:02}'

    def sucet(self, iny):
        return Cas(self.hod + iny.hod, self.min + iny.min, self.sek + iny.sek)

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

    def pocet_sekund(self):
        return 3600*self.hod + 60*self.min + self.sek

    def vacsi(self, iny):
        return self.pocet_sekund() > iny.pocet_sekund()

cas1 = Cas(10, 22, 30)
cas2 = Cas(9, 58, 45)
print('cas1 =', cas1)
print('cas2 =', cas2)
print('sucet =', cas1.sucet(cas2))
print('cas1 > cas2 =', cas1.vacsi(cas2))
print('cas2 > cas1 =', cas2.vacsi(cas1))
print('cas1 - cas2 =', cas1.rozdiel(cas2))
print('cas2 - cas1 =', cas2.rozdiel(cas1))
```

Pomocnú funkciu `pocet_sekund()` sme využili nielen v porovnávaní dvoch časov (metóda `vacsi()`) ale aj v novej metóde `rozdiel()`.

Celá trieda sa dá ešte viac zjednodušiť, ak by samotný objekt nemal 3 atribúty `hod`, `min` a `sek`, ale len jediný atribút `sek` pre celkový počet sekúnd. Vďaka tomu už nebudeme musieť pri každej operácii čas prepočítavať na sekundy. Len pri výpise budeme sekundy prevádzať na hodiny a minúty. Napríklad takto:

```
class Cas:

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

    def __str__(self):
        return f'{self.sek // 3600}:{self.sek // 60 % 60:02}:{self.sek % 60:02}'
```

```
    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
```

Ak budeme teraz vytvárať zoznam časov podobne, ako v jednej úlohe z predchádzajúceho cvičenia: prvý z nich je `8:10` a každý ďalší je o 50 minút posunutý, môžeme to zapísať napríklad takto:

```
zoznam = [Cas(8, 10)]

for i in range(14):
    zoznam.append(zoznam[-1].sucet(Cas(0, 50)))

for cas in zoznam:
    print(cas, end=' ')
```

Zápis `zoznam[-1].sucet(Cas(0, 50))` označuje, že k času, ktorý je momentálne posledným prvkom v zozname pripočítame 50 minút (teda čas, ktorý je 0 hodín a 50 minút). Ak by sme vedeli zabezpečiť sčitovanie časov rovnakým zápisom ako je napríklad sčitovanie čísel alebo zreťazovanie reťazcov, tento zápis by vyzeral `zoznam[-1] + Cas(0, 50)`, čo už vyzerá zaujímavo, ale žiaľ nefunguje.

Triedne a inštančné atribúty

Už vieme, že

- triedy sú kontajnery na súkromné funkcie (metódy)
- inštancie sú kontajnery na súkromné premenné (atribúty)

Napríklad

```
>>> class Test: pass
>>> t.x = 100              # nový atribút v inštancii
>>> t.y = 200
```

Lenže atribúty ako premenné môžeme definovať aj v triede, vtedy sú to tzv. **triedne atribúty** (atribúty na úrovni inštancií sú **inštančné atribúty**). Ak teda definujeme triedny atribút:

```
>>> Test.z = 300           # nový atribút v triede
```

tak tento atribút automaticky získavajú (vidia) aj všetky inštancie tejto triedy (tak ako všetky inštancie vidia všetky metódy triedy):

```
>>> print(t.x, t.y, t.z)
    100 200 300
```

Aj novovytvorená inštancia získava (teda vidí) tento triedny atribút:

```
>>> t2 = Test()
>>> t2.z
    300
```

Lenže tento atribút sa zatiaľ nachádza iba v kontajneri triedy `Test`, v kontajneroch inštancií atribút s takýmto menom nie je. Inštancie tento triedny atribút vidia (môžu ho čítať), ale tento sa v ich kontajneri nenachádza.

Ak ho chceme mať ako súkromnú premennú v inštancii (inštančný atribút), musíme mu **v inštancii** priradiť hodnotu:

```
>>> t2.z = 400
>>> Test.z
    300
>>> t.z
    300
>>> t2.z
    400
```

Kým do inštancie nepriradíme tento atribút, inštancia „vidí“ hodnotu triedy, keď už vyrobíme vlastný atribút, tak už vidí túto novú hodnotu. Uvedomte si, že momentálne existuje **triedny atribút** `Test.z` a s rovnakým menom aj inštančný atribút `t2.z`. Inštancia `t2` teraz po zapísaní `t2.z` vidí už len tento svoj súkromný atribút.

Triedne atribúty môžeme vytvoriť už pri definovaní triedy, napríklad:

```
class Test:
    z = 300

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

    def __str__(self):
        return f'test {self.x},{self.y},{self.z}'

>>> t1 = Test(100, 200)
>>> print(t1)
    test 100,200,300
>>> t2 = Test(10, 20)
>>> t2.z = 30
>>> print(t2)
    test 10,20,30
```

Triedny atribút `z` má stále hodnotu 300, hoci inštancia `t2` má už svoju vlastnú verziu inštančného atribútu `z` s hodnotou 30 a preto pri výpise `t2.z` vidí len svoj atribút `z`. Keď zmeníme obsah triedneho atribútu, dostaneme:

```
>>> Test.z = 9
>>> print(t1)
    test 100,200,9
>>> print(t2)
    test 10,20,30
```

Dobrou zásadou pri definovaní triedy a jej metód je **nepoužívať žiadne globálne premenné**. Pozrime teraz takýto príklad triedy:

```
import tkinter
import random

class Bodka:

    def __init__(self, canvas, x, y):
        self.id = canvas.create_oval(x - 5, y - 5, x + 5, y + 5)
        self.canvas = canvas

    def prefarbi(self):
```

```
        if random.randrange(2):
            farba = 'red'
        else:
            farba = 'blue'
        self.canvas.itemconfig(self.id, fill=farba)

canvas = tkinter.Canvas()
canvas.pack()
bodky = []
for i in range(100):
    bodky.append(Bodka(canvas, random.randint(10, 300), random.randint(10, 250)))
for b in bodky:
    b.prefarbi()
```

V programe sme vytvorili 100-prvkový zoznam bodiek (inštancií triedy `Bodka`), tieto sa už pri inicializácii nakreslili ako malé nezafarbené krúžky. Na záver sme všetky bodky náhodne prefarbili na modro alebo na červeno.

Keďže sme v metódach triedy nechceli pracovať s globálnou premennou `canvas`, poslali sme `canvas` ako parameter do inicializácie. Tu sa `canvas` zapamätal ako atribút každej jednej inštancie.

Ak by sme teraz dostali úlohu na záver vypísať počet modrých a červených, zdá sa, že bez globálnej premennej to bude veľmi ťažké.

Tu nám pomôžu triedne atribúty:

- `canvas` nemusíme posielať zvlášť každému objektu (takto v každom objekte vzniká inštančný atribút `canvas`, pričom všetky objekty triedy `Bodka` majú rovnakú hodnotu tohto atribútu), môžeme vytvoriť jediný triedny atribút, ktorý budú vidieť všetky inštancie
- pridáme ďalšie dva triedne atribúty pre počítanie počtu modrých a červených bodiek, pričom v metóde `prefarbi()` budeme tieto dve počítadla zvyšovať

Dostávame takúto verziu programu:

```
import tkinter
import random

class Bodka:
    canvas = None
    pocet_modrych = pocet_cervenych = 0

    def __init__(self, x, y):
        self.id = self.canvas.create_oval(x - 5, y - 5, x + 5, y + 5)

    def prefarbi(self):
        if random.randrange(2):
            farba = 'red'
            Bodka.pocet_cervenych += 1
        else:
            farba = 'blue'
            Bodka.pocet_modrych += 1
        self.canvas.itemconfig(self.id, fill=farba)

Bodka.canvas = tkinter.Canvas()
Bodka.canvas.pack()
bodky = []
for i in range(100):
    bodky.append(Bodka(random.randint(10, 300), random.randint(10, 250)))
for b in bodky:
    b.prefarbi()
print('pocet modrych =', Bodka.pocet_modrych)
print('pocet cervenych =', Bodka.pocet_cervenych)
```

Zamyslite sa nad tým, čo sa stane, keď v metóde `prefarbi()` zmeníme `Bodka.pocet_cervenych += 1` na `self.pocet_cervenych += 1`.

Príklad s grafickými objektmi

Postupne zadefinujeme niekoľko tried, ktoré pracujú s rôznymi objektmi v grafickej ploche.

Objekt Kruh

Zadefinujeme:

```
import tkinter

class Kruh:
    canvas = None

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

Kruh.canvas = tkinter.Canvas()
Kruh.canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)
```

Na začiatok definície triedy `Kruh` sme pridali vytvorenie triedneho atribútu `canvas` zatiaľ s hodnotou `None`. Robiť sme to nemuseli - funguje to rovnako dobre aj bez tohto priradenia, ale čitateľ nášho programu bude vidieť, že na tomto mieste počítame s triednym atribútom.

Aby sme mohli nakreslený objekt posunúť alebo zmeniť jeho veľkosť alebo farbu, musíme si zapamätať jeho identifikačné číslo, ktoré vráti funkcia `create_oval()`. Opäť využijeme mechanizmus metód objektu `canvas`, ktoré menia už nakreslený útvar:

- `canvas.move(id, dx, dy)` - posúva ľubovoľný útvar
- `canvas.itemconfig(id, nastavenie=hodnota, ...)` - zmení ľubovoľné nastavenie (napríklad farbu, hrúbku, …)
- `canvas.coords(id, x1, y1, x2, y2, ...)` - zmení súradnice útvaru

Zapíšme triedu `Kruh` aj s týmito ďalšími metódami:

```
import tkinter

class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self.x = x
        self.y = y
        self.r = r
        self.farba = farba
        self.id = self.canvas.create_oval(
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r,
            fill=self.farba)
```

```
    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, r):
        self.r = r
        self.canvas.coords(self.id,
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

Kruh.canvas = tkinter.Canvas()
Kruh.canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)

k1.posun(30,10)
k2.zmen(50)
k1.prefarbi('green')
```

Trieda Obdlznik

Skopírujeme triedu `Kruh` a zmeníme na `Obdlznik`:

```
import tkinter

class Obdlznik:
    canvas = None

    def __init__(self, x, y, sirka, vyska, farba='red'):
        self.x = x
        self.y = y
        self.sirka = sirka
        self.vyska = vyska
        self.farba = farba
        self.id = self.canvas.create_rectangle(
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska,
            fill=self.farba)

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, sirka, vyska):
        self.sirka = sirka
        self.vyska = vyska
        self.canvas.coords(self.id,
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

Obdlznik.canvas = tkinter.Canvas()
Obdlznik.canvas.pack()
r1 = Obdlznik(50, 50, 50, 30, 'blue')
```

```
r2 = Obdlznik(150, 100, 80, 80)
```

Trieda Skupina

Vyrobíme triedu `Skupina`, pomocou ktorej budeme ukladať rôzne útvary do jednej štruktúry:

```
import tkinter

class Skupina:
    def __init__(self):
        self.zoznam = []

    def pridaj(self, utvar):
        self.zoznam.append(utvar)

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

sk = Skupina()
sk.pridaj(Kruh(50, 50, 30, 'blue'))
sk.pridaj(Obdlznik(100, 20, 100, 50))
sk.zoznam[0].prefarbi('green')
sk.zoznam[1].posun(50)
```

Vidíme, ako môžeme teraz meniť už nakreslené útvary.

Ak budeme potrebovať meniť viac útvarov, použijeme cyklus:

```
for i in range(len(sk.zoznam)):
    sk.zoznam[i].prefarbi('orange')
```

alebo krajšie:

```
for utvar in sk.zoznam:
    utvar.posun(dy=15)
```

Do triedy `Skupina` môžeme doplniť metódy, ktoré pracujú so všetkými útvarmi v skupine, napríklad:

```
class Skupina:
    ...

    def prefarbi(self, farba):
        for utvar in self.zoznam:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.zoznam:
            utvar.posun(dx, dy)
```

Môžeme navrhnúť metódy, ktoré nebudú pracovať so všetkými útvarmi, ale len s útvarmi nejakého konkrétneho typu (napríklad len s kruhmi). Preto do tried `Kruh` aj `Obdlznik` doplníme ďalší atribút:

```
class Kruh:
    canvas = None
    typ = 'kruh'

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

```
class Obdlznik:
    canvas = None
    typ = 'obdlznik'

    def __init__(self, x, y, sirka, vyska, farba='red'):
        ...

class Skupina:
    ...
    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.prefarbi(farba)
```

Môžeme vygenerovať skupinu 20 náhodných útvarov - kruhov a obdĺžnikov:

```
import tkinter
import random

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

sk = Skupina()

for i in range(20):
    if random.randrange(2) == 0:
        sk.pridaj(Kruh(random.randint(50, 200), random.randint(50, 200), 30, 'blue'))
    else:
        sk.pridaj(Obdlznik(random.randint(50, 200), random.randint(50, 200), 40, 40))

sk.prefarbi_typ('kruh', 'yellow')
sk.posun_typ('obdlznik', -10, -25)
```

Metóda __str__()

Do oboch tried `Kruh` aj `Obdlznik` pridáme magickú metódu `__str__()` a vďaka tomu môžeme veľmi elegantne vypísať všetky útvary v skupine:

```
class Kruh:
    ...
    def __str__(self):
        return f'Kruh({self.x},{self.y},{self.r},{self.farba!r})'

class Obdlznik:
    ...
    def __str__(self):
        return f'Obdlznik({self.x},{self.y},{self.sirka},{self.vyska},{self.farba!r})'

...
for utvar in sk.zoznam:
    print(utvar)
```

a dostávame niečo takéto:

```
Obdlznik(185,50,40,40,'red')
Kruh(95,115,30,'blue')
Obdlznik(63,173,40,40,'red')
```

```
Kruh(138,176,30,'blue')
Obdlznik(92,50,40,40,'red')
Obdlznik(180,80,40,40,'red')
...
```

Cvičenia

L.I.S.T.

- riešenia **aspoň 8 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- pozri si Riešenie úloh 15. cvičenia

1. Zadefinuj triedu, pomocou ktorej budeš vedieť reprezentovať obdĺžniky. Pri obdĺžnikoch nás budú zaujímať len veľkosti strán a na základe toho sa bude dať vypočítať ich obsah aj obvod. Dopíš všetky metódy:

```
class Obdlznik:

    def __init__(self, a, b):
        # inicializuje
        ...

    def __str__(self):
        # vráti reťazec v tvare 'Obdlznik(100, 70)'
        ...

    def obsah(self):
        # vráti obsah
        ...

    def obvod(self):
        # vráti obvod
        ...

    def zmen_velkost(self, pomer):
        # vynásobí obe veľkosti strán zadaným pomerom
        ...

    def kopia(self):
        # vyrobí kópiu samého seba
        ...
```

   Otestuj, napríklad:

```
>>> obd1 = Obdlznik(20, 7)
>>> print('obvod =', obd1.obvod())
    obvod = 54
>>> print(obd1)
    Obdlznik(20, 7)
>>> obd2 = obd1.kopia()
>>> obd2.zmen_velkost(2)
>>> print(obd2)
    Obdlznik(40, 14)
```

2. Zober riešenie (11) úlohy z predchádzajúceho cvičenia: triedu `TelefonnyZoznam`, ktorá udržiava informácie o telefónnych číslach (ako zoznam `list` dvojíc `tuple`). Trieda mala 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.
- malo by fungovať:

```
3. tz = TelefonnyZoznam()
4. tz.pridaj('Jana', '0901020304')
5. tz.pridaj('Juro', '0911111111')
6. tz.pridaj('Jozo', '0212345678')
7. tz.pridaj('Jana', '0999020304')
8. tz.vypis()
```

Doplň do tejto triedy nové metódy tak, aby fungovalo aj zapisovanie aj čítanie s textovým súborom:

- metóda `__init__(meno_suboru)` si zapamätá meno súboru (nič ešte nezapisuje ani nečíta)
- metóda `zapis()` momentálny obsah telefónneho zoznamu zapíše do súboru: v každom riadku bude jedna dvojica hodnôt meno a číslo, pričom budú navzájom oddelené znakom `';'` (v jednom riadku súboru môže byť, napríklad `Jana;0999020304`)
- metóda `citaj()` prečíta zadaný súbor a vyrobí z neho zoznam dvojíc (`list` s prvkami `tuple`) - starý obsah zoznamu v pamäti sa zruší a nahradí novým
- malo by fungovať napríklad:

```
tz = TelefonnyZoznam('tel.txt')
tz.pridaj('Jana', '0901020304')
tz.pridaj(...
...
tz.zapis()     # zapísal do súboru
t2 = TelefonnyZoznam('tel.txt')
t2.citaj()
t2.vypis()     # pôvodny obsah
```

3. Zadefinuj triedu `Stv`, ktorá zabezpečí definovanie farebného štvorčeka v grafickej ploche. Trieda bude mať tieto matódy:

   - metóda `__init__(x, y, a=20, farba='')` vytvorí v grafickej ploche štvorček so stredom `(x, y)`, so stranou `a` a s výplňou `farba`; ak má parameter `farba` hodnotu prázdny reťazec, tak sa nahradí vygenerovanou náhodnou farbou
   - metóda `posun(dx, dy)` posunie objekt o `(dx, dy)`
   - metóda `zmen_farbu(farba)` prefarbí štvorček na zadanú farbu

   Okrem týchto metód definuj aj triedny atribút `canvas`, ktorý bude spoločný pre všetky definované štvorčeky. Otestuj napríklad:

```
for i in range(30):
   Stv(random.randint(50, 300), random.randint(50, 200))
```

   Otestuj aj posúvanie všetkých štvorčekov metódou `posun` aj prefarbovanie metódou `zmen_farbu`

4. Okrem triedy `Stv` z (3) úlohy zadefinuj triedu `Dvojica`, pomocou ktorej sa budú vytvárať dvojice „zlepených“ štvorčekov (inštancií `Stv`). Trieda bude mať tieto matódy:

   - metóda `__init__(x, y, a=20)` zadefinuje dva štvorčeky, pričom prvý z nich má stred v `(x, y)` a druhý je k nemu prilepený sprava (má posunutý `x`); oba majú veľkosť strán `a` a náhodné farby
   - metóda `posun(dx, dy)` posunie oba štvorčeky o `(dx, dy)`
   - metóda `vymen()` navzájom vymení farby oboch štvorčekov

Otestuj tak, že najprv vytvoríš 20 dvojíc na náhodných pozíciách s náhodnými veľkosťami z intervalu `<20, 50>`. Potom ich všetky poposúvaj o nejaké `(dx, dy)` a otestuj aj výmenu farieb v štvorčekoch.

5. Zadefinuj triedu `Klikanie` s dvomi metódami `__init__` a `klik`:
   - metóda `__init__` (okrem `self` nemá ďalšie parametre) vytvorí grafickú plochu (do `self.canvas`) a zviaže ju (`bind`) s metódou `self.klik` pri kliknutí do plochy (`'<ButtonPress>'`)
   - metóda `klik` zrejme musí mať jeden parameter `event` (okrem `self`), z ktorého získa `x` a `y` kliknutého miesta; metóda na toto kliknuté miesto nakreslí kružnicu s polomerom `5`
   - keď bude táto trieda hotová, program naštartuj a otestuj pomocou:

```
6. k = Klikanie()      # týmto sa zavolá konštruktor __init__()
```

6. Do triedy `Klikanie` z (5) úlohy pridaj ďalšiu metódu:
   - metóda `vypis` vypíše momentálny zoznam všetkých kliknutých bodov (dvojíc čísel), napríklad v tvare:

```
7. >>> k.vypis()          # zavolané po troch kliknutých bodoch
8.     (162, 129)
9.     (231, 51)
10.    (273, 199)
```

7. V triede `Klikanie` zo (6) úlohy uprav metódu `klik` tak, aby okrem kreslenia malej kružnice spojila úsečkou posledne nakreslený bod s predchádzajúcim bodom (zrejme okrem prvého kliknutia). Nepoužívaj globálne premenné.

8. Zadefinuj triedu `Tahanie` s tromi metódami `__init__`, `klik` a `tahanie`:
   - metódy `__init__` a `klik` budú veľmi podobné metódam z (5) úlohy: `__init__` vytvorí grafickú plochu a zviaže udalosť kliknutia s metódou `self.klik` a udalosť ťahania (`'<B1-Motion>'`) s metódou `self.tahanie`
   - metóda `klik` nakreslí malú kružnicu
   - metóda `tahanie` umožní pri ťahaní myšou kresliť čiaru (posledný kliknutý/ťahaný bod spojí úsečkou s novým bodom)

9. Zadefinuj triedu `VyrobPolygon`, ktorá bude fungovať takto:
   - metóda `__init__(meno_suboru)` si zapamätá meno súboru a vytvorí prázdny súbor s týmto menom (súbor zatvorí), vytvorí grafickú plochu (`self.canvas`) a v nej jeden polygón s jediným bodom (0, 0), s čiernym obrysom a bielym vnútrom; zároveň zviaže (`bind`) dve metódy `self.klik` a `self.enter` na udalosti kliknutia a stlačenie klávesu Enter (udalosť `'<Return>'` zviaž pomocou `bind_all`); okrem toho vytvorí atribút `zoznam` s prázdnym obsahom
   - metóda `klik(event)` pridá do zoznamu `self.zoznam` kliknuté súradnice (nie ako dvojicu, ale dve celé čísla za sebou) a pomocou `self.canvas.coords(...)` zmení vykresľovaný polygón na obsah tohto zoznamu
   - metóda `enter(event)` zapíše momentálny obsah zoznamu (súradnice polygónu) na koniec súboru do jedného riadka ako postupnosť celých čísel oddelených medzerou; atribút zoznam potom vyprázdni a ďalšie klikania potom už vytvárajú nový polygón (opätovný Enter zapíše nový polygón ako ďalší riadok súboru)
   - keď bude táto trieda hotová, program naštartuj pomocou:

```
10. VyrobPolygon('poly.txt')    # týmto sa zavolá konštruktor __init__()
```

- naklikaj niekoľko polygónov a skontroluj obsah súboru

10. Zadefinuj triedu `CitajPolygon`, ktorá vykreslí polygóny uložené v súbore z úlohy (9). Zapíš tieto dve metódy:
    - metóda `__init__(meno_suboru)` vytvorí grafickú plochu, prečíta súradnice z daného súboru a vykreslí všetky polygóny s čiernym obrysom a bielym vnútrom; zároveň zviaže metódu `self.prefarbi` s udalosťou kliknutia
    - metóda `prefarbi` zmení vnútro (`fill`) všetkých nakreslených polygónov na náhodné farby
    - keď bude táto trieda hotová, program naštartuj pomocou:

```
11. CitajPolygon('poly.txt')    # týmto sa zavolá konštruktor __init__()
```

11. Zadefinuj triedu `MojaGrafika` s týmito metódami:
    - metóda `__init__()` vytvorí grafickú plochu veľkosti `400x300` (atribút `self.canvas`)
    - metóda `kruh(r, x, y, farba=None)` nakreslí kruh s polomerom `r` so stredom `(x, y)` s danou výplňou (`None` označuje náhodnú farbu)
    - metóda `stvorec(a, x, y, farba=None)` nakreslí štvorec so stranou `a` so stredom `(x, y)` s danou výplňou (`None` označuje náhodnú farbu)
    - metóda `text(text, x, y, farba=None)` vypíše daný text na súradnice `(x, y)` s danou farbou (`None` označuje náhodnú farbu)
    - metóda `zapis(meno_suboru)` zapíše všetky nakreslené útvary do textového súboru: každý do samostatného riadka v tvare,
      napríklad `kruh 40 100 150 red` alebo `text Python 100 50 #12ff3a`, …
    - metóda `citaj(meno_suboru)` zruší všetky nakreslené objekty (`self.canvas.delete('all')`), prečíta súbor a nakreslí všetky v ňom zapísané útvary

    Napríklad:

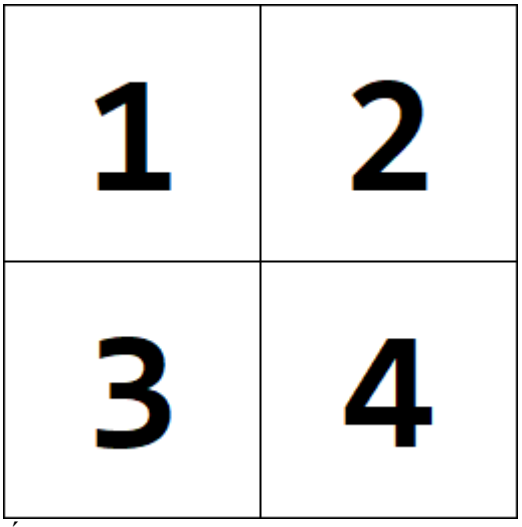
```
g = MojaGrafika()
g.stvorec(280, 200, 150, 'yellow')
for x in range(20, 400, 40):
    g.kruh(20, x, 100)                # náhodné farby
g.text('Python', 200, 150, 'red')
g.zapis('grafika.txt')                # vytvorí súbor

g = MojaGrafika()
g.citaj('grafika.txt')                # znovu ho prečíta a vykreslí
```

8. Týždenný projekt

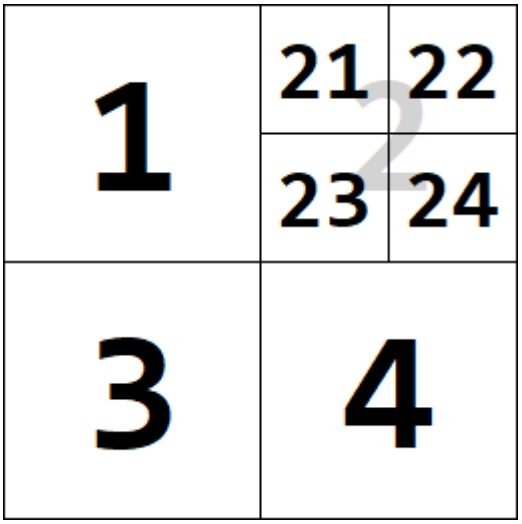
L.I.S.T.

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

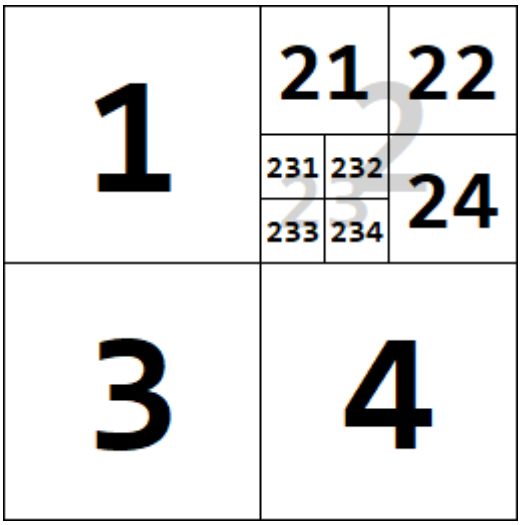
Veľký štvorec môžeme rozdeliť na 4 kvadranty takto:



Úplne rovnako môžeme každý kvadrant rozdeliť na menšie kvadranty, napríklad 2. kvadrant takto:



Môžeš vidieť, že tieto menšie kvadranty v `2` sú očíslované ako `21`, `22`, `23`, `24`. Ľubovoľný kvadrant môžeš rozdeliť na 4 menšie. Napríklad, rozdelením kvadrantu `23` dostaneš:



Takýmto delením môžeš ísť do nejakej danej hĺbky `n`, kým neprídeš na elementárny štvorček, ktorý sa už deliť nedá. Napríklad, hĺbka `n=1` znamená, že plocha sa skladá z 2x2 elementárnych štvorčekov a deliť ju môžeš maximálne raz na 4 základné kvadranty. Hĺbka `n=4` označuje 16x16 elementárnych štvorčekov, ktorú môžeš deliť maximálne 4-krát. Potom, napríklad kvadrant `2` je veľký 8x8 elementárnych štvorčekov, kvadrant `23` zaberá 4x4 štvorčekov, kvadrant `234` zaberá 2x2 štvorčeky a zrejme `2343` už len jeden. Teda pre dané `n` poradové číslo kvadrantu má zmysel s maximálnym počtom cifier `n`. Čím má číselné označenie kvadrantu menej cifier, tým je kvadrant väčší (napríklad 0-ciferné číslo kvadrantu označuje celý štvorec).

Napíš pythonovský modul, ktorý bude obsahovať jedinú triedu `Stvorce` a žiadne iné globálne premenné:

```
class Stvorce:
    def __init__(self, n):
        ...

    def urob(self, index):
        ...

    def pocet(self):
        ...

    def vypis(self):
        ...
```

Metódy majú fungovať takto:

- inicializácia `__init__(self, n)` vyhradí takú dvojrozmernú tabuľku, aby sa dali deliť kvadranty do hĺbky `n`; každý elementárny štvorček môže obsahovať `0` alebo `1`, pri inicializácii budú všade `0`
- metóda `urob(self, index)` dostáva číslo kvadrantu ako znakový reťazec (môže byť aj prázdny) a pre zadaný kvadrant vyznačí všetky elementárne štvorčeky tak, že hodnoty `0` nahradí `1` a hodnoty `1` nahradí `0`
- metóda `pocet(self)` vráti dvojicu celých čísel: počet núl a počet jednotiek v dvojrozmernej tabuľke
- metóda `vypis(self)` vypíše (pomocou `print()`) momentálny obsah dvojrozmernej tabuľky, pričom namiesto `0` použije znak `'-'` a namiesto `1` znak `'X'`

Napríklad:

```
>>> stv = Stvorce(2)
>>> stv.pocet()
    (16, 0)
>>> stv.urob('23')
>>> stv.vypis()
    ----
    --X-
    ----
    ----
>>> stv.pocet()
    (15, 1)
>>> stv.urob('2')
>>> stv.urob('3')
>>> stv.pocet()
    (9, 7)
>>> stv.vypis()
    --XX
    ---X
    XX--
    XX--
```

Pre inšpiráciu môžeš otestovať tieto postupnosti indexov:

```
['1', '11', '131', '1314', '143', '12', '1233', '1234', '1243', '1244', '23', '234',
'2133', '2134', '2143', '2144', '24', '242', '2423', '31', '313', '3132', '32', '324',
'3232', '3411', '3412', '3421', '3422', '4', '44', '424', '4241', '413', '4141', '43',
'4311', '4312', '4321', '4322']

----------------
----------------
----------------
----XXXXXXXX----
--XXXXXXXXXX--
-XXXXXXXXXXXX-
XXXX--XXXX--XXXX
XXXX--XXXX--XXXX
XXXXXXXXXXXXXXXX
```

```
XXXXXXXXXXXXXXX
-XXXX------XXXX-
--XXXX----XXXX--
----XXXXXXXX----
---------------
---------------
---------------
```

```
['14', '141','1412','1431','124', '1241', '2', '21', '22', '24', '213', '2113',
'2143', '2324', '2433', '3', '31', '3142', '3211', '343', '3441', '3443', '33',
'332', '3321', '41', '4213', '4231', '4232', '43', '434', '441', '4423']
```

```
---------------
--------X-------
-------XXX------
------XXXXX-----
-----XXXXXXX----
------XXXXX-----
-----XXXXXXX----
----XXXXXXXXX---
-----XXXXXXX----
----XXXXXXXXX---
---XXXXXXXXXXX--
----XXXXXXXX----
---XXXXXXXXXXX--
--XXXXXXXXXXXXX-
-------XXX------
-------XXX------
```

```
['2', '23224', '2411', '24123', '24131', '24132', '2422', '24223', '2444', '24441',
'212', '21412', '21421', '21422', '211', '2113', '21132', '21143', '22', '2233',
'2234', '22342', '14', '14111', '14113', '1433', '14332', '132', '1322', '13223',
'134', '1344', '13441', '1312', '13142', '13144', '13322', '13324', '1334', '1134',
'11341', '11333', '11334', '11433', '1224', '12243', '12344', '124', '1241', '12421',
'12423', '12431', '3112', '31123', '31111', '31112', '31211', '32122', '322', '3223',
'32213', '3224', '32242', '411', '412', '4124', '41241', '41234', '4211', '42121',
'42122', '42211', '4131', '413114', '41321']
```

```
--------------------------------
--------------------------------
--------------XXX---------------
---------------XXXX-------------
---------------XXXXXX-----------
---------------XXXXXXXXX--------
---X---------XXXXXXXXXXXXXX-----
XXXXX------XXXXXXXXXXXXXXXXX----
--XXXX---XXXXXXXXXXXXXXX--XXXX--
--XXXXX--XXXXXXXXXXXXXX----XXXX-
---XXXXXXXXXXXXXXXXXXXXX--XXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XXXXX--XXXXXXXXXXXXXXXXXXXXXX-
--XXXX----XXXXXXXXXXXXXXXXXXX--
XXXXX------XXXXXXXXXXXXXXXXX---
---X---------XXXXXXXXXXXXX------
---------------XXXXXXXX---------
----------------XXXXX-----------
----------------XXX-------------
----------------XX--------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
```

```
--------------------------------
--------------------------------
--------------------------------
--------------------------------
```

Tvoj odovzdaný program s menom `riesenie.py` musí začínať tromi riadkami komentárov:

```
# 8. zadanie: stvorce
# autor: Janko Hraško
# datum: 3.12.2021
```

Projekt `riesenie.py` odovzdaj na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **3. decembra**. Môžeš zaň získať **5 bodov**.