# Стиль программирования

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2020

# Содержание лекции

1. Оформление кода
2. Анализаторы кода

# PEP8 - стиль кода в языке Python

**PEP8**

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

- документ создан на основе рекомендаций Гуидо ван Россума с добавлениями от Барри.
- этот PEP фактически, наверное, никогда не будет закончен.

  *A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.*
  *Ральф Уолдо Эмерсон, «Доверие к себе»*

- Гвидо Ван Россум хотел, чтобы его стилю соответствовали каждая функция, каждый модуль и каждый проект.
- можно довериться себе и отклонится от PEP8 в угоду читаемости, сформированного в проекте стилю и другим моментам.
- Прямое следование гайду ни к чему хорошему не приведет.

# PEP8 - стиль кода в языке Python

**Ключевая идея**: код читается намного больше раз, чем пишется.

- Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читабельность кода и сделать его согласованным между большим числом проектов.
- В идеале, весь код будет написан в едином стиле, и любой сможет легко его прочесть.

Согласованность:

- с PEP8;
- внутри проекта;
- внутри функции или модуля.

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.

# 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?**

- 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.
- Никогда не смешивайте символы табуляции и пробелы.

# Максимальная длина строки

- Ограничьте максимальную длину строки 79 символами.
  - Пока еще существует немало устройств, где длина строки равна 80 символам;
  - 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.
- 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.

```
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):
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)

raise ValueError("I don't think so -- values are %s, %s" %
                 (width, height))
```

# 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 $\backslash x$, $\backslash u$ или $\backslash U$ для не-ASCII символов в строках.

# Кодировки (PEP 263)

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).
- 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.

# 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
```

# Import-секции

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 (список публичных объектов данного модуля).

```
__all__ = ["MyClass", "MyClass2"]).
```

При импорте

```
from mymodule import *
```

импортированы будут только те объекты, которые описаны в __all__.

# Пробелы в выражениях и инструкциях

Избегайте использования пробелов в следующих ситуациях:

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

правильно:

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

неправильно:

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

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

правильно:

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

неправильно:

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

# Пробелы в выражениях и инструкциях

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

правильно:

```
spam(1)
```

неправильно:

```
spam (1)
```

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

правильно:

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

неправильно:

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

# Пробелы в выражениях и инструкциях

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

правильно:

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

неправильно:

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

# Прочие рекоммендации:

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.

*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
```

# Прочие рекомендации:

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

# Прочие рекоммендации:

4. 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()
```

# Прочие рекоммендации:

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.

*можно иногда:*

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

*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, 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.
- 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.

# 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 #.

# Встрочные комментарии

«Встрочные» комментарии - комментарии в строке с кодом

- 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   # Место для рамки окна
```

# 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.

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.

# Соглашения по наименованию

## Главный принцип

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

- Соглашения по именованию переменных в python немного туманны, поэтому их список никогда не будет полным.
- 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.

# Стили имен

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).
- mixedCase (отличается от CapitalizedWords тем, что первое слово начинается с маленькой буквы)
- Capitalized_Words_With_Underscores (слова с заглавными буквами и подчеркиваниями – уродливо!)

# Стили имен

- _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..

# Предписания: соглашения по именованию

## Имена модулей и пакетов

- Модули должны иметь короткие имена, состоящие из маленьких букв. Можно использовать символы подчеркивания, если это улучшает читабельность.
- 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.
- 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.

# Предписания: соглашения по именованию

## Имена классов

- Имена классов должны обычно следовать соглашению CapWords.
- 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 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).

# Предписания: соглашения по именованию

## 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).

# Предписания: соглашения по именованию

## Имена функций

- Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания.
- 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.

# Предписания: соглашения по именованию

## 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.

## Константы

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

# Общие рекомендации

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

# Рекомендации по обработке исключений

- Наследуйте свой класс исключения от Exception, а не от BaseException. Прямое наследование от BaseException зарезервировано для исключений, которые не следует перехватывать.
- Когда вы генерируете исключение, пишите raise ValueError('message')
- Когда код перехватывает исключения, перехватывайте конкретные ошибки вместо простого выражения except:

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

# Рекомендации по обработке исключений

- 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. Обычно же лучше пользоваться конструкцией "try...finally".
- Постарайтесь заключать в каждую конструкцию try...except минимум кода, чтобы легче отлавливать ошибки.

# Рекомендации по обработке исключений

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 сг
    return key_not_found(key)
```

1. 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):
```

2. 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)
```

3. Не пользуйтесь строковыми константами, которые имеют важные пробелы в конце — они невидимы, а многие редакторы (а теперь и reindent.py) обрезают их.

4. Не сравнивайте логические типы с True и False с помощью ==:
*Правильно:*

```
if greeting:
```

*Неправильно:*

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

*Совсем неправильно:*

```
if greeting is True:
```

# Содержание лекции

1 Оформление кода

2 Анализаторы кода

Два типа ошибок, которые обнаруживают анализаторы кода:

- ошибки стиля (неправильные отступы, длинные строки);
- ошибки в логике программы и ошибки синтаксиса языка программирования;
  - опечатки при написании названий стандартных функций;
  - неиспользуемые импорты;
  - дублирование кода.

Существуют и другие виды ошибок, например — оставленные в коде пароли или высокая цикломатическая сложность.

# Тестовый файл

```
import os
import notexistmodule

def Function(num,num_two):
    return num

class MyClass:
    """class MyClass """

    def __init__(self,var):
        self.var=var

    def out(var):
        print(var)

if __name__ == "__main__":
    my_class = MyClass("var")
    my_class.out("var")
    notexistmodule.func(5)
```

В коде допущено несколько ошибок:

- импорт неиспользуемого модуля os,
- импорт не существующего модуля notexistmodule,
- имя функции начинается с заглавной буквы,
- лишние аргументы в определении функции,
- отсутствие self первым аргументом в методе класса,
- неверное форматирование.

# Pycodestyle

Pycodestyle – простая консольная утилита для анализа кода Python, а именно для проверки кода на соответствие PEP8.

Запустим проверку на нашем коде:

```
> python3 -m pycodestyle example.py
```

```
example.py:4:1: E302 expected 2 blank lines, found 1
example.py:4:17: E231 missing whitespace after ','
example.py:7:1: E302 expected 2 blank lines, found 1
example.py:10:22: E231 missing whitespace after ','
example.py:11:17: E225 missing whitespace around operator
```

Вывод показывает строки, в которых, по мнению анализатора, есть нарушение соглашений PEP8.

Формат вывода: <имя файла>: <номер строки> :<положение символа>: <код и короткая расшифровка ошибки>

# Pycodestyle

Ограничения программы:

- можно настроить уровень проверок;
- нет проверка на правильность именования;
- проверка документации сводится к проверки длины docstring.

```
> python3 -m pycodestyle --statistics -qq example.py

1 E225 missing whitespace around operator
2 E231 missing whitespace after ','
2 E302 expected 2 blank lines, found 1
```

Более наглядный вывод: ключ –show-source.

```
> python3 -m pycodestyle --show-source example.py

example.py:4:1: E302 expected 2 blank lines, found 1
def Function(num,num_two):
^
example.py:4:17: E231 missing whitespace after ','
def Function(num,num_two):
^
example.py:7:1: E302 expected 2 blank lines, found 1
class MyClass:
^
example.py:10:22: E231 missing whitespace after ','
def __init__(self,var):
^
example.py:11:17: E225 missing whitespace around operator
self.var=var
^
```

# Pydocstyle

Pydocstyle проверяет наличие docstring у модулей, классов, функций и их соответствие официальному соглашению PEP257.

```
> python3 -m pydocstyle example.py

example.py:1 at module level:
D100: Missing docstring in public module
example.py:4 in public function `Function`:
D103: Missing docstring in public function
example.py:7 in public class `MyClass`:
D400: First line should end with a period (not 's')
example.py:7 in public class `MyClass`:
D210: No whitespaces allowed surrounding docstring text
example.py:10 in public method `__init__`:
D107: Missing docstring in __init__
example.py:13 in public method `out`:
D102: Missing docstring in public method
```

# Pyflakes

Pyflakes – анализатор кода, осуществляет поиск логических и синтаксических ошибок.

```
> python3 -m pyflakes example.py

example.py:1: 'os' imported but unused
```

# Pylint

Pylint совместил в себе поиск стилистических и логических ошибок.

```
> python3.6 -m pylint --reports=y text example.py
...
```

Полный отчет в *pylint_result.txt*

Программа имеет свою внутреннюю маркировку проблемных мест в коде:

- **R**efactor – требуется рефакторинг,
- **C**onvention – нарушено следование стилистике и соглашениям,
- **W**arning – потенциальная ошибка,
- **E**rror – ошибка,
- **F**atal – ошибка, которая препятствует дальнейшей работе программы.

# Pylint

Основные возможности:

- генерация файла настроек (–generate-rcfile).
- отключение вывода в коде.

```
# pylint: disable=unused-argument
```

- создание отчетов в формате json (–output-format=json)
- запуск в нескольких параллельных потоках на многоядерных процессорах;
- встроенная документация

```
> python3.6 -m pylint --help-msg=import-error

:import-error (E0401): *Unable to import %s*
Used when pylint has been unable to import a module. This m
the imports checker.
```

- возможность подключения плагинов;
- вывод «прогресса».