**Введение**

PEP8 можно определить, как документ, описывающий общепринятый стиль написания кода на языке Python. Python Enhanced Proposal (PEP) - переводится, как заявки по улучшению языка Python.

Помимо PEP8, так же имеются и другие документы под индексами о которых можно прочитать в PEP0. Но наибольшего внимания заслуживает именно PEP8, а так же PEP7 (В нем описывается какого стиля следует придерживаться при написании кода на C в реализации языка python)

На этой странице представлено полное описание PEP8 на русском языке. Так же вы можете ознакомится с коротким описанием PEP8.

**1. Внешний вид кода**

**Отступы**

Используйте 4 пробела на каждый уровень отступа.

На каждый уровень отсупа используйте 4 пробела. Для действительно старого кода, который вы не хотите трогать, можите продолжать использовать 8 пробелов.

**Табуляция или пробелы?**

Пробелы - наиболее предпочтительный метод отступов.

Табуляция может быть использоваться только для поддержки кода,  в котором были использованы отступы с помощью табуляции.

В Python 3 запрещается смешивание табуляции и пробелов в отступах.

В Python 2 табуляция преобразовыватеся в пробелы.

Интерпретатор Python 2 выдает предупреждения (warnings) при использовании смешанного стиля в отступах при использовании параметра -t. При использовании прараметра -tt, интерпретатор выдаст ошибки (errors) в этих местах. Использование этих параметров очень рекомендуется.

В новых проектах для отступов настоятельно рекомендется использовать пробелы. В этом вам могут помочь многие редакторы

**Максимальная длина строки**

Длину строки рекомендуется ограничить 79 символами

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

Сейчас все еще существует устройства, где длина строки ограничена 80 символам. Автоматический перенос строк на таких устройствах нарушит форматирование, и код будет труднее понять. Так же, ограничив ширину окна 80 символами, мы сможем расположить несколько окон рядом друг с другом.

Предпочтительный способ переноса длинных строк — использование подразумевающегося продолжения строки между обычными, квадратными и фигурными скобками. Если необходимо - можно добавить еще одну пару скобок вокруг выражения, но лучше использовать обратный слэш ("\").

Пример:

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())

Другой вариант - инструкция assert.

Постарайтесь сделать правильные отступы для перенесённой строки. Предпочтительнее ставить перенос после бинарного оператора, а не перед ним.

Пример:

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)

**Пустые строки**

Функции верхнего уровня и определения классов отделяются двумя пустыми строками.

Определения методов внутри класса разделяются одной пустой строкой.

Дополнительные отступы пустыми строками могут быть использованы для выделения группы логически связанных функций. Пустые строки могут быть пропущены, между несколькими выражениями, записанными в одну строку, например, «заглушки» функций.

Используйте пустые строки в коде функций, чтобы указать логические разделы.

Python расценивает символ Сontrol+L как незначащий (whitespace). Вы можете использовать его, так как многие редакторы обрабатывают его как разрыв страницы — таким образом логические части в файле будут на разных страницах.

**Кодировка исходного файла**

Кодировка Python 3 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 тестирования, либо если 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. Кроме того, строки и комментарии тоже должны содержать лишь ASCII символы. Исключения составляют:

1) test case, тестирующий не-ASCII особенности программы;

2) Bмена авторов. Авторы, чьи имена основаны не на латинском алфавите, должны транслитерировать свои имена в латиницу.

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

**Импорты**

* Для каждого импорта - отдельная строка.

Правильно:

import os

import datetime

Неправильно:

import os, datetime

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

Между каждой группой импортов вставляйте пустую строку.

Указывайте спецификации \_\_all\_\_ после импортов.

* Когда вы импортируете класс из модуля, разрешается 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

Использование - "myclass.MyClass" и "foo.bar.yourclass.YourClass".

* Следует избегать шаблонов импортов (from import \*), так как они не дают информации о том, какие имена присутствуют в глобальном пространстве имён, что осложняет как чтение кода программистом, так и другим автоматизированным средства.

**Пробелы в выражениях и инструкциях**

**Следует избегать использования пробелов в следующих ситуациях:**

* Внутри круглых, квадратных или фигурных скобок.

Правильно:

animals(tiger[2], {eagle: 6})

Неправильно:

animals( tiger[ 1 ], { eagle: 2 } )

* Перед запятой, точкой с запятой или двоеточием:

Правильно:

if x == y:

* Неправильно:

if x == y :

* Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции:

Правильно:

tiger(1)

Неправильно:

tiger (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 любого оператора для того, чтобы выровнять его с другим:

Правильно:

x = 1

y = 2

long\_variable = 3

Неправильно:

x = 1

y = 2

long\_variable = 3

**Другие рекомендации**

* Всегда окружайте следующие бинарные операторы одним пробелом с каждой стороны: операторы присваивания (=, +=, -= и другие), операторы сравнения (==, <, >, !=, <>, <=, >=, in, not in, is, is not), логические операторы (and, or, not).
* Если используются операторы с разными приоритетами, попробуйте добавить пробелы вокруг операторов с самым низким приоритетом для того, чтобы сразу визуально отличить очередность выполнения. Используйте свои собственные суждения, но никогда не используйте более одного пробела, а так же 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 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()

* Тело циклов **for**, **while**а так же ветку **if**можно писать в одну строку, если команда короткая и она одна.

Возможно неправильно:

if foo == 'blah': do\_blah\_thing()

for x in lst: total += x

while t < 10: t = delay()

Точно 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()

**Комментарии**

Комментарии, которые противоречат коду, хуже, их отсутсвие. Всегда исправляйте комментарии, когда меняете код!

Комментарии должны представлять собой 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 не ставить 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.

Если вы пишете по-английски, не забывайте о книге “Elements of style” Странка и Уайта. Эта книга является эталонным руководством по правильному написанию текстов на английском языке.

Программистам, которые не говорят на английском языке следует писать комментарии на английском. За исключением случаев, когда Вы на 126% уверены, 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 строчка такого блока должна начинаться с символа **#**и одного пробела после него.

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

**"Встрочные" комментарии**

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. "Встрочные" комментарии должны отделяться как минимум двумя пробелами от самой инструкции. И должны начинаться с символа # и одного пробела.

Если комментарии объясняют очевидное - они не нужны, т.к. только отвлекают от чтения кода

Пример как писать не нужно:

x = x + 1 # Increment x

**Строки документации**

* Пишите документацию для всех public модулей, функций, классов, методов. Для не-public методов cтроки документации необязательны, но лучше описать, что делает метод. Комментарий нужно писать после строки с def.
* [PEP 257](http://pythonworld.ru/osnovy/dokumentirovanie-koda-v-python-pep-257.html) объясняет, как правильно писать документацию. Очень важно, чтобы закрывающие кавычки стояли на отдельной строке. А еще лучше, если перед ними будет ещё и пустая строка.

        Пример:

* """Return a foobang
* Optional plotz says to frobnicate the bizbaz first.
* """
* Для однострочной документации можно оставить закрывающие кавычки на той же строке.

**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.

**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 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. Выглядит уродливо!)

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 «L»), O (заглавная латинская буква «o») или I (заглавная латинская буква «i») как однобуквенные идентификаторы.

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

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 С или C++, имеет сопутствующий python-модуль (содержащий интерфейс высокого уровня), С/С++ модуль начинается с символа подчеркивания, например, \_socket.

Имена классов

Имена классов должны обычно следовать соглашению CapitalizedWords .

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 CapitalizedWords используется только для именования исключений и встроенных констант.

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).

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

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

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 clss. (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.

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) — public или не-public. Если вы сомневаетесь, выберите не-public атрибут. Потом будет проще сделать его public.

Public атрибуты — это те, которые будут использовать другие программисты, и вы должны быть уверены в обратной совместимости.

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

**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. 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.
* 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.

**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.
* 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](http://pythonworld.ru/osnovy/dekoratory.html) functools.total\_ordering() для реализации недостающих методов.

PEP 207 указывает, что интерпретатор может поменять y > х на х < y, y >= х на х <= y, и может поменять местами аргументы х == y и х != 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.
* При генерации исключения, пишите raise ValueError('message') вместо старого синтаксиса raise ValueError, message.
* Перехватывайте конкретные ошибки вместо простого выражения except:

Пример:

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;
  2. 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](http://pythonworld.ru/osnovy/with-as-menedzhery-konteksta.html" \t "_blank) 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)

* 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):

* 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 обрезают их.
* Не сравнивайте логические типы с True и False с помощью ==:

Правильно:

if greeting:

Неправильно:

if greeting == True: