18. Polymorfizmus, typ množina

video prezentácie

- polymorfizmus
- typ množina

Ešte raz zopakujme, aké sú najdôležitejšie vlastnosti objektového programovania:

- **zapuzdrenie** (encapsulation)
- **dedičnosť** (inheritance)
- **polymorfizmus** (polymorphism)

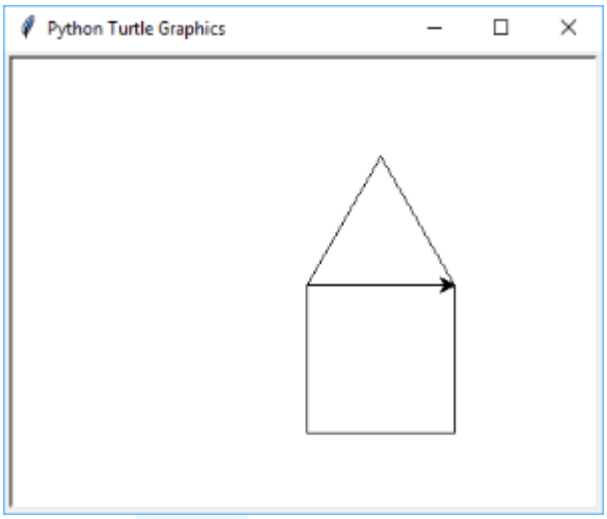
V dnešnej prednáške sa sústredíme na poslednú vlastnosť **polymorfizmus**.

Vráťme sa k príkladu zo 16. prednášky, v ktorom korytnačka kreslila domček:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)       # fd z triedy Turtle
            self.rt(uhol)        # rt z triedy Turtle

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



V metóde `domcek()` sme predpokladali, že pri kreslení domčeka inštancia `t` triedy `MojaTurtle` použije zdedenú metódu `fd()` (z triedy `Turtle`) a tiež zdedenú metódu `rt()` z triedy `Turtle`.

Do triedy `MojaTurtle` sme ešte pridali vlastnú verziu metódy `fd()`, ktorá **prekryla** (override) kreslenie obyčajných čiar na kreslenie cikcakových čiar:

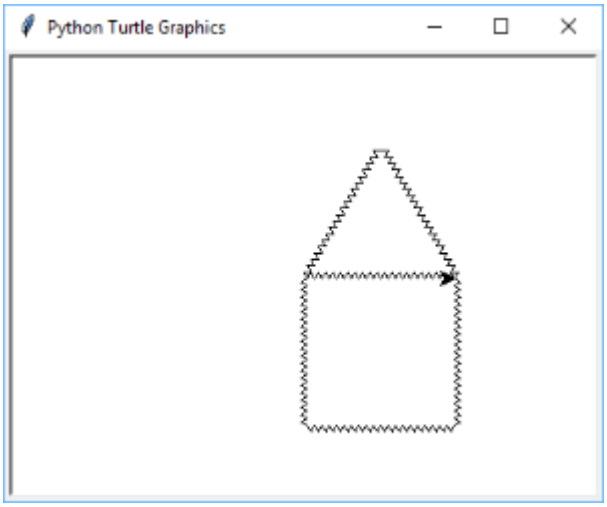
```
import turtle

class MojaTurtle(turtle.Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)       # fd z triedy MojaTurtle
            self.rt(uhol)        # rt z triedy Turtle
```

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

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

Vidíme, že kreslenie domčeka teraz už nepoužíva obyčajné `fd()` z triedy `Turtle`, ale využíva našu zmenenú metódu `fd()` a kreslí cikcakové čiary:



Lenže kreslenie cikcakového domčeka môžeme prepísať do takýchto dvoch definícií tried:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)        # fd z triedy ??? Turtle
            self.rt(uhol)         # rt z triedy Turtle

class MojaTurtle1(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)         # fd z triedy Turtle
            self.rt(120)
            super().fd(5)         # fd z triedy Turtle
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)         # fd z triedy Turtle

t = MojaTurtle1()
t.domcek(100)
```

Z triedy `MojaTurtle` sme **odvodili** novú triedu `MojaTurtle1`. Táto nová trieda teda zdedila od svojej základnej triedy `MojaTurtle` všetko (aj `domcek()`) okrem metódy `fd()`, ktorú **prekryla** (override) svojou vlastnou verziou tejto metódy.

Takže teraz:

- inštancia triedy `MojaTurtle` pomocou metódy `domcek()` nakreslí domček zo 6 obyčajných úsečiek

- inštancia triedy `MojaTurtle1` pomocou metódy `fd()` kreslí cikcakové čiary
- táto inštancia triedy `MojaTurtle1` bude takýmito cikcakovými čiarami kresliť aj domček (volaním metódy `domcek()`)

Ako je to možné? Veď predsa v metóde `domcek()`, keď sme ju definovali, sme jasne zapísali, že kreslenie čiar `fd()` sa bude robiť tak, ako bolo definované v základnej triede `Turtle`. Nikde sme tu nijako nezaznačovali (ani nás to vtedy nenapadlo), že tento `fd()` niekto v budúcnosti možno nahradí svojou vlastnou verziou metódy (napríklad cikcak).

Tak práve tomuto mechanizmu sa hovorí **polymorfizmus** a označuje:

- keď Python vykonáva nejakú metódu (napríklad `domcek()`), tak sa toto vykonávanie **prispôsobí** (adaptuje) tej inštancii, ktorá túto metódu zavolala
- vždy sa použijú aktuálne verzie metód objektu, pre ktorý sa niečo vykonáva
- funguje to aj spätne, teda vo všetkých zdedených metódach: ak sa v nich nachádza volanie niečoho, čo sme práve prekryli svojou novou verziou, tak sa to naozaj uplatní

Uvedomte si, čo by sa stalo, keby tu nefungoval polymorfizmus:

- metóda `domcek()` by vždy kreslila úplne rovnaký domček z rovných čiar bez ohľadu na to, kto túto metódu zavolal (kto bol `self`)
- keby sme potrebovali domček z cikcakových čiar aj pre objekt typu `MojaTurtle1`, museli by sme túto metódu skopírovať aj do tejto triedy, hoci dedičnosť nám hovorí, že by sme to nemali robiť

Pridajme k týmto dvom triedam aj `MojaTurtle2`, pomocou ktorej korytnačka po každej kreslenej čiare prešla trikrát:

```
import turtle
import random

class MojaTurtle(turtle.Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)       # fd z triedy ???Turtle
            self.rt(uhol)        # rt z triedy Turtle

class MojaTurtle1(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)        # fd z triedy Turtle
            self.rt(120)
            super().fd(5)        # fd z triedy Turtle
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)        # fd z triedy Turtle

class MojaTurtle2(MojaTurtle):
    def fd(self, dlzka):
        super().fd(dlzka)        # fd z triedy Turtle
        self.rt(180 - random.randint(-3, 3))
        super().fd(dlzka)        # fd z triedy Turtle
        self.rt(180 - random.randint(-3, 3))
        super().fd(dlzka)        # fd z triedy Turtle
```

Ďalej vytvoríme 100-prvkový zoznam korytnačiek, pričom pre každú z nich sa náhodne rozhodneme, aký typ vyberieme:

```
turtle.delay(0)
zoz = []
moja = (MojaTurtle, MojaTurtle1, MojaTurtle2)
for i in range(100):
    t = random.choice(moja)()   # všimnite si zátvorky na konci
    t.ht()
```

```
    t.speed(0)
    t.pu()
    t.setpos(random.randint(-300, 250), random.randint(-250, 250))
    t.pd()
    zoz.append(t)

for t in zoz:
    t.domcek(50)
```

Vytvorili sme tu zoznam korytnačiek troch rôznych typov. Každá z korytnačiek dokáže nakresliť `domcek()` ale každá to robí po svojom. Keďže tento zoznam obsahuje inštancie rôznych typov, niekedy hovoríme, že je to tzv. **polymorfný zoznam** (prípadne polymorfné pole).

V Pythone ale nie je problém so zoznamami, resp. poľami, ktorých prvky sú rôznych typov. Toto ale nie je bežné v iných programovacích jazykoch (Pascal, C++, …), kde väčšinou určujeme nejaký jeden konkrétny typ ako typ všetkých prvkov poľa (napríklad pole celých čísel, pole reťazcov, …). Aj v týchto jazykoch sa dá vytvárať polymorfné pole, ale už to nebude také jednoduché ako v Pythone.

Toto ale nie sú jediné významy polymorfizmu - tento pojem sa objavuje na mnohých miestach aj v situáciách, s ktorými sme sa zoznámili dávnejšie a už sme sa zmierili s takýmto správaním Pythonu. Pripomeňme si triedu `Cas` z 15. prednášky:

```
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

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

Tu vidíme použitie aj magickej metódy `__str__()`:

```
>>> c1 = Cas(10, 22, 30)
>>> c1.__str__()
    '10:22:30'
>>> c2 = Cas(4, 55, 18)
>>> str(c2)
    '4:55:18'
>>> print('sucet =', c1.sucet(c2))
    sucet = 15:17:48
```

Už vieme, že `c1.__str__()` priamo zavolá metódu `__str__()`, teda vráti reťazcovú reprezentáciu hodnoty čas. Volanie `str(c2)` tiež zavolá `__str__()`, ale neurobí sa to priamo, ale cez nejaký „magický“ mechanizmus:

- štandardná funkcia `str()` má za úlohu ľubovoľnú Pythonovskú hodnotu (napríklad číslo, zoznam, n-ticu, …) vyjadriť ako znakový reťazec
- keďže túto štandardnú funkciu naprogramovali vo firme „Python“ pred veľa rokmi, nemohli vtedy myslieť aj na to, že v roku 2021 niekto zadefinuje vlastný typ `Cas` a bude ho potrebovať pomocou `str(c2)` previesť na znakový reťazec

- preto má táto štandardná funkcia v sebe skrytý mechanizmus, pomocou ktorého veľmi jednoducho zistí reťazcovú reprezentáciu ľubovoľného typu: namiesto toho aby sama vyrábala znakový reťazec, zavolá metódu `__str__()` danej hodnoty; pritom každá trieda má vždy zadefinovanú náhradnú verziu tejto metódy, ktorá (keď ju neprekryjeme vlastnou metódou) vypisuje známe `'<__main__.Cas object at 0x035B92D0>'`

Štandardná funkcia `print()`, ktorá má za úlohu vypísať všetky svoje parametre, najprv všetky neznakové parametre prevedie na znakové reťazce: pomocou štandardnej funkcie `str()` z nich vyrobí reťazce a tieto vypíše.

Takže aj prevod hodnoty typu `Cas` na znakový reťazec pomocou štandardnej funkcie `str()` funguje vďaka **polymorfizmu**: aj táto funkcia sa prispôsobí (adaptuje) k zadanému typu a snaží sa z neho získať reťazec volaním jeho metódy `__str__()`.

Operátorový polymorfizmus

Už máme skúsenosti s tým, že napríklad operácia `+` funguje nielen s číslami ale aj s reťazcami a zoznamami:

```
>>> 12 + 34
    46
>>> 'Pyt' + 'hon'
    Python
>>> [1, 2] + [3, 4, 5]
    [1, 2, 3, 4, 5]
>>> 12 + '34'
    ...
    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Hovoríme tomu **operátorový polymorfizmus**, lebo táto operácia funguje rôzne pre rôzne typy. Python sa v tomto prípade nemusí pre každú dvojicu typov rozhodovať, či ich súčet je realizovateľný alebo je to chyba `TypeError`. Jednoducho prvému operandu oznámi, aby pripočítal druhý operand, t.j. zavolá nejakú jeho metódu a pošle mu druhý operand. Tou metódou je samozrejme magická metóda `__add__()` a preto pri vyhodnocovaní súčtu Python vlastne volá magickú metódu:

```
>>> 12 + 34
    46
>>> (12).__add__(34)   # 12 tu musí byť v zátvorkách
    46
>>> 'Pyt' + 'hon'
    Python
>>> 'Pyt'.__add__('hon')
    Python
>>> (12).__add__('34')
    NotImplemented
```

Tiež si uvedomte, že `a.__add__(b)` pre `a` napríklad celé číslo je to isté ako `int.__add__(a, b)`. Práve táto metóda je zodpovedná za to, či a ako sa dá k celému číslu pripočítať hodnota nejakého iného typu.

Teraz už vieme, že keď v Pythone zapíšeme `a + b`, v skutočnosti sa volá metóda `a.__add__(b)` a preto aj pre našu triedu `Cas` stačí dodefinovať túto metódu, teda vlastne stačí len premenovať `sucet()` na `__add__()`. Vyskúšajme:

```
class Cas:

    ...

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

```
    ...

c1 = Cas(10, 22, 30)
c2 = Cas(4, 55, 18)
print('sucet =', c1 + c2)
```

a vidíme, že to naozaj funguje. Zrejme na rovnakom princípe fungujú nielen všetky aritmetické operácie ale aj relačné operátory:

aritmetické operácie

metóda	operácia
`x.__add__(y)`	`x + y`
`x.__sub__(y)`	`x - y`
`x.__mul__(y)`	`x * y`
`x.__truediv__(y)`	`x / y`
`x.__floordiv__(y)`	`x // y`
`x.__mod__(y)`	`x % y`
`x.__pow__(y)`	`x ** y`
`x.__neg__()`	`- x`

Tomuto sa hovorí **preťažovanie operátorov** (operator overloading): existujúca operácia dostáva pre našu triedu nový význam, t.j. prekryli sme štandardné správanie Pythonu, keď niektoré operácie pre neznáme operandy hlásia chybu. Stretnete sa s tým aj v iných programovacích jazykoch.

relačné operácie

metóda	relácia
`x.__eq__(y)`	`x == y`
`x.__ne__(y)`	`x != y`
`x.__lt__(y)`	`x < y`
`x.__le__(y)`	`x <= y`
`x.__gt__(y)`	`x > y`
`x.__ge__(y)`	`x >= y`

Teraz môžeme vylepšiť kompletnú triedu `Cas`:

```
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 __add__(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

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

    def __gt__(self, iny):
        return self.sek > iny.sek

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

Vďaka tomuto môžeme časy nielen sčitovať ale aj odčitovať a porovnávať relačnými operátormi.

Pozrime si ešte takúto funkciu:

```
def sucet(a, b):
   return a + b
```

Zrejme táto funkcia bude dávať správne výsledky pre rôzne typy parametrov, môžeme im hovoriť **polymorfné parametre** a niekedy sa stretnete aj s pojmom **parametrický polymorfizmus**.

Trieda Zlomok

Na 14. cvičeniach ste riešili aj úlohu, v ktorej ste definovali triedu `Zlomok` aj s metódami `str()` a `float()`. My toto riešenie trochu vylepšíme:

```
class Zlomok:
    def __init__(self, citatel=0, menovatel=1):
        self.cit = citatel
        self.men = menovatel

    def __str__(self):
        return f'{self.cit}/{self.men}'

    def __int__(self):
        return self.cit // self.men

    def __float__(self):
        return self.cit / self.men
```

a jednoduchý test:

```
>>> z1 = Zlomok(3, 8)
>>> z2 = Zlomok(2, 4)
>>> print('desatinne cislo z', z1, 'je', float(z1))
    desatinne cislo z 3/8 je 0.375
>>> print('cela cast', z2, 'je', int(z2))
    cela cast 2/4 je 0
```

Magické metódy `__int__()` a `__float__()` slúžia na to, aby sme objekt typu `Zlomok` mohli poslať do konvertovacích funkcií `int()` a `float()`.

Tento dátový typ by mohol byť naozaj užitočný, keby obsahoval aj nejaké operácie. S týmto už máme nejaké skúsenosti z definovania triedy `Cas`. Tiež by bolo veľmi vhodné, keby sa v tejto triede zlomok automaticky upravil na základný tvar. Túto úpravu budeme robiť v inicializácii `__init__()`: z matematiky na základnej škole vieme, že na to potrebujeme zistiť **najväčší spoločný deliteľ**. Použijeme známy Euklidov algoritmus (programovali sme ho v 5. prednáške):

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

Ak budeme túto funkciu potrebovať len v metóde `__init__()`, nemusíme ju definovať ako globálnu funkciu, ale ju prenesieme do tela inicializačnej funkcie, čím z nej urobíme lokálnu funkciu (vidí ju len samotná metóda `__init__()`). Všimnite si, že sme sem doplnili niekoľko zatiaľ neznámych magických metód:

```
class Zlomok:

    def __init__(self, citatel=0, menovatel=1):

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

        if menovatel == 0:
            menovatel = 1
        delitel = nsd(citatel, menovatel)
        self.cit = citatel // delitel
        self.men = menovatel // delitel

    def __str__(self):
        return f'{self.cit}/{self.men}'

    __repr__ = __str__

    def __add__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.cit*m + self.men*c, self.men*m)

    __radd__ = __add__

    def __sub__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.cit*m - self.men*c, self.men*m)

    def __rsub__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.men*c - self.cit*m, self.men*m)

    def __mul__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
```

```
        return Zlomok(self.cit * c, self.men*m)

    __rmul__ = __mul__

    def __abs__(self):
        return Zlomok(abs(self.cit), self.men)

    def __int__(self):
        return self.cit // self.men

    def __float__(self):
        return self.cit / self.men

    def __lt__(self, iny):
        return self.cit*iny.men < self.men*iny.cit

    def __eq__(self, iny):
        return self.men==iny.men and self.cit==iny.cit
```

Niekoľko noviniek v tomto kóde:

- atribút `__repr__` je tu definovaný pomocou priradenia `__repr__ = __str__` a znamená:
    - aj `__repr__` bude metódou triedy `Zlomok` a týmto sme ju definovali ako identickú k `__str__` (triedny atribút `__repr__` obsahuje rovnakú referenciu ako `__str__`, teda obsahuje rovnakú definíciu metódy)
    - magická metóda `__repr__` špecifikuje, čo sa bude vypisovať, ak inštanciu zadáme priamo v shelli alebo sa objaví pri vypisovaní prvkov zoznamu, napríklad:

```
>>> z = Zlomok(1, 3)
>>> z
    1/3
>>> zoznam = [Zlomok(1, 5), Zlomok(2, 5), Zlomok(3, 5), Zlomok(4, 5)]
>>> zoznam
    [1/5, 2/5, 3/5, 4/5]
```

- magická metóda `__radd__` (jej definícia je identická s `__add__`) je potrebná v situáciách, keď chceme sčitovať celé číslo so zlomkom:
    - samotná `__add__` zvláda sčítať len zlomok s číslom (súčet `Zlomok(1, 3) + 1` označuje volanie `Zlomok(1, 3).__add__(1)`)
    - sčitovanie čísla so zlomkom `1 + Zlomok(1, 3)` označuje `(1).__add__(Zlomok(1, 3))`, čo by znamenalo, že metóda `__add__` triedy `int` by mala vedieť sčitovať aj zlomky (je nemožné predefinovať štandardnú metódu `int.__add__()` aby fungovala s nejakým divným typom)
    - preto pri sčitovaní `1 + Zlomok(1, 3)`, keď Python zistí, že nefunguje `(1).__add__(Zlomok(1, 3))`, vyskúša vymeniť operandy operácie a namiesto `__add__()` zavolať `__radd__()`
- podobne je definovaná aj metóda `__rmul__`, pričom odčitovanie `__rsub__` nemôže byť identická funkcia s metódou `__sub__`, preto je zadefinovaná zvlášť
- pridali sme magickú metódu `__abs__()`, vďaka ktorej bude fungovať aj štandardná funkcia `abs(zlomok)`

Uvedomte si, že všetky nami definované metódy triedy `Zlomok` (okrem `__init__()`) sú **pravé funkcie** a preto aj náš nový typ `Zlomok` môžeme považovať za nemeniteľný (immutable).

Vďaka relačným operátorom `__lt__()` a `__eq__()` a schopnosti sčitovať zlomky s číslami bude fungovať aj takáto ukážka:

```
>>> zoznam = []
>>> for m in range(2, 8):
```

```
        for c in range(1, m):
            zoznam.append(Zlomok(c, m))

>>> zoznam
    [1/2, 1/3, 2/3, 1/4, 1/2, 3/4, 1/5, 2/5, 3/5, 4/5, 1/6, 1/3,
     1/2, 2/3, 5/6, 1/7, 2/7, 3/7, 4/7, 5/7, 6/7]
>>> min(zoznam)
    1/7
>>> max(zoznam)
    6/7
>>> sum(zoznam)
    21/2
>>> sorted(zoznam)
    [1/7, 1/6, 1/5, 1/4, 2/7, 1/3, 1/3, 2/5, 3/7, 1/2, 1/2, 1/2,
     4/7, 3/5, 2/3, 2/3, 5/7, 3/4, 4/5, 5/6, 6/7]
```

Typ množina

Na 14. cvičeniach (11. úloha) ste riešili aj príklad s triedou `Zoznam`, pomocou ktorej sa uchovávali nejaké texty v zozname. Tu je možné riešenie:

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

    def __str__(self):
        p = []
        for prvok in self.zoznam:
            p.append(str(prvok))
        return ', '.join(p)

    def pridaj(self, prvok):
        if prvok not in self.zoznam:
            self.zoznam.append(prvok)

    def vyhod(self, prvok):
        if prvok in self.zoznam:
            self.zoznam.remove(prvok)

    def je_v_zozname(self, prvok):
        return prvok in self.zoznam

    def pocet(self):
        return len(self.zoznam)
```

Jednoduchý test:

```
z = Zoznam()
z.pridaj('behat')
z.pridaj('upratat')
z.pridaj('ucit sa')
if z.je_v_zozname('behat'):
    print('musis behat')
else:
    print('nebehaj')
z.pridaj('upratat')
print('zoznam =', z)
z.vyhod('spievat')
print('pocet prvkov v zozname =', z.pocet())
```

```
musis behat
zoznam = behat, upratat, ucit sa
pocet prvkov v zozname = 3
```

V Pythone je zaužívané použiť operáciu `in` vtedy, keď potrebujeme zistiť, či sa v nejakej postupnosti hodnôt nachádza nejaká konkrétna hodnota, napríklad:

```
 >>> 3 in [1, 2, 3, 4, 5]
     True
 >>> 'x' in 'Python'
     False
```

Zrejme by bolo prirodzené, keby sme aj našu metódu `je_v_zozname()` vedeli prerobiť na pythonovský štýl (pythonic). Aj na toto existuje magická metóda `__contains__()` a predchádzajúce dva príklady sú vlastne krajšími zápismi (tzv. *syntactic sugar*) pre:

```
 >>> [1, 2, 3, 4, 5].__contains__(3)
     True
 >>> 'Python'.__contains__('x')
     False
```

Podobne aj štandardná funkcia `len()`, ktorá vie zistiť počet prvkov zoznamu alebo dĺžku reťazca (počet znakov v reťazci), využíva polymorfizmus, teda v skutočnosti, aby zistila počet prvkov nejakej štruktúry, sa jej na to opýta pomocou magickej metódy `__len__()`. Preto nasledovné trojice príkazov robia to isté:

```
>>> len([1, 2, 3, 4, 5])
    5
>>> [1, 2, 3, 4, 5].__len__()
    5
>>> list.__len__([1, 2, 3, 4, 5])
    5

>>> len('Python')
    6
>>> 'Python'.__len__()
    6
>>> str.__len__('Python')
    6
```

Upravme aj našu triedu `Zoznam`, pričom premenujeme aj metódy `pridaj()` a `vyhod()` na anglické ekvivalenty:

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

    def __str__(self):
        p = []
        for prvok in self.zoznam:
            p.append(str(prvok))
        return ', '.join(p)

    def __contains__(self, prvok):
        return prvok in self.zoznam

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

    def add(self, prvok):
        if prvok not in self.zoznam:
            self.zoznam.append(prvok)
```

```
    def discard(self, prvok):
        if prvok in self.zoznam:
            self.zoznam.remove(prvok)
```

Otestujeme rovnako ako predtým:

```
z = Zoznam()
z.add('behat')
z.add('upratat')
z.add('ucit sa')
if 'behat' in z:
    print('musis behat')
else:
    print('nebehaj')
z.add('upratat')
print('zoznam =', z)
z.discard('spievat')
print('pocet prvkov v zozname =', len(z))
```

```
musis behat
zoznam = behat, upratat, ucit sa
pocet prvkov v zozname = 3
```

Uvedomte si, že do takéhoto zoznamu nemusíme vkladať len znakové reťazce, ale rovnako by fungoval aj pre ľubovoľné iné typy hodnôt. Tento typ je vlastne jednoduchá realizácia matematickej množiny hodnôt: každý prvok sa tu môže nachádzať maximálne raz.

Štandardný typ množina - set

Python má medzi štandardnými typmi aj typ **množina**, ktorý má v Pythone meno `set`. Podobne ako aj iné typy `str`, `list` a `tuple` aj tento množinový typ je postupnosťou hodnôt, ktorú môžeme prechádzať for-cyklom (je to iterovateľný typ) alebo ju poslať ako parameter pri konštruovaní iného typu (kde sa očakáva postupnosť). Napríklad:

```
>>> mnozina = {'behat', 'ucit sa', 'upratat'}
>>> mnozina
    {'upratat', 'behat', 'ucit sa'}
>>> zoznam = list(mnozina)
>>> zoznam
    ['upratat', 'behat', 'ucit sa']
>>> ntica = tuple(mnozina)
>>> ntica
    ('upratat', 'behat', 'ucit sa')
>>> for prvok in mnozina:
        print(prvok, end=', ')

    'upratat', 'behat', 'ucit sa',
```

Podobne ako vieme skonštruovať zoznam pomocou generátora postupnosti `range()`, vieme to urobiť aj s množinami:

```
>>> list(range(7))
    [0, 1, 2, 3, 4, 5, 6]
>>> set(range(7))
    {0, 1, 2, 3, 4, 5, 6}
```

alebo vytvorenie zoznamu a množiny zo znakového reťazca:

```
>>> list('mama ma emu')
    ['m', 'a', 'm', 'a', ' ', 'm', 'a', ' ', 'e', 'm', 'u']
>>> set('mama ma emu')
    {' ', 'm', 'u', 'a', 'e'}
```

Štandardný Pythonovský typ **set** má kompletnú sadu množinových operácií a veľa užitočných metód. Pre prvky množiny ale platí, že to nemôžu byť ľubovoľné hodnoty, ale musia to byť nemenné typy (immutable), napríklad čísla, reťazce, n-tice.

Predchádzajúci príklad, v ktorom sme definovali triedu `Zoznam` vieme prepísať l s použitím pythonovských množín, napríklad takto:

```
z = set()                   # prázdna pythonovská množina
z.add('behat')
z.add('upratat')
z.add('ucit sa')
if 'behat' in z:
    print('musis behat')
else:
    print('nebehaj')
z.add('upratat')
print('zoznam =', z)
z.discard('spievat')
print('pocet prvkov v zozname =', len(z))
```

```
musis behat
zoznam = {behat, upratat, ucit sa}
pocet prvkov v zozname = 3
```

Operácie a metódy s množinami

Štandardný typ množina (`set`) je meniteľný (**mutable**) a z toho vyplývajú všetky dôsledky podobne ako pre pythonovské zoznamy (`list`).

V ďalších tabuľkách predpokladáme, že `M`, `M1` a `M2` sú nejaké množiny:

množinové operácie

	popis
`M1 | M2`	zjednotenie dvoch množín
`M1 & M2`	prienik dvoch množín
`M1 - M2`	rozdiel dvoch množín
`M1 ^ M2`	vylučovacie zjednotenie dvoch množín (symetrický rozdiel)
`M1 == M2`	dve množiny majú rovnaké prvky
`M1 is M2`	dve množiny sú identické štruktúry v pamäti (je to tá istá hodnota)
`M1 < M2`	M1 je podmnožinou M2 (funguje aj pre zvyšné relačné operátory)
`prvok in M`	zistí, či prvok patrí do množiny

množinové operácie

	popis
`prvok not in M`	zistí, či prvok nepatrí do množiny
`for prvok in M: ...`	cyklus, ktorý prechádza cez všetky prvky množiny

štandardné funkcie

	popis
`len(M)`	počet prvkov
`min(M)`	minimálny prvok (ale všetky prvky sa musia dať navzájom porovnávať)
`max(M)`	maximálny prvok (ale všetky prvky sa musia dať navzájom porovnávať)
`list(M)`	vráti neusporiadaný zoznam prvkov z množiny
`sorted(M)`	vráti usporiadaný zoznam (ale všetky prvky sa musia dať navzájom porovnávať)

niektoré metódy

	popis
`M.add(prvok)`	pridá prvok do množiny (ak už v množine bol, neurobí nič)
`M.remove(prvok)`	vyhodí daný prvok z množiny (ak neexistuje, vyhlási chybu)
`M.discard(prvok)`	vyhodí daný prvok z množiny (ak neexistuje, neurobí nič)
`M.pop()`	vyhodí nejaký neurčený prvok z množiny a vráti jeho hodnotu (ak je množina prázdna, vyhlási chybu)
`M.clear()`	vyčistí množinu

Všetky tieto metódy sú **mutable**, teda zmenia obsah premennej a ak je na ňu viac referencií, ovplyvní to všetky.

Metód, ktoré pracujú s množinami, je oveľa viac.

Vytvorenie množiny

niektoré metódy

	popis
`M = set()`	vytvorí prázdnu množinu
`M = {hodnota, hodnota, ...}`	vytvorí neprázdnu množinu so zadanými prvkami

niektoré metódy

	popis
`M = set(zoznam)`	so zadaného zoznamu vytvorí množinu
`M = set(M1)`	vytvorí kópiu množiny `M1`

Uvedomte si, že niektoré situácie vieme riešiť rôznymi spôsobmi, napríklad

- pridať `prvok` do množiny `mnoz`:

```
mnoz.add(prvok)          # mutable
```

alebo:

```
mnoz = mnoz | {prvok}    # immutable
```

čo je to isté ako:

```
mnoz |= {prvok}          # mutable
```

- vyhodiť jeden `prvok` z množiny `mnoz`:

```
mnoz.discard(prvok)      # mutable
```

alebo:

```
mnoz = mnoz - {prvok}    # immutable
```

čo je to isté ako:

```
mnoz -= {prvok}          # mutable
```

ak máme istotu, že prvok je v množine (inak to spadne na chybe):

```
mnoz.remove(prvok)       # mutable
```

- zistí, či je množina `mnoz` prázdna:

```
mnoz == set()
```

alebo:

```
len(mnoz) == 0
```

alebo veľmi nečitateľne:

```
not mnoz
```

Príklady s množinami

Napíšme funkciu, ktorá vráti počet rôznych samohlások v danom slove:

```
def pocet_samohlasok(slovo):
    return len(set(slovo) & set('aeiouy'))
```

Bez použitia množiny by sme ju zapísali asi takto:

```
def pocet_samohlasok(slovo):
    vysl = 0
    for znak in 'aeiouy':
        if znak in slovo:
            vysl += 1
    return vysl
```

Ďalšia funkcia skonštruuje množinu s takouto vlastnosťou:

- `1` patrí do množiny
- ak do množiny patrí nejaké `i`, tak tam patrí aj `2*i+1` aj `3*i+1`

Funkcia vytvorí všetky prvky množiny ktoré nie sú väčšie ako zadané `n`:

```
def urob(n):
    m = {1}
    for i in range(n//2):
        if i in m:
            if 2*i + 1 <= n:
                m.add(2*i + 1)
            if 3*i + 1 <= n:
                m.add(3*i + 1)
    return m
```

Ďalšia funkcia je rekurzívna a zisťuje, či nejaké dané `i` je prvkom množiny z predchádzajúceho príkladu (bez toho, aby sme museli túto množinu najprv skonštruovať):

```
def test(i):
    if i == 0:
        return False
    if i == 1:
        return True
    if i%2 == 1 and test(i//2):
        return True
    if i%3 == 1 and test(i//3):
        return True
    return False
```

To isté sa dá zapísať trochu úspornejšie (a výrazne menej čitateľne):

```
def test(i):
    if i <= 1:
        return i == 1
    return i%2 == 1 and test(i//2) or i%3 == 1 and test(i//3)
```

Pomocou funkcie `test()` vieme zapísať aj funkciu `urob()`:

```
def urob(n):
    m = set()
    for i in range(n+1):
        if test(i):
            m.add(i)
    return m
```

Ďalšia funkcia skonštruuje množinu prvočísel pomocou algoritmu Eratostenovo sito:

- zoberieme zoznam všetkých celých čísel od 2 po nejaké zadané `n`
- prvé číslo v zozname `2` je prvočíslo, zo zoznamu odstránime všetky jeho násobky
- druhé číslo v tomto novom zozname `3` je tiež prvočíslo, zo zoznamu odstránime všetky jeho násobky
- aj tretie číslo v tomto novom zozname `5` je prvočíslo, zo zoznamu odstránime všetky jeho násobky
- takto to budeme opakovať, kým neprejdeme celý zoznam čísel

Zapíšme to ako funkciu:

```
def eratostenovo_sito(n):
    mnozina = set(range(2, n+1))
    for i in range(n):
        if i in mnozina:
            mnozina -= set(range(i+i, n+1, i))
    return mnozina

>>> eratostenovo_sito(100)
    {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
     73, 79, 83, 89, 97}
```

Python nezaručuje, že prvky v množine sú v rastúcej postupnosti. Preto niekedy množinu prevedieme na usporiadaný zoznam, napríklad:

```
>>> m1 = {1, 5, 10, 30, 100, 1000}
>>> m2 = {2, 6, 11, 31, 101, 1001}
>>> m1 | m2
    {1, 2, 100, 5, 101, 6, 1000, 1001, 10, 11, 30, 31}
>>> sorted(m1 | m2)
    [1, 2, 5, 6, 10, 11, 30, 31, 100, 101, 1000, 1001]
```

Cvičenia

L.I.S.T.

- riešenia **aspoň 12 úloh** odovzdaj na úlohový server https://list.fmph.uniba.sk/
- pozri si Riešenie úloh 18. cvičenia

Polymorfné korytnačky

1. Do triedy `MojaTurtle` z prednášky pridaj inicializáciu `__init__(self)`, ktorá nastaví `speed(0)` a náhodnú pozíciu, kde `x` aj `y` je z `<-250, 250>`. Trieda `MojaTurtle1` je odvodena z `MojaTurtle` a kresli cikcakové čiary. Otestuj:

```
2. turtle.delay(0)
3. for i in range(30):
4.     MojaTurtle1().domcek(30)
```

2. Vytvor novú triedu `MojaTurtle0`, ktorá bude odvodená z `MojaTurtle`. V tejto triede prekryješ `lt(uhol)` aj `rt(uhol)`, v ktorých sa zmení otočenie na `uhol+randint(-5,5)`. Otestuj:

```
3. turtle.delay(0)
4. for i in range(20):
5.     MojaTurtle0().domcek(50)
```

Teraz zmeň triedy `MojaTurtle1` aj `MojaTurtle2` tak, aby boli odvodené z `MojaTurtle0`. Opäť otestuj kreslenie 20 domčekov s týmito dvoma novými triedami.

Polymorfizmus

3. Prepíš tento výraz:

```
4. (3 + 4) * 5 + 2 ** (100 // 5)
```

tak, aby si všetky operácie nahradil magickými metódami. Potom skontroluj, či jeho vykonaním dostaneš rovnaký výsledok:

```
>>> (3 + 4) * 5 + 2 ** (100 // 5)
    1048611
>>> (3).__add__(4)...
```

Ďalej prepíš nasledovný výraz teraz bez magických funkcii a opäť skotroluj:

```
>>> (7).__pow__(8).__str__().__len__().__add__('xy'.__rmul__(8).__len__().__add__(1)).__
mul__(13)
    312
>>> ... 7 ** 8 ...
```

4. Triedu `Cas` z prednášky doplň tak, aby operácie sčitovania a odčitovania fungovali aj s celými číslami (pripočítava, resp. odpočítava sekundy), ale aj s n-ticami (prvým prvkom sú hodiny, druhým minúty a tretím sekundy). Napríklad:

```
5. >>> c = Cas(8, 10, 34)
6. >>> c
7.     8:10:34
8. >>> c + 640
9.     8:21:14
10.>>> (1, 55) + c
11.    10:05:34
12.>>> c - 100
13.    8:08:54
```

Ak IDLE nevypisuje hodnoty časov, asi ti chýba magická metóda `__repr__`.

5. Pomocou modulu `time` a funkcie vieme zistiť momentálny čas v počítači. Napríklad:

```
6. >>> import time
7. >>> time.localtime()
8.     time.struct_time(tm_year=2017, tm_mon=11, tm_mday=22, tm_hour=8, tm_min=26, tm_sec=1
   2,
9.     tm_wday=1, tm_yday=327, tm_isdst=0)
10.>>> time.localtime()[3:6]
11.    (8, 26, 24)
```

Napíš funkciu `teraz()`, ktorá vráti inštanciu triedy `Cas` s momentálnym časom. Napríklad:

```
>>> c = teraz()
>>> type(c)
    <class '__main__.Cas'>
>>> c
    8:34:07
```

```
>>> teraz()
    8:35:22
```

6. Vytvor zoznam rôznych časov (napríklad ich generuj náhodným generátorom). Otestuj, či funguje triedenie pomocou štandardnej funkcie `sorted()`. Napríklad:

```
>>> zoznam = [Cas(20, 15), Cas(7), ...]
>>> zoznam1 = sorted(zoznam)
>>> zoznam1
    [... usporiadaný zoznam časov ...]
```

Funkcia `sorted()` by mala fungovať pre ľubovoľnú postupnosť prvkov, ktoré sa navzájom dajú porovnávať reláciou menší `<`.

Množiny - set

7. Napíš funkciu `mnozina1(n)`, ktorá vráti množinu všetkých čísel z intervalu `<0, n>`, ktoré sú deliteľné `3` a súčasne ich zvyšok po delení `5` je `1` alebo `2`. Vo funkcii nepoužiť žiaden cyklus, len množinové operácie a funkciu `range`. Napríklad:

```
>>> m = mnozina1(21)
>>> type(m)
    <class 'set'>
>>> m
    {6, 12, 21}
```

Teraz napíš funkciu `mnozina2(n1, n2)`, ktorá robí to isté ako funkcia `mnozina1` ale pri interval `<n1, n2>`. Funkciu zapíš tak, aby sa využilo volanie `mnozina1`. Napríklad:

```
>>> mnozina2(20, 100)
    {21, 27, 36, ... }
```

8. Napíš funkciu `len_v_jednom(retazec1, retazec2)`, ktorá vráti množinu znakov, ktoré sa vyskytujú iba v jednom z oboch reťazcov. Funkciu zapíš len jedným množinovým výrazom:

```
def len_v_jednom(retazec1, retazec2):
    return ...
```

Napríklad:

```
>>> mn = len_v_jednom('isiel macek do malaciek', 'sosovicku mlacit')
>>> mn
    {'d', 'e', 't', 'u', 'v'}
```

9. Napíš funkciu `vsetky_rozne(postupnost)`, ktorá zistí, či sú všetky prvky danej postupnosti navzájom rôzne. Funkcia vráti `True` alebo `False`. Funkciu zapíš len jedným výrazom, v ktorom využiješ množinu:

```
def vsetky_rozne(postupnosť):
    return ...
```

Napríklad:

```
>>> vsetky_rozne((2, 5, 7, 'x', 11, 13, 17, 19, 23, 'x', 29))
    False
```

10. Napíš funkciu `rozdel(mnozina)`, ktorá vráti dve množiny: množinu čísel a množinu reťazcov. Napríklad:

```
>>> m1, m2 = rozdel({7, 7.5, '12', 3, 'python'})
>>> m1
    {7, 7.5, 3}
>>> m2
    {'12', 'python'}
```

11. Napíš funkciu `bez_parnych(mnozina)`, ktorá z danej množiny vyhodí všetky párne celé čísla. Funkcia nič nevypisuje ani nevracia. Funkcia len modifikuje vstupnú množinu. Napríklad:

```
>>> a = {7, 6.0, '12', 4, 'python', 124}
>>> bez_parnych(a)
>>> a
    {7, 6.0, '12', 'python'}
>>> b = set(range(4, 100, 2))
>>> bez_parnych(b)
>>> b
    set()
```

12. Napíš funkciu `len_raz(retazec)`, ktorá vráti množinu znakov z daného reťazca, ktoré sa v ňom vyskytujú iba raz. Rieš tak, že najprv zostrojíš dve množiny: množinu všetkých znakov a množinu tých, ktoré boli viac ako raz. Potom z nich vytvoríš výsledok. Napríklad:

```
>>> znaky = len_raz('anicka dusicka kde si bola')
>>> znaky
    {'b', 'e', 'l', 'n', 'o', 'u'}
>>> len_raz('mama ma emu a ema ma mamu')
    set()
```

13. Napíš funkciu `opakuje_sa(meno_suboru)`, ktorá vráti množinu slov z daného súboru, ktoré sa v ňom objavujú viac ako raz (slová sú v ňom navzájom oddelené medzerami). Napríklad pre súbor `text1.txt`:

```
Ján Botto
Žltá ľalija

Stojí stojí mohyla
Na mohyle zlá chvíľa
na mohyle tŕnie chrastie
a v tom tŕní chrastí rastie
rastie kvety rozvíja
jedna žltá ľalija
Tá ľalija smutno vzdychá
Hlávku moju tŕnie pichá
a nožičky oheň páli
pomôžte mi v mojom žiali
```

```
>>> viac = opakuje_sa('text1.txt')
>>> viac
    {'a', 'mohyle', 'rastie', 'tŕnie', 'v', 'ľalija'}
```

14. Napíš funkciu `vsetky(a, b, c, d)`, ktorá vráti zoznam všetkých dvojprvkových množín z prvkov `a`, `b`, `c`, `d`. Môžeš predpokladať, že všetky hodnoty parametrov sú navzájom rôzne. Napríklad:

```
>>> zoz = vsetky(1, 2, 3, 4)
>>> zoz
    [{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

15. Vylepši funkciu z predchádzajúceho príkladu `vsetky(mnozina)`, tak aby generovala všetky dvojprvkové množiny z prvkov zadanej množiny. Napríklad:

```
>>> z = vsetky(set('java'))
>>> z
    [{'j', 'v'}, {'a', 'v'}, {'j', 'a'}]
>>> vsetky(set(range(5)))
    [{0, 1}, {0, 2}, {0, 3}, {0, 4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
>>> vsetky({3, 1, 'x', 4, 1, 2, 'x'})
    [{1, 2}, {1, 'x'}, {1, 3}, {1, 4}, {2, 'x'}, {2, 3}, {2, 4}, {3, 'x'}, {'x', 4}, {3,
4}]
>>> vsetky({'python'})
    []
```

16. Napíš funkciu `kartez_sucin(m1, m2)`, ktorá vráti **karteziánsky súčin množín**, t.j. množinu všetkých usporiadaných dvojíc (`tuple`), ktorých prvá zložka je z množiny `m1` a druhá zložka z množiny `m2`. Napríklad:

```
>>> k = kartez_sucin({1, 2, 3, 4}, {'a', 'b', 'c'})
>>> type(k)
    <class 'set'>
>>> k
    {(3, 'c'), (3, 'b'), (2, 'a'), (4, 'a'), (4, 'c'), (4, 'b'), (1, 'b'),
     (1, 'c'), (1, 'a'), (3, 'a'), (2, 'c'), (2, 'b')}
```

Uvedom si, že výsledná množina môže obsahovať tieto dvojice v ľubovoľnom inom poradí.

10. Týždenný projekt

L.I.S.T.

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

Asi už poznáš logickú hru Sudoku, v ktorej je úlohou hráča zaplniť voľné políčka štvorcovej siete 9x9 (dvojrozmerná tabuľka) číslami od 1 do 9. Pritom by mali byť splnené tieto podmienky:

- v každom riadku tabuľky bude každé číslo práve raz
- v každom stĺpci tabuľky bude každé číslo práve raz
- v každom vyznačenom štvorci 3x3 tabuľky bude každé číslo práve raz

Zadanie hry môže vyzerať, napríklad takto:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Tvoj program sa bude snažiť riešiť túto hru veľmi zjednodušeným spôsobom: postupne prejde všetky voľné políčka a pre každé z nich zistí množinu kandidátov, t.j. všetkých takých čísel, ktoré by sme na toto políčko mohli položiť a nevznikla by kolízia z už položenými číslami v riadku, v stĺpci a ani vo príslušnom štvorci 3x3. Zrejme, ak je niektorá z týchto množín prázdna, táto hra už nemá riešenie a netreba sa ju ďalej snažiť riešiť. Ak na niektorom voľnom políčku je táto množina jednoprvková, znamená to, že práve toto číslo je jediné, ktoré sem môžeme (musíme) zapísať. Ak toto urobíme so všetkými jednoprvkovými množinami, trochu sa priblížime k celkovému riešeniu tohto Sudoku. Postupné riešenie pre vás teda bude znamenať toto:

1. všetky voľné políčka (označené znakom `'.'`) nahraď množinami kandidátov
2. ak je niektorá z množín prázdna, úloha nemá riešenie, bude treba skončiť
3. ak tam nie je žiadna množina (nie je tam voľné políčko), úloha je zrejme vyriešená a bude treba skončiť
4. ak majú všetky množiny viac ako 1 prvok, bude treba skončiť, lebo tento algoritmus už viac robiť nevie
5. ak je tam niekoľko jednoprvkových množín, všetky sa nahradia týmto svojim jediným prvkom, všetky ostatné množiny sa nahradia znakom `'.'`
6. ďalej sa pokračuje v 1. kroku

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

```
class Sudoku:
    def __init__(self, meno_suboru):
        self.tab = []
        ...

    def __str__(self):
        ...

    def urob(self):
        ...

    def nahrad(self):
        ...

    def ries(self):
        ...

    def pocet_nezaplnenych(self):
        ...
```

Metódy majú fungovať takto:

- inicializácia `__init__(meno_suboru)` prečíta textový súbor s počiatočným zaplnením čísel, voľné políčka sú vyznačené znakmi `'.'`
    - súbor obsahuje 9 riadkov, v každom je 9 číslic alebo bodiek oddelených medzerou
    - inicializácia zaplní dvojrozmernú tabuľku `self.tab` (prvkami musia byť celé čísla `int` a znaky `'.'`)
- metóda `__str__()` vyrobí reťazcovú reprezentáciu hracej plochy: reťazec bude obsahovať 9 riadkov v každom po 9 hodnôt (čísel alebo bodiek) v rovnakom formáte, ako bol zadaný vstupný súbor
- metóda `urob()` všetky voľné políčka (v tabuľke sú tam `'.'`) nahradí množinami kandidátov, t.j. čísel, ktoré by sa na tejto pozícii mohli nachádzať
    - funkcia vráti `None`, ak sa medzi týmito množinami objavila **prázdna** množina, inak funkcia vráti celkový počet **jednoprvkových** množín
- metóda `nahrad()` všetky políčka s jednoprvkovými množinami sa nahradia priamo hodnotou v tejto množine, ostatné políčka s množinami sa nahradia znakom `'.'`
- metóda `ries()` bude postupne volať metódy `urob()` a `nahrad()`, kým bude metóda `urob()` vracať číslo a nie `None` (zrejme bude vykonávať hore uvedený cyklus); funkcia vráti dvojicu: počet prechodov cyklu (volaní `urob()` a `nahrad()`) a touto hodnotou:
    - `None`, ak táto hra nemá riešenie (metóda `urob()` vrátila `None`)
    - počet voľných políčok (zrejme `0` označuje, že úloha je úplne vyriešená)

  po skončení metódy `ries()` by dvojrozmerná tabuľka `self.tab` mala obsahovať len čísla a znaky `'.'`
- metóda `pocet_nezaplnenych()` vráti momentálny počet voľných políčok

Napríklad pre vstupný súbor `'subor1.txt'`:

```
. . . . . 9 . . .
. . 7 . 8 6 . . .
6 . . 3 . . . . .
. 4 . . . 7 . . 8
. . . . . . . 3 2
. . 3 6 . 5 1 . .
. 6 . 7 . . . 8 .
3 . 2 . . . 4 9 .
. 5 4 8 . . . . 3
```

Tento test nám bude postupne vypisovať:

```
>>> s = Sudoku('subor1.txt')
>>> print(s)
    . . . . . 9 . . .
    . . 7 . 8 6 . . .
    6 . . 3 . . . . .
    . 4 . . . 7 . . 8
    . . . . . . . 3 2
    . . 3 6 . 5 1 . .
    . 6 . 7 . . . 8 .
    3 . 2 . . . 4 9 .
    . 5 4 8 . . . . 3
>>> s.pocet_nezaplnenych()
    55
>>> s.urob()
    1
>>> for r in s.tab:
        print(r)

    [{1, 2, 4, 5, 8}, {1, 2, 3, 8}, {1, 5, 8}, {1, 2, 4, 5}, {1, 2, 4, 5, 7}, 9, {2, 3, 5, 6, 7
, 8}, {1, 2, 4, 5, 6, 7}, {1, 4, 5, 6, 7}]
    [{1, 2, 4, 5, 9}, {1, 2, 3, 9}, 7, {1, 2, 4, 5}, 8, 6, {2, 3, 5, 9}, {1, 2, 4, 5}, {1, 4, 5
, 9}]
```

```
    [6, {1, 2, 8, 9}, {1, 5, 8, 9}, 3, {1, 2, 4, 5, 7}, {1, 2, 4}, {2, 5, 7, 8, 9}, {1, 2, 4, 5
, 7}, {1, 4, 5, 7, 9}]
    [{1, 2, 5, 9}, 4, {1, 5, 6, 9}, {1, 2, 9}, {1, 2, 3, 9}, 7, {5, 6, 9}, {5, 6}, 8]
    [{1, 5, 7, 8, 9}, {1, 7, 8, 9}, {1, 5, 6, 8, 9}, {1, 4, 9}, {1, 4, 9}, {1, 4, 8}, {5, 6, 7,
9}, 3, 2]
    [{2, 7, 8, 9}, {2, 7, 8, 9}, 3, 6, {2, 4, 9}, 5, 1, {4, 7}, {4, 7, 9}]
    [{1, 9}, 6, {1, 9}, 7, {1, 2, 3, 4, 5, 9}, {1, 2, 3, 4}, {2, 5}, 8, {1, 5}]
    [3, {1, 7, 8}, 2, {1, 5}, {1, 5, 6}, {1}, 4, 9, {1, 5, 6, 7}]
    [{1, 7, 9}, 5, 4, 8, {1, 2, 6, 9}, {1, 2}, {2, 6, 7}, {1, 2, 6, 7}, 3]
>>> s.nahrad()
>>> for r in s.tab:
        print(r)

    ['.', '.', '.', '.', '.', 9, '.', '.', '.']
    ['.', '.', 7, '.', 8, 6, '.', '.', '.']
    [6, '.', '.', 3, '.', '.', '.', '.', '.']
    ['.', 4, '.', '.', '.', 7, '.', '.', 8]
    ['.', '.', '.', '.', '.', '.', '.', 3, 2]
    ['.', '.', 3, 6, '.', 5, 1, '.', '.']
    ['.', 6, '.', 7, '.', '.', '.', 8, '.']
    [3, '.', 2, '.', '.', 1, 4, 9, '.']
    ['.', 5, 4, 8, '.', '.', '.', '.', 3]
>>> s.ries()
    (12, 28)
>>> print(s)
    . . . . . 9 . . .
    . . 7 . 8 6 . . .
    6 . . 3 . 4 . . .
    2 4 1 9 3 7 5 6 8
    . . . 4 1 8 . 3 2
    . . 3 6 2 5 1 . .
    1 6 9 7 4 3 2 8 5
    3 8 2 5 6 1 4 9 7
    7 5 4 8 9 2 6 1 3
```

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

```
# 10. zadanie: sudoku
# autor: Janko Hraško
# datum: 17.12.2021
```

Projekt `riesenie.py` odovzdaj na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **17. decembra**. Testovač bude spúšťať tvoje riešenie s rôznymi vstupmi (môžeš si ich stiahnuť z LISTu). Za projekt môžeš získať **5 bodov**.