PEP 8 - руководство по написанию кода на Python

Аннотация

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

PEP 8 создан на основе рекомендаций Guido van Rossum с добавлениями от Barry Warsaw. Если где-то возникал конфликт, мы выбирали стиль Гуидо. И, конечно, этот PEP может быть неполным (фактически, он, наверное, никогда не будет закончен).

Ключевая идея Гуидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. 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.

Две причины для того, чтобы нарушить данные правила:

Когда применение правила сделает код менее читаемым даже для того, кто привык читать код, который следует правилам.

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.

Источники:

https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html

Первоисточник:

http://legacy.python.org/dev/peps/pep-0008/#tabs-or-spaces

Содержание

1. Общие рекомендации

1.1. Внешний вид кода

1.1.1. 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 линии не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение линии.

Правильно:

```
# Выровнено по открывающему разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
# Больше отступов включено для отличения его от остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
  print(var_one)
```

Неправильно:

```
# Аргументы на первой линии запрещены, если не используется вертикальное выравнивание
foo = long_function_name(var_one, var_two,
  var_three, var_four)
# Больше отступов требуется, для отличения его от остальных
def long_function_name(
  var_one, var_two, var_three,
```

```
    var_four):
    print(var_one)
```

Опционально:

```
# Нет необходимости в большем количестве отступов.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

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',
)
```

1.2. Табуляция или пробелы?

Пробелы - самый предпочтительный метод отступов.

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

Python 3 запрещает смешивание табуляции и пробелов в отступах.

Python 2 пытается преобразовать табуляцию в пробелы.

Когда вы вызываете интерпретатор Python 2 в командной строке с параметром -t, он выдает предупреждения (warnings) при использовании смешанного стиля в отступах, а запустив интерпретатор с параметром -tt, вы получите в этих местах ошибки (errors). Эти параметры очень рекомендуются!

2. 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 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.

Сделайте правильные отступы для перенесённой строки. Предпочтительнее вставить перенос строки после логического оператора, но не перед ним. Например:

```
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)
```

2.1. 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.

3. Кодировка исходного файла

Кодировка Python должна быть UTF-8 (ASCII в Python 2).

Файлы в ASCII (Python 2) или UTF-8 (Python 3) не должны иметь объявления кодировки.

В стандартной библиотеке, нестандартные кодировки должны использоваться только для целей тестирования, либо когда комментарий или строка документации требует упомянуть имя автора, содержащего не ASCII символы; в остальных случаях использование \x, \u, \U или \N - наиболее предпочтительный способ включить не ASCII символы в строковых литералах.

Начиная с версии python 3.0 в стандартной библиотеке действует следующее соглашение: все идентификаторы обязаны содержать только ASCII символы, и означать английские слова везде, где это возможно (во

многих случаях используются сокращения или неанглийские технические термины). Кроме того, строки и комментарии тоже должны содержать лишь ASCII символы. Исключения составляют: (а) test case, тестирующий не-ASCII особенности программы, и (б) имена авторов. Авторы, чьи имена основаны не на латинском алфавите, должны транслитерировать свои имена в латиницу.

Проектам с открытым кодом для широкой аудитории также рекомендуется использовать это соглашение.

4. 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:

- импорты из стандартной библиотеки
- импорты сторонних библиотек
- импорты модулей текущего проекта

Вставляйте пустую строку между каждой группой импортов.

Указывайте спецификации __all__ после 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".

Шаблоны импортов (from 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).

5. Пробелы в выражениях и инструкциях

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 } )
```

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
```

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 выровнять его с другим:

Правильно:

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

Неправильно:

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

5.1. 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)
```

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)
```

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()
```

6. Комментарии

6.1. Общие рекомендации

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 переменной!).

Если комментарий короткий, можно опустить точку в конце предложения. Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое предложение должно оканчиваться точкой.

Ставьте два пробела после точки в конце предложения.

Программисты, которые не говорят на английском языке, 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.

6.2. 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 строчка 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 #.

6.3. "Встрочные" комментарии

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. "Встрочные" 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
```

6.4. Строки документации

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 можно оставить закрывающие кавычки на той же строке.

7. Контроль версий

Если вам нужно использовать Subversion, CVS или RCS в ваших исходных кодах, делайте вот так:

```
__version__ = "$Revision: 1a40d4eaa00b $"
# $Source$
```

Вставляйте эти строки после документации модуля перед любым другим кодом и отделяйте их пустыми строками по одной до и после.

8. Соглашения по именованию

8.1. О соглашении 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.

8.2. Главный принцип

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

8.3. Описание: Стили имен

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

Обычно различают следующие стили:

b (одиночная маленькая буква)

B (одиночная заглавная буква)

lowercase (слово в нижнем регистре)

lower_case_with_underscores (слова из маленьких букв с подчеркиваниями)

UPPERCASE (заглавные буквы)

UPPERCASE_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.

9. Предписания: соглашения по именованию

9.1. 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.

9.2. Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие 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.

Так как имена модулей отображаются в имена файлов, а некоторые файловые системы являются нечувствительными к регистру символов и обрезают длинные имена, очень важно использовать достаточно короткие имена модулей — это не проблема в Unix, но, возможно, код окажется непереносимым в старые версии Windows, Mac, или DOS.

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.

9.3. 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.

9.4. Имена исключений

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

9.5. 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).

9.6. Имена функций

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы увеличить читабельность.

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

9.7. 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 clss. (Возможно, хорошим вариантом будет подобрать синоним).

9.8. 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 __ имена (см. ниже).

9.9. Константы

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

10.Проектирование наследования

10.1. Общие требования

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.

10.2. Рекомендации по проектированию наследования

Теперь сформулируем рекомендации:

Открытые 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: Свойства (properties) работают только в классах нового стиля (в Python 3 все классы являются таковыми).

Примечание 2: Постарайтесь избавиться от побочных эффектов, связанным с функциональным поведением; впрочем, такие вещи, как кэширование, вполне допустимы.

Примечание 3: 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.

10.3. Общие рекомендации

Код должен быть написан так, чтобы не зависеть от разных реализаций языка (PyPy, Jython, IronPython, Pyrex, Psyco 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!

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 осуществить 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
```

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. В Python 3, "raise X from Y" следует использовать для указания явной замены без потери отладочной информации.

Когда намеренно заменяется исключение (использование "raise X" в Python 2 или "raise X from None" в Python 3.3+), проследите, чтобы соответствующая информация передалась в новое исключение (такие, как сохранение имени атрибута при преобразовании KeyError в AttributeError или вложение текста исходного исключения в новом).

Когда вы генерируете исключение, пишите raise ValueError('message') вместо старого синтаксиса raise ValueError, message.

Старая форма записи запрещена в python 3.

Такое использование предпочтительнее, потому что из-за скобок не нужно использовать символы для продолжения перенесенных строк, если эти строки длинные или если используется форматирование.

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:

– Если обработчик выводит пользователю всё о случившейся ошибке; по крайней мере, пользователь будет знать, что произошла ошибка.

– Если нужно выполнить некоторый код после перехвата исключения, а потом вновь "бросить" его для обработки где-то в другом месте. Обычно же лучше пользоваться конструкцией "try...finally".

При связывании перехваченных исключений с именем, предпочитайте явный синтаксис привязки, добавленный в Python 2.6:

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

Это единственный синтаксис, поддерживающийся в Python 3, который позволяет избежать проблем неоднозначности, связанных с более старым синтаксисом на основе запятой.

При перехвате ошибок операционной системы, предпочитайте использовать явную иерархию исключений, введенную в Python 3.3, вместо анализа значений errno.

Постарайтесь заключать в каждую конструкцию try...except 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.

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 дает никакой информации, указывающей на то, что __enter__ и __exit__ делают что-то кроме закрытия соединения после транзакции. Быть явным важно в данном случае.

Используйте строковые методы вместо модуля string — они всегда быстрее и имеют тот же API для unicode-строк. Можно отказаться от этого правила, если необходима совместимость с версиями python младше 2.0.

В Python 3 остались только строковые методы.

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):
```

Когда вы проверяете, является ли объект строкой, обратите внимание на то, что строка может быть unicode-строкой. В python 2 у str и unicode есть общий базовый класс, поэтому вы можете написать:

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

Отметим, что в Python 3, unicode и basestring больше не существуют (есть только str) и bytes больше не является своего рода строкой (это последовательность целых чисел).

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:
```