Pokročilé aspekty Pythonu

Veci, ktoré sme nestihli

. . . ale je dobré vedieť, že existujú.

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

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) ]
```

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)

x = [1, 2]
add_to_list(l, 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)
```

Trieda a typ

Nasledujúce slidy "preletia v rýchlosti" nad konceptom triedy – triedy sú relatívne komplikované.

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 MyLittleClass:
    def my_method(self, arg1, arg2):
        ...

my_object = MyLittleClass()
my_object.my_method(1, 2)
```

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

p = Point(1.0, 3.0)
print(p.x, p.y)
```

Magické metódy

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

Sčítanie bodov:

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

p1 = Point(1.0, 3.0)
p2 = Point(-1.0, 3.0)
p3 = p1 + p2
print(p3.x, p3.y)
```

Funkcia `dir`

Vypíše čo objekt obsahuje (metódy, dáta):

```
p = Point(1, 2)
print(dir(p))
```

Výhody tried

Malo by byť jasné, že triedy nutne nepotrebujete – funkcie a základné dátové typy v bohate stačia.

Výhody používania tried:

- Organizácia kódu a kreativita - trieda je vyššia abstrakcia, ktorá spája funkcie a dáta.
- Ľahké vyhľadávanie - metódy ľahko nájdete, funkcie môžu byť kdekoľvek.
- Dokumentácia - docstringy môžete dať pod jednotlivé metódy a triedy.
- Recyklácia kódu pomocou *dedičnosti*.
- Polymorfizmus a *Duck typing - keď to kváka ako kačka, je to kačka*:

```
class Swan:
    def quack(self):
        ...
```

```
class Duck:
    def quack(self):
        ...
```

Nevýhody tried

- Dodatočná komplexita.
- Výber toho, aké triedy implementujete, a ako spolu budú interagovať nemusí byť jednoznačný. Je viac možných abstrakcií pre každý daný problém, a ak zvolíte zlú, tak si to uvedomíte väčšinou neskoro.

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.