Triedy a pokročilé aspekty Pythonu

Osnova

- Triedy
  - Tvorba nových tried
  - Magické metódy
  - Dedičnosť
- List comprehensions
- Pokročilé používanie funkcií
  - Funkcie s ľubovoľným počtom argumentov
  - Docstring
  - `*args` a `**kwargs`
- Moduly a balíčky

Trieda a typ

Pre Python sú slová *trieda* a *typ* viac-menej ekvivalentné pojmy.

- `1` je z triedy/typu `int`
- `[1,2]` je z triedy/typu `list`

Konkrétne realizácie (inštancie) triedy sú objekty.

(Pre Python sú slová *objekt* a *inštancia* viac-menej ekvivalentné pojmy.)

Trieda je teda nejaký vzor pre objekt - určuje ako sa objekt chová. Napr.:

- operátor + je definovaný pre oba typy `int` a `list`, ale chová sa inak

Trieda

Trieda teda určuje chovanie objektu:

- chovanie operátorov
- definuje metódy
- definuje asociované dáta

Vlastné triedy

Python vám dovoľuje vytvoriť si vlastné triedy:

```
class Vector:
    x = 4
    y = 3

    def scale(self, scalar):
        self.x = self.x * scalar
        self.y = self.y * scalar

v = Vector()
print(v.x, v.y) # 4 3

v.scale(2)
print(v.x, v.y) # 8 6
```

Čo je vlastne metóda?

Metóda je funkcia, ktorá je asociovaná k triede.

Metódu je možné zavolať dvoma spôsobmi:

```
str.replace('A D C', 'D', 'B')
```

alebo

```
'A D C'.replace('D', 'B')
```

Druhý spôsob vidíte častejšie.

Metóda je teda funkcia, ktorej ako prvý argument dáme objekt, pre ktorý je metóda definovaná.

Inicializácia triedy

Na inicializáciu je špeciálna metóda `__init__`.

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

v = Vector(1.0, 3.0)
print(v.x, v.y) # 1.0 3.0
```

Magic methods (Dunder methods)

Iné tzv. magické metódy existujú pre rôzne aplikácie, najčastejšie pre definíciu chovania operátorov a základných funkcií (`len`, `str`,...) – Kompletný zoznam.

Sčítanie vektorov:

```
class Vector:
    def __init__(self, x, y):
        ...
    def __add__(self, other):
        x_new = self.x + other.x
        y_new = self.y + other.y
        return Vector(x_new, y_new)

v1 = Vector(1.0, 3.0)
v2 = Vector(-1.0, 3.0)
v3 = v1 + v2
print(v3.x, v3.y) # 0.0 6.0
```

Cvičenie 1

Naša trieda `Vector` má implementovanú metódu `scale`, ktorá vynásobí jednotlivé komponenty daného vektora argumentom `scalar`.

```
class Vector:
    ...
    def scale(self, scalar):
        self.x = self.x * scalar
        self.y = self.y * scalar


v = Vector(1, 3)
```

Nahraďte túto metódu magickou metódou `__mul__` (nápoveda tu) tak, aby sme namiesto

```
v.scale(2)
```

mohli písať

```
v * 2
```

Čo keď chcem vytvoriť novú triedu `VectorQuantity`? Určite zdieľa veľa funkcionality s `Vector`, ale predsa nebudem kopírovať kód...

Dedičnosť

Dedičnosť (inheritance) označuje, že novú triedu nevytvárame z nuly, ale využijeme už existujúcu triedu.

```
class VectorQuantity(Vector):
    unit = 'm/s'


v = VectorQuantity(2, 5)
print(v.x, v.y) # 2 5
print(v.unit) # m/s
```

V tomto prípade je:

- `Vector` tzv. `parent class`
- `VectorQuantity` tzv. `child class`.

Inicializácia pri dedení

```
class VectorQuantity(Vector):
    def __init__(self, x, y, unit):
        Vector.__init__(self, x, y) # inicializácia rodiča
        self.unit = unit

v = VectorQuantity(2, 5, 'm/s')
print(v.x, v.y) # 2 5
print(v.unit) # m/s
```

Cvičenie 2

Vytvorte triedu `Particle`, ktorá bude mať atribút `mass`. Potom vytvorte triedu `ChargedParticle`, ktorá podedí z `Particle`, a bude mať navyše atribút `charge`. Otestujte, že triedy je možné takto použiť:

```
p = Particle(1.67e-27)
print(p.mass) # 1.67e-27

cp = ChargedParticle(1.67e-27, 1.6e-19)
print(cp.charge) # 1.6e-19
print(cp.mass) # 1.67e-27
```

Funkcia `dir`

Vypíše čo objekt obsahuje (metódy, dáta):

```
v = Vector(1, 2)
print(dir(v))
```

Výhody tried

- Modulárnosť: Triedy umožňujú programátorom rozdeliť program do menších a zrozumiteľnejších častí. To uľahčuje údržbu kódu a jeho rozšíriteľnosť.
- Opätovné použitie: Triedy umožňujú programátorom znovu použiť kód, ktorý bol už vytvorený a overený v inom kontexte. To ušetrí čas a úsilie a znižuje riziko chýb.
- Abstrakcia: Triedy umožňujú abstrakciu zložitých systémov a návrh programu na vyššej úrovni. To umožňuje programátorom sústrediť sa na funkčnosť programu a nezaoberať sa podrobne jeho vnútornou implementáciou.
- Zapuzdrenie: Triedy umožňujú programátorom skryť podrobnosti o svojom kóde a vytvoriť tak zložitý systém, ktorý je jednoduchý na použitie a zrozumiteľný.

Nevýhody tried

- Komplexnosť: Triedy môžu byť veľmi zložité a ťažké na pochopenie, ak sa používajú neefektívne. Programátori musia starostlivo navrhovať triedy, aby boli jednoduché na použitie a zrozumiteľné.
- Výkon: Používanie tried môže viesť k zníženiu výkonu, pretože triedy musia byť inicializované a triedy musia byť prechádzané cez viacero funkcií.

Moduly a balíčky

- Modul v Pythone je jednoduchý textový súbor s Pythoním kódom a príponou `.py`
- Modul je možné importovať pomocou kľúčového slova `import`:

```
import názov_súboru_bez_koncovky
```

- Balíček je kolekcia modulov.

Ako organizovať balíček a viac o moduloch v dokumentácií Pythonu.

Cvičenie 3

Skopírujte triedu `Particle`, ktorú ste vytvorili v predchádzajúcom cvičení, a vložte ju do nového pythonieho modulu `particle.py`.

Overte, že sa dá `Particle` importovať a použiť:

```
from particle import Particle

p = Particle(5e-27)
print(p.mass)
```

List comprehensions

Namiesto:

```
lst = []
for i in range(10):
    lst.append(i**2)
```

je možné v Pythone napísať:

```
lst = [ i**2 for i in range(10) ]
```

Pre matematikov:

$$L = \{i^2 | i \in \mathbb{Z}, 0 \leq i < 10\}$$

List comprehensions

Namiesto:

```
lst = []

for i in range(10):
    if i % 2 == 0:
        lst.append(i**2)
```

je možné použiť:

```
lst = [ i**2 for i in range(10) if i % 2 == 0 ]
```

Funkcia `help`

Ak chcete dokumentáciu a nechcete chodiť na internet:

```
help(len)
```

V interaktívnych Pythonoch (`IPython`, `Jupyter Notebook`) je skratka:

```
?len
```

alebo

```
len?
```

Docstring

Ak chcete mať definovaný `help` a vlastných funkciách:

```
def is_prime(x):
    """Determines whether a number is a prime number.
       - Input: `x` - a positive integer.
       - Output: `True` or `False`.
    """


    if x == 1:
        return False
    for i in range(2, x):
        if x % i == 0:
            return False
    return True
```

Reťazcu na začiatku funkcie sa hovorí *docstring* (documentation string).

```
help(is_prime)
```

Argumenty funkcie vo väčšom detaile

K argumentu vo funkcií je možné pristúpiť aj pomocou jeho mena:

```
def f(a, b):
    print(a, b)

f(1, 2)         # 1 2
f(a=1, b=2)     # 1 2
f(b=1, a=2)     # 2 1
```

Argumenty funkcie vo väčšom detaile

Je možné prednastaviť hodnoty argumentov:

```
def f(a, b=1):
    print(a, b)


f(3)      # 3 1
f(1,2)    # 1 2
f(a=4)    # 4 1
```

Toto ale nefunguje (`SyntaxError`):

```
def f(a=1, b):
    ...
```

Prednastavené argumenty musia byť **za** neprednastavenými argumentmi.

*args

Funkcie je možné definovať tak, aby brali ľubovoľný počet parametrov:

```
def print_args(*args):
    print(type(args)) # tuple
    for i in args:
        print(i)
```

`args` je len (štandardne zaužívaný) názov premennej, iný názov funguje tiež:

```
def print_args(*x):
    ...
```

*args

Je možné to kombinovať s inými argumentami:

```
def add_to_list(lst, *args):
    for i in args:
        lst.append(i)
    print(lst)

x = [1, 2]
add_to_list(x, 3, 4, 'abc')
```

**kwargs

Ak chcete variabilný počet argumentov, ale zároveň vám záleží na mene argumentu:

```
def print_kwargs(**kwargs):
    print(type(kwargs)) # dict
    for i in kwargs:
        print(i, kwargs[i])

print_kwargs(a=1, b=2, c='red')
```

Znovu, `kwargs` je len (štandardne zaužívaný) názov premennej, iný názov funguje tiež. `kwargs` je skratka pre *keyword arguments*.

**kwargs

Znova, rôzne kombinácie sú dovolené:

```
def print_all(a, *args, b=33, **kwargs):
    print(a, b)
    for i in args:
        print('arg:', i)
    for k, v in kwargs.items():
        print('kwarg', k, v)

print_all(1, 2, 3, 4, c='red', d=5)
```