16. Triedy a dedičnosť

video prezentácia

triedy a dedičnosť

Čo už vieme o triedach a ich inštanciách:

- triedy sú kontajnery atribútov:
  - atribúty sú väčšinou funkcie, hovoríme im metódy
  - niekedy sú to premenné, hovoríme im triedne atribúty
  - niektoré metódy sú „magické“: začínajú aj končia dvomi znakmi ‚__‘ a každá z nich má pre Python svoje špeciálne využitie
- triedy sú vzormi na vytvárame inštancií (niečo ako formičky na vyrábanie nejakých výrobkov)
- aj inštancie sú kontajnery atribútov:
  - väčšinou sú to súkromné premenné inštancií
- ak nejaký atribút nie je v inštancii definovaný, tak Python zabezpečí, že sa použije atribút z triedy (inštancia automaticky „vidí“ triedne atribúty) - samozrejme, len ak tam existuje, inak sa o tom vyhlási chyba

Objektové programovanie

je v programovacom jazyku charakterizované týmito tromi vlastnosťami:

1. Zapuzdrenie

Zapuzdrenie (enkapsulácia, encapsulation) označuje:

- v objekte sa nachádzajú premenné aj metódy, ktoré s týmito premennými pracujú (hovoríme, že údaje a funkcie sú zapuzdrené v jednom celku)
- vďaka metódam môžeme premenné v objekte ukryť tak, že zvonku sa s údajmi bude pracovať len pomocou týchto metód

2. Dedičnosť

Dedičnosť (inheritance) označuje, že

- novú triedu nevytvárame z nuly, ale využijeme už existujúcu triedu
- tejto vlastnosti sa budeme venovať v tejto prednáške

3. Polymorfizmus

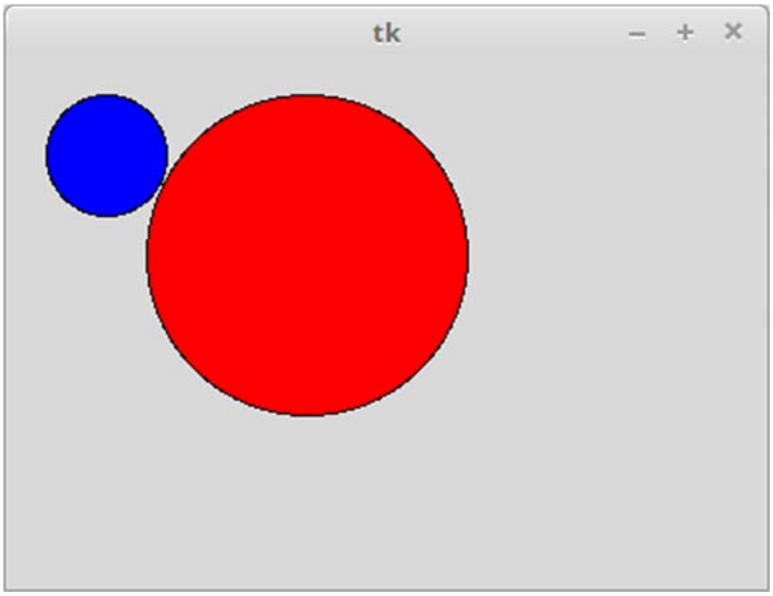
Tejto vlastnosti objektového programovania sa budeme venovať až v ďalších prednáškach.

Zapuzdrenie

Pripomeňme si triedu `Kruh`: v atribútoch `x`, `y`, `r` a `farba` sú hodnoty, ktoré presne zodpovedajú vykreslenému objektu:

```
import tkinter

class Kruh:
    canvas = None

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

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

    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)
```

Dostávame:



Ak by sme teraz s atribútmi inštancií pracovali priamo mimo metód, mohli by sme byť prekvapení, ako to nefunguje. Napríklad, zmeníme atribúty `r` a `farba`:

```
>>> k1.r = 100
>>> k2.farba = 'green'
>>> print(k1)
    Kruh(50, 50, 100, 'blue')
```

```
>>> print(k2)
    Kruh(150, 100, 80, 'green')
```

Niečo iné je v atribútoch a niečo iné vidíme v canvase (obrázok sa vôbec nezmenil). Aby sme korektne menili atribúty aj mimo metód, programátorom sa **odporúča** zadefinovať sadu metód, ktoré pri zmene hodnoty zároveň urobia aj ďalšie akcie. Tu by sa hodilo, aby sme ku každému atribútu (ktorý dovolíme zmeniť) vytvorili zodpovedajúcu metódu na zmenu. Zapíšme:

```
class Kruh:
    canvas = None

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

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

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

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

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

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

a môžeme to použiť, napríklad takto:

```
>>> k1.zmen_x(80)
>>> k1.zmen_r(100)
>>> k2.zmen_farbu('green')
>>> k2.zmen_y(130)
```

Takéto riešenie je už na dobrej ceste, ale ešte stále sa programátor môže pomýliť a napíše:

```
>>> k2.r = 50
```

To znamená, že zmení nejaký atribút priamo bez našej metódy. Tu sa v takýchto prípadoch **odporúča** označiť takéto kritické atribúty ako **nie verejné**, tzv. **súkromné** tak, že budú začínať jedným znakom podčiarkovník. Programátor dáva takto najavo, že tieto atribúty by sa **nemali používať** mimo metód, lebo by to mohlo mať zlé dôsledky. Preto okrem metód, ktoré korektne menia atribúty (budeme im hovoriť **setter**), napríklad `zmen_r` alebo `zmen_farbu`, zadefinujeme metódy, ktoré vrátia hodnoty týchto atribútov (aby sme nepoužívali identifikátory začínajúce znakom podčiarkovník `_`). Takýmto metódam budeme hovoriť **getter**.

Prepíšme triedu `Kruh` tak, aby v nej boli pre všetky atribúty definované metódy **setter** aj **getter**:

```
class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self._x, self._y, self._r = x, y, 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=farba)

    def __str__(self):
        return f'Kruh({self._x}, {self._y}, {self._r}, {self._farba!r})'

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)

    def zmen_r(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 zmen_farbu(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

    def zmen_x(self, x):
        self._x = x
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)

    def zmen_y(self, y):
        self._y = y
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)

    def daj_r(self):
        return self._r

    def daj_farbu(self):
        return self._farba

    def daj_x(self):
        return self._x

    def daj_y(self):
        return self._y
```

Pozrite, ako môžeme teraz korektne pracovať s atribútmi:

```
>>> k1.zmen_x(k1.daj_x() + 10)
>>> k2.zmen_r(k2.daj_r() - 10)
>>> k2.r
    ...
    AttributeError: 'Kruh' object has no attribute 'r'
```

Keďže takýto zápis robí naše programy výrazne horšie čitateľné (oproti zápisu `k1.x = k1.x + 10`), Python ponúka špeciálne využitie **getter**-ov a **setter**-ov. V triede môžeme okrem definícií takýchto metód zadefinovať aj špeciálny atribút, tzv. **property**:

```
class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self._x, self._y, self._r = x, y, 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=farba)

    def __str__(self):
        return f'Kruh({self._x}, {self._y}, {self._r}, {self._farba!r})'

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)

    def zmen_r(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 zmen_farbu(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

    def zmen_x(self, x):
        self._x = x
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)

    def zmen_y(self, y):
        self._y = y
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)

    def daj_r(self):
        return self._r

    def daj_farbu(self):
        return self._farba

    def daj_x(self):
        return self._x

    def daj_y(self):
        return self._y
```

```
    x = property(daj_x, zmen_x)
    y = property(daj_y, zmen_y)
    r = property(daj_r, zmen_r)
    farba = property(daj_farbu, zmen_farbu)
```

Pribudli nám tu štyri priradenia. Už vieme, že priradenia pri definovaní triedy, vytvárajú **triedne atribúty**. V našom prípade sa vytvoril, napríklad triedny atribút `x`, ktorého hodnotou je špeciálna dvojica **getter** `daj_x` a **setter** `zmen_x`. Odteraz Python vie, že pri práci s inštanciou nebude meniť, resp. získavať atribút `x` bežným spôsobom, ale volaním zodpovedajúcich metód. Python teraz vie, že to nie sú skutočné atribúty premenné, ale sú to len zamaskované volania metód **getter** a **setter**. Teraz môžeme zapísať aj toto:

```
>>> k1.x = k1.x + 150
>>> k2.farba = 'indian' + k2.farba
>>> k2.r /= 2
```

Toto isté môžete v Pythone zapísať aj takto:

```
class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self._x, self._y, self._r = x, y, 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=farba)

    def __str__(self):
        return f'Kruh({self._x}, {self._y}, {self._r}, {self._farba!r})'

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)

    @property
    def r(self):
        return self._r

    @r.setter
    def r(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)

    @property
    def farba(self):
        return self._farba

    @farba.setter
    def farba(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

    @property
    def x(self):
        return self._x
```

```
    @x.setter
    def x(self, x):
        self._x = x
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, y):
        self._y = y
        self.canvas.coords(self._id,
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r)
```

V tomto zápise obe metódy **getter** aj **setter** majú rovnaké mená a zhodujú sa s menom **property** atribútu. Pred každú definíciu triedy sme teraz museli pridať tzv. **dekorátor** buď `@property` pre definovanie **getter** alebo `@r.setter` pre definovanie **setter**. Je na vás, ktorú z týchto verzií zápisu použijete. Pre začiatočníkov sa odporúča prvá verzia (s priradeniami typu `r = property(daj_r, zmen_r)`), prípadne **property** zatiaľ vo svojich triedach nedefinujte.

Dedičnosť

Začneme definíciou jednoduchej triedy:

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

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

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

bod = Bod(100, 50)
bod.posun(-10, 40)
print('bod =', bod)
```

Toto by nemalo byť pre nás nič nové. Tiež sme sa už stretli s tým, že:

```
>>> dir(Bod)
    ['__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__', 'posun']
```

V tomto výpise všetkých atribútov triedy `Bod` vidíme nielen nami definované tri metódy: `__init__`, `__str__` a `posun`, ale aj veľké množstvo neznámych identifikátorov, o ktorých asi netušíme odkiaľ sa tu nabrali a na čo slúžia.

V Pythone, keď vytvárame novú triedu, tak sa táto „nenarodí“ úplne prázdna, ale získava niektoré dôležité atribúty od základnej Pythonovskej triedy `object`. Keď pri definovaní triedy zapíšeme:

```
class Bod:
    ...
```

v skutočnosti to znamená:

```
class Bod(object):
    ...
```

Do okrúhlych zátvoriek píšeme triedu (v tomto prípade triedu `object`), z ktorej sa vytvára naša nová trieda `Bod`. Vďaka tomuto naša nová trieda už pri „narodení“ pozná základnú množinu atribútov a my našimi definíciami metód tieto atribúty buď prepisujeme alebo pridávame nové. Tomuto mechanizmu sa hovorí **dedičnosť** a znamená to, že z existujúcej triedy vytvárame nejakú novú:

- triede, z ktorej vytvárame nejakú novú, sa hovorí **základná trieda**, alebo **bázová trieda**, alebo **super trieda** (base class, super class)
- triede, ktorá vznikne dedením z inej triedy, hovoríme **odvodená trieda**, alebo **podtrieda** (derived class, subclass)

Niekedy sa vzťahu základná trieda a odvodená trieda hovorí aj terminológiou **rodič** a **potomok** (potomok zdedil nejaké vlastnosti od svojho rodiča).

Odvodená trieda

Vytvorme nový typ (triedu) z triedy, ktorú sme definovali my, napríklad z triedy `Bod` vytvoríme novú triedu `FarebnyBod`:

```
class FarebnyBod(Bod):
    def zmen_farbu(self, farba):
        self.farba = farba
```

Vďaka takémuto zápisu trieda `FarebnyBod` získava už pri narodení metódy `__init__`, `__str__` a `posun`, pritom metódu `zmen_farbu` sme jej dodefinovali teraz. Teda môžeme využívať všetko z definície triedy, z ktorej sme **odvodili** novú triedu (t.j. všetky atribúty, ktoré sme **zdedili**). Môžeme teda zapísať:

```
fbod = FarebnyBod(200, 50)          # volá __init__ z triedy Bod
fbod.zmen_farbu('red')              # volá zmen_farbu z triedy FarebnyBod
fbod.posun(dy=50)                   # volá posun z triedy Bod
print('fbod =', fbod)               # volá __str__ z triedy Bod
```

Zdedené metódy môžeme v novej triede nielen využívať, ale aj predefinovať - napríklad môžeme zmeniť inicializáciu `__init__`:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        self.x = x
        self.y = y
        self.farba = farba

    def zmen_farbu(self, farba):
        self.farba = farba

fbod = FarebnyBod(200, 50, 'green')
fbod.posun(dy=50)
print('fbod =', fbod)
```

Pôvodná verzia inicializačnej metódy `__init__` z triedy `Bod` sa teraz prekryla novou verziou tejto metódy, ktorá má teraz už tri parametre. Ak by sme v metóde `__init__` chceli využiť pôvodnú verziu tejto metódy zo základnej triedy `Bod`, môžeme ju z tejto metódy zavolať, ale **nesmieme** to urobiť takto:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        self.__init__(x, y)                  # !!! chybné volanie !!!
        self.farba = farba
    ...
```

Toto je totiž **rekurzívne volanie**, ktoré spôsobí spadnutie programu `RecursionError: maximum recursion depth exceeded`. Musíme to zapísať takto:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        Bod.__init__(self, x, y)             # inicializácia zo základnej triedy
        self.farba = farba
    ...
```

T.j. pri inicializácii inštancie triedy `FarebnyBod` najprv použi inicializáciu ako keby to bola inicializácia základnej triedy `Bod` (inicializuje atribúty `x` a `y`) a potom ešte inicializuj niečo navyše - t.j. atribút `farba`. Dá sa to zapísať ešte univerzálnejšie:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        super().__init__(x, y)
        self.farba = farba
    ...
```

Štandardná funkcia `super()` na tomto mieste označuje: urob tu presne to, čo by na tomto mieste urobil môj rodič (t.j. moja super trieda). Tento zápis uvidíme aj v ďalších ukážkach.

Grafické objekty

Trochu sme upravili grafické objekty `Kruh`, `Obdlznik` a `Skupina` zo začiatku prednášky (bez **property**) aj z prednášky: 15. Triedy a metódy:

```
import tkinter

class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self._x, self._y, self._r = x, y, 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=farba)

    def __str__(self):
        return f'Kruh({self._x}, {self._y}, {self._r}, {self._farba!r})'

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)
```

```
    def zmen_r(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 zmen_farbu(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

class Obdlznik:
    canvas = None

    def __init__(self, x, y, sirka, vyska, farba='red'):
        self._x, self._y, self._sirka, self._vyska = x, y, sirka, vyska
        self._farba = farba
        self._id = self.canvas.create_rectangle(
            self._x, self._y,
            self._x + self._sirka, self._y + self._vyska,
            fill=farba)

    def __str__(self):
        return f'Obdlznik({self._x}, {self._y}, {self._sirka}, {self._vyska}, {self._farba!r})'

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)

    def zmen_velkost(self, sirka, vyska):
        self._sirka, self._vyska = sirka, vyska
        self.canvas.coords(self._id,
            self._x, self._y,
            self._x + self._sirka, self._y + self._vyska)

    def zmen_farbu(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

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

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

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

    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self._zoznam:
            if type(utvar) == typ:
                utvar.posun(dx, dy)

    def zmen_farbu(self, farba):
        for utvar in self._zoznam:
            utvar.zmen_farbu(farba)

    def zmen_farbu_typ(self, typ, farba):
        for utvar in self._zoznam:
            if type(utvar) == typ:
                utvar.zmen_farbu(farba)

#---------------------------------------
```

```
c = Kruh.canvas = Obdlznik.canvas = tkinter.Canvas(bg='white')
c.pack()

k = Kruh(50, 50, 30, 'blue')
r = Obdlznik(100, 20, 100, 50)
k.zmen_farbu('green')
r.posun(50)
```

Všimnite si:

- zrušili sme triedny atribút `typ`, pomocou ktorého metódy `posun_typ` a `zmen_farbu_typ` vedeli pracovať len s objektmi daného typu - namiesto toho teraz testujeme samotnú inštanciu pomocou funkcie `type` - tá nám vráti buď typ `Kruh` alebo `Obdlznik`
- obe triedy `Kruh` aj `Obdlznik` majú niektoré atribúty aj metódy úplne rovnaké (napríklad `x`, `y`, `farba`, `posun`, `zmen_farbu`)
- ak by sme chceli využiť dedičnosť (jedna trieda zdedí nejaké atribúty a metódy od inej), nie je rozumné, aby `Kruh` niečo dedil z triedy `Obdlznik`, alebo naopak `Obdlznik` bol odvodený z triedy `Kruh`

Zadefinujeme preto novú triedu `Utvar`, ktorá bude predkom (rodičom, bude základnou triedou) oboch tried `Kruh` aj `Obdlznik` - táto trieda bude obsahovať všetky spoločné atribúty týchto tried, t.j. aj niektoré metódy:

```
class Utvar:
    canvas = None

    def __init__(self, x, y, farba='red'):
        self._x, self._y, self._farba = x, y, farba
        self._id = None

    def posun(self, dx=0, dy=0):
        self._x += dx
        self._y += dy
        self.canvas.move(self._id, dx, dy)

    def zmen_farbu(self, farba):
        self._farba = farba
        self.canvas.itemconfig(self._id, fill=farba)

Utvar.canvas = tkinter.Canvas(width=400, height=400)
Utvar.canvas.pack()
```

Uvedomte si, že nemá zmysel vytvárať objekty tejto triedy, lebo okrem inicializácie zvyšné metódy nebudú fungovať. Teraz dopíšme triedy `Kruh` a `Obdlznik`:

```
class Kruh(Utvar):
    def __init__(self, x, y, r, farba='red'):
        super().__init__(x, y, farba)
        self._r = r
        self._id = self.canvas.create_oval(
            self._x - self._r, self._y - self._r,
            self._x + self._r, self._y + self._r,
            fill=farba)

    def zmen_r(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)

class Obdlznik(Utvar):
    def __init__(self, x, y, sirka, vyska, farba='red'):
```

```
        super().__init__(x, y, farba)
        self._sirka, self._vyska = sirka, vyska
        self._id = self.canvas.create_rectangle(
            self._x, self._y,
            self._x + self._sirka, self._y + self._vyska,
            fill=farba)

    def zmen_velkost(self, sirka, vyska):
        self._sirka, self._vyska = sirka, vyska
        self.canvas.coords(self._id,
            self._x, self._y,
            self._x + self._sirka, self._y + self._vyska)
```

Zrušili sme atribút `canvas`, ktorý sa nachádzal v každej z tried `Kruh` a `Obdlznik`. Teraz sa tento atribút nachádza iba v základnej triede `Utvar` a obe odvodené triedy ho vidia.

Testovanie typu inštancie

Pomocou štandardnej funkcie `type` vieme otestovať, či je inštancia konkrétneho typu, napríklad

```
>>> t1 = Kruh(10, 20, 30)
>>> t2 = Obdlznik(40, 50, 60, 70)
>>> type(t1) == Kruh
    True
>>> type(t2) == Kruh
    False
>>> type(t2) == Obdlznik
    True
>>> type(t1) == Utvar
    False
```

Okrem tohto testu môžeme použiť štandardnú funkciu `isinstance(i, t)`, ktorá zistí, či je inštancia `i` typu `t` alebo je typom niektorého jeho predka, preto budeme radšej písať:

```
>>> t1 = Kruh(10, 20, 30)
>>> t2 = Obdlznik(40, 50, 60, 70)
>>> isinstance(t1, Kruh)
    True
>>> isinstance(t1, Utvar)
    True
>>> isinstance(t2, Kruh)
    False
>>> isinstance(t2, Utvar)
    True
```

Môžeme teraz prepísať metódy `posun_typ` a `prefarbi_typ` triedy `Skupina` takto:

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

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

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

    def posun_typ(self, typ, dx=0, dy=0):
```

```
        for utvar in self._zoznam:
            if isinstance(utvar, typ):
                utvar.posun(dx, dy)

    def zmen_farbu(self, farba):
        for utvar in self._zoznam:
            utvar.zmen_farbu(farba)

    def zmen_farbu_typ(self, typ, farba):
        for utvar in self._zoznam:
            if isinstance(utvar, typ):
                utvar.zmen_farbu(farba)
```

a použiť takto:

```
import random

sk = Skupina()
for i in range(20):
    if random.randrange(2):
        sk.pridaj(Kruh(random.randint(50, 350), random.randint(50, 350),
                       random.randint(10, 25)))
    else:
        sk.pridaj(Obdlznik(random.randint(50, 350), random.randint(50, 350),
                           random.randint(10, 50), random.randint(10, 50)))

sk.zmen_farbu_typ(Kruh, 'yellow')
sk.posun_typ(Obdlznik, -10, -25)
```

- volanie `zmen_farbu_typ` zmení farbu všetkých kruhov v skupine na žltú
- volanie `posun_typ` posunie len všetky obdĺžniky

Odvodená trieda od Turtle

Aj od triedy `Turtle` z prednášky: 11. Korytnačky (turtle) môžeme odvádzať nové triedy, napríklad:

```
import turtle

class MojaTurtle(turtle.Turtle):
    pass

t = MojaTurtle()
for i in range(5):
    t.fd(100)
    t.lt(144)
```

Zadefinovali sme novú triedu `MojaTurtle`, ktorá je odvodená od triedy `Turtle` (z modulu `turtle`, preto musíme písať `turtle.Turtle`) a keďže sme do nej nedodefinovali nič nové, táto naša trieda dokáže presne to isté ako `turtle.Turtle`. Aj inštancia odvodená z obyčajnej `turtle.Turtle` dokáže robiť len to, čo táto základná trieda.

Zadefinujme teraz triedu `MojaTurtle` tak, aby jej kolekcia metód bola rozšírená o metódu `stvorec`:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def stvorec(self, velkost):
        for i in range(4):
            self.fd(velkost)
```

```
            self.rt(90)

t = MojaTurtle()
t.stvorec(100)
t.lt(30)
t.stvorec(200)
```

Samozrejme, že túto metódu môžu volať len korytnačky typu `MojaTurtle`, obyčajné korytnačky pri takomto volaní metódy `stvorec` hlásia chybu:

```
>>> t0 = turtle.Turtle()
>>> t0.stvorec(50)
    ...
    AttributeError: 'Turtle' object has no attribute 'stvorec'
```

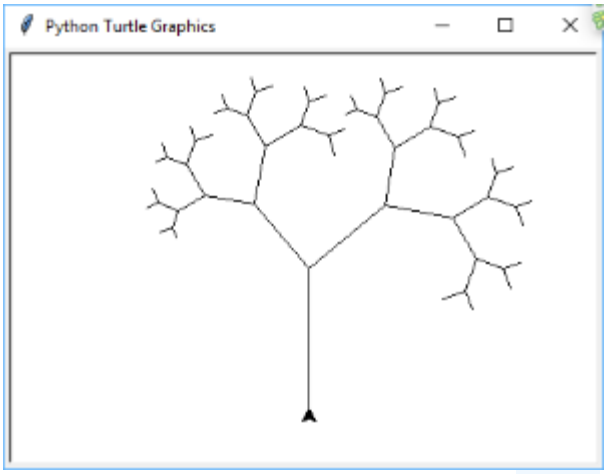
Tu si môžete otestovať, ako rozumiete volaniam metód a skúste to zavolať takto:

```
>>> MojaTurtle.stvorec(t0, 50)
```

Takto to ale používať **nebudeme**!

Môžeme definovať aj zložitejšie metódy, napríklad aj rekurzívny strom:

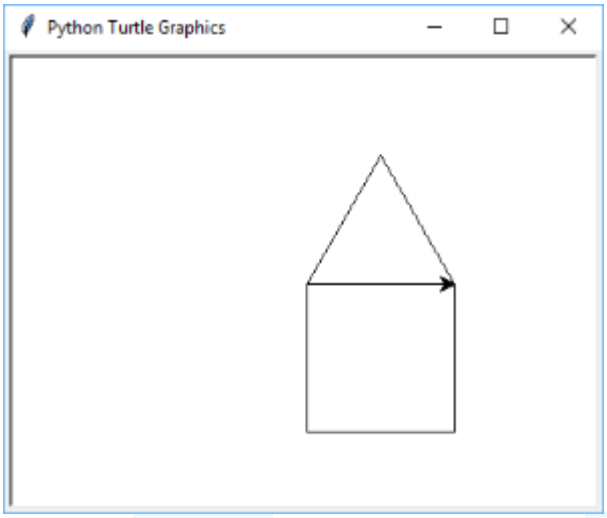
```
import turtle

class MojaTurtle(turtle.Turtle):
    def strom(self, n, d):
        self.fd(d)
        if n > 0:
            self.lt(40)
            self.strom(n-1, d*0.6)
            self.rt(90)
            self.strom(n-1, d*0.7)
            self.lt(50)
        self.bk(d)

t = MojaTurtle()
t.lt(90)
t.strom(5, 100)
```



Niekedy nám môže chýbať to, že trieda `Turtle` neumožňuje vytvoriť korytnačku inde ako v strede plochy. Predefinujme inicializáciu našej novej korytnačky a zároveň sme tu zadefinujme metódu `domcek`, ktorá nakreslí domček zadanej veľkosti:

```
import turtle
```

```
class MojaTurtle(turtle.Turtle):
    def __init__(self, x=0, y=0):
        super().__init__()
        self.speed(0)
        self.pu()
        self.setpos(x, y)
        self.pd()

    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)
            self.rt(uhol)

t = MojaTurtle(-200, 100)
t.domcek(100)
```



Do triedy `MojaTurtle` dodefinujme aj metódu `fd`. Zrejme chceme **prekryť** existujúcu metódu `fd` z pôvodnej triedy `turtle.Turtle` našou vlastnou verziou. Zapíšme metódu `fd`, ktorá namiesto rovnej čiary nakreslí „cikcakovú“ a pritom skončí v rovnakom bode, ako pôvodná čiara tejto dĺžky:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def __init__(self, x=0, y=0):
        super().__init__()
        self.speed(0)
        self.pu()
        self.setpos(x, y)
        self.pd()

    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)
            self.rt(uhol)

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

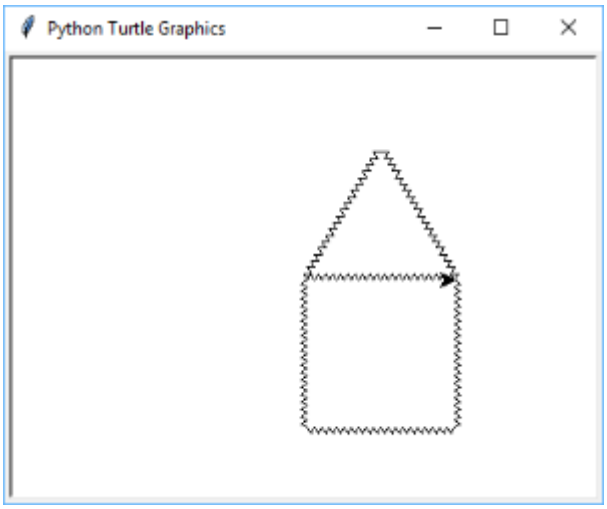
Otestujeme:

```
>>> t = MojaTurtle()
>>> t.fd(300)
```

Korytnačka naozaj prešla vzdialenosť `300`, ale zanechala pritom „cikcakovú“ čiaru. Ak otestujeme namiesto jednej čiary nakreslenie domčeka:

```
>>> t = MojaTurtle()
>>> t.domcek(100)
```

Dostávame domček s „cikcakovými“ čiarami:

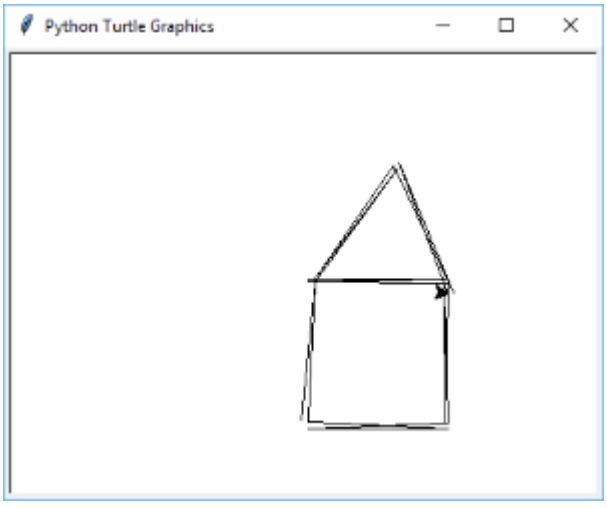


Zmeňme cikcakové čiary na trochu náhodné:

```
import turtle
import random

class MojaTurtle(turtle.Turtle):
    ...

    def fd(self, dlzka):
        super().fd(dlzka)
        self.rt(180 - random.randint(-3, 3))
        super().fd(dlzka)
        self.rt(180 - random.randint(-3, 3))
        super().fd(dlzka)
```

Domček teraz vyzerá takto:

```
MojaTurtle().domcek(100)
```



Metóda `fd` namiesto jednej rovnej čiary danej dĺžky, nakreslí tri čiary tejto dĺžky, pričom sa zakaždým otočí o 180 stupňov plus nejaká malá náhodná odchýlka <-3, 3> stupne. Vďaka tejto odchýlke môže vzniknúť taký efekt, akokeby kresba domčeka vznikla kreslením od ruky

Zapis `MojaTurtle().domcek(100)` označuje, že najprv vytvoríme novú inštanciu `MojaTurtle()`, ale namiesto toho, aby sme ju priradili do nejakej premennej, napríklad `t = MojaTurtle()` a s ňou ďalej pracovali, napríklad `t.domcek(100)`, tak sme priamo bez priradenia zavolali danú metódu. Toto sa zvykne robiť vtedy, keď inštanciu potrebujeme len na jedno zavolanie jej metódy a nepredpokladáme, že budeme potrebovať premennú na prístup k tejto inštancii.

Cvičenia

L.I.S.T.

- riešenia **aspoň 8 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- pozri si Riešenie úloh 16. cvičenia

1. Zisti, ako fungujú tieto triedy. Skús najprv bez počítača, potom skontroluj na počítači:

```
2. class Zviera:
3.     def __init__(self, meno):
4.         self.meno = meno.capitalize()
5.
6.     def __str__(self):
7.         return f'zviera {self.typ} má meno {self.meno} a robí {self.zvuk}'
8.
9. class Pes(Zviera):
10.    typ = 'pes'
11.    zvuk = 'haf-haf'
12.
13.class Macka(Zviera):
14.    typ = 'mačka'
15.    zvuk = 'mnau-mnau'
16.
17.class Kacka(Zviera):
18.    typ = 'kačka'
19.    zvuk = 'ga-ga'
20.
21.z1 = Pes('dunčo')
22.z2 = Macka('mica')
23.z3 = Pes('bono')
24.z4 = Kacka('gréta')
25.z3.zvuk = 'vrr-vrr'
26.for z in z1, z2, z3, z4:
27.    print(z)
```

2. Atribúty `meno`, `typ` a `zvuk` prerob na **property**:
    - najprv ich všetky premenuj tak, aby začínali znakom podčiarkovník
    - v triede `Zviera` pre všetky z nich zadefinuj zodpovedajúci **getter** v tvare `daj_atribút`
    - atribúty `meno` a `zvuk` budú mať aj svoj **setter**:
        - metóda `zmen_meno` nastaví prvé písmeno mena na veľké a ostatné na malé
        - metóda `zmen_zvuk` najprv zistí, či nový zvuk obsahuje znak `'-'` a ak nie, tak zvuk zreťazí za seba a vloží znak `'-'`,

    Napríklad:

```
>>> z3.zmen_zvuk('vrr')
>>> z3.daj_zvuk()
    'vrr-vrr'
>>> z4.zmen_zvuk('GA-GA')
>>> z4.daj_zvuk()
    'GA-GA'
```

- vyrob `meno`, `typ` a `zvuk` ako **property**, pričom `typ` nebude mať definovaný **setter** (zapíšeš `typ = property(daj_typ)`)

Otestuj, napríklad:

```
>>> z3.zvuk = 'vrr'
>>> z4.meno = 'grETA'
>>> z2.typ = 'cat'
```

3. Zadefinuj triedu `Ucet` s týmito metódami:
   - `__init__(meno, suma)` - meno účtu a počiatočná suma, napríklad `Ucet('mbank', 100)` alebo `Ucet('jbanka')`
   - `__str__()` - reťazec v tvare `'ucet mbank -> 100 euro'` alebo `ucet jbanka -> 0 euro`
   - `stav()` - vráti momentálny stav účtu (vráti sumu na účte)
   - `vklad(suma)` - danú sumu pripočíta k účtu
   - `vyber(suma)` - vyberie sumu z účtu (len ak je to kladné číslo), ak je na účte menej ako požadovaná suma, vyberie len toľko koľko sa dá, metóda vráti (`return`) vybranú sumu

   Otestuj, napríklad takto:

```
mbank = Ucet('mbank')
csob = Ucet('csob', 100)
tatra = Ucet('tatra', 17)
sporo = Ucet('sporo', 50)
mbank.vklad(sporo.vyber(30) + tatra.vyber(30))
csob.vyber(-5)
spolu = 0
for ucet in mbank, csob, tatra, sporo:
    print(ucet)
    spolu += ucet.stav()
print('spolu = ', spolu)
```

   vypíše:

```
ucet mbank -> 47 euro
ucet csob -> 100 euro
ucet tatra -> 0 euro
ucet sporo -> 20 euro
spolu =  167
```

4. Zadefinuj triedu `UcetHeslo`, ktorá je **odvodená** z triedy `Ucet` a má takto zmenené správanie:
   - `__init__(meno, heslo, suma)` - k účtu si zapamätá aj heslo
   - `vklad(suma)` - si najprv vypýta heslo a až keď je správne, zrealizuje vklad
   - `vyber(suma)` - si najprv vypýta heslo a až keď je správne, zrealizuje výber, inak vráti `None`
   - pri definovaní týchto metód využite volania ich pôvodných verzií z triedy `Ucet`

   Otestujte napríklad:

```
mbank = UcetHeslo('mbank', 'gigi')
csob = Ucet('csob', 100)
tatra = UcetHeslo('tatra', 'gogo', 17)
```

```
sporo = Ucet('sporo', 50)
mbank.vklad(sporo.vyber(30) + tatra.vyber(30))
csob.vyber(-5)
spolu = 0
for ucet in mbank, csob, tatra, sporo:
    print(ucet)
    spolu += ucet.stav()
print('spolu = ', spolu)
```

Tento program si najprv dvakrát vypýta heslo:

```
zadaj heslo uctu tatra: gogo
zadaj heslo uctu mbank: gigi
```

- a až potom (po správnom zadaní hesiel) vypíše to isté, ako predtým
- zisti, čo sa stane s účtami, keď pre `'mbank'` určíme chybné heslo

5. Z prednášky skopíruj triedu `MojaTurtle`, v ktorej sa definovali dve metódy `__init__(x, y)` a `domcek(dlzka)`. Teraz vytvor 10 inštancií tejto triedy, ktoré budú pravidelne rozostavené na jednej priamke. Potom každá z nich nakreslí svoj domček náhodnej veľkosti z intervalu `<30, 50>`.

6. Vytvor dve odvodené triedy od `MojaTurtle` (z úlohy (5)) `MojaTurtle1` a `MojaTurtle2` (obe budú potomkami `MojaTurtle`). Prvá z nich `MojaTurtle1` bude definovať novú verziu metódy `fd`, ktorá kreslí cikcakovú čiaru a druhá `MojaTurtle2` kreslí `fd` ako tri náhodné čiary vedľa seba (riešenie v prednáške). Teraz pooprav 10 domčekov vedľa seba z úlohy (5) tak, aby každý z nich bol náhodnej inštancie z `(MojaTurtle, MojaTurtle1, MojaTurtle2)`.

7. Zadefinuj triedu `Turtle1` odvodenú od `turtle.Turtle`, v ktorej bude definova metóda `trojuholník`. Metóda nakreslí rovnostranný trojuholník s danou veľkosťou strany. Otestuj, napríklad:

```
t = Turtle1()
for i in range(5):
    t.trojuholnik(150)
    t.lt(72)
```

8. Zadefinuj `Turtle2` odvodenú od `Turtle1` z úlohy (7), v ktorej predefinuješ metódu `trojuholnik`. Táto nová verzia metódy najprv nastaví náhodnú farbu výplne (`self.fillcolor(...)`), naštartuje vypĺňanie (`self.begin_fill()`), zavolá metódu `trojuholník` z rodičovskej triedy (super triedy) a ukončí vypĺňanie (`self.end_fill()`). Otestuj, napríklad:

```
t = Turtle2()
for i in range(5):
    t.trojuholnik(150)
    t.lt(72)
```

Teraz v triede `Turtle1` oprav metódu `trojuholník` tak, aby namiesto otáčania napríklad `self.rt(120)` bola trojica príkazov `self.rt(60)`, `self.fd(10)`, `self.rt(60)` a znovu otestuj:

```
t = Turtle2()
for i in range(5):
    t.trojuholnik(150)
    t.lt(72)
```

9. Naprogramuj triedu `Pero`, pomocou ktorej budeme vedieť kresliť do grafickej plochy. Trieda má tieto metódy:
    - `__init__(x=0, y=0)`, ak ešte nebola vytvorená grafická plocha (`canvas` má hodnotu `None`), vytvorí ju s danou šírkou a výškou, zapamätá si súradnice pera a stav, že pero je spustené dolu (bude kresliť)
    - `pu()` zdvihne pero, odteraz pohyb pera nekreslí
    - `pd()` spustí pero, pohyb bude zanechávať čiaru
    - `setpos(x, y)` presunie pero na novú pozíciu, ak je pero spustené, zanecháva čiernu čiaru hrúbky 1

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

class Pero:
    canvas = None
    sirka, vyska = 400, 300

    def __init__(...):
        ...
    ...
```

Otestuj vytvorením dvoch inštancií pera, ktoré nakreslia napríklad dva štvorce:

```
p1 = Pero(100, 200)
p2 = Pero(200, 150)
...
```

10. Zadefinuj novú triedu `Korytnacka`, ktorá bude odvodená od triedy `Pero` z úlohy (9):
    - metóda `__init__()` vytvorí pero v strede plochy a do nového atribútu `uhol` nastaví 0 (teda otočenie smerom na východ)
    - metódy `lt(uhol)` a `rt(uhol)` zmenšia, resp. zväčšia atribút `uhol` o zadanú hodnotu, uhly sa budú počítať v stupňoch
    - metóda `fd(dlzka)` presunie pero (zavolá metódu `setpos()`) o zadanú dĺžku, ktorá je v momentálnom smere natočenia
        - asi použiješ približne takýto vzorec pre nové `x` a `y`: `x+dlzka*cos(uhol)`, `y+dlzka*sin(uhol)`
        - nezabudni, že `sin()` a `cos()` fungujú v radiánoch, pričom atribút `uhol` pracuje v stupňoch
    - nepoužíj modul `turtle`

Otestuj napríklad takto:

```
class Korytnacka(Pero):
    ...

#---- test -------

t = Korytnacka()
for i in range(1, 200, 2):
    t.fd(i)
    t.lt(89)
```

11. Z triedy `Korytnacka` z (10) úlohy odvoď triedu `Kor1`, do ktorej dopíšeš metódu `strom(n, d)` (napríklad z prednášky). Potom otestuj:

```
t = Kor1()
t.pu()
```

```
14. t.setpos(200, 280)
15. t.pd()
16. t.lt(90)
17. t.strom(5, 100)
```

12. Otestuj, ako v tejto novej triede `Korytnacka` fungujú príklady z prednášky (alebo (6) úlohy z cvičení) s kreslením domčeka rôznym typom čiar:

```
13. class MojaKor(Korytnacka):
14.     def domcek(...):
15.         ...
16.
17.     def fd(...):              # cikcaková čiara alebo náhodná čiara
18.         ...
19.
20. MojaKor().domcek(100)
```

9. Týždenný projekt

L.I.S.T.

- riešenie odovzdaj na úlohový server https://list.fmph.uniba.sk/
- príklad zo skúšky z roku 2018/2019

Robot Karel

Robot Karel sa pohybuje po štvorcovej sieti, v ktorej sa na niektorých políčkach nachádzajú kartičky s nejakými symbolmi. Robot prechádza ponad tieto políčka, pričom na niektorých môže kartičku pod sebou zdvihnúť (vloží si ju do svojho batoha), resp. karičku z batoha vybrať a položiť na políčko pod seba. Robot reaguje na povely `'vlavo'`, `'vpravo'`, `'krok'`, `'zdvihni'`, `'poloz'`:

- robot je natočený v jednom zo štyroch smerov, označovať ich budeme takto: **0** na východ, **1** na juh, **2** na západ, **3** na sever
- príkazy `'vlavo'`, resp. `'vpravo'` otočia robota v danom smere
- príkazom `'krok'` robot prejde v momentálnom smere na susedné políčko, ak je už na okraji siete, z plochy nevypadne, ale v danom smere nevykoná nič
- príkazom `'zdvihni'` zoberie kartičku z políčka pod sebou a vloží ju do batoha; ak na danom políčku nebola žiadna kartička, príkaz nevykoná nič; ak na danom políčku bolo na sebe viac kartičiek, robot zdvihne najvrchnejšiu z nich; kartičky vkladá do batoha na seba v poradí ako ich zdvíhal z plochy (naspodku je prvá, na vrchu je naposledy zdvihnutá)
- príkazom `'poloz'` vyberie najvrchnejšiu kartičku z batoha a vloží ju na políčko pod seba; ak bol batoh prázdny, príkaz neurobí nič; ak na políčku už boli nejaké kartičky pred tým, novú kartičku položí na vrch týchto kartičiek.

Zadanie štvorcovej siete s počiatočným rozložením kartičiek je v textovom súbore. V prvom riadku je dvojica celých čísel, ktorá popisuje veľkosť štvorcovej siete: počet riadkov a počet stĺpcov. Za tým nasleduje informácia o kartičkách v ploche - v každom riadku je symbol na kartičke a dvojica celých čísel, ktoré označujú riadok a stĺpec pozície kartičky (číslujeme od 0). Na jednom políčku sa môže nachádzať aj viac kartičiek.

Naprogramuj triedu `RobotKarel`:

```
class RobotKarel:
    def __init__(self, meno_suboru):
```

```
        ...

    def __str__(self):
        return ''

    def robot(self, riadok, stlpec, smer):
        ...

    def rob(self, prikaz):
        return 0

    def batoh(self):
        return []
```

kde

- `init` prečíta súbor - robot tam zatiaľ nie je
- `__str__` vráti znakovú reprezentáciu plochy: pozíciu robota zapíš (podľa momentálneho natočenia) jedným zo znakov `'>'`, `'v'`, `'<'`, `'^'`, ak je na políčku viac kartičiek, zobrazí sa iba najvrchnejšia z nich, prázdne políčko zobraz znakom `'.'`; ak je robot na políčku s kartičkami, zobrazí sa iba robot
- `robot` položí robota na zadaný riadok a stĺpec s daným otočením (číslo od 0 do 3)
- `rob` dostáva jeden povel, alebo postupnosť za sebou nasledujúcich povelov, pričom povel je jeden z reťazcov `'vlavo'`, `'vpravo'`, `'krok'`, `'zdvihni'`, `'poloz'`, ktorý môže mať na začiatku aj celé číslo, vtedy to označuje počet opakovaní; napríklad `'3 krok'` označuje tri kroky za sebou; robot sa postupne pohybuje v danom smere, pričom zbiera, resp. kladie kartičky; povely, ktoré sa nedajú vykonať, ignoruje; funkcia vráti počet tu vykonaných ale neignorovaných povelov
- metóda `batoh` vráti momentálny zoznam kartičiek so symbolmi v batohu (prvým prvkom je najspodnejšia kartička, posledným je najvrchnejšia)
- odporúčame štvorcovú sieť reprezentovať ako dvojrozmernú tabuľku (zoznam zoznamov), v ktorej každý prvok je buď reťazec (postupnosť znakov) alebo zoznam znakov, políčka bez kartičiek reprezentuj prázdnym reťazcom, resp. zoznamom

Napríklad, pre súbor `'subor1.txt'`:

```
3 4
N 1 3
O 1 2
H 1 1
P 0 1
Y 0 2
T 0 3
```

aa testovanie môžeš využiť tento kód (umiestni ho za definíciu triedy, môže ostať v module, aj keď ho budeš odovzdávať na testovanie):

```
if __name__ == '__main__':
    k = RobotKarel('subor1.txt')
    k.robot(0, 0, 0)
    print(k)
    print(k.rob('krok'))
    print(k.rob('2 zdvihni'))
    k.rob('krok')
    k.rob('vpravo')
    k.rob('krok')
    k.rob('2 zdvihni')
    k.rob('2 krok')
    print(k)
    print('batoh =', k.batoh())
    k.rob('poloz vlavo')
    k.rob('krok 6 vlavo')
    print(k)
```

```
    print('batoh =', k.batoh())
```

vypíše:

```
>PYT
.HON
....
1
1
..YT
.H.N
..v.
batoh = ['P', 'O']
..YT
.H.N
..O<
batoh = ['P']
```

Z úlohového servera L.I.S.T. si stiahni kostru programu `riesenie.py`. Pozri si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, …, ktoré bude používať testovač.

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

```
# 9. zadanie: karel
# autor: Janko Hraško
# datum: 10.12.2021
```

Projekt `riesenie.py` odovzdaj (bez dátových súborov) na úlohový server L.I.S.T. najneskôr do **10. decembra**. Môžeš zaň získať **5 bodov**.