нетология
центр онлайн-образования

ZEN OF PYTHON

МегаФон

Кирилл Табельский

chigiwar

- 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!

- Красивое лучше уродливого.
- Явное лучше неявное.
- Простое лучше сложного.
- Сложное лучше запутанного.
- Плоское лучше, вложенного.
- Разреженное лучше плотного.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если они не замалчиваются явно.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один и, желательно, только один очевидный способ сделать это.
- Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда.
- Хотя никогда зачастую лучше, чем *прямо* сейчас.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имен — отличная штука! Будем делать их больше!

Whitespaces

- 4 spaces per indentation level.
- No hard tabs.
- Never mix tabs and spaces.
- One blank line between functions.
- Two blank lines between classes.

Whitespaces

- Add a space after “,” in dicts, lists, tuples & argument lists & after “:” in dicts, but not before.
- Put spaces around assignments & comparisions (except in argument lists).
- No spaces just inside parentheses or just before argument lists.
- No spacec just inside docstrings.

```
def make_squares(key, value=0):
    """Return a dictionary and a list..."""
    result_dict = {key: value}
    result_list = [key, value]
    return result_dict, result_list
```

Naming

- **joined_lower** for functions, methods, attributes.
- **joined_lower** or **ALL_CAPS** for constants.
- **StudlyCaps** for classes.
- **camelCase** only to conform to pre-existing conventions.
- Attributes: **interface**, **_internal**, **__private**
- But try to avoid the **__private** form.

Long Lines & Continuations

- Keep lines below 80 (or 120) characters in length.
- Use implied line continuation inside parentheses/brackets/braces:

```
def __init__(self, first, second,
             third, fourth, fifth, sixth):
    output = (first + second + third
              + fourth + fifth + sixth)
    my_value = my_function(
        first, second, third,
        fourth, fifth, sixth
    )
```

Long Lines & Continuations

- Use backslashes as a last resort:

```
VeryLong.left_hand_side \
    = even_longer.right_hand_side()
```

- Multiline strings use triple quotes:

```
"""Triple
double
quotes"""
```

Compound Statements

- Good:

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

- Bad:

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

Docstrings & Comments

Docstrings = How to use code
Comments = Why (rationale) & how code works

- Docstrings explain how to use code, and are for the users of your code. Uses of docstrings:
  - Explain the purpose of the function even if it seems obvious to you, because it might not be obvious to someone else later on.
  - Describe the parameters expected, the return values, and any exceptions raised.
  - If the method is tightly coupled with a single caller, make some mention of the caller (though be careful as the caller might change later).

Docstrings & Comments

- Comments explain why, and are for the maintainers of your code. Examples include notes to yourself, like:

```
# !!! BUG: ...
# !!! FIX: This is a hack
# ??? Why is this here?
```

Practicality Beats Purity

- When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
- To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

Swap values

- In other languages:

```
temp = a
a = b
b = temp
```

- In Python:

```
b, a == a, b
```

```
foo = ['David', 'Pythonista', '9-00-34’]
name, title, phone = foo
```

Tuple comma

```
value = 1,
print(value)  # (1,)
```

- If you see a tuple where you don’t expect one, look for a comma!

Interactive “_”

- Interactive interpreter:

```
>>> 1 + 1
2
>>> _
2
```

_ stores the last printed expression.

Build Strings from Substrings

```
colors = ['red', 'blue', 'green', 'yellow’]
```

We want to join all the strings together into one large string. Especially when the number of substrings is large…

```
# Don’t do this:
result = ’’
for s in colors:
    result += s
```

This is very inefficient. It has terrible memory usage and performance patterns. The “summation” will compute, store, and then throw away each intermediate step.

```
# Do this instead:
result = ''.join(colors)
```

Use in where possible

- Good:

```
for key in d:
    print key
```

- **in** is generally faster.
- This pattern also works for items in arbitrary containers (such as lists, tuples and sets).

- Bad:

```
for key in d.keys():
    print key
```

- This is limited to objects with a keys() method.

Dictionary get method

Bad

```
navs = {}
for (portfolio, equity, position) in data:
    if portfolio not in navs:
        navs[portfolio] = 0
        navs[portfolio] += position * prices[equity]
```

dict.get(key, default) removes the need for the test:

Good

```
navs = {}
for (portfolio, equity, position) in data:
    navs[portfolio] = (navs.get(portfolio, 0)
                       + position * prices[equity])
```

Building & splitting dictionaries

- Interactive interpreter:

```
>>> given = ['John', 'Eric', 'Terry', 'Michael']
>>> family = ['Cleese', 'Idle', 'Gilliam', 'Palin']
>>> pythons = dict(zip(given, family))
>>> pprint.pprint(pythons)
{'John': 'Cleese',
 'Eric': 'Idle',
 'Terry': 'Gilliam',
 'Michael': 'Palin'}
```

Testing for Truth Values

```
# do this:
if x:
    pass
```

```
# not this:
if x == True:
    pass
```

```
# do this:
if items:
    pass
```

```
# not this:
if len(items) != 0:
    pass

# and definitely not this:
if items != []:
    pass
```

Truth Values

FALSE	TRUE
False (== 0)	True (== 1)
"" (empty string)	any string but "" (" ", "anything")
0, 0.0	any number but 0 (1, 0.1, -1, 3.14)
[], (), {}, set()	any non-empty container ([0], (None,), [''])
None	almost any object that's not explicitly False

Index & Item

- Split items:

```
>>> items = 'zero one two three'.split()
>>> print(items)
['zero', 'one', 'two', 'three']
```

- Get indexes:

```
>>> print(list(enumerate(items)))
[(0, 'zero'), (1, 'one'), (2, 'two'), (3, 'three')]
```

Formatting

- Use **.format()** method not **%**

```
x = 1
y = 3
z = x + y
text = '{0} + {1} = {3}'
foo = text.format(x, y, z)
print(foo) # 1 + 3 = 4
```

```
# or you can:
args = [x, y, z]
gee = text.format(*args)
print(gee) # 1 + 3 = 4
```

```
# or even better:
foo = f'{x} + {y} = {z}'
print(foo) # 1 + 3 = 4
```

List comprehensions

- The traditional way, with for and if statements:

```
new_list = []
for item in a_list:
    if condition(item):
        new_list.append(fn(item))
```

- As a list comprehension:

```
new_list = [fn(item) for item in a_list
            if condition(item)]
```

List comprehensions

- For example, a list of the squares of 0-9:

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- A list of the squares of odd 0-9:

```
>>> [n ** 2 for n in range(10) if n % 2]
[1, 9, 25, 49, 81]
```

try/except/finally

- Use this construction:

```
try:
    do_something
except ValueError:
    catch_exception()
else:
    hail_success()
finally:
    do_smth_anyway()
```

Don’t reinvent the wheel

- Check Python’s standard library.
- Check the Python Package Index: https://pypi.org
- Search the web. Google (Yandex) is your friend.

- https://www.python.org/dev/peps/pep-0008
- https://www.python.org/dev/peps/pep-0020
- https://david.goodger.org/projects/pycon/2007/idiomatic/handout.html

нетология
центр онлайн-образования

Спасибо за внимание!