Dobre praktyki

PEP20 - Zen of Python

```
1 import this
```

The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Zen Pythona

- Piękne jest lepsze niż brzydkie.
- Wyrażone wprost jest lepsze niż domniemane.
- Proste jest lepsze niż złożone.
- Złożone jest lepsze niż skomplikowane.
- Płaskie jest lepsze niż wielopoziomowe.
- Rzadkie jest lepsze niż gęste.
- Czytelność się liczy.
- Sytuacje wyjątkowe nie są na tyle wyjątkowe, aby łamać reguły.
- Choć praktyczność przeważa nad konsekwencją.
- Błędy zawsze powinny być sygnalizowane.
- Chyba że zostaną celowo ukryte.
- W razie niejasności powstrzymaj pokusę zgadywania.
- Powinien być jeden -- i najlepiej tylko jeden -- oczywisty sposób na zrobienie danej rzeczy.
- Choć ten sposób może nie być oczywisty jeśli nie jest się Holendrem.
- Teraz jest lepsze niż nigdy.

- Chociaż nigdy jest często lepsze niż natychmiast.
- Jeśli rozwiązanie jest trudno wyjaśnić, to jest ono złym pomysłem.
- Jeśli rozwiązanie jest łatwo wyjaśnić, to może ono być dobrym pomysłem.
- Przestrzenie nazw to jeden z niesamowicie genialnych pomysłów -- miejmy ich więcej!

PEP8

Wcięcia

Tak:

```
# Wyrównany z otwarciem ogranicznika.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Dodano więcej wcięć, aby odróżnić to od reszty.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Nie:

```
# Argumenty w pierwszym wierszu są zabronione, gdy nie używa się pionowego
wyrównania.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Dalsze wcięcia wymagane jako wcięcia nie są rozróżnialne.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Zamykanie nawiasów

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

Lub:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Lub:

```
my_list = [
    1, 2, 3,
    4, 5, 6]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f')
```

Tabulacje i czy spacje?

Głębokość wcięć równa czterem spacjom.

Długość linii

To jest dość kontrowersyjna klauzula mówiąca o tym, że długość linii powinna być nie dłuższa niż 79 znaków. Przy obecnych wielkich szerokokątnych monitorach jest to dość uciążliwe. Jednakże należy przestrzegać konwencji.

Linie możemy łamać poprzez stawianie znaku ukośnika `\` na końcu:

```
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())
```

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):

        if (width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")

        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
```

```
                                (width, height))

        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Puste linie

Kodowanie plików

Przy Pythonie 3 kodownaie plików powinno być w UTF-8.

Importy

Importy powinny być poukładane alfabetycznie w grupach. Na górze importy z bibliotek standardowych Pythona. Następnie linia przerwy i zewnętrzne zależności. Znów linia przerwy i zależności wewnątrz Twoich aplikacji.

Każdy z importów powinien być w osobnej linii.

Tak:

```
import os
import sys
```

Nie:

```
import sys, os
```

Ale można:

```
from subprocess import Popen, PIPE
```

Cudzysłowia

Python interpretuje cudzysłowia pojedyncze i podwójne tak samo. Ważne jest aby wybrać jeden sposób i konsekwentnie się go trzymać w całej aplikacji.

Białe spacje w wyrażeniach

Tak:

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

ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]

spam(1)
```

```

dct['key'] = lst[index]

x = 1
y = 2
long_variable = 3

i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

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

def munge(input: AnyStr):
def munge(sep: AnyStr = None):
def munge() -> AnyStr:
def munge(input: AnyStr, sep: AnyStr = None, limit=1000):

if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Nie:

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

ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]

spam (1)

dct ['key'] = lst [index]

x             = 1
y             = 2
long_variable = 3

i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)

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

```

```
def munge(input: AnyStr=None):
def munge(input:AnyStr):
def munge(input: AnyStr)->PosInt:

if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()

if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

Komentarze

Google style comments

Konwencje nazewnicze

- `zmienne`
- `STALE`
- `NazwyKlas`
- `nazwy_metod()` i `nazwy_funkcji()`
- `nazwymodulow`, `nazwy_modulow`
- `self`
- `cls`

Używanie `___` i `_` w nazwach

Konstrukcje warunkowe

Yes:

```
if foo is not None:
```

No:

```
if not foo is None:
```

Zwracanie z funkcji

Tak:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

Nie:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)
```

Sprawdzanie warunków

Tak:

```
if not seq:
if seq:

if greeting:
```

Nie:

```
if len(seq)
if not len(seq)

if greeting == True:
if greeting is True:
```

Korzystanie z `help()`, `dir()` i `object.__dict__`

Kilka przykładów z praktyki

`html.append('<tag>')`

```
## Wydajność - metoda łączy ciągi znaków za pomocą + w pętli
def make_html1(lista):
    html = '<table>'
```

```
    for element in lista:
        html += '<tr><td>%s</td></tr>' % element
    html += '</table>'

    return html

## Problem rozwiązany - tak jest lepiej
def make_html2(lista):
    html = ['<table>']

    for element in lista:
        html.append('<tr><td>%s</td></tr>' % element)

    html.append('</table>')
    return '\r\n'.join(html)
```

Magic number i Magic string

Passowords

- Sticky bit
- setuid
- configparser

Wczytywanie konfiguracji programów

- configparser

Wersjonowanie API

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding:gzip, deflate, sdch
Accept-Language:en-US,en;q=0.8,pl;q=0.6
```