Metodika - Code style Python

Užitočné linky:

- Skrátený výcuc z používaného Python codestylu
- Typovanie v Pythone
- Docstring v Pythone

Verzia Pythonu

Projekt používa Python3. Konkrétne **3.10**. Nie je to ešte úplne stable verzia - napríklad sa stáva v Python 3.10 shelli, že nefunguje história, ale to nie je problém, keďže v tomto projekte nie je potrebné riešiť interaktívne veci. 3.10 so sebou ale prináša ďalšie cool featurky, hlavne:

- Výpovednejšie Error Message
- Nové featurky pre Type Hinting
- Structural Pattern Matching

Code style - čoho sa držať

Oficiálny code style Pythonu je PEP 8. Problém je, že je príliš dlhý a čítanie by človeku zabralo aj 30 minút, no aj tak by si z toho asi veľa nezapamätal. Preto z neho existuje výcuc a z toho ďalší výcuc.

A z toho je potrebné si zobrať hlavne tieto body, ktorých nedodržanie môže mať za následok odseknutie končatín samotným scrum masterom:

Základ

```
# Good:
result = some_function_that_takes_arguments(
    'argument one,
    'argument two',
    'argument three'
)

long_foo_dict_with_many_elements = {
    'foo': 'cat',
    'bar': 'dog'
}

with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())

income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)

# Bad:
result = some_function_that_takes_arguments(
'argument one,
'argument two', 'argument three')
result2 = some_function_that_takes_arguments('argument one', 'argument two', 'argument three')
```

1. Always use 4 spaces for indentation (don’t use tabs)
    a. Pozor! Toto neznamená, že bude potrebné ťukať vkuse do medzerníka. V editore si vie človek nastaviť, že používa na odsadzovanie 4 medzery a pritom normálne používa taby, keď píše kód, iba editor mu to rovno mení na medzery.

2. Max line-length: 80 characters (especially in comments)
3. Always indent wrapped code for readability

Importy

```
# Good:
import os # STD lib imports first
import sys # alphabetical

from mypkg.sibling import example # 3rd party stuff next
from subprocess import Popen, PIPE # Acceptable

from .sibling import example # local stuff last

# Bad:
import os, sys # multiple packages
import sibling # local module without "."
from mypkg import * # wildcards
```

4. Don’t use wildcards (from xxx import *)
5. Don’t import multiple packages per line
6. Import standard libs first, 3rd-party libs next, and local stuff last
7. Try to import in a alphabetical order
8. When using relative imports, be explicit (with .)

Medzery a riadky

```
# Good:
spam(ham[1], {eggs: 2})

if x == 4:
    print x, y
    x, y = y, x
dict['key'] = list[index]

y = 2
long_variable = 3
hypot2 = x*x + y*y
c = (a+b) * (a-b)

def complex(real, imag=0.0):
    return magic(r=real, i=imag)

do_one()
do_two()

# Bad
spam ( ham[ 1 ], { eggs: 2 } ) # spaces inside brackets
if x == 4 : print x , y ; x , y = y , x # inline statements, space before commas
dict ['key'] = list [index] # space before dictionary key
y            = 2 # Using spaces to line up assignment operators
long_variable = 3
hypot2 = x * x + y * y # Too much space around operators
c = (a + b) * (a - b) # Too much space around operators

def complex(real, imag = 0.0):
    return magic(r = real, i = imag) # Spaces in default values
```

9. 2 blank lines before top-level function and class definitions
10. 1 blank line before class method definitions
11. Use blank lines in functions sparingly
12. Don’t use whitespace to line up assignment operators (=, :)
13. Spaces around = for assignment
14. No spaces around = for default parameter values
15. Spaces around mathematical operators, but group them sensibly
16. Multiple statements on the same line are discouraged

Komentáre

```
def my_function():
    """ A one-line docstring """

def my_other_function(parameter=False):
    """
    A multiline docstring.

    Keyword arguments:
    parameter -- an example parameter (default False)

    """
```

17. Keep comments up to date - incorrect comments are worse than no comments
18. Try to write in plain and easy-to-follow English
19. Use inline comments sparingly & avoid obvious comments
20. Each line of block comments should start with #
21. Docstring one-liners can be all on the same line
22. In docstrings, list each argument on a separate line
23. Docstrings should have a blank line before the final """

Názvy

```
A_CONSTANT = 'ugh.'

class MyClass:
    """ A purely illustrative class """

    __property = None

    def __init__(self, property_value):
        self.__property = property_value

    def get_property(self):
        """ A simple getter for "property" """

        return self.__property

    @classmethod
    def default(cls):
        instance = MyClass("default value")
        return instance
```

24. Class names in CapWords
25. Method, function and variables names in lowercase_with_underscores
26. Private methods and properties start with __double_underscore
27. “Protected” methods and properties start with _single_underscore
28. Use all-uppercase FIXED_TERM for constant variables
29. Always use self for the first argument to instance methods

Ešte niečo k deleniu kódu na viacero riadkov

Na obrázku je vyjavený príklad polovične pekného kódu. Nie je to 300-znakový riadok, ale je tam pekne rozdelený dictionary do 3 riadkov. Avšak, prvý riadok dictionary má viac ako 80 znakov. Netreba sa teda báť rozdeliť napríklad list comprehension.

```
# BAD
index[term] = {
    'collection_frequency': sum([song['frequency'] for song in posting_lists[term]]),
    'document_frequency': len(posting_lists[term]),
    'posting_list': copy.deepcopy(posting_lists[term])
}
```

Tuto je rovnaký kód už korektne naformátovaný:

```
# GOOD
index[term] = {
    'collection_frequency': sum([
        song['frequency'] for song in posting_lists[term]
    ]),
    'document_frequency': len(posting_lists[term]),
    'posting_list': copy.deepcopy(posting_lists[term])
}
```

Na tomto obrázku je možné vidieť príklad toho, že aj dictionary, ktorého niektoré key:value páry by sa zmestili do jedného riadka, je lepšie rozdeliť do osobitných riadkov. K tomuto je dobré si zapamätať ešte 2 veci:

1. otváracia zátvorka sa nachádza na konci predošlého riadka, zatváracia zátvorka je zarovnaná na úroveň riadka s otváracou a telo objektu je odsadené o jednu úroveň
2. v takomto formáte je vhodné písať čiarku aj za posledný prvok. Nijako to neovplyvňuje funkcionalitu, ale uľahčuje to ďalšie potenciálne doplnenie alebo zmenu poradia v danom objekte

```
labels: dict[str, str] = {
    'chorus': 'chorus',
    'pre-chorus': 'chorus',
    'verse': 'verse',
    'ver': 'verse',
    'pre-verse': 'verse',
    'bridge': 'bridge',
}
```

Na záver máme na dvoch obrázkoch znázornené nesprávne a správne volanie funkcie s dlhým a komplexným zoznamom parametrov.

```
# BAD
join_datasets(outdir=DATA, indirs=[TEST_JSON_EN_DIR, TRAIN_JSON_DIR], pipe=[lambda x: get_fields(x, ( 'Artis

join_datasets(outdir=DATA, indirs=[TEST_JSON_EN_DIR, TRAIN_JSON_DIR],
pipe=[lambda x: get_fields(x, ( 'Artist', 'Song', 'Genre', 'Lyrics')),
remove_trash, lambda x: split_song_to_verses(x, 70), syllable_song,
lemmatize_song, split_song_to_parts])

join_datasets(outdir=DATA, indirs=[TEST_JSON_EN_DIR, TRAIN_JSON_DIR],
    pipe=[lambda x: get_fields(x, ( 'Artist', 'Song', 'Genre', 'Lyrics')),
    remove_trash, lambda x: split_song_to_verses(x, 70), syllable_song,
    lemmatize_song, split_song_to_parts])

join_datasets(
    outdir=DATA,
    indirs=[
    TEST_JSON_EN_DIR,
    TRAIN_JSON_DIR,
    ],
    pipe=[
    lambda x: get_fields(
    x, (
    'Artist',
    'Song',
    'Genre',
    'Lyrics',
    )
    ),
    remove_trash,
    lambda x: split_song_to_verses(x, 70),
    syllable_song,
    lemmatize_song,
    split_song_to_parts,
    ]
)
```

```
# GOOD
join_datasets(
    outdir=DATA,
    indirs=[
        TEST_JSON_EN_DIR,
        TRAIN_JSON_DIR,
    ],
    pipe=[
        lambda x: get_fields(
            x, (
                'Artist',
                'Song',
                'Genre',
                'Lyrics',
            )
        ),
        remove_trash,
        lambda x: split_song_to_verses(x, 70),
        syllable_song,
        lemmatize_song,
        split_song_to_parts,
    ]
)
```

Docstring

Predpísaný formát, ktorým sa v Pythone vysvetľujú objekty (funkcie, classy, všetko), ich parametre, atribúty, návratové hodnoty a podobne. Nie je to vôbec dlhý článok, je fajn si to rovno na tej stránke.

Mal by existovať v každej jednej funkcii na začiatku, teda aspoň jednoriadkový opis, čo robí tá funkcia.

Typovanie

Áno, Python v podstate nepozná typy premenných… ale je možné mu to aj tak povedať. Kód je potom robustnejší, lebo programátor má explicitne špecifikované, aký typ chce kam poslať. Malo by byť použité určite v hlavičkách funkcií. Optimálne aj vo zvyšku kódu.

Niečo od autorov PEP 484:

“This PEP aims to provide a standard syntax for type annotations, opening up Python code to easier static analysis and refactoring, potential runtime type checking, and (perhaps, in some contexts) code generation utilizing type information.

Of these goals, static analysis is the most important. This includes support for off-line type checkers such as mypy, as well as providing a standard notation that can be used by IDEs for code completion and refactoring."

Viaže sa k tomu PEP 484 a PEP 526. PEPy sú extrémne dlhé, preto existuje napr. tento cheatsheet. Vyzerá to v zásade nejako takto:

```python
# This is how you annotate a function definition
def stringify(num: int) -> str:
    return str(num)

# And here's how you specify multiple arguments
def plus(num1: int, num2: int) -> int:
    return num1 + num2
```

```python
# Use Union when something could be one of a few types
x: List[Union[int, str]] = [3, 5, "test", "fun"]

# Use Any if you don't know the type of something or it's too
# dynamic to write a type for
x: Any = mystery_function()

# If you initialize a variable with an empty container or "None"
# you may have to help mypy a bit by providing a type annotation
x: List[str] = []
x: Optional[str] = None

# This makes each positional arg and each keyword arg a "str"
def call(self, *args: str, **kwargs: str) -> str:
    request = make_request(*args, **kwargs)
    return self.do_api_query(request)
```

Odporúčané rozšírenia pre IDE

K typovaniu sú vhodné tieto dve rozšírenia:

- PyRight
- Python Type Hint

Vo VS Code je ich možné nájsť normálne v extensions. Pre JetBrains určite existujú tiež.

Okrem toho je vhodné mať nainštalovaný nejaký klasický Python syntax highlighter.