



PEP 8 – руководство по написанию кода

Этот документ описывает соглашение о том, как писать код для языка Python, включая стандартную библиотеку, входящую в состав Python.

Этот документ и PEP 257 (Docstring Conventions) были адаптированы из оригинальных рекомендаций Гвидо ван Россума "Руководство по стилю Python", с некоторыми дополнениями из руководства по стилю Барри.

Многие проекты имеют свои собственные руководства по стилю кодирования. В случае возникновения каких-либо противоречий, такие руководства имеют приоритет для конкретного проекта.

Глупое постоянство – хобгоблин маленьких умов

Ключевая идея Гвидо такова: код читается намного больше раз, чем пишется. Приведенные здесь рекомендации призваны улучшить читабельность кода и сделать его единообразным в широком спектре кода Python. Как говорится в PEP 20, "Читабельность имеет значение". В идеале, весь код будет написан в едином стиле, и любой сможет легко его прочесть.

Это руководство о согласованности и единстве. Согласованность с этим руководством очень важна. Согласованность внутри одного проекта еще важнее. А согласованность внутри модуля или функции — самое важное. Но важно помнить, что иногда это руководство неприменимо, и понимать, когда можно отойти от рекомендаций. Когда вы сомневаетесь, просто посмотрите на другие примеры и решите, какой выглядит лучше. И не стесняйтесь спрашивать!

В частности: не нарушайте обратную совместимость только для того, чтобы соответствовать этому PEP!

Веские причины для того, чтобы нарушить данные правила:

1. Применение правила сделает код менее читаемым даже для того, кто привык читать код, который следует этим правилам
2. Чтобы писать в едином стиле с кодом, который уже есть в проекте и который нарушает правила (возможно, в силу исторических причин) — впрочем, это возможность переписать чужой код
3. Код, о котором идет речь, появился до введения этого руководства, и нет других причин для его изменения
4. Код должен оставаться совместимым со старыми версиями Python, которые не поддерживают функцию, рекомендуемую руководством по стилю

Содержание

- [Внешний вид кода](#)
 - [Отступы](#)
 - [Табуляция или пробелы?](#)
 - [Максимальная длина строки](#)
 - [Должна ли строка переноситься до или после бинарного оператора?](#)
 - [Пустые строки](#)
 - [Кодировка исходного файла](#)
 - [Импорты](#)
 - [Имена "дандеры" на уровне модуля](#)
- [Кавычки литерала строки](#)

- [Пробелы в выражениях и инструкциях](#)
 - [Домашние питомцы](#)
 - [Другие рекомендации](#)
- [Когда использовать запятые в конце строки](#)
- [Комментарии](#)
 - [Блоки комментариев](#)
 - [Внутристрочные комментарии](#)
 - [Строки документации](#)
- [Соглашения по именованию](#)
 - [Главный принцип](#)
 - [Описание: Стили наименования](#)
 - [Предписания: соглашения по именованию](#)
 - [Имена, которых следует избегать](#)
 - [ASCII совместимость](#)
 - [Имена модулей и пакетов](#)
 - [Имена классов](#)
 - [Имена переменных типа](#)
 - [Имена исключений](#)
 - [Имена глобальных переменных](#)
 - [Имена функций и переменных](#)
 - [Аргументы функций и методов](#)
 - [Имена методов и переменных экземпляров классов](#)
 - [Константы](#)
 - [Проектирование наследования](#)
 - [Публичные и внутренние интерфейсы](#)
- [Общие рекомендации](#)
 - [Аннотации функций](#)
 - [Аннотации переменных](#)

Внешний вид кода

Отступы

Используйте 4 пробела на каждый уровень отступа.

Продолжительные строки должны выравнивать обернутые элементы либо вертикально, используя неявную линию в скобках (круглых, квадратных или фигурных), либо с использованием висячего отступа. При использовании висячего отступа следует применять следующие соображения: на первой строке не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение.

Правильно:

```
# Выровнено по открывающему разделителю
foo = long_function_name(var_one, var_two,
                          var_three, var_four)
```

Добавлено 4 пробела (следующий уровень отступа) для выделения аргумента от остальных

```
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Неправильно:

```
# Аргументы на первой строке запрещены, если не используется вертикальное выравнивание
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

```
# Больше отступов требуется для выделения аргумента от остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Правило 4 пробелов опционально для строк-продолжений. Опционально:

```
# Всякие отступы *могут* быть отступом, отличным от 4 пробелов.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Если условная часть if-выражения достаточно длинная и требует написания нескольких строк, стоит отметить, что комбинация из двух символов ключевого слова (т. е. if), плюс один пробел, плюс открывающая скобка создает естественный отступ из 4 пробелов для последующих строк многострочного условного выражения. Это может привести к визуальному конфликту с отступом набора кода, вложенного внутри if-выражения, который также естественно будет отступать на 4 пробела. Данный PEP не занимает четкой позиции относительно того, как (и нужно ли) дополнительно визуально отличать такие условные строки от вложенного набора внутри if-выражения. Приемлемые варианты в этой ситуации включают, но не ограничиваются следующими примерами:

```
# Без дополнительных отступов
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
```

```
# Добавьте комментарий, который обеспечит некоторое разграничение в редакторах, поддерживаи
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
```

```
# Добавьте несколько дополнительных отступов в строке продолжении
if (this_is_one_thing
    and that_is_another_thing):
    do_something()
```

(Также см. ниже обсуждение вопроса о том, следует ли разбивать до или после бинарных операторов.)

Закрывающие круглые/квадратные/фигурные скобки в многострочных конструкциях могут находиться под первым непробельным символом последней строки списка, например:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
```

```
'd', 'e', 'f',  
)
```

либо быть под первым символом строки, начинающей многострочную конструкцию:

```
my_list = [  
    1, 2, 3,  
    4, 5, 6,  
]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

Табуляция или пробелы?

Пробелы - самый предпочтительный метод отступов.

Табуляция должна использоваться только для поддержки кода, написанного с отступами с помощью табуляции.

Python запрещает смешивание табуляции и пробелов в отступах.

Максимальная длина строки

Ограничьте длину строки максимум 79 символами.

Для более длинных блоков текста с меньшими структурными ограничениями (строки документации или комментарии), длину строки следует ограничить 72 символами.

Ограничение необходимой ширины окна редактора позволяет иметь несколько открытых файлов бок о бок, и хорошо работает при использовании инструментов анализа кода, которые предоставляют две версии в соседних столбцах.

Некоторые команды предпочитают большую длину строки. Для кода, поддерживающегося исключительно или преимущественно этой группой, в которой могут прийти к согласию по этому вопросу, нормально увеличение длины строки с 80 до 100 символов (фактически увеличивая максимальную длину до 99 символов), при условии, что комментарии и строки документации все еще будут 72 символа.

Стандартная библиотека Python консервативна и требует ограничения длины строки в 79 символов (а строк документации/комментариев в 72).

Предпочтительный способ переноса длинных строк является использование подразумеваемых продолжений строк Python внутри круглых, квадратных и фигурных скобок. Длинные строки могут быть разбиты на несколько строк, обернутые в скобки. Это предпочтительнее использования обратной косой черты для продолжения строки.

Обратная косая черта все еще может быть использована время от времени. Например, длинная конструкция with до Python версии 3.10 не может использовать неявные продолжения, так что обратная косая черта является приемлемой:

```
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())
```

Ещё один случай - assert.

Должна ли строка переноситься до или после бинарного оператора?

На протяжении десятилетий рекомендовалось разрывать строку после бинарных операторов. Но это может ухудшить читабельность в двух отношениях: операторы обычно разбросаны по разным колонкам экрана, и каждый оператор перемещается от своего операнда на предыдущую строку. Глазу приходится

проделявать дополнительную работу, чтобы определить, какие элементы складываются, а какие вычитаются:

Неправильно:

```
# Операторы находятся далеко от своих операндов
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

Чтобы решить эту проблему читабельности, математики и их издатели придерживаются противоположного соглашения. Дональд Кнут объясняет традиционное правило в своей серии "Компьютеры и набор текста": "Хотя формулы внутри абзаца всегда обрываются после бинарных операций и отношений, отображаемые формулы всегда обрываются перед бинарными операциями".

Следование традициям математики обычно приводит к созданию более читабельного кода:

Правильно:

```
# Легко сопоставлять операторы с операндами
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

В коде Python допускается перенос перед или после двоичного оператора, если это поведение согласовано на локальном уровне. Для нового кода рекомендуется использовать стиль Кнута.

Пустые строки

Отделяйте функции верхнего уровня и определения классов двумя пустыми строками.

Определения методов внутри класса разделяются одной пустой строкой.

Дополнительные пустые строки возможно использовать для разделения различных групп похожих функций. Пустые строки могут быть опущены между несколькими связанными однострочниками (например, набор фиктивных реализаций).


Используйте пустые строки в функциях, чтобы указать логические разделы.

Python расценивает символ control+L как незначащий (whitespace), и вы можете использовать его, потому что многие редакторы обрабатывают его как разрыв страницы — таким образом логические части в файле будут на разных страницах. Однако, не все редакторы распознают control+L и могут на его месте отображать другой символ.

Кодировка исходного файла

Кодировка Python должна быть UTF-8, и не должны содержать объявления кодировки.

В стандартной библиотеке, нестандартные кодировки должны использоваться только для целей тестирования.

Используйте символы, отличные от ASCII, редко, желательно только для обозначения мест и имен людей. Если в качестве данных используются не ASCII-символы, избегайте неприятных символов Unicode, таких как  и знаки порядка байтов.

Все идентификаторы в стандартной библиотеке Python ДОЛЖНЫ использовать идентификаторы только ASCII и ДОЛЖНЫ использовать английские слова везде, где это возможно (во многих случаях используются аббревиатуры и технические термины, которые не являются английскими).

Проектам с открытым исходным кодом, ориентированным на глобальную аудиторию, рекомендуется придерживаться аналогичной политики.

Импорты

- Каждый импорт, как правило, должен быть на отдельной строке.

Правильно:

```
import os
import sys
```

Неправильно:

```
import sys, os
```

В то же время, можно писать так:

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

- Импорты всегда помещаются в начале файла, сразу после комментариев к модулю и строк документации, и перед объявлением констант.

Импорты должны быть сгруппированы в следующем порядке:

1. импорты из стандартной библиотеки
2. импорты сторонних библиотек
3. импорты модулей текущего проекта

Вставляйте пустую строку между каждой группой импортов.

- Рекомендуется абсолютное импортирование, так как оно обычно более читаемо и ведет себя лучше (или, по крайней мере, даёт понятные сообщения об ошибках), если импортируемая система настроена неправильно (например, когда каталог внутри пакета заканчивается на `sys.path`):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Тем не менее, явный относительный импорт является приемлемой альтернативой абсолютному импорту, особенно при работе со сложными пакетами, где использование абсолютного импорта было бы излишне подробным:

```
from . import sibling
from .sibling import example
```

В стандартной библиотеке следует избегать сложной структуры пакетов и всегда использовать абсолютные импорты.

Неявные относительно импорты никогда не должны быть использованы, и были удалены в Python 3.

- Когда вы импортируете класс из модуля, вполне можно писать вот так:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

Если такое написание вызывает конфликт имен, тогда пишите:

```
import myclass
import foo.bar.yourclass
```

И используйте `"myclass.MyClass"` и `"foo.bar.yourclass.YourClass"`.

- wildcard-импорты (`from <module> import *`) следует избегать, так как они делают неясным то, какие имена присутствуют в глобальном пространстве имён, что вводит в заблуждение как читателей, так и многие автоматизированные средства. Существует один оправданный пример

использования шаблона импорта, который заключается в опубликовании внутреннего интерфейса как часть общественного API (например, переписав реализацию на чистом Python в модуле акселератора (и не будет заранее известно, какие именно функции будут перезаписаны)).

Имена "дандеры" на уровне модуля

"Дандеры" (т.е. имена с двумя ведущими и двумя завершающими подчеркиваниями), такие как `__all__`, `__author__`, `__version__` и т.д., должны располагаться после строки документации модуля, но перед любым выражением `import`, за исключением импортов `__future__`. Python предписывает, что импорты `__future__` должны появляться в модуле перед любым другим кодом, за исключением строки документации.

```
"""This is the example module.
```

```
This module does stuff.
"""
```

```
from __future__ import barry_as_FLUFL
```

```
__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'
```

```
import os
import sys
```

Кавычки литерала строки

В Python строки с одинарными и двойными кавычками - это одно и то же. В данном PEP нет рекомендаций на этот счет. Выберите правило и придерживайтесь его. Однако, если строка содержит символы одинарной или двойной кавычки, используйте вторую, чтобы избежать обратных слешей в строке. Это улучшает читаемость кода.

Для строк с тройными кавычками всегда используйте двойные кавычки, чтобы соответствовать соглашению о docstring в PEP 257.

Пробелы в выражениях и инструкциях

Домашние питомцы

Избегайте использования пробелов в следующих ситуациях:

- Непосредственно внутри круглых, квадратных или фигурных скобок.

Правильно:

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

Неправильно:

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

- Между запятой и последующей закрытой скобкой.

Правильно:

```
foo = (0,)
```

Неправильно:

```
bar = (0, )
```

- Непосредственно перед запятой, точкой с запятой или двоеточием.

Правильно:

```
if x == 4: print(x, y); x, y = y, x
```

Неправильно:

```
if x == 4 : print(x , y) ; x , y = y , x
```

- Однако в [срезах](#) двоеточие действует как бинарный оператор и должно иметь равное количество пробелов с обеих сторон (рассматривая его как оператор с наименьшим приоритетом). В расширенном срезе оба двоеточия должны иметь одинаковое количество интервалов. Исключение: если параметр среза опущен, пробел не ставится.

Правильно:

```
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]
```

Неправильно:

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

- Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции.

Правильно:

```
spam(1)
```

Неправильно:

```
spam (1)
```

- Сразу перед открывающей скобкой, после которой следует индекс или срез.

Правильно:

```
dict['key'] = list[index]
```

Неправильно:

```
dict ['key'] = list [index]
```

- Использование более одного пробела вокруг оператора присваивания (или любого другого) для того, чтобы выравнивать его с другим.

Правильно:

```
x = 1  
y = 2  
long_variable = 3
```

Неправильно:


```
x = 1
y = 2
long_variable = 3
```

Другие рекомендации

- Всегда окружайте эти бинарные операторы одним пробелом с каждой стороны: присваивания (=, +=, -= и другие), сравнения (==, <, >, !=, <>, ≤, ≥, in, not in, is, is not), логические (and, or, not).
- Если используются операторы с разными приоритетами, попробуйте добавить пробелы вокруг операторов с самым низким приоритетом. Используйте свои собственные суждения, однако, никогда не используйте более одного пробела, и всегда используйте одинаковое количество пробелов по обе стороны бинарного оператора.

Правильно:

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

Неправильно:

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

- Аннотации функций должны использовать обычные правила для двоеточий и всегда иметь пробелы вокруг стрелки ->, если она присутствует.

Правильно:

```
def munge(input: AnyStr): ...
def munge() -> PosInt: ...
```

Неправильно:

```
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

- Не используйте пробелы вокруг знака =, если он используется для обозначения именованного аргумента или значения параметров по умолчанию.

Правильно:

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

Неправильно:

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

Однако при объединении аннотации аргумента со значением по умолчанию используйте пробелы вокруг знака =.

Правильно:

```
def munge(sep: AnyStr = None): ...
```

```
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

Неправильно:

```
def munge(input: AnyStr=None): ...  
def munge(input: AnyStr, limit = 1000): ...
```

- Не используйте составные инструкции (несколько команд в одной строке).

Правильно:

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

Неправильно:

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

- Иногда можно писать тело циклов while, for или ветку if в той же строке, если команда короткая, но если команд несколько, никогда так не пишите. А также избегайте длинных строк!

Точно неправильно:

```
if foo == 'blah': do_blah_thing()  
for x in lst: total += x  
while t < 10: t = delay()
```

Вероятно, неправильно:

```
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()
```

Когда использовать запятые в конце строки

Конечные запятые обычно необязательны, но они обязательны при создании кортежа из одного элемента. Для ясности рекомендуется заключать их в (технически лишние) круглые скобки.

Правильно:

```
FILES = ('setup.cfg',)
```

Неправильно:

```
FILES = 'setup.cfg',
```

Хотя запятые в конце строки являются лишними, они часто полезны при использовании системы контроля версий, когда ожидается, что список значений, аргументов или импортируемых элементов будет расширяться с течением времени. Обычно каждое значение (и т. д.) помещается в отдельную строку, всегда добавляя запятую в конце, а закрывающая скобка добавляется на следующей строке.

Однако не имеет смысла располагать запятую в той же строке, что и закрывающий разделитель (за исключением описанного выше случая одноэлементных кортежей).

Правильно:

```
FILES = [  
    'setup.cfg',  
    'tox.ini',  
]  
initialize(FILES,  
            error=True,  
            )
```

Неправильно:

```
FILES = ['setup.cfg', 'tox.ini',]  
initialize(FILES, error=True,)
```

Комментарии

Комментарии, противоречащие коду, хуже, чем отсутствие комментариев. Всегда исправляйте комментарии, если меняете код!

Комментарии должны являться законченными предложениями. Если комментарий — фраза или предложение, первое слово должно быть написано с большой буквы, если только это не имя переменной, которая начинается с маленькой буквы (никогда не изменяйте регистр переменных!).

Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое предложение должно оканчиваться точкой.

В комментариях, состоящих из нескольких предложений, следует использовать один или два пробела после точки, завершающей предложение, за исключением последнего предложения.

Убедитесь, что ваши комментарии ясны и понятны людям, говорящим на языке, на котором вы пишете.

Программисты, из неанглоязычных стран: пожалуйста, пишите комментарии на английском, если только вы не уверены на 120%, что ваш код никогда не будут читать люди, не знающие вашего родного языка.

Блоки комментариев

Блок комментариев обычно объясняет код (весь, или только некоторую часть), идущий после блока, и должен иметь тот же отступ, что и сам код. Каждая строка такого блока должна начинаться с символа # и одного пробела после него (если только сам текст комментария не имеет отступа).

Абзацы внутри блока комментариев разделяются строкой, состоящей из одного символа #.

Внутристрочные комментарии

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. Внутристрочные комментарии должны отделяться по крайней мере двумя пробелами от инструкции. Они должны начинаться с символа # и одного пробела.

Комментарии в строке с кодом не нужны и только отвлекают от чтения, если они объясняют очевидное. Не пишите вот так:

```
x = x + 1 # Increment x
```

Впрочем, такие комментарии иногда полезны:

```
x = x + 1 # Компенсация границы
```

Строки документации

Соглашения по написанию хороших строк документации (они же "docstrings") увековечены в PEP 257.

- Пишите документацию для всех публичных модулей, функций, классов, методов. Строки документации необязательны для приватных методов, но лучше написать, что делает метод. Комментарий нужно писать после строки с `def`
- PEP 257 объясняет, как правильно и хорошо документировать. Заметьте, очень важно, чтобы закрывающие кавычки стояли на отдельной строке. А еще лучше, если перед ними будет ещё и пустая строка, например

```
"""Return a foobang
```

```
Optional plotz says to frobnicate the bizbaz first.
```

```
"""
```

- Для однострочной документации оставляйте закрывающие кавычки на той же строке

```
"""Return an ex-parrot."""
```

Соглашения по именованию

Соглашения по именованию переменных в python немного туманны, поэтому их список никогда не будет полным — тем не менее, ниже мы приводим список рекомендаций, действующих на данный момент. Новые модули и пакеты должны быть написаны согласно этим стандартам, но если в какой-либо уже существующей библиотеке эти правила нарушаются, предпочтительнее писать в едином с ней стиле.

Главный принцип

Имена, которые видны пользователю как часть общественного API должны следовать конвенциям, которые отражают использование, а не реализацию.

Описание: Стили наименования

Существует много разных стилей. Поможем вам распознать, какой стиль именования используется, независимо от того, для чего он используется.

Обычно различают следующие стили:

- `b` (одионочная маленькая буква)
- `B` (одионочная заглавная буква)
- `lowercase` (слово в нижнем регистре)
- `lower_case_with_underscores` (слова из маленьких букв с подчеркиваниями)
- `UPPERCASE` (заглавные буквы)
- `UPPER_CASE_WITH_UNDERSCORES` (слова из заглавных букв с подчеркиваниями)
- `CapitalizedWords` (слова с заглавными буквами, или `CapWords`, или `CamelCase`). Замечание: когда вы используете аббревиатуры в таком стиле, пишите все буквы аббревиатуры заглавными — `HTTPServerError` лучше, чем `HttpServerError`.
- `mixedCase` (отличается от `CapitalizedWords` тем, что первое слово начинается с маленькой буквы!)
- `Capitalized_Words_With_Underscores` (слова с заглавными буквами и подчеркиваниями — уродливо!)

Ещё существует стиль, в котором имена, принадлежащие одной логической группе, имеют один короткий префикс. Этот стиль редко используется в python, но мы упоминаем его для полноты. Например,

функция `os.stat()` возвращает кортеж, имена в котором традиционно имеют вид `st_mode`, `st_size`, `st_mtime` и так далее. (Так сделано, чтобы подчеркнуть соответствие этих полей структуре системных вызовов POSIX, что помогает знакомым с ней программистам).

В библиотеке X11 используется префикс `X` для всех public-функций. В python этот стиль считается излишним, потому что перед полями и именами методов стоит имя объекта, а перед именами функций стоит имя модуля.

В дополнение к этому, используются следующие специальные формы записи имен с добавлением символа подчеркивания в начало или конец имени:

- `_single_leading_underscore`: слабый индикатор того, что имя используется для внутренних нужд. Например, `from M import *` не будет импортировать объекты, чьи имена начинаются с символа подчеркивания.
- `single_trailing_underscore_`: используется по соглашению для избежания конфликтов с ключевыми словами языка python, например:

```
tkinter.Toplevel(master, class_='ClassName')
```

- `_double_leading_underscore`: изменяет имя атрибута класса, то есть в классе `FooBar` поле `_boo` становится `_FooBar_boo`.
- `_double_leading_and_trailing_underscore_` (двойное подчеркивание в начале и в конце имени): магические методы или атрибуты, которые находятся в пространствах имен, управляемых пользователем. Например, `_init_`, `_import_` или `_file_`. Не изобретайте такие имена, используйте их только так, как написано в документации.

Предписания: соглашения по именованию

Имена, которых следует избегать

Никогда не используйте символы `l` (маленькая латинская буква «эль»), `O` (заглавная латинская буква «о») или `I` (заглавная латинская буква «ай») как однобуквенные идентификаторы.

В некоторых шрифтах эти символы неотличимы от цифры один и нуля. Если очень нужно `l`, пишите вместо неё заглавную `L`.

ASCII совместимость

Идентификаторы, используемые в стандартной библиотеке, должны быть совместимы с ASCII, как описано в разделе политики PEP 3131.

Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие только из маленьких букв. Можно использовать символы подчеркивания, если это улучшает читабельность. То же самое относится и к именам пакетов, однако в именах пакетов не рекомендуется использовать символ подчёркивания.

Когда модуль расширения, написанный на C или C++, имеет сопутствующий python-модуль (содержащий интерфейс высокого уровня), C/C++ модуль начинается с символа подчеркивания, например, `_socket`.

Имена классов

Имена классов должны обычно следовать соглашению `CapWords`.

Вместо этого могут использоваться соглашения для именования функций, если интерфейс документирован и используется в основном как функции.

Обратите внимание, что существуют отдельные соглашения о встроенных именах: большинство встроенных имен - одно слово (либо два слитно написанных слова), а соглашение `CapWords` используется только для именования исключений и встроенных констант.

Имена переменных типа

В именах переменных типов, введенных в PEP 484, обычно следует использовать CapWords, предпочитая короткие имена: T, AnyStr, Num. Рекомендуется добавлять суффиксы `_co` или `_contra` к переменным, используемым для объявления ковариантного или контравариантного поведения соответственно:

```
from typing import TypeVar

VT_co = TypeVar('VT_co', covariant=True)
KT_contra = TypeVar('KT_contra', contravariant=True)
```

Имена исключений

Так как исключения являются классами, к исключениям применяется стиль именования классов. Однако вы можете добавить `Error` в конце имени (если, конечно, исключение действительно является ошибкой).

Имена глобальных переменных

(Будем надеяться, что глобальные переменные используются только внутри одного модуля.)
Руководствуйтесь теми же соглашениями, что и для имен функций.

Добавляйте в модули, которые написаны так, чтобы их использовали с помощью `from M import *`, механизм `__all__`, чтобы предотвратить экспортирование глобальных переменных. Или же, используйте старое соглашение, добавляя перед именами таких глобальных переменных один символ подчеркивания (которым вы можете обозначить те глобальные переменные, которые используются только внутри модуля).

Имена функций и переменных

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания, если это необходимо для того, чтобы улучшить читабельность.

Имена переменных следуют тем же соглашениям, что и названия функций.

Стиль `mixedCase` допускается в тех местах, где уже преобладает такой стиль, для сохранения обратной совместимости.

Аргументы функций и методов

Всегда используйте `self` в качестве первого аргумента метода экземпляра объекта.

Всегда используйте `cls` в качестве первого аргумента метода класса.

Если имя аргумента конфликтует с зарезервированным ключевым словом python, обычно лучше добавить в конец имени символ подчеркивания, чем исказить написание слова или использовать аббревиатуру. Таким образом, `class_` лучше, чем `class`. (Возможно, хорошим вариантом будет подобрать синоним).

Имена методов и переменных экземпляров классов

Используйте тот же стиль, что и для имен функций: имена должны состоять из маленьких букв, а слова разделяться символами подчеркивания.

Используйте один символ подчёркивания перед именем для непубличных методов и атрибутов.

Чтобы избежать конфликтов имен с подклассами, используйте два ведущих подчеркивания.

Python искажает эти имена: если класс `Foo` имеет атрибут с именем `_a`, он не может быть доступен как `Foo._a`. (Настойчивый пользователь все еще может получить доступ, вызвав `Foo._Foo__a`.) Вообще, два ведущих подчеркивания должны использоваться только для того, чтобы избежать конфликтов имен с атрибутами классов, предназначенных для наследования.

Примечание: есть некоторые разногласия по поводу использования `_`имя (см. ниже).

Константы

Константы обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания. Например: `MAX_OVERFLOW`, `TOTAL`.

Проектирование наследования

Обязательно решите, каким должен быть метод класса или экземпляра класса (далее - атрибут) — публичный или непубличный. Если вы сомневаетесь, выберите непубличный атрибут. Потом будет проще сделать его публичным, чем наоборот.

Публичные атрибуты — это те, которые будут использовать другие программисты, и вы должны быть уверены в отсутствии обратной несовместимости. Непубличные атрибуты, в свою очередь, не предназначены для использования третьими лицами, поэтому вы можете не гарантировать, что не измените или не удалите их.

Мы не используем термин "приватный атрибут", потому что на самом деле в python таких не бывает.

Другой тип атрибутов классов принадлежит так называемому API подклассов (в других языках они часто называются `protected`). Некоторые классы проектируются так, чтобы от них наследовали другие классы, которые расширяют или модифицируют поведение базового класса. Когда вы проектируете такой класс, решите и явно укажите, какие атрибуты являются публичными, какие принадлежат API подклассов, а какие используются только базовым классом.

Теперь сформулируем рекомендации:

- Публичные атрибуты не должны иметь в начале имени символа подчеркивания.
- Если имя открытого атрибута конфликтует с ключевым словом языка, добавьте в конец имени один символ подчеркивания. Это более предпочтительно, чем аббревиатура или искажение написания (однако, у этого правила есть исключение — аргумента, который означает класс, и особенно первый аргумент метода класса (`class method`) должен иметь имя `cls`).
- Назовите простые публичные атрибуты понятными именами и не пишите сложные методы доступа и изменения (`accessor/mutator`, `get/set`, — прим. перев.) Помните, что в python очень легко добавить их потом, если потребуется. В этом случае используйте свойства (`properties`), чтобы скрыть функциональную реализацию за синтаксисом доступа к атрибутам.

Примечание 1: Постарайтесь избавиться от побочных эффектов, связанным с функциональным поведением; впрочем, такие вещи, как кэширование, вполне допустимы.

Примечание 2: Избегайте использования вычислительно затратных операций, потому что из-за записи с помощью атрибутов создается впечатление, что доступ происходит (относительно) быстро.

- Если вы планируете класс таким образом, чтобы от него наследовались другие классы, но не хотите, чтобы подклассы унаследовали некоторые атрибуты, добавьте в имена два символа подчеркивания в начало, и ни одного — в конец. Механизм изменения имен в python сработает так, что имя класса добавится к имени такого атрибута, что позволит избежать конфликта имен с атрибутами подклассов.

Примечание 1: Будьте внимательны: если подкласс будет иметь то же имя класса и имя атрибута, то вновь возникнет конфликт имен.

Примечание 2: Механизм изменения имен может затруднить отладку или работу с `__getattr__()`, однако он хорошо документирован и легко реализуется вручную.

Примечание 3: Не всем нравится этот механизм, поэтому старайтесь достичь компромисса между необходимостью избежать конфликта имен и возможностью доступа к этим атрибутам.

Публичные и внутренние интерфейсы

Любые гарантии обратной совместимости распространяются только на публичные интерфейсы. Соответственно, важно, чтобы пользователи могли четко различать публичные и внутренние интерфейсы.

Документированные интерфейсы считаются общедоступными, если только в документации они явно не объявлены временными или внутренними интерфейсами, на которые не распространяются обычные гарантии обратной совместимости. Все недокументированные интерфейсы должны считаться внутренними.

Чтобы лучше поддерживать интроспекцию, модули должны явно объявлять имена в своем публичном API с помощью атрибута `__all__`. Установка `__all__` в пустой список означает, что модуль не имеет публичного API.

Даже если `__all__` установлен соответствующим образом, внутренние интерфейсы (пакеты, модули, классы, функции, атрибуты или другие имена) все равно должны иметь префикс с одним ведущим подчеркиванием.

Интерфейс также считается внутренним, если любое содержащее его пространство имен (пакет, модуль или класс) считается внутренним.

Импортированные имена всегда должны рассматриваться как детали реализации. Другие модули не должны полагаться на косвенный доступ к таким импортированным именам, если только они не являются явно документированной частью API содержащего модуля, например, `os.path` или модуль `__init__` пакета, который раскрывает функциональность подмодулей.

Общие рекомендации

- Код должен быть написан так, чтобы не зависеть от разных реализаций языка (PyPy, Jython, IronPython, Pyrex, Psyc и пр.).

Например, не полагайтесь на эффективную реализацию в CPython конкатенации строк в выражениях типа `a+=b` или `a=a+b`. Такие инструкции выполняются значительно медленнее в Jython. В критичных к времени выполнения частях программы используйте `".join()"` — таким образом склеивание строк будет выполнено за линейное время независимо от реализации python.

- Сравнения с `None` должны обязательно выполняться с использованием операторов `is` или `is not`, а не с помощью операторов сравнения. Кроме того, не пишите `if x`, если имеете в виду `if x is not None` — если, к примеру, при тестировании такая переменная может принять значение другого типа, отличного от `None`, но при приведении типов может получиться `False`!
- Используйте оператор `is not`, а не `not ... is`. Хотя оба выражения функционально идентичны, первое более читабельно и предпочтительно:

Правильно:

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

Неправильно:

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

- При реализации методов сравнения, лучше всего реализовать все 6 операций сравнения (`_eq_`, `_ne_`, `_lt_`, `_le_`, `_gt_`, `_ge_`), чем полагаться на то, что другие программисты будут использовать только конкретный вид сравнения.

Для минимизации усилий можно воспользоваться [декоратором](#) `functools.total_ordering()` для реализации недостающих методов.

PEP 207 указывает, что интерпретатор может поменять `y > x` на `x < y`, `y ≥ x` на `x ≤ y`, и может поменять местами аргументы `x == y` и `x != y`. Гарантируется, что операции `sort()` и `min()` используют оператор `<`, а `max()` использует оператор `>`. Однако, лучше всего определить все шесть операций, чтобы не возникало путаницы в других местах.

- Всегда используйте выражение `def`, а не присваивание лямбда-выражения к имени.

Правильно:

```
def f(x): return 2*x
```

Неправильно:

```
f = lambda x: 2*x
```

Первая форма означает, что имя результирующего объекта функции будет конкретно `f`, а не общее `<lambda>`. Это более удобно для трассировки и строкового представления в целом. Использование оператора присваивания устраняет единственное преимущество лямбда-выражения перед явным оператором `def` (т. е. его можно встраивать внутрь более крупного выражения).

- Наследуйте свой класс исключения от `Exception`, а не от `BaseException`. Прямое наследование от `BaseException` зарезервировано для исключений, которые не следует перехватывать.
- Используйте цепочки исключений соответствующим образом. Следует использовать `raise X from Y` для указания явной замены без потери отладочной информации.

Когда намеренно заменяется исключение (использование `raise X from None`), проследите, чтобы соответствующая информация передалась в новое исключение (такие, как сохранение имени атрибута при преобразовании `KeyError` в `AttributeError` или вложение текста исходного исключения в сообщение нового).

- Когда код перехватывает исключения, перехватывайте конкретные ошибки вместо простого выражения `except:`.

К примеру, пишите вот так:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

Простое написание `except:` также перехватит и `SystemExit`, и `KeyboardInterrupt`, что породит проблемы, например, сложнее будет завершить программу нажатием `control+C`. Если вы действительно собираетесь перехватить все исключения, пишите `except Exception:`.

Хорошим правилом является ограничение использования `except:`, кроме двух случаев:

1. Если обработчик выводит пользователю всё о случившейся ошибке; по крайней мере, пользователь будет знать, что произошла ошибка.
 2. Если нужно выполнить некоторый код после перехвата исключения, а потом вновь "бросить" его для обработки где-то в другом месте с помощью `raise`. Обычно же лучше пользоваться конструкцией `try...finally`.
- При перехвате ошибок операционной системы, предпочитайте использовать явную иерархию исключений, введенную в Python 3.3, вместо анализа значений `errno`.
 - Постарайтесь заключать в каждый блок `try` минимум кода, чтобы легче отлавливать ошибки. Опять же, это позволяет избежать замаскированных ошибок.

Правильно:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

Неправильно:

```
try:
    # Здесь много действий!
    return handle_value(collection[key])
except KeyError:
    # Здесь также перехватится KeyError, который может быть сгенерирован handle_value
    return key_not_found(key)
```

- Когда ресурс является локальным на участке кода, используйте [выражение with](#) для того, чтобы после выполнения он был очищен оперативно и надёжно. try...finally тоже подойдёт.
- Менеджеры контекста следует вызывать с помощью отдельной функции или метода, всякий раз, когда они делают что-то другое, чем получение и освобождение ресурсов. Например:

Правильно:

```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

Неправильно:

```
with conn:
    do_stuff_in_transaction(conn)
```

Последний пример не даёт никакой информации, указывающей на то, что `_enter_` и `_exit_` делают что-то кроме закрытия соединения после транзакции. Быть явным важно в данном случае.

- Будьте последовательны в операторах `return`. Либо все операторы `return` в функции должны возвращать выражение, либо ни один из них. Если любой оператор `return` возвращает выражение, то все операторы `return`, в которых не возвращается значение, должны явно указывать на это как на `return None`, а в конце функции (если это возможно) должен присутствовать явный оператор `return`:

Правильно:

```
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)
```

Неправильно:

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

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

- Пользуйтесь `".startswith()"` и `".endswith()"` вместо обработки срезов строк для проверки суффиксов или префиксов.

`startswith()` и `endswith()` выглядят чище и порождают меньше ошибок. Например:

Правильно:

```
if foo.startswith('bar'):
```

Неправильно:

```
if foo[:3] == 'bar':
```

- Сравнение типов объектов нужно делать с помощью `isinstance()`, а не прямым сравнением типов:

Правильно:

```
if isinstance(obj, int):
```

Неправильно:

```
if type(obj) is type(1):
```

- Для последовательностей (строк, списков, кортежей) используйте тот факт, что пустая последовательность есть `false`:

Правильно:

```
if not seq:  
if seq:
```

Неправильно:

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

- Не пользуйтесь строковыми константами, которые имеют важные пробелы в конце — они невидимы, а многие редакторы (а теперь и `reindent.py`) обрезают их.
- Не сравнивайте логические типы с `True` и `False` с помощью `==`:

Правильно:

```
if greeting:
```

Неправильно:

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

Совсем неправильно:

```
if greeting is True:
```

- Использование управляющих операторов `return/break/continue` внутри набора `finally` в `try...finally`, где управляющий оператор выходил бы за пределы набора `finally`, не рекомендуется. Это связано с тем, что такие операторы неявно отменяют любое активное исключение, распространяющееся через `finally`.

Неправильно:

```
def foo():  
    try:  
        1 / 0  
    finally:  
        return 42
```

С принятием PEP 484 правила стиля для аннотаций функций изменились.

- Аннотации функций должны использовать синтаксис PEP 484 (некоторые рекомендации по форматированию аннотаций приведены в предыдущем разделе).
- Эксперименты со стилями аннотаций, которые рекомендовались ранее в этом PEP, больше не поощряются.
- Однако за пределами `stdlib` эксперименты в рамках правил PEP 484 теперь поощряются. Например, разметка большой сторонней библиотеки или приложения с аннотациями типов в стиле PEP 484, анализ того, насколько легко было добавить эти аннотации, и наблюдение за тем, повышает ли их наличие понятность кода.
- Стандартная библиотека Python должна быть консервативна в принятии таких аннотаций, но их использование разрешено для нового кода и для больших рефакторингов.
- Для кода, который хочет по-другому использовать аннотации функций, рекомендуется поместить комментарий вида:

```
# type: ignore
```

в верхней части файла; это указывает программам проверки типов игнорировать все аннотации. (Более тонкие способы отключения жалоб от средств проверки типов можно найти в PEP 484).

- Как и линтеры, программы проверки типов являются необязательными, отдельными инструментами. Интерпретаторы Python по умолчанию не должны выдавать никаких сообщений из-за проверки типов и не должны изменять свое поведение на основе аннотаций.
- Пользователи, которые не хотят использовать средства проверки типов, могут их игнорировать. Однако предполагается, что пользователи пакетов сторонних библиотек могут захотеть запустить проверку типов в этих пакетах. Для этого PEP 484 рекомендует использовать файлы-заглушки: файлы `.pyi`, которые читаются программой проверки типов вместо соответствующих файлов `.py`. Файлы-заглушки могут распространяться вместе с библиотекой или отдельно (с разрешения автора библиотеки) через репозиторий `typeshed`.

Аннотации переменных

В PEP 526 были введены аннотации переменных. Рекомендации по стилю для них аналогичны описанным выше для аннотаций функций:

- Аннотации для переменных уровня модуля, переменных класса и экземпляра, а также локальных переменных должны иметь один пробел после двоеточия.
- Перед двоеточием не должно быть пробела.
- Если у присваивания есть правая часть, то знак равенства должен иметь ровно один пробел с обеих сторон.

Правильно:

```
code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
```

Неправильно:

```
code:int
code : int

class Test:
    result: int=0
```

- Хотя PEP 526 принят для Python 3.6, синтаксис аннотации переменных является предпочтительным синтаксисом для файлов-заглушек во всех версиях Python (подробнее см. PEP 484).

[В категорию "Продвинутый Python"](#)

→ Telegram

Поддержать ₽

(с) 2024 ruplanet - Python с нуля до бесконечности