Python 编码规范

- 前言
- 一、编码风格
  - 1.1 缩进
  - 1.2 每行最大长度
  - 1.3 空白符
  - 1.4 操作符
  - 1.5 括号
  - 1.6 空行
  - 1.7 源文件编码
  - 1.8 Shebang
  - 1.9 模块引用(import)
  - 1.10 模块中的魔术变量(dunders)
  - 1.11 注释
  - 1.12 文档字符串

前言

为了让不同编码习惯的开发者更好的协作配合，并且形成良好的基础编码规范与风格，我们以 PEP8 为基础，修改了陈旧的规则，丰富了示例，并整理了工作中常见的不规范操作，最终形成此 Python 编码规范与风格。

本规范适用于所有使用 Python 语言作为开发语言的软件产品。

由于 Python2 在 2020 年停止维护，建议新增的项目使用 Python3.6+，可以使用到更多的高级特性。如果项目有兼容性需求需要支持老版本 Python 的，那么不涉及的特性可以忽略。 本规范的示例采用符合 Python3.6+ 的语法。

- **必须（Mandatory）**：用户必须采用；
- **推荐（Preferable）**：用户理应采用，但如有特殊情况，可以不采用；
- **可选（Optional）**：用户可参考，自行决定是否采用；

未明确指明的则默认为 **必须（Mandatory）**。

一、编码风格

规范地代码布局有助于帮助开发者更容易地理解业务逻辑。

1.1 缩进

1.1.1 **【必须】** 对于每级缩进，统一要求使用4个 空格 ，而非 tab 键。 pylint: `bad-indentation`.

1.1.2 **【必须】** 续行，要求使用括号等定限界符，并且需要垂直对齐。

正确示范

```
 # 与定界（括号）符对齐
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 换行并增加4个额外的空格（一级缩进）
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂需要增加一级缩进
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

错误示范

```


# 当不使用垂直对齐时，第一行不允许加参数
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 下面这种情况，需要增加额外的缩进，否则无法区分代码所在的缩进级别
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

1.1.3 **【推荐】** 如果包含定界符（括号，中括号，大括号）的表达式跨越多行，那么定界符的扩回符， 可以放置与最后一行的非空字符对齐或者与构造多行的开始第一个字符对齐。

正确示范

```
 # 与最后一行的非空字符对齐
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
```

```
    )

# 或者与开始构造多行的第一个字符对齐
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

1.1.4 **【推荐】** 对于会经常改动的函数参数、列表、字典定义，建议每行一个元素，并且每行增加一个 `,` 。

正确示范

```
 yes = ('y', 'Y', 'yes', 'TRUE', 'True', 'true', 'On', 'on', '1')  # 基本不再
改变

kwlist = [
    'False',
    'None',
    'True',
    'and',
    'as',
    'assert',
    ...
    'yield',  # 最后一个元素也增加一个逗号，方便以后diff不显示此行
]

person = {
    'name': 'bob',
    'age': 12,      # 可能经常增加字段
}
```

错误示范

```
 # 关键字会不断增加，每个元素都换行
kwlist = ['False', 'None', 'True', 'and', 'as',
          'assert', 'async', ...
]
```

```
person = {'name': 'bob', 'age': 12}
```

1.1.5 **【可选】** 对于 if 判断，一般来说尽量不要放置过多的判断条件。换行时增加 4 个额外的空格。
pycodestyle: `E129 visually indented line with same indent as next logical line`.

> *备注：PEP8没有明确规定，以下几种都是允许的。建议使用前面2种方法，后2种会与已有的开源工具冲突。*

正确示范

```
 # 更推荐：在续行中，增加额外的缩进级别。允许 and 操作符在前
if (this_is_one_thing
        and that_is_another_thing):
    do_something()

# 更推荐：在续行中，增加额外的缩进级别
if (this_is_one_thing and
        that_is_another_thing):
    do_something()

# 允许：与定界符（括号）对齐，不需要额外的缩进
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 允许：增加注释，编辑器会提示语法高亮，有助于区分
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()
```

1.2 每行最大长度

1.2.1 **【必须】** 每行最多不超过 `120` 个字符。每行代码最大长度限制的根本原因是过长的行会导致阅读障碍，使得缩进失效。 pylint: `line-too-long`.

除了以下两种情况例外：

1. 导入模块语句。
2. 注释中包含的URL。

如果需要一个长的字符串，可以用括号实现隐形连接。

正确示范

```
x = ('This will build a very long long '
    'long long long long long long string')
```

1.3 空白符

1.3.1 **【必须】** 在表达式的赋值符号、操作符左右至少有一个空格。

正确示范

```
x = y + 1
```

错误示范

```
x=y+1
x = y+1
```

1.3.2 **【必须】** 禁止行尾空白。 pylint: `trailing-whitespace`.

行尾空白虽然不会造成功能性异常，但是这些空白字符会被源码管理系统标记出来显示为差异，对开发人员造成困恼。

正确示范

```
# YES: 尾部没有空白符号
+ para = {}
+ para = {}  # comment
```

错误示范

```
 # No: 结尾存在多余空白符号
- para = {}•••••
- para = {}  # comment••••••
```

1.4 操作符

1.4.1 **【推荐】** Python 没有三目操作符，对于二目操作符来说，操作符允许在换行符之后出现。

> *备注：pycodestyle 工具与此条目相反，PEP8 推荐操作符在这之前，更具备可读性。*PEP8: Should a Line Break Before or After a Binary Operator?*。 屏蔽 pycodestyle:* `W503 line break before binary operator`

正确示范

```
 # YES: 易于将运算符与操作数匹配，可读性高
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

错误示范

```
 # No: 运算符的位置远离其操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

1.5 括号

1.5.1 **【必须】** tuple 元组不允许逗号结尾，显式增加括号规避。即使一个元素也加上括号。
pylint: `trailing-comma-tuple`.

行尾的逗号可能导致本来要定义一个简单变量，结果变成 tuple 变量。

正确示范

```
 trailingcomma = (['f'],)

return (1,)
```

错误示范

```
 trailingcomma = ['f'],   # tuple

return 1,
```

1.6 空行

1.6.1 **【必须】** 模块中的一级函数和类定义之间，需要`空两行`。 pycodestyle: `E302 expected 2 blank lines`.

1.6.2 **【必须】** 类中函数定义之间，`空一行`。 pycodestyle: `E302 expected 1 blank line`.

1.6.3 **【必须】** 源文件末尾有且仅有`一行空行`。 pylint: `missing-final-newline`, `trailing-newlines`.

1.6.4 **【必须】** 通常每个语句应该独占一行。 pylint: `multiple-statements`.

如果测试结果与测试语句在一行放得下，你也可以将它们放在同一行。 如果是`if`语句, 只有在没有`else`时才能这样做。 特别地，绝不要对`try`/`except`这样做，因为`try`和`except`不能放在同一行。

正确示范

```
 if foo:
    bar(foo)
else:
    baz(foo)

try:
    bar(foo)
except ValueError:
    baz(foo)
```

错误示范

```
 if foo: bar(foo)
else:   baz(foo)

try:               bar(foo)
except ValueError: baz(foo)

try:
  bar(foo)
except ValueError: baz(foo)
```

1.6.5 **【推荐】** 可以在代码段中的 空一行 来区分不同业务逻辑块。

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

This module does stuff.
"""


import os


def foo():
    pass


class MyClass():

    def __init__(self):
        pass

    def foo(self):
        pass


class AnotherClass(object):
    """Another class.
```

```
    This is some comments for another class
    """

    def __init__(self,
                 a,
                 b):
        if a > b:
            self._min = b
            self._max = a

        else:
            self._min = a
            self._max = b

        self._gap = self._max = self._min

    def foo(self):
        pass
```

1.7 源文件编码

1.7.1 **【必须】** 源文件编码需统一使用 `UTF-8` 编码，以下内容需要增加到每一个python文件的头部。

```
# -*- coding: utf-8 -*-
```

1.7.2 **【必须】 避免不同操作系统对文件换行处理的方式不同，一律使用** `LF` 。 pylint: `mixed-line-endings` , `unexpected-line-ending-format` .

1.8 Shebang

1.8.1 **【必须】** 程序的main文件应该以 `#!/usr/bin/env python2` 或者 `#!/usr/bin/env python3` 开始， 可以同时支持Python2、Python3的 `#!/usr/bin/env python` 。

非程序入口的文件不应该出现 Shebang 。

正确示范

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
```

1.9 模块引用(import)

1.9.1 **【必须】** 每个导入应该独占一行。 pylint: `multiple-imports`.

正确示范

```
 import os
import sys
```

错误示范

```
 import os, sys
```

1.9.2 **【必须】** 导入总应该放在文件顶部，位于模块注释和文档字符串之后，模块全局变量和常量之前。 pylint: `wrong-import-order`.

导入应该按照从最通用到最不通用的顺序分组, 每个分组之间，需要空一行:

- 标准库导入
- 第三方库导入
- 本地导入

每种分组中，建议每个模块的完整包路径按 ***字典序*** 排序，并忽略大小写。

正确示范

```
 import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

1.9.3 **【必须】** 避免使用 `from <module> import \*`，因为可能会造成命名空间的污染。
pylint: `wildcard-import`.

1.9.4 **【必须】** 禁止导入了模块却不使用它。 pylint: `unused-import`.

正确示范

```
 import os  # used

dir_path = os.path.abspath('.')
```

错误示范

```
 import os   # unused !

# dir_path = os.path.abspath('.')
```

1.10 模块中的魔术变量(dunders)

1.10.1 **【必须】** 对于两个 `\_` 开头和两个 `\_` 结尾的变量， 如 `\__all_\_`， `\__author_\_`，`\__version_\_` 等，应该放在模块文档之后， 其他模块导入之前（`\__future_\_` 除外）。

1.10.2 **【必须】** Python 要求 `future` 导入必须出现在其他模块导入之前。 pylint: `misplaced-future`.

正确示范

```
 # -*- coding: utf-8 -*-
#
# Copyright @ 2020 Tencent.com

"""This is the example module.

This module does stuff.
"""
```

```
from __future__ import barry_as_FLUFL

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

import os
import sys
```

1.11 注释

> *有效的注释有助于帮助开发者更快地理解代码，模块，函数，方法，以及行内注释的都有各自的风格。*

1.11.1 **【必须】** 所有 `#` 开头的注释，必须与所在的代码块同级，并置放在代码之上。 pycodestyle: `E262 inline comment should start with '# '`.

1.11.2 **【必须】** 注释的每一行都应该以 `#` 和一个空格开头。 pycodestyle: `E266 too many leading '#' for block comment`, `E262 inline comment should start with '# '`.

1.11.3 **【必须】** 行内注释 `#` 与代码离开至少2个空格。 pycodestyle: `E261 at least two spaces before inline comment`.

1.11.4 **【必须】** 块注释：对于复杂的操作，可以在代码之前写若干行注释，对简单的代码，可以放在行内。与代码离开至少2个空格。

正确示范

```
 # this is a very complex operation, please
# read this carefully

if i & (i-1) == 0:
    # do my job ...

# 单行注释，为可读性，至少离开代码2个空格
x = x + 1                 # Compensate for border
```

1.11.5 **【必须】** TODO 注释需要加上名字。

TODO注释应该在所有开头处包含 `TODO` 字符串，紧跟着是用括号括起来的你的名字， email地址或其它标识符，然后是一个可选的冒号。 接着必须有一行注释，解释要做什么。 主要目的是为了有一个统一的TODO格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节)。 写了TODO注释并不保证写的人会亲自解决问题。 当你写了一个TODO，请注上你的名字。

> *为临时代码使用TODO注释, 它是一种短期解决方案。常见的IDE在提交代码时， 会检查变更中包含了TODO并提醒开发者，防止提交是忘记还有未完成的代码。 如果TODO是 将来做某事 的形式, 那么请确保包含一个指定的日期或者一个特定的事件（条件）。 相同地，也可以留下 `FIXME`, `NOTES` 注释。*

正确示范

```
 # TODO(zhangsan): Change this to use relations.
# FIXME(zhangsan@xx.com): Please fix me here.
# NOTES(zhangsan): This is some notes.
```

1.12 文档字符串

Docstring 文档字符串提供了将文档与 Python 模块，函数，类和方法相关联的便捷方法。

```
def foobar():
   """Return a foobang

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

PEP 257 全面描述了文档字符串的风格。

1.12.1 **【推荐】** 需对外发布的public 模块，函数，类，方法等需要包含文档字符串。内部使用的方法，函数等，要求使用简单的注释描述功能。 pylint: `missing-module-docstring`, `missing-class-docstring`, `missing-function-docstring`.

一个函数或方法，如果可以直接被其他开发者使用，需要提供文档明确其含义，需要指出输入，输出，以及异常内容。

1.12.2 **【必须】** 第一行应为文档名，空一行后，输入文档描述。

1.12.3 **【推荐】** 在使用文档字符串时，推荐使用 `reStructuredText` 风格类型。

正确示范

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
   """Fetches rows from a Bigtable.

   Retrieves rows pertaining to the given keys from the Table instance
   represented by big_table.  Silly things may happen if
   other_silly_variable is not None.
```

```
    :param big_table: An open Bigtable Table instance.
    :param keys: A sequence of strings representing the key of each table
row
        to fetch.
    :param other_silly_variable: Another optional variable, that has a much
        longer name than the other args, and which does nothing.

    :return: A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    :raises ValueError: if `keys` is empty.
    :raises IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

1.12.4 **【推荐】** 类应该在其定义下有一个用于描述该类的文档字符串。 如果类有公共属性(Attributes)，那么文档中应该有一个属性(Attributes)段， 并且应该遵守和函数参数相同的格式。

正确示范

```
 class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    :ivar likes_spam: A boolean indicating if we like SPAM or not.
    :ivar eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

1.13 类型提示

Python 是动态语言，在运行时无需指定变量类型。 虽然运行时不会执行函数与变量类型注解，但类型提示有助于阅读代码、重构、静态代码检查与IDE的语法提示。 推荐在项目中使用该特性。 更多使用可以参考 类型标注支持。

类型提示示例代码

```
 from typing import List


class Container(object):
    def __init__(self) -> None:
        self.elements: List[int] = []

    def append(self, element: int) -> None:
        self.elements.append(element)


def greeting(name: str) -> str:
    return 'Hello ' + name


# 变量定义
lang: str = 'zh'
success_code: int = 0
```

1.13.1 **【必须】** 模块级变量，类和实例变量以及局部变量的注释应在冒号后面有一个空格。 pycodestyle: `E231 missing whitespace after ':'`.

1.13.2 **【必须】** 冒号前不应有空格。

1.13.3 **【必须】** 如果有赋值符，则等号在两边应恰好有一个空格。 pycodestyle: `E225 missing whitespace around operator`.

正确示范

```
 code: int = 10

class Point(object):
    coords: Tuple[int, int] = (0, 0)
    label: str = '<unknown>'

def broadcast(servers: Sequence[Server],
              message: str = 'spaces around equality sign') -> None:
    pass
```

错误示范

```
 code:int  # No space after colon
code : int  # Space before colon

class Test(object):
    result: int=0  # No spaces around equality sign
```

1.13.4 **【推荐】** 当使用类型提示出现循环引用时，可以在导入的头部使用 `if typing.TYPE_CHECKING`，且对类型注解使用 `双引号` 或 `单引号` 进行修饰。

正确示范

```
 import typing

if typing.TYPE_CHECKING:  # 运行时不导入
    # For type annotation
    from typing import Any, Dict, List, Sequence  # NOQA
    from sphinx.application import Sphinx  # NOQA


class Parser(docutils.parsers.Parser):

    def set_application(self, app: "Sphinx") -> None:  # 同时采用引号
        pass
```

1.14 字符串

1.14.1 **【推荐】** 即使参数都是字符串, 也要使用%操作符或者格式化方法格式化字符串。不过也不能一概而论, 你需要在+和%之间权衡。

正确示范

```
 # 更推荐
x = f'name: {name}; score: {n}'  # Python3.6+ 以上支持
x = 'name: {name}; score: {n}'.format(name=name, n=n)
x = 'name: {name}; score: {n}'.format(**{"name": name, "n": n})
x = 'name: %(name)s; score: %(n)d' % {"name": name, "n": n}

# 可接受
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}!'.format(imperative, expletive)
x = 'name: %s; score: %d' % (name, n)
x = 'name: {}; score: {}'.format(name, n)
```

错误示范

```
 x = '%s%s' % (a, b)  # use + in this case
x = '{}{}'.format(a, b)  # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

1.14.2 **【推荐】** 避免在循环中用 `+` 和 `+=` 操作符来累加字符串。 由于字符串是不可变的, 这样做会创建不必要的临时对象, 并且导致二次方而不是线性的运行时间。 作为替代方案, 你可以将每个子串加入列表, 然后在循环结束后用 `.join` 连接列表。 (也可以将每个子串写入一个 io.StringIO 缓存中。) pylint: `consider-using-join`.

正确示范

```
 items = ['<table>']
for last_name, first_name in employee_list:
    items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
items.append('</table>')
employee_table = ''.join(items)
```

错误示范

```
 employee_table = '<table>'
for last_name, first_name in employee_list:
    employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
employee_table += '</table>'
```

1.14.3 **【推荐】** 在同一个文件中, 保持使用字符串引号的一致性。 使用单引号 `'` 或者双引号 `"` 引用字符串, 并在同一文件中一直沿用这种风格。 当字符串中包含单引号或者双引号时，为提高可读性，使用另外一种引号，代替转义字符。

正确示范

```
 Python('Why are you hiding your eyes?')
Gollum("I'm scared of lint errors.")
Narrator('"Good!" thought a happy Python reviewer.')
```

错误示范

```
 Python("Why are you hiding your eyes?")
Gollum('The lint. It burns. It burns us.')
Gollum("Always the great lint. Watching. Watching.")
```

1.14.4 **【必须】** 如果要引用的字符串为多行时，需要使用双引号引用字符串。

1.14.5 **【必须】** 文档字符串（docstring）必须使用三重双引号 `"""` 。

1.14.6 **【可选】** 避免在代码中使用三重引号 `"""` ，因为当使用三重引号时，缩进方式与其他部分不一致，容易引起误导。

正确示范

```
 print("This is much nicer.\n"
      "Do it this way.\n")
```

错误示范

```
 print("""This is pretty ugly.
Don't do this.
""")
```

1.14.7 **【推荐】** 检查前缀和后缀时，使用 `.startswith()` 和 `.endswith()` 代替字符串切片。

正确示范

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

错误示范

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

1.15 文件和sockets

1.15.1 **【必须】** 在文件和sockets结束时，显式的关闭它。

除文件外， sockets或其他类似文件的对象在没有必要的情况下打开，会有许多副作用，例如：

1. 它们可能会消耗有限的系统资源，如文件描述符。 如果这些资源在使用后没有及时归还系统，那么用于处理这些对象的代码会将资源消耗殆尽。
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作。
3. 仅仅是从逻辑上关闭文件和sockets，那么它们仍然可能会被其共享的程序在无意中进行读或者写操作。只有当它们真正被关闭后，对于它们尝试进行读或者写操作将会抛出异常, 并使得问题快速显现出来。

而且，幻想当文件对象析构时，文件和sockets会自动关闭，试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的。因为有如下原因：

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的Python实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长。

2. 对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪, 包含有全局变量等)。

1.15.2 **【推荐】** 推荐使用 `with` 语句管理文件。

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print(line)
```

对于不支持使用 `with` 语句的类似文件的对象,使用 `contextlib.closing()` :

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as
front_page:
    for line in front_page:
        print(line)
```

Legacy AppEngine 中Python 2.5的代码如使用 `with` 语句, 需要添加 `from \__future_\_ import with_statement` .

1.16 访问控制

> *在Python中，对于琐碎又不太重要的访问函数, 应该直接使用公有变量来取代它们，这样可以避免额外的函数调用开销。 当添加更多功能时，也可以用属性(property)来保持语法的一致性。*

1.16.1 **【推荐】** 如果访问属性后需要复杂的逻辑处理，或者变量的访问开销很显著， 那么应该使用像 `get_foo()` 和 `set_foo()` 这样的函数调用。 如果之前的代码行为已经通过属性(property)访问，那么就不要将新的访问函数与属性绑定。 否则，任何试图通过老方法访问变量的代码就没法运行，使用者也就会意识到复杂性发生了变化。 （如果可以重构这个代码是最好的了）

1.17 Main

> *即使是一个打算被用作脚本的文件, 也应该是可导入的。 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行，这是一种副作用。 主功能应该放在一个main()函数中。*

1.17.1 **【必须】** 所有的文件都应该可以被导入。 对不需要作为程序入口地方添加 `if \__name_\_ == '\__main_\_'` 。

在Python中，pydoc以及单元测试要求模块必须是可导入的。 你的代码应该在执行主程序前总是检查 `if \__name_\_ == '\__main_\_'` ， 这样当模块被导入时主程序就不会被执行。 所有的顶级代码在模块导入时都会被执行。 要小心不要去调用函数，创建对象，或者执行那些不应该在使用pydoc时执行的操作。

正确示范

```
 def main():
      ...

if __name__ == '__main__':
    main()
```

1.18 命名

> *module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_VAR_NAME, instance_var_name, function_parameter_name, local_var_name.*

应该避免的名称：

1. 单字符名称，除了计数器和迭代器。
2. 包/模块名中的连字符(-)。
3. 双下划线开头并结尾的名称(Python保留，例如*__init__*)。

1.18.1 **【推荐】** 命名约定规则如下： pylint: `invalid-name`.

1. 所谓`内部(Internal)`表示仅模块内可用, 或者, 在类内是保护或私有的。
2. 用单下划线`(\_)`开头表示模块变量或函数是protected的(使用from module import *时不会包含)。
3. 用双下划线`(\_\_)`开头的实例变量或方法表示类内私有。
4. 将相关的类和顶级函数放在同一个模块里。不像Java，没必要限制一个类一个模块。
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格)，但是模块名应该用小写加下划线的方式(如lower_with_under.py)。尽管已经有很多现存的模块使用类似于CapWords.py这样的命名，但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致，这会让人困扰。

Python之父Guido推荐的规范：

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	

Type	Public	Internal
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

二、编码规范

2.1 三目运算符

2.1.1 **【必须】** 三目操作符判断，python 不支持三目运算符，但可使用如下方式，禁止使用复杂难懂的逻辑判断。

正确示范

```
x = a if a >= b else b
```

错误示范

```
x = a >= b and a or b
```

2.2 None 条件的判断

2.2.1 **【必须】** 为提升可读性，在判断条件中应使用 `is not`，而不使用 `not ... is`。 pycodestyle: `E714 test for object identity should be 'is not'`.

正确示范

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

错误示范

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

2.3 lambda匿名函数

2.3.1 **【必须】** 使用 def 定义简短函数而不是使用 lambda。 pycodestyle: `E731 do not assign a lambda expression, use a def`.

使用def的方式有助于在trackbacks中打印有效的类型信息，明确使用 `f` 函数而不是一个lambda的调用。

正确示范

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

错误示范

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

2.4 异常

2.4.1 **【必须】** 异常类继承自 `Exception`，而不是 `BaseException`。

2.4.2 **【必须】** 使用新版本抛出异常的方式，禁止使用废弃的方式。 pycodestyle: `W602 deprecated form of raising exception`.

正确示范

```
raise ValueError('message')
```

错误示范

```
raise ValueError, 'message'
```

2.4.3 **【必须】** 捕获异常时，需要指明具体异常，而不是捕获所有异常。 除非已经在当前线程的最外层（记得还是要打印一条traceback）。 pylint: `broad-except`, `bare-except`.

正确示范

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

try:
    do_something()
except Exception as ex:
    print(traceback.format_exc())
    do_handle_exception()
```

错误示范

```
 try:
    import platform_specific_module
except Exception:  # broad-except
    platform_specific_module = None

try:
    do_something()
except Exception:  # 框架等未明确异常场景，建议增加 traceback 打印
    platform_specific_module = None
```

2.4.4 **【推荐】** 建议在代码中用异常替代函数的错误返回码。

正确示范

```
 def write_data():
    if check_file_exist():
        do_something()
    else:
        raise FileNotExist()
```

错误示范

```
 def write_data():
    if check_file_exist():
        do_something()
        return 0
    else:
        return FILE_NOT_EXIST
```

2.4.5 **【推荐】** 在 `except` 子句中重新抛出原有异常时，不能用 `raise ex`，而是用 `raise`。

正确示范

```
 try:
    raise MyException()
except MyException as ex:
    try_handle_exception()
    raise  # 可以保留原始的 traceback
```

```
try:
    raise MyException()
except MyException as ex:
    raise AnotherException(str(ex))  # 允许的，建议保留好之前的异常栈信息，用于定位问题
```

错误示范

```
 try:
    raise MyException()
except MyException as ex:
    try_handle_exception()
    raise ex  # 异常栈信息从这里开始，原始的raise异常栈信息消息
```

2.4.6 **【推荐】** 所有 `try` / `except` 子句的代码要尽可的少，以免屏蔽其他的错误。

正确示范

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

错误示范

```
 try:
    # 范围太广
    return handle_value(collection[key])
except KeyError:
    # 会捕捉到 handle_value() 中的 KeyError
    return key_not_found(key)
```

2.5 条件表达式

2.5.1 **【推荐】** 函数或者方法在没有返回时要明确返回`None`。 pylint: `inconsistent-return-statements`.

正确示范

```
 def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

错误示范

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

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

2.5.2 **【推荐】** 对于未知的条件分支或者不应该进入的分支，建议抛出异常，而不是返回一个值（比如说`None`或`False`）。

正确示范

```
 def f(x):
    if x in ('SUCCESS',):
        return True
    else:
        raise MyException()  # 如果一定不会走到的条件，可以增加异常，防止将来未知的语句执行。
```

错误示范

```
 def f(x):
    if x in ('SUCCESS',):
        return True
    return None
```

2.5.3 **【可选】** `if` 与 `else` 尽量一起出现，而不是全部都是 `if` 子句。

正确示范

```
 if condition:
    do_something()
else:
    # 增加说明注释
    pass

if condition:
    do_something()
# 这里增加注释说明为什么不用写else子句
# else:
#     pass
```

错误示范

```
 if condition:
    do_something()
if another_condition:  # 不能确定是否笔误为 elif ，还是开启全新一个if条件
    do_another_something()
else:
```

```
    do_else_something()
```

2.6 True/False 布尔运算

2.6.1 **【必须】** 不要用 `==` 与 `True`、 `False` 进行布尔运算。 pylint: `singleton-comparison`.

正确示范

```
 if greeting:
   pass
```

错误示范

```
 if greeting == True:
   pass

if greeting is True:  # Worse
   pass
```

2.6.2 **【必须】** 对序列（字符串、列表 、元组），空序列为 false 的情况。 pylint: `len-as-condition`.

正确示范

```
 if not seq:
   pass

if seq:
   pass
```

错误示范

```
 if len(seq):
   pass

if not len(seq):
   pass
```

2.7 列表推导式

2.7.1 **【必须】** 禁止超过1个for语句或过滤器表达式，否则使用传统 `for` 循环语句替代。

正确示范

```
 number_list = [1, 2, 3, 10, 20, 55]
odd = [i for i in number_list if i % 2 == 1]

result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
```

错误示范

```
 result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]  # for
语句
```

2.7.2 **【推荐】** 列表推导式适用于简单场景。 如果语句过长，每个部分应该单独置于一行： 映射表达式， for 语句， 过滤器表达式。

正确示范

```
 fizzbuzz = []
for n in range(100):
    if n % 3 == 0 and n % 5 == 0:
        fizzbuzz.append(f'fizzbuzz {n}')
    elif n % 3 == 0:
        fizzbuzz.append(f'fizz {n}')
```

```
    elif n % 5 == 0:
        fizzbuzz.append(f'buzz {n}')
    else:
        fizzbuzz.append(n)


for n in range(1, 11):
    print(n)
```

错误示范

```
 # 条件过于复杂，应该采用for语句展开
fizzbuzz = [
    f'fizzbuzz {n}' if n % 3 == 0 and n % 5 == 0
    else f'fizz {n}' if n % 3 == 0
    else f'buzz {n}' if n % 5 == 0
    else n
    for n in range(100)
]


[print(n) for n in range(1, 11)]  # 无返回值
```

2.8 函数

2.8.1 **【必须】** 模块内部禁止定义重复函数声明。 pylint: `function-redefined`.

禁止重复的函数定义。

错误示范

```
 def get_x(x):
    return x

def get_x(x):  # 模块内重复定义
    return x
```

2.8.2 **【必须】** 函数参数中，不允许出现可变类型变量作为默认值。 pylint: `dangerous-default-value`.

正确示范

```
def f(x=0, y=None, z=None):
   if y is None:
       y = []
   if z is None:
       z = {}
```

错误示范

```
def f(x=0, y=[], z={}):
   pass

def f(a, b=time.time()):
   pass
```

2.9 变量

2.9.1 **【必须】** 禁止定义了变量却不使用它。 pylint: `unused-variable`.

在代码里到处定义变量却没有使用它，不完整的代码结构看起来像是个代码错误。 即使没有使用，但是定义变量仍然需要消耗资源，并且对阅读代码的人也会造成困惑，不知道这些变量是要做什么的。

正确示范

```
def get_x_plus_y(x, y):
   return x + y
```

错误示范

```
some_unused_var = 42

# 定义了变量不意味着就是使用了
y = 10
y = 5
```

```
# 对自身的操作并不意味着使用了
z = 0
z = z + 1

# 未使用的函数参数
def get_x(x, y):
    return x
```

2.9.2 **【推荐】** 使用双下划线 `\_\_` 来代表不需要的变量，单下划线 `\_` 容易与 `gettext()` 函数的别名冲突。

正确示范

```
 path = '/tmp/python/foobar.txt'
dir_name, __ = os.path.split(path)
```

工具与配置

flake8

`flake8` 是一个结合了 pycodestyle，pyflakes，mccabe 检查 Python 代码规范的工具。

使用方法：

```
 flake8 {source_file_or_directory}
```

在项目中创建 `setup.cfg` 或者 `tox.ini` 或者 `.flake8` 文件，添加 `\[flake8\]` 部分。

推荐的配置文件如下：

```
 [flake8]
ignore =
    ;W503 line break before binary operator
    W503,
    ;E203 whitespace before ':'
```

```
    E203,

; exclude file
exclude =
    .tox,
    .git,
    __pycache__,
    build,
    dist,
    *.pyc,
    *.egg-info,
    .cache,
    .eggs

max-line-length = 120
```

如果需要屏蔽告警可以增加行内注释 `# noqa`，例如：

```
example = lambda: 'example'  # noqa: E731
```

pylint

`pylint` 是一个能够检查编码质量、编码规范的工具。

配置项较多，单独一个配置文件配置，详情可查阅：.pylintrc

使用方法：

```
pylint {source_file_or_directory}
```

如果遇到一些实际情况与代码冲突的，可以在行内禁用相关检查，例如 ：

```
try:
   do_something()
except Exception as ex:  # pylint: disable=broad-except
    pass
```

如果需要对多行的进行禁用规则，可以配套使用 `# pylint: disable=具体错误码` / `# pylint: enable=具体错误码` 。

```
 # pylint: disable=invalid-name
这里的代码块会被忽略相关的告警
app = Flask(__name__)
# pylint: enable=invalid-name
```

black

`black` 是一个官方的 Python 代码格式化工具。

使用方法：

```
 black {source_file_or_directory}
```

如果不想格式化部分代码，可以配套使用 `# fmt: off` / `# fmt: on` 临时关闭格式化。

```
 # fmt: off
在这的代码不会被格式化
# fmt: on
```

EditorConfig

EditorConfig 可以帮助开发同一项目下的跨多 IDE 的开发人员保持一致编码风格。

在项目的根目录下放置 `.editorconfig` 文件，可以让编辑器规范文件对格式。参考配置如下：

```
 # https://editorconfig.org

root = true

[*]
indent_style = space
indent_size = 4
trim_trailing_whitespace = true
insert_final_newline = true
charset = utf-8
end_of_line = lf

[*.py]
max_line_length = 120

[*.bat]
indent_style = tab
end_of_line = crlf

[LICENSE]
insert_final_newline = false

[Makefile]
indent_style = tab
```

支持常见的 IDE，配置说明及 IDE 的支持情况可参考官网 editorconfig.org。

TOC

前言

为了让不同编码习惯的开发者更好的协作配合，并且形成良好的基础编码规范与风格，我们以 PEP8 为基础，修改了陈旧的规则，丰富了示例，并整理了工作中常见的不规范操作，最终形成此 Python 编码规范与风格。

本规范适用于所有使用 Python 语言作为开发语言的软件产品。

由于 Python2 在 2020 年停止维护，建议新增的项目使用 Python3.6+，可以使用到更多的高级特性。如果项目有兼容性需求需要支持老版本 Python 的，那么不涉及的特性可以忽略。 本规范的示例采用符合 Python3.6+ 的语法。

- **必须（Mandatory）**：用户必须采用；
- **推荐（Preferable）**：用户理应采用，但如有特殊情况，可以不采用；
- **可选（Optional）**：用户可参考，自行决定是否采用；

未明确指明的则默认为 **必须（Mandatory）**。

一、编码风格

规范地代码布局有助于帮助开发者更容易地理解业务逻辑。

1.1 缩进

1.1.1 **【必须】** 对于每级缩进，统一要求使用4个 空格 ，而非 tab 键。 pylint: `bad-indentation` .

1.1.2 **【必须】** 续行，要求使用括号等定限界符，并且需要垂直对齐。

正确示范

```
 # 与定界（括号）符对齐
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 换行并增加4个额外的空格（一级缩进）
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂需要增加一级缩进
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

错误示范

```


# 当不使用垂直对齐时，第一行不允许加参数
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 下面这种情况，需要增加额外的缩进，否则无法区分代码所在的缩进级别
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

1.1.3 **【推荐】** 如果包含定界符（括号，中括号，大括号）的表达式跨越多行，那么定界符的扩回符， 可以放置与最后一行的非空字符对齐或者与构造多行的开始第一个字符对齐。

正确示范

```
 # 与最后一行的非空字符对齐
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )

# 或者与开始构造多行的第一个字符对齐
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

1.1.4 **【推荐】** 对于会经常改动的函数参数、列表、字典定义，建议每行一个元素，并且每行增加一个`,`。

正确示范

```
 yes = ('y', 'Y', 'yes', 'TRUE', 'True', 'true', 'On', 'on', '1')  # 基本不再
改变

kwlist = [
    'False',
    'None',
    'True',
    'and',
    'as',
    'assert',
    ...
    'yield',  # 最后一个元素也增加一个逗号，方便以后diff不显示此行
]

person = {
    'name': 'bob',
    'age': 12,      # 可能经常增加字段
}
```

错误示范

```
 # 关键字会不断增加，每个元素都换行
kwlist = ['False', 'None', 'True', 'and', 'as',
          'assert', 'async', ...
]

person = {'name': 'bob', 'age': 12}
```

1.1.5 **【可选】** 对于 if 判断，一般来说尽量不要放置过多的判断条件。换行时增加 4 个额外的空格。

pycodestyle: `E129 visually indented line with same indent as next logical line`.

> *备注：PEP8没有明确规定，以下几种都是允许的。建议使用前面2种方法，后2种会与已有的开源工具冲突。*

正确示范

```
 # 更推荐：在续行中，增加额外的缩进级别。允许 and 操作符在前
if (this_is_one_thing
        and that_is_another_thing):
    do_something()

# 更推荐：在续行中，增加额外的缩进级别
if (this_is_one_thing and
        that_is_another_thing):
    do_something()

# 允许：与定界符（括号）对齐，不需要额外的缩进
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 允许：增加注释，编辑器会提示语法高亮，有助于区分
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()
```

1.2 每行最大长度

1.2.1 **【必须】** 每行最多不超过 `120` 个字符。每行代码最大长度限制的根本原因是过长的行会导致阅读障碍，使得缩进失效。 pylint: `line-too-long`.

除了以下两种情况例外：

1. 导入模块语句。
2. 注释中包含的URL。

如果需要一个长的字符串，可以用括号实现隐形连接。

正确示范

```
x = ('This will build a very long long '
    'long long long long long long string')
```

1.3 空白符

1.3.1 **【必须】** 在表达式的赋值符号、操作符左右至少有一个空格。

正确示范

```
x = y + 1
```

错误示范

```
 x=y+1
x = y+1
```

1.3.2 **【必须】** 禁止行尾空白。 pylint: `trailing-whitespace`.

行尾空白虽然不会造成功能性异常，但是这些空白字符会被源码管理系统标记出来显示为差异，对开发人员造成困恼。

正确示范

```
 # YES: 尾部没有空白符号
+ para = {}
+ para = {}  # comment
```

错误示范

```
 # No: 结尾存在多余空白符号
- para = {}•••••
- para = {}  # comment••••••
```

1.4 操作符

1.4.1 **【推荐】** Python 没有三目操作符，对于二目操作符来说，操作符允许在换行符之后出现。

> *备注：pycodestyle 工具与此条目相反，PEP8 推荐操作符在这之前，更具备可读性。*PEP8: Should a Line Break Before or After a Binary Operator?*。 屏蔽 pycodestyle:* `W503 line break before binary operator`

正确示范

```
 # YES: 易于将运算符与操作数匹配，可读性高
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

错误示范

```
 # No: 运算符的位置远离其操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

1.5 括号

1.5.1 **【必须】** tuple 元组不允许逗号结尾，显式增加括号规避。即使一个元素也加上括号。
pylint: `trailing-comma-tuple`.

行尾的逗号可能导致本来要定义一个简单变量，结果变成 tuple 变量。

正确示范

```
 trailingcomma = (['f'],)

return (1,)
```

错误示范

```
 trailingcomma = ['f'],   # tuple

return 1,
```

1.6 空行

1.6.1 **【必须】** 模块中的一级函数和类定义之间，需要 空两行 。 pycodestyle: `E302 expected 2 blank lines`.

1.6.2 **【必须】** 类中函数定义之间， 空一行 。 pycodestyle: `E302 expected 1 blank line`.

1.6.3 **【必须】** 源文件末尾有且仅有 一行空行 。 pylint: `missing-final-newline`, `trailing-newlines`.

1.6.4 **【必须】** 通常每个语句应该独占一行。 pylint: `multiple-statements`.

如果测试结果与测试语句在一行放得下，你也可以将它们放在同一行。 如果是 `if` 语句, 只有在没有 `else` 时才能这样做。 特别地，绝不要对 `try` / `except` 这样做，因为 `try` 和 `except` 不能放在同一行。

正确示范

```
 if foo:
    bar(foo)
else:
    baz(foo)

try:
    bar(foo)
except ValueError:
    baz(foo)
```

错误示范

```
 if foo: bar(foo)
else:   baz(foo)

try:               bar(foo)
except ValueError: baz(foo)

try:
  bar(foo)
except ValueError: baz(foo)
```

1.6.5 **【推荐】** 可以在代码段中的`空一行`来区分不同业务逻辑块。

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

This module does stuff.
"""


import os


def foo():
    pass


class MyClass():

    def __init__(self):
```

```
        pass

    def foo(self):
        pass


class AnotherClass(object):
    """Another class.

    This is some comments for another class
    """

    def __init__(self,
                 a,
                 b):
        if a > b:
            self._min = b
            self._max = a

        else:
            self._min = a
            self._max = b

        self._gap = self._max = self._min

    def foo(self):
        pass
```

1.7 源文件编码

1.7.1 **【必须】** 源文件编码需统一使用 `UTF-8` 编码，以下内容需要增加到每一个python文件的头部。

```
 # -*- coding: utf-8 -*-
```

1.7.2 **【必须】 避免不同操作系统对文件换行处理的方式不同，一律使用** `LF` 。 pylint: `mixed-line-endings` , `unexpected-line-ending-format` .

1.8 Shebang

1.8.1 **【必须】** 程序的main文件应该以 `#!/usr/bin/env python2` 或者 `#!/usr/bin/env python3` 开始， 可以同时支持Python2、Python3的 `#!/usr/bin/env python` 。

非程序入口的文件不应该出现 Shebang 。

正确示范

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
```

1.9 模块引用(import)

1.9.1 **【必须】** 每个导入应该独占一行。 pylint: `multiple-imports` .

正确示范

```
 import os
import sys
```

错误示范

```
 import os, sys
```

1.9.2 **【必须】** 导入总应该放在文件顶部，位于模块注释和文档字符串之后，模块全局变量和常量之前。 pylint: `wrong-import-order` .

导入应该按照从最通用到最不通用的顺序分组, 每个分组之间，需要空一行:

- 标准库导入
- 第三方库导入
- 本地导入

每种分组中，建议每个模块的完整包路径按 ***字典序*** 排序，并忽略大小写。

正确示范

```
 import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

1.9.3 **【必须】** 避免使用 `from <module> import \*`，因为可能会造成命名空间的污染。
pylint: `wildcard-import`.

1.9.4 **【必须】** 禁止导入了模块却不使用它。 pylint: `unused-import`.

正确示范

```
 import os  # used

dir_path = os.path.abspath('.')
```

错误示范

```
 import os   # unused !

# dir_path = os.path.abspath('.')
```

1.10 模块中的魔术变量(dunders)

1.10.1 **【必须】** 对于两个 `\_` 开头和两个 `\_` 结尾的变量， 如 `\__all_\_`， `\__author_\_`，
`\__version_\_` 等，应该放在模块文档之后， 其他模块导入之前（ `\__future_\_` 除外）。

1.10.2 **【必须】** Python 要求 `future` 导入必须出现在其他模块导入之前。 pylint: `misplaced-future`.

正确示范

```
 # -*- coding: utf-8 -*-
#
# Copyright @ 2020 Tencent.com
```

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

This module does stuff.
"""

from __future__ import barry_as_FLUFL

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

import os
import sys
```

1.11 注释

> *有效的注释有助于帮助开发者更快地理解代码，模块，函数，方法，以及行内注释的都有各自的风格。*

1.11.1 **【必须】** 所有`#`开头的注释，必须与所在的代码块同级，并置放在代码之上。 pycodestyle: `E262 inline comment should start with '# '`.

1.11.2 **【必须】** 注释的每一行都应该以`#`和一个空格开头。 pycodestyle: `E266 too many leading '#' for block comment`, `E262 inline comment should start with '# '`.

1.11.3 **【必须】** 行内注释`#`与代码离开至少2个空格。 pycodestyle: `E261 at least two spaces before inline comment`.

1.11.4 **【必须】** 块注释：对于复杂的操作，可以在代码之前写若干行注释，对简单的代码，可以放在行内。与代码离开至少2个空格。

正确示范

```
 # this is a very complex operation, please
# read this carefully

if i & (i-1) == 0:
    # do my job ...

# 单行注释，为可读性，至少离开代码2个空格
x = x + 1                 # Compensate for border
```

1.11.5 **【必须】** TODO 注释需要加上名字。

TODO注释应该在所有开头处包含 `TODO` 字符串，紧跟着是用括号括起来的你的名字， email地址或其它标识符，然后是一个可选的冒号。 接着必须有一行注释，解释要做什么。 主要目的是为了有一个统一的TODO格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节)。 写了TODO注释并不保证写的人会亲自解决问题。 当你写了一个TODO，请注上你的名字。

> *为临时代码使用TODO注释, 它是一种短期解决方案。常见的IDE在提交代码时， 会检查变更中包含了TODO并提醒开发者，防止提交是忘记还有未完成的代码。 如果TODO是 将来做某事 的形式, 那么请确保包含一个指定的日期或者一个特定的事件（条件）。 相同地，也可以留下 `FIXME` , `NOTES` 注释。*

正确示范

```
 # TODO(zhangsan): Change this to use relations.
# FIXME(zhangsan@xx.com): Please fix me here.
# NOTES(zhangsan): This is some notes.
```

1.12 文档字符串

Docstring 文档字符串提供了将文档与 Python 模块，函数，类和方法相关联的便捷方法。

```
def foobar():
   """Return a foobang

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

PEP 257 全面描述了文档字符串的风格。

1.12.1 **【推荐】** 需对外发布的public 模块，函数，类，方法等需要包含文档字符串。内部使用的方法，函数等，要求使用简单的注释描述功能。 pylint: `missing-module-docstring` , `missing-class-docstring` , `missing-function-docstring` .

一个函数或方法，如果可以直接被其他开发者使用，需要提供文档明确其含义，需要指出输入，输出，以及异常内容。

1.12.2 **【必须】** 第一行应为文档名，空一行后，输入文档描述。

1.12.3 **【推荐】** 在使用文档字符串时，推荐使用 `reStructuredText` 风格类型。

正确示范

```
 def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    :param big_table: An open Bigtable Table instance.
    :param keys: A sequence of strings representing the key of each table
row
        to fetch.
    :param other_silly_variable: Another optional variable, that has a much
        longer name than the other args, and which does nothing.

    :return: A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    :raises ValueError: if `keys` is empty.
    :raises IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

1.12.4 **【推荐】** 类应该在其定义下有一个用于描述该类的文档字符串。 如果类有公共属性(Attributes)，那么文档中应该有一个属性(Attributes)段， 并且应该遵守和函数参数相同的格式。

正确示范

```
 class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    :ivar likes_spam: A boolean indicating if we like SPAM or not.
    :ivar eggs: An integer count of the eggs we have laid.
    """
```

```
    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

1.13 类型提示

Python 是动态语言，在运行时无需指定变量类型。 虽然运行时不会执行函数与变量类型注解，但类型提示有助于阅读代码、重构、静态代码检查与IDE的语法提示。 推荐在项目中使用该特性。 更多使用可以参考 类型标注支持。

类型提示示例代码

```
 from typing import List


class Container(object):
    def __init__(self) -> None:
        self.elements: List[int] = []

    def append(self, element: int) -> None:
        self.elements.append(element)


def greeting(name: str) -> str:
    return 'Hello ' + name


# 变量定义
lang: str = 'zh'
success_code: int = 0
```

1.13.1 **【必须】** 模块级变量，类和实例变量以及局部变量的注释应在冒号后面有一个空格。 pycodestyle: `E231 missing whitespace after ':'`.

1.13.2 **【必须】** 冒号前不应有空格。

1.13.3 **【必须】** 如果有赋值符，则等号在两边应恰好有一个空格。 pycodestyle: `E225 missing whitespace around operator`.

正确示范

```
 code: int = 10

class Point(object):
    coords: Tuple[int, int] = (0, 0)
    label: str = '<unknown>'

def broadcast(servers: Sequence[Server],
              message: str = 'spaces around equality sign') -> None:
    pass
```

错误示范

```
 code:int  # No space after colon
code : int  # Space before colon

class Test(object):
    result: int=0  # No spaces around equality sign
```

1.13.4 **【推荐】** 当使用类型提示出现循环引用时，可以在导入的头部使用 `if typing.TYPE_CHECKING`，且对类型注解使用 `双引号` 或 `单引号` 进行修饰。

正确示范

```
 import typing

if typing.TYPE_CHECKING:  # 运行时不导入
    # For type annotation
    from typing import Any, Dict, List, Sequence  # NOQA
    from sphinx.application import Sphinx  # NOQA


class Parser(docutils.parsers.Parser):

    def set_application(self, app: "Sphinx") -> None:  # 同时采用引号
        pass
```

1.14 字符串

1.14.1 **【推荐】** 即使参数都是字符串, 也要使用%操作符或者格式化方法格式化字符串。不过也不能一概而论, 你需要在+和%之间权衡。

正确示范

```
 # 更推荐
x = f'name: {name}; score: {n}'  # Python3.6+ 以上支持
x = 'name: {name}; score: {n}'.format(name=name, n=n)
x = 'name: {name}; score: {n}'.format(**{"name": name, "n": n})
x = 'name: %(name)s; score: %(n)d' % {"name": name, "n": n}

# 可接受
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}!'.format(imperative, expletive)
x = 'name: %s; score: %d' % (name, n)
x = 'name: {}; score: {}'.format(name, n)
```

错误示范

```
 x = '%s%s' % (a, b)  # use + in this case
x = '{}{}'.format(a, b)  # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

1.14.2 **【推荐】** 避免在循环中用 `+` 和 `+=` 操作符来累加字符串。 由于字符串是不可变的, 这样做会创建不必要的临时对象, 并且导致二次方而不是线性的运行时间。 作为替代方案, 你可以将每个子串加入列表, 然后在循环结束后用 `.join` 连接列表。 (也可以将每个子串写入一个 io.StringIO 缓存中。) pylint: `consider-using-join`.

正确示范

```
 items = ['<table>']
for last_name, first_name in employee_list:
    items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
items.append('</table>')
employee_table = ''.join(items)
```

错误示范

```
 employee_table = '<table>'
for last_name, first_name in employee_list:
    employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
employee_table += '</table>'
```

1.14.3 **【推荐】** 在同一个文件中, 保持使用字符串引号的一致性。 使用单引号 `'` 或者双引号 `"` 引用字符串, 并在同一文件中一直沿用这种风格。 当字符串中包含单引号或者双引号时，为提高可读性，使用另外一种引号，代替转义字符。

正确示范

```
 Python('Why are you hiding your eyes?')
Gollum("I'm scared of lint errors.")
Narrator('"Good!" thought a happy Python reviewer.')
```

错误示范

```
 Python("Why are you hiding your eyes?")
Gollum('The lint. It burns. It burns us.')
Gollum("Always the great lint. Watching. Watching.")
```

1.14.4 **【必须】** 如果要引用的字符串为多行时，需要使用双引号引用字符串。

1.14.5 **【必须】** 文档字符串（docstring）必须使用三重双引号 `"""` 。

1.14.6 **【可选】** 避免在代码中使用三重引号 `"""` ，因为当使用三重引号时，缩进方式与其他部分不一致，容易引起误导。

正确示范

```
 print("This is much nicer.\n"
      "Do it this way.\n")
```

错误示范

```
 print("""This is pretty ugly.
Don't do this.
""")
```

1.14.7 **【推荐】** 检查前缀和后缀时，使用 `.startswith()` 和 `.endswith()` 代替字符串切片。

正确示范

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

错误示范

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

1.15 文件和sockets

1.15.1 **【必须】** 在文件和sockets结束时，显式的关闭它。

除文件外， sockets或其他类似文件的对象在没有必要的情况下打开，会有许多副作用，例如：

1. 它们可能会消耗有限的系统资源，如文件描述符。 如果这些资源在使用后没有及时归还系统，那么用于处理这些对象的代码会将资源消耗殆尽。
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作。
3. 仅仅是从逻辑上关闭文件和sockets，那么它们仍然可能会被其共享的程序在无意中进行读或者写操作。只有当它们真正被关闭后，对于它们尝试进行读或者写操作将会抛出异常, 并使得问题快速显现出来。

而且，幻想当文件对象析构时，文件和sockets会自动关闭，试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的。因为有如下原因：

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的Python实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长。
2. 对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪, 包含有全局变量等)。

1.15.2 **【推荐】** 推荐使用 `with` 语句管理文件。

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print(line)
```

对于不支持使用 `with` 语句的类似文件的对象,使用 `contextlib.closing()` ：

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as
front_page:
    for line in front_page:
        print(line)
```

Legacy AppEngine 中Python 2.5的代码如使用 `with` 语句, 需要添加 `from \__future_\_ import with_statement` .

1.16 访问控制

> *在Python中，对于琐碎又不太重要的访问函数, 应该直接使用公有变量来取代它们，这样可以避免额外的函数调用开销。 当添加更多功能时，也可以用属性(property)来保持语法的一致性。*

1.16.1 **【推荐】** 如果访问属性后需要复杂的逻辑处理，或者变量的访问开销很显著， 那么应该使用像 `get_foo()` 和 `set_foo()` 这样的函数调用。 如果之前的代码行为已经通过属性(property)访问，那么就不要将新的访问函数与属性绑定。 否则，任何试图通过老方法访问变量的代码就没法运行，使用者也就会意识到复杂性发生了变化。 （如果可以重构这个代码是最好的了）

1.17 Main

> *即使是一个打算被用作脚本的文件, 也应该是可导入的。 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行，这是一种副作用。 主功能应该放在一个main()函数中。*

1.17.1 **【必须】** 所有的文件都应该可以被导入。 对不需要作为程序入口地方添加 `if \__name_\_ == '\__main_\_'` 。

在Python中，pydoc以及单元测试要求模块必须是可导入的。 你的代码应该在执行主程序前总是检查 `if \__name_\_ == '\__main_\_'` ， 这样当模块被导入时主程序就不会被执行。 所有的顶级代码在模块导入时都会被执行。 要小心不要去调用函数，创建对象，或者执行那些不应该在使用pydoc时执行的操作。

正确示范

```
 def main():
      ...

if __name__ == '__main__':
    main()
```

1.18 命名

> *module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_VAR_NAME, instance_var_name, function_parameter_name, local_var_name.*

应该避免的名称：

1. 单字符名称，除了计数器和迭代器。
2. 包/模块名中的连字符(-)。
3. 双下划线开头并结尾的名称(Python保留，例如*__init__*)。

1.18.1 **【推荐】** 命名约定规则如下： pylint: `invalid-name`.

1. 所谓`内部(Internal)`表示仅模块内可用, 或者, 在类内是保护或私有的。
2. 用单下划线`(\_)`开头表示模块变量或函数是protected的(使用from module import *时不会包含)。
3. 用双下划线`(\_\_)`开头的实例变量或方法表示类内私有。
4. 将相关的类和顶级函数放在同一个模块里。不像Java，没必要限制一个类一个模块。
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格)，但是模块名应该用小写加下划线的方式(如lower_with_under.py)。尽管已经有很多现存的模块使用类似于CapWords.py这样的命名，但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致，这会让人困扰。

Python之父Guido推荐的规范：

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	

Type	Public	Internal
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

二、编码规范

2.1 三目运算符

2.1.1 **【必须】** 三目操作符判断，python 不支持三目运算符，但可使用如下方式，禁止使用复杂难懂的逻辑判断。

正确示范

```
x = a if a >= b else b
```

错误示范

```
x = a >= b and a or b
```

2.2 None 条件的判断

2.2.1 **【必须】** 为提升可读性，在判断条件中应使用 `is not`，而不使用 `not ... is`。 pycodestyle: `E714 test for object identity should be 'is not'`.

正确示范

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

错误示范

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

2.3 lambda匿名函数

2.3.1 **【必须】** 使用 def 定义简短函数而不是使用 lambda。 pycodestyle: `E731 do not assign a lambda expression, use a def`.

使用def的方式有助于在trackbacks中打印有效的类型信息，明确使用 `f` 函数而不是一个lambda的调用。

正确示范

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

错误示范

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

2.4 异常

2.4.1 **【必须】** 异常类继承自 `Exception`，而不是 `BaseException`。

2.4.2 **【必须】** 使用新版本抛出异常的方式，禁止使用废弃的方式。 pycodestyle: `W602 deprecated form of raising exception`.

正确示范

```
raise ValueError('message')
```

错误示范

```
raise ValueError, 'message'
```

2.4.3 **【必须】** 捕获异常时，需要指明具体异常，而不是捕获所有异常。 除非已经在当前线程的最外层（记得还是要打印一条traceback）。 pylint: `broad-except`, `bare-except`.

正确示范

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

try:
    do_something()
except Exception as ex:
    print(traceback.format_exc())
    do_handle_exception()
```

错误示范

```
 try:
    import platform_specific_module
except Exception:  # broad-except
    platform_specific_module = None

try:
    do_something()
except Exception:  # 框架等未明确异常场景，建议增加 traceback 打印
    platform_specific_module = None
```

2.4.4 **【推荐】** 建议在代码中用异常替代函数的错误返回码。

正确示范

```
 def write_data():
    if check_file_exist():
        do_something()
    else:
        raise FileNotExist()
```

错误示范

```
 def write_data():
    if check_file_exist():
        do_something()
        return 0
    else:
        return FILE_NOT_EXIST
```

2.4.5 **【推荐】** 在 `except` 子句中重新抛出原有异常时，不能用 `raise ex`，而是用 `raise`。

正确示范

```
 try:
    raise MyException()
except MyException as ex:
    try_handle_exception()
    raise  # 可以保留原始的 traceback
```

```
try:
    raise MyException()
except MyException as ex:
    raise AnotherException(str(ex))  # 允许的，建议保留好之前的异常栈信息，用于定位问题
```

错误示范

```
 try:
    raise MyException()
except MyException as ex:
    try_handle_exception()
    raise ex  # 异常栈信息从这里开始，原始的raise异常栈信息消息
```

2.4.6 **【推荐】** 所有 `try` / `except` 子句的代码要尽可的少，以免屏蔽其他的错误。

正确示范

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

错误示范

```
 try:
    # 范围太广
    return handle_value(collection[key])
except KeyError:
    # 会捕捉到 handle_value() 中的 KeyError
    return key_not_found(key)
```

2.5 条件表达式

2.5.1 **【推荐】** 函数或者方法在没有返回时要明确返回`None`。 pylint: `inconsistent-return-statements`.

正确示范

```
 def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

错误示范

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

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

2.5.2 **【推荐】** 对于未知的条件分支或者不应该进入的分支，建议抛出异常，而不是返回一个值（比如说`None`或`False`）。

正确示范

```
 def f(x):
    if x in ('SUCCESS',):
        return True
    else:
        raise MyException()  # 如果一定不会走到的条件，可以增加异常，防止将来未知的语句执行。
```

错误示范

```
 def f(x):
    if x in ('SUCCESS',):
        return True
    return None
```

2.5.3 **【可选】** `if` 与 `else` 尽量一起出现，而不是全部都是 `if` 子句。

正确示范

```
 if condition:
    do_something()
else:
    # 增加说明注释
    pass

if condition:
    do_something()
# 这里增加注释说明为什么不用写else子句
# else:
#     pass
```

错误示范

```
 if condition:
    do_something()
if another_condition:  # 不能确定是否笔误为 elif ，还是开启全新一个if条件
    do_another_something()
else:
```

```
    do_else_something()
```

2.6 True/False 布尔运算

2.6.1 **【必须】** 不要用 `==` 与 `True`、 `False` 进行布尔运算。 pylint: `singleton-comparison`.

正确示范

```
 if greeting:
   pass
```

错误示范

```
 if greeting == True:
   pass

if greeting is True:  # Worse
   pass
```

2.6.2 **【必须】** 对序列（字符串、列表 、元组），空序列为 false 的情况。 pylint: `len-as-condition`.

正确示范

```
 if not seq:
   pass

if seq:
   pass
```

错误示范

```
 if len(seq):
   pass

if not len(seq):
   pass
```

2.7 列表推导式

2.7.1 **【必须】** 禁止超过1个for语句或过滤器表达式，否则使用传统 `for` 循环语句替代。

正确示范

```
 number_list = [1, 2, 3, 10, 20, 55]
odd = [i for i in number_list if i % 2 == 1]

result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
```

错误示范

```
 result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]  # for
语句
```

2.7.2 **【推荐】** 列表推导式适用于简单场景。 如果语句过长，每个部分应该单独置于一行： 映射表达式， for 语句， 过滤器表达式。

正确示范

```
 fizzbuzz = []
for n in range(100):
    if n % 3 == 0 and n % 5 == 0:
        fizzbuzz.append(f'fizzbuzz {n}')
    elif n % 3 == 0:
        fizzbuzz.append(f'fizz {n}')
```

```
    elif n % 5 == 0:
        fizzbuzz.append(f'buzz {n}')
    else:
        fizzbuzz.append(n)


for n in range(1, 11):
    print(n)
```

错误示范

```
 # 条件过于复杂，应该采用for语句展开
fizzbuzz = [
    f'fizzbuzz {n}' if n % 3 == 0 and n % 5 == 0
    else f'fizz {n}' if n % 3 == 0
    else f'buzz {n}' if n % 5 == 0
    else n
    for n in range(100)
]


[print(n) for n in range(1, 11)]  # 无返回值
```

2.8 函数

2.8.1 **【必须】** 模块内部禁止定义重复函数声明。 pylint: `function-redefined`.

禁止重复的函数定义。

错误示范

```
 def get_x(x):
    return x

def get_x(x):  # 模块内重复定义
    return x
```

2.8.2 **【必须】** 函数参数中，不允许出现可变类型变量作为默认值。 pylint: `dangerous-default-value`.

正确示范

```
def f(x=0, y=None, z=None):
   if y is None:
       y = []
   if z is None:
       z = {}
```

错误示范

```
def f(x=0, y=[], z={}):
   pass

def f(a, b=time.time()):
   pass
```

2.9 变量

2.9.1 **【必须】** 禁止定义了变量却不使用它。 pylint: `unused-variable`.

在代码里到处定义变量却没有使用它，不完整的代码结构看起来像是个代码错误。 即使没有使用，但是定义变量仍然需要消耗资源，并且对阅读代码的人也会造成困惑，不知道这些变量是要做什么的。

正确示范

```
def get_x_plus_y(x, y):
   return x + y
```

错误示范

```
some_unused_var = 42

# 定义了变量不意味着就是使用了
y = 10
y = 5
```

```
# 对自身的操作并不意味着使用了
z = 0
z = z + 1

# 未使用的函数参数
def get_x(x, y):
    return x
```

2.9.2 **【推荐】** 使用双下划线 `\_\_` 来代表不需要的变量，单下划线 `\_` 容易与 `gettext()` 函数的别名冲突。

正确示范

```
 path = '/tmp/python/foobar.txt'
dir_name, __ = os.path.split(path)
```

工具与配置

flake8

`flake8` 是一个结合了 pycodestyle，pyflakes，mccabe 检查 Python 代码规范的工具。

使用方法：

```
 flake8 {source_file_or_directory}
```

在项目中创建 `setup.cfg` 或者 `tox.ini` 或者 `.flake8` 文件，添加 `\[flake8\]` 部分。

推荐的配置文件如下：

```
 [flake8]
ignore =
    ;W503 line break before binary operator
    W503,
    ;E203 whitespace before ':'
```

```
    E203,

; exclude file
exclude =
    .tox,
    .git,
    __pycache__,
    build,
    dist,
    *.pyc,
    *.egg-info,
    .cache,
    .eggs

max-line-length = 120
```

如果需要屏蔽告警可以增加行内注释 `# noqa`，例如：

```
 example = lambda: 'example'  # noqa: E731
```

pylint

`pylint` 是一个能够检查编码质量、编码规范的工具。

配置项较多，单独一个配置文件配置，详情可查阅：.pylintrc

使用方法：

```
 pylint {source_file_or_directory}
```

如果遇到一些实际情况与代码冲突的，可以在行内禁用相关检查，例如 ：

```
 try:
    do_something()
except Exception as ex:  # pylint: disable=broad-except
    pass
```

如果需要对多行的进行禁用规则，可以配套使用 `# pylint: disable=具体错误码` / `# pylint: enable=具体错误码` 。

```
 # pylint: disable=invalid-name
这里的代码块会被忽略相关的告警
app = Flask(__name__)
# pylint: enable=invalid-name
```

black

`black` 是一个官方的 Python 代码格式化工具。

使用方法：

```
 black {source_file_or_directory}
```

如果不想格式化部分代码，可以配套使用 `# fmt: off` / `# fmt: on` 临时关闭格式化。

```
 # fmt: off
在这的代码不会被格式化
# fmt: on
```

EditorConfig

EditorConfig 可以帮助开发同一项目下的跨多 IDE 的开发人员保持一致编码风格。

在项目的根目录下放置 `.editorconfig` 文件，可以让编辑器规范文件对格式。参考配置如下：

```
 # https://editorconfig.org

root = true

[*]
indent_style = space
indent_size = 4
trim_trailing_whitespace = true
insert_final_newline = true
charset = utf-8
end_of_line = lf

[*.py]
max_line_length = 120

[*.bat]
indent_style = tab
end_of_line = crlf

[LICENSE]
insert_final_newline = false

[Makefile]
indent_style = tab
```

支持常见的 IDE，配置说明及 IDE 的支持情况可参考官网 editorconfig.org。