Стиль кода в языке Python

ITcoder

Введение

Этот документ описывает соглашение о том, как писать код для языка python, включая стандартную библиотеку, входящую в состав python. Пожалуйста, посмотрите также на сопутствующий PEP (python enhanced proposal — заявки на улучшение языка python), описывающий, какого стиля следует придерживаться при написании кода на C в реализации языка python[1].

Этот документ создан на основе рекомендаций Гуидо ван Россума с добавлениями от Барри. Если где-то возникал конфликт, мы выбирали стиль Гуидо. И, конечно, этот PEP может быть неполным (фактически, он, наверное, никогда не будет закончен).

[1] PEP 7, Style Guide for C Code, van Rossum

A Foolish Consistency is the Hobgoblin of Little Minds[2]

Ключевая идея Гуидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читбельность кода и сделать его согласованным между большим числом проектов. В идеале, весь код будет написан в едином стиле, и любой сможет легко его прочесть. Как говорится в PEP 20[3], «Читабельность имеет значение».

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, чтобы нарушить правила:

- Когда применение правила сделает код менее читабельным 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 (может быть, в силу исторических причин) — впрочем, это возможность подчистить чужой код.

---

[2] «A foolish consistency is the hobgoblin of little minds, adored by little statesman and philosophers and divines. With consistency a great soul has simply nothing to do» — цитата Ральфа Валдо Эмерсона, известного американского писателя.

[3] PEP 20, The Zen of Python

Внешний вид кода

Отступы

Используйте 4 пробела на один уровень отступа. В старом коде, который вы не хотите трогать, можно продолжить пользоваться 8 пробелами для отступа.

Табуляция или пробелы?

Никогда не смешивайте символы табуляции и пробелы.

Самый распространенный способ отступов — пробелы. На втором месте — отступы только с использованием табуляции. Код, в котором используются и те, и другие типы отступов, должен быть исправлен так, чтобы отступы в нем были расставлены только с помощью пробелов. Когда вы вызываете интерпретатор 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). Используйте этии опции!

В новых проектах для отступов мы настоятельно рекомендуем использовать пробелы. К тому же, многие редакторы позволяют легко делать.

Максимальная длина строки

Ограничьте максимальную длину строки 79 символами.

Пока еще существует немало устройств, где длина строки равна 80 символам; к тому же, ограничив ширину окна 80 символами, мы сможем расположить несколько окон рядом друг с другом. Автоматический перенос строк на таких устройствах нарушит форматирование, и код будет труднее понять. Так что, пожалуйста, ограничьте длину строки 79 символами, и 72 символами в случае длинных блоков текста (строки документации или комментарии).

Предпочтительный способ переноса длинных строк — использование подразумевающегося продолжения строки между обычными, квадратными и фигурными скобками. В случае необходимости можно добавить еще одну пару скобок вокруг выражения, но часто лучше выглядит обратный слэш. Постарайтесь сделать правильные отступы для перенесённой строки. Предпочтительнее вставить перенос строки *после* бинарного оператора, но не перед ним.

Вот несколько примеров:

```
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

Отделяйте функции (верхнего уровня, не функции внутри функций) и определения классов двумя пустыми строчками.

Определения методов внутри класса отделяйте одной пустой строкой.

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

Используйте (без энтузиазма) пустые строки в коде функций, чтобы отделить друг от друга логические части.

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.

Кодировки (PEP 263)

Код ядра python всегда должен использовать ASCII или Latin-1 кодировку (также известную как ISO-8859-1). Начиная с версии python 3.0, предпочтительной является кодировка UTF-8 (смотрите PEP 3120).

Files using ASCII (or UTF-8, for Python 3.0) should not have a coding cookie. Используйте Latin-1 (или UTF-8), только если это необходимо, чтобы указать в комментарии или строке документации имя автора, содержащее в себе символ из Latin-1. В противном случае предпочтительнее использовать escape-символы \x, \u или \U для не-ASCII символов в строках.

Начиная с версии python 3.0 в стандартной библиотеке действует следующая политика (смотрите PEP 3131): 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, буквы в именах которых не из латинского алфавита, должны транслитерировать свои имена в латиницу.

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

Import-секции

Импортирование разных модулей должно быть на разных строчках, например:

правильно:

```
import os
import sys
```

неправильно:

```
import os, sys
```

В то же время, можно писать вот так:

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

Импортирование всегда нужно делать сразу после комментариев к модулю и строк документации, перед объявлением глобальных переменных и констант.

Группируйте импорты в следующем порядке:

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

Вставляйте пустую строку между каждой группой импортов.

Указывайте спецификации `__all__` после импортов.

Относительные импорты крайне не рекомендуются — всегда указывайте абсолютный путь к модулю для всех импортирований. Даже сейчас, когда PEP 328[4] реализован в версии python 2.5, использовать явные относительные импорты нежелательно, потому что абсолютные импорты лучше переносимы и читабельны.

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».

Пробелы в выражениях и инструкциях

Избегайте использования пробелов в следующих ситуациях:

1. Сразу после или перед скобками (обычными, фигурными и квадратными)

   можно:

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

   нельзя:

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

2. Сразу перед запятой, точкой с запятой, двоеточием:

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

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

[4] PEP 328, Imports: Multi-Line and Absolute/Relative

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

```
spam(1)

spam (1)
```

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

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

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

5. Использование более одного пробела вокруг оператора присваивания (или любого другого) для того, чтобы выровнять его с другим таким же оператором на соседней строке:

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

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

Прочие рекомендации:

1. Вседа окружайте эти бинарные операторы одним пробелом с каждой стороны: присваивание (=, +=, -= и прочие), сравнения (==, <, >, !=, <>, <=, >=, `in`, `not in`, `is`, `is not`), логические операторы (`and`, `or`, `not`).

2. Ставьте пробелы вокруг арифметических операций.

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

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

3. Не используйте пробелы для отделения знака =, когда он употребляется для обозначения аргумента-ключа (keyword argument) или значения параметра по умолчанию.

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

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

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

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

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

5. 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.

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

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, 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 (кстати, никогда не отступайте от этого правила для имен переменных).

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.

Если вы пишете по-английски, не забывайте о Странке и Уайте (имеется в виду книга Strunk & White, “Elements of style”, которая является практически эталонным руководством по правильному написанию текстов на английском языке, *— прим. перев.*)

Программисты, которые не говорят на английском языке, пожалуйста, пишите комментарии на английском, если только вы не уверены на 120 процентов, что ваш код никогда не будут читать люди, не знающие вашего родного языка.

Блок 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 лучше отделять строкой, состоящей из одного символа #.

Комментарии в строке с кодом

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. «Встрочные» комментарии должны отделяться хотя бы 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, иногда такие комметарии полезны:

```
x = x + 1                  # Compensate for border
```

Строки документации

Соглашения о написании хорошей документации (docstrings) увековечены (да, забавно, но автор использует именно такое слово, — *прим. перев.*) в PEP 257[5].

Пишите документацию для всех модулей, функций, классов, методов, которые объявлены как public. Строки документации необязательны для не-public 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 """ стояли на отдельной строчке. 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 """ на той же строке.

[5] PEP 257, Docstring Conventions, Goodger, van Rossum

Учёт версий

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

```
version__ = "$Revision: 68852 $"
# $Source$
```

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

Имена

Соглашения об именах 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.

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[6]. Иногда называется StudlyCaps). 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, т.е. в `class FooBar` поле `__boo` становится `_FooBar__boo`.
- `__double_leading_and_trailing_underscore__` (двойное подчеркивание в начале и в конце имени): «волшебные» объекты или атрибуты, которые «живут» в пространствах имен, управляемых пользователем (user-controlled namespaces). Например, `__init__`, `__import__` или `__file__`. Не изобретайте такие имена, используйте их только так, как написано в документации.

[6] CamelCase в википедии

Стили имен

Имена, которых следует избегать

Никогда не используйте символы `l` (малелькая 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 (и символа вертикальной палочки, — *прим. перев.*) Если очень нужно использовать `l` имена, пишите вместо неё заглавную `L`.

Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие из маленьких букв. Можно использовать и символы подчеркивания, если это улучшает читабельность. То же, за исключением символов подчеркивания, относится и к именам пакетов.

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 или 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`.

Имена классов

Все имена классов должны следовать соглашению CapWords почти без исключений. Классы внутреннего использования могут начинаться с символа подчеркивания.

Имена исключений (exceptions)

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 такие имена 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 увеличить читабельность.

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

Аргументы функций и методов

Всегда используйте `self` в качестве первого аргумента метода экземпляра объекта (instance method).

Всегда используйте `cls` в качестве первого аргумента метода класса (class method).

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, `print_` лучше, чем `prnt`. (Возможно, хорошим вариантом будет подобрать синоним).

Имена методов и переменные экземпляров классов

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

Чтобы избежать конфликта имен с подклассами, добавьте два символа подчеркивания, чтобы включить механизм изменения имен. Если класс `Foo` имя атрибут с именем `__foo`, к нему нельзя обратиться, написав `Foo.__a`. (Настойчивый пользователь всё равно может получить доступ, написав `Foo._Foo__a`). Вообще, двойное подчеркивание в именах должно использоваться, чтобы избежать конфликта имен с атрибутами классов, спроектированных так, чтобы от них наследовали подклассы.

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 переменная экземпляра класса (в общем, атрибут) — public или не-public. Если вы сомневаетесь, выберите закрытый, не-public атрибут. Потом будет проще сделать их public, чем наоборот.

Открытые атрибуты — это те, которые будут использовать потребители ваших классов, и вы должны быть уверены в отсутствии обратной несовместимости. Не-public атрибуты, в свою очередь, не предназначены для использования третьими лицами, поэтому вы можете не гарантировать, что не измените или не удалите эти атрибуты.

Мы не используем термин «закрытый член» (private), потому что на самом деле в python таких членов не бывает.

Другой тип аттрибутов 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 открытыми (public), какие принадлежат API подклассов (subclass API), а какие используются только базовым классом.

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 открытые 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.
    - Свойства (properties) работают только в классах нового стиля (new-style classes)
    - Постарайтесь избавиться от побочных эффектов, связанным с функциональным поведением; впрочем, такие вещи, как кэширование, вполне допустимы.
    - 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 (name mangling, — *прим. перев.*) сработает так, что имя класса добавится к имени такого атрибута, что позволит избежать конфликта имен с атрибутами подклассов.
    - Будьте внимательны: если подкласс будет иметь то же имя класса и имя атрибута, то вновь возникнет конфликт имен.
    - Механизм изменения имен может затруднить отладку или работу с `__getattr__()`, однако он хорошо документирован и легко реализуется вручную.
    - Не всем нравится этот механизм, поэтому старайтесь достичь компромисса между необходимостью избежать конфликта имен и возможностью доступа к этим атрибутам.

Общие рекомменда́ции

Код должен быть написан так, чтобы не зависеть от разных реализация 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 равенства или неравенства. Кроме того, не пишите if x, если имеете в виду if x is not None — если, к примеру, при тестировании такая переменная или аргумент примет значение иного типа, то при приведении к булевскому типу получится `false`.

Создавайте исключения на основе классов. Впрочем, начиная с версии python 2.6, мы уже не можем использовать строки в качестве исключений. В модулях или пакетах создавайте свои базовые классы исключений, наследуя их от встроенного класса `Exception` и обязательно их документируйте:

```
class MessageError(Exception):
    """Base class for errors in the email package."""
```

Здесь применимы те же правила, что и для именования классов. Если исключение по своему смыслу является ошибкой, вы можете добавить в конце имени `Error`.

Когда вы генерируете исключение, пишите raise `ValueError('message')` вместо старого синтаксиса `raise ValueError, message`. Такое использование предпочтительнее, потому что из-за скобок не нужно использовать символы для продолжения перенесенных строк, если эти строки длинные или если используется форматирование. Старая форма записи запрещена в python 3000.

Когда код перехватывает исключения, «ловите» конкретные ошибки вместо простого выражнения `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:'`.

Ограничьтесь использованием чистого `'except:'` в двух случаях:

1. Если обработчик исключения выводит пользователю всё о случившейся ошибке (например, traceback)
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'`.

Постарайтесь заключать в каждую конструкцию try...except минимум кода, чтобы легче отлавливать ошибки.

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

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

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.

Пользуйтесь `''.startswith()` и `''.endswith()` вместо обработки частей строк (string slicing) для проверки суффиксов или префиксов. `startswith()` и `endswith()` выглядят чище и порождают меньше ошибок. Например:

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

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

Исключением является случай, когда вам нужна совместимость с python 1.5.2.

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

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

```
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 2.3 у `str` и `unicode` есть общий базовый класс, поэтому пишите вот так:

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

В python 2.2 в модуле `types` для этой цели определен тип `StringTypes`:

```
from types import StringTypes
if isinstance(obj, StringTypes):
```

В python 2.0 и 2.1 нужно писать:

```
from types import StringType, UnicodeType
if isinstance(obj, StringType) or \
isinstance(obj, UnicodeType) :
```

Для последовательностей (строк, списков, кортежей) можно использовать тот факт, что пустая последовательность есть false:

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

```
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 ==:

```
if greeting:
```

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

А вот так писать совсем плохо:

```
if greeting is True:
```