Python语言规范

Lint

Tip

对你的代码运行pylint

定义:
pylint是一个在Python源代码中查找bug的工具. 对于C和C++这样的不那么动态的(译者注: 原文是less dynamic)语言, 这些bug通常由编译器来捕获. 由于Python的动态特性, 有些警告可能不对. 不过伪告警应该很少.

优点:
可以捕获容易忽视的错误, 例如输入错误, 使用未赋值的变量等.

缺点:
pylint不完美. 要利用其优势, 我们有时侯需要: a) 围绕着它来写代码 b) 抑制其告警 c) 改进它, 或者d) 忽略它.

结论:
确保对你的代码运行pylint.抑制不准确的警告,以便能够将其他警告暴露出来。

你可以通过设置一个行注释来抑制告警. 例如:

```
dict = 'something awful'  # Bad Idea... pylint: disable=redefined-builtin
```

pylint警告是以一个数字编号(如 `C0112`)和一个符号名(如 `empty-docstring`)来标识的. 在编写新代码或更新已有代码时对告警进行抑制, 推荐使用符号名来标识.

如果警告的符号名不够见名知意，那么请对其增加一个详细解释。

采用这种抑制方式的好处是我们可以轻松查找抑制并回顾它们.

你可以使用命令 `pylint --list-msgs` 来获取pylint告警列表. 你可以使用命令 `pylint --help-msg=C6409` , 以获取关于特定消息的更多信息.

相比较于之前使用的 `pylint: disable-msg` , 本文推荐使用 `pylint: disable` .

要抑制"参数未使用"告警, 你可以用"_"作为参数标识符, 或者在参数名前加"unused_". 遇到不能改变参数名的情况, 你可以通过在函数开头"提到"它们来消除告警. 例如:

```
def foo(a, unused_b, unused_c, d=None, e=None):
    _ = d, e
    return a
```

导入

Tip

仅对包和模块使用导入

定义:
模块间共享代码的重用机制.

优点:
命名空间管理约定十分简单. 每个标识符的源都用一种一致的方式指示. x.Obj表示Obj对象定义在模块x中.

缺点:
模块名仍可能冲突. 有些模块名太长, 不太方便.

结论:
使用 `import x` 来导入包和模块.

使用 `from x import y` , 其中x是包前缀, y是不带前缀的模块名.

使用 `from x import y as z` , 如果两个要导入的模块都叫做y或者y太长了.

例如, 模块 `sound.effects.echo` 可以用如下方式导入:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

导入时不要使用相对名称. 即使模块在同一个包中, 也要使用完整包名. 这能帮助你避免无意间导入一个包两次.

包

Tip

使用模块的全路径名来导入每个模块

优点:
避免模块名冲突. 查找包更容易.

缺点:
部署代码变难, 因为你必须复制包层次.

结论:
所有的新代码都应该用完整包名来导入每个模块.

应该像下面这样导入:

```
# Reference in code with complete name.
import sound.effects.echo

# Reference in code with just module name (preferred).
from sound.effects import echo
```

异常

Tip

允许使用异常, 但必须小心

定义:
异常是一种跳出代码块的正常控制流来处理错误或者其它异常条件的方式.

优点:
正常操作代码的控制流不会和错误处理代码混在一起. 当某种条件发生时, 它也允许控制流跳过多个框架. 例如, 一步跳出N个嵌套的函数, 而不必继续执行错误的代码.

缺点:
可能会导致让人困惑的控制流. 调用库时容易错过错误情况.

结论:
异常必须遵守特定条件:

1. 像这样触发异常: `raise MyException("Error message")` 或者 `raise MyException` . 不要使用两个参数的形式(`raise MyException, "Error message"`)或者过时的字符串异常(`raise "Error message"`).
2. 模块或包应该定义自己的特定域的异常基类, 这个基类应该从内建的Exception类继承. 模块的异常基类应该叫做"Error".

   ```
   class Error(Exception):
       pass
   ```

3. 永远不要使用 `except:` 语句来捕获所有异常, 也不要捕获 `Exception` 或者 `StandardError` , 除非你打算重新触发该异常, 或者你已经在当前线程的最外层(记得还是要打印一条错误消息). 在异常这方面, Python非常宽容, `except:` 真的会捕获包括Python语法错误在内的任何错误. 使用 `except:` 很容易隐藏真正的bug.
4. 尽量减少try/except块中的代码量. try块的体积越大, 期望之外的异常就越容易被触发. 这种情况下, try/except块将隐藏真正的错误.
5. 使用finally子句来执行那些无论try块中有没有异常都应该被执行的代码. 这对于清理资源常常很有用, 例如关闭文件.
6. 当捕获异常时, 使用 `as` 而不要用逗号. 例如

```
try:
    raise Error
except Error as error:
    pass
```

全局变量

Tip

避免全局变量

定义:
定义在模块级的变量.

优点:
偶尔有用.

缺点:
导入时可能改变模块行为, 因为导入模块时会对模块级变量赋值.

结论:
避免使用全局变量, 用类变量来代替. 但也有一些例外:

1. 脚本的默认选项.
2. 模块级常量. 例如: PI = 3.14159. 常量应该全大写, 用下划线连接.
3. 有时候用全局变量来缓存值或者作为函数返回值很有用.
4. 如果需要, 全局变量应该仅在模块内部可用, 并通过模块级的公共函数来访问.

嵌套/局部/内部类或函数

Tip

鼓励使用嵌套/本地/内部类或函数

定义:
类可以定义在方法, 函数或者类中. 函数可以定义在方法或函数中. 封闭区间中定义的变量对嵌套函数是只读的.

优点:
允许定义仅用于有效范围的工具类和函数.

缺点:
嵌套类或局部类的实例不能序列化(pickled).

结论:
推荐使用.

列表推导(List Comprehensions)

Tip

可以在简单情况下使用

定义:
列表推导(list comprehensions)与生成器表达式(generator expression)提供了一种简洁高效的方式来创建列表和迭代器, 而不必借助map(), filter(), 或者lambda.

优点:
简单的列表推导可以比其它的列表创建方法更加清晰简单. 生成器表达式可以十分高效, 因为它们避免了创建整个列表.

缺点:
复杂的列表推导或者生成器表达式可能难以阅读.

结论:
适用于简单情况. 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. 禁止多重for语句或过滤器表达式. 复杂情况下还是使用循环.

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

  for x in xrange(5):
      for y in xrange(5):
          if x != y:
              for z in xrange(5):
                  if y != z:
                      yield (x, y, z)

  return ((x, complicated_transform(x))
          for x in long_generator_function(parameter)
          if x is not None)

  squares = [x * x for x in range(10)]

  eat(jelly_bean for jelly_bean in jelly_beans
      if jelly_bean.color == 'black')
```

```
No:
  result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]

  return ((x, y, z)
          for x in xrange(5)
          for y in xrange(5)
          if x != y
          for z in xrange(5)
          if y != z)
```

默认迭代器和操作符

Tip

如果类型支持, 就使用默认迭代器和操作符. 比如列表, 字典及文件等.

定义:
容器类型, 像字典和列表, 定义了默认的迭代器和关系测试操作符(in和not in)

优点:
默认操作符和迭代器简单高效, 它们直接表达了操作, 没有额外的方法调用. 使用默认操作符的函数是通用的. 它可以用于支持该操作的任何类型.

缺点:
你没法通过阅读方法名来区分对象的类型(例如, has_key()意味着字典). 不过这也是优点.

结论:
如果类型支持, 就使用默认迭代器和操作符, 例如列表, 字典和文件. 内建类型也定义了迭代器方法. 优先考虑这些方法, 而不是那些返回列表的方法. 当然，这样遍历容器时，你将不能修改容器.

```
Yes:  for key in adict: ...
      if key not in adict: ...
      if obj in alist: ...
      for line in afile: ...
      for k, v in dict.iteritems(): ...
```

```
No:   for key in adict.keys(): ...
      if not adict.has_key(key): ...
      for line in afile.readlines(): ...
```

生成器

Tip

按需使用生成器.

定义:
所谓生成器函数, 就是每当它执行一次生成(yield)语句, 它就返回一个迭代器, 这个迭代器生成一个值. 生成值后, 生成器函数的运行状态将被挂起, 直到下一次生成.

优点:
简化代码, 因为每次调用时, 局部变量和控制流的状态都会被保存. 比起一次创建一系列值的函数, 生成器使用的内存更少.

缺点:
没有.

结论:
鼓励使用. 注意在生成器函数的文档字符串中使用”Yields:”而不是”Returns:”.

(译者注: 参看 注释)

Lambda函数

Tip

适用于单行函数

定义:
与语句相反, lambda在一个表达式中定义匿名函数. 常用于为 `map()` 和 `filter()` 之类的高阶函数定义回调函数或者操作符.

优点:
方便.

缺点:
比本地函数更难阅读和调试. 没有函数名意味着堆栈跟踪更难理解. 由于lambda函数通常只包含一个表达式, 因此其表达能力有限.

结论:
适用于单行函数. 如果代码超过60-80个字符, 最好还是定义成常规(嵌套)函数.

对于常见的操作符，例如乘法操作符，使用 `operator` 模块中的函数以代替lambda函数. 例如, 推荐使用 `operator.mul` , 而不是 `lambda x, y: x * y` .

条件表达式

Tip

适用于单行函数

定义:
条件表达式是对于if语句的一种更为简短的句法规则. 例如: `x = 1 if cond else 2` .

优点:
比if语句更加简短和方便.

缺点:
比if语句难于阅读. 如果表达式很长， 难于定位条件.

结论:
适用于单行函数. 在其他情况下，推荐使用完整的if语句.

默认参数值

Tip

适用于大部分情况.

定义:
你可以在函数参数列表的最后指定变量的值, 例如, `def foo(a, b = 0):` . 如果调用foo时只带一个参数, 则b被设为0. 如果带两个参数, 则b的值等于第二个参数.

优点:
你经常会碰到一些使用大量默认值的函数, 但偶尔(比较少见)你想要覆盖这些默认值. 默认参数值提供了一种简单的方法来完成这件事, 你不需要为这些罕见的例外定义大量函数. 同

时, Python也不支持重载方法和函数, 默认参数是一种"仿造"重载行为的简单方式.

缺点:

默认参数只在模块加载时求值一次. 如果参数是列表或字典之类的可变类型, 这可能会导致问题. 如果函数修改了对象(例如向列表追加项), 默认值就被修改了.

结论:

鼓励使用, 不过有如下注意事项:

不要在函数或方法定义中使用可变对象作为默认值.

```
Yes: def foo(a, b=None):
         if b is None:
             b = []
```

```
No:  def foo(a, b=[]):
         ...
No:  def foo(a, b=time.time()):  # The time the module was loaded???
         ...
No:  def foo(a, b=FLAGS.my_thing):  # sys.argv has not yet been parsed...
         ...
```

属性(properties)

Tip

访问和设置数据成员时, 你通常会使用简单, 轻量级的访问和设置函数. 建议用属性（properties）来代替它们.

定义:

一种用于包装方法调用的方式. 当运算量不大, 它是获取和设置属性(attribute)的标准方式.

优点:

通过消除简单的属性(attribute)访问时显式的get和set方法调用, 可读性提高了. 允许懒惰的计算. 用Pythonic的方式来维护类的接口. 就性能而言, 当直接访问变量是合理的, 添加访问方法就显得琐碎而无意义. 使用属性(properties)可以绕过这个问题. 将来也可以在不破坏接口的情况下将访问方法加上.

缺点:

属性(properties)是在get和set方法声明后指定, 这需要使用者在接下来的代码中注意: set和get是用于属性(properties)的(除了用 `@property` 装饰器创建的只读属性). 必须继承自object类. 可能隐藏比如操作符重载之类的副作用. 继承时可能会让人困惑.

结论:

你通常习惯于使用访问或设置方法来访问或设置数据, 它们简单而轻量. 不过我们建议你在新的代码中使用属性. 只读属性应该用 `@property` 装饰器 来创建.

如果子类没有覆盖属性, 那么属性的继承可能看上去不明显. 因此使用者必须确保访问方法间接被调用, 以保证子类中的重载方法被属性调用(使用模板方法设计模式).

```
Yes: import math

     class Square(object):
         """A square with two properties: a writable area and a read-only perimeter.

         To use:
         >>> sq = Square(3)
         >>> sq.area
         9
         >>> sq.perimeter
         12
         >>> sq.area = 16
         >>> sq.side
         4
         >>> sq.perimeter
         16
         """

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

         def __get_area(self):
             """Calculates the 'area' property."""
             return self.side ** 2

         def ___get_area(self):
             """Indirect accessor for 'area' property."""
             return self.__get_area()

         def __set_area(self, area):
             """Sets the 'area' property."""
             self.side = math.sqrt(area)

         def ___set_area(self, area):
             """Indirect setter for 'area' property."""
             self._SetArea(area)

         area = property(___get_area, ___set_area,
                         doc="""Gets or sets the area of the square.""")

         @property
         def perimeter(self):
             return self.side * 4
```

(译者注: 老实说, 我觉得这段示例代码很不恰当, 有必要这么蛋疼吗?)

True/False的求值

Tip

尽可能使用隐式false

定义:
Python在布尔上下文中会将某些值求值为false. 按简单的直觉来讲, 就是所有的"空"值都被认为是false. 因此0， None, [], {}, "" 都被认为是false.

优点:
使用Python布尔值的条件语句更易读也更不易犯错. 大部分情况下, 也更快.

缺点:
对C/C++开发人员来说, 可能看起来有点怪.

结论:

尽可能使用隐式的false, 例如: 使用 `if foo:` 而不是 `if foo != []:` . 不过还是有一些注意事项需要你铭记在心:

1. 永远不要用==或者!=来比较单件, 比如None. 使用is或者is not.
2. 注意: 当你写下 `if x:` 时, 你其实表示的是 `if x is not None` . 例如: 当你要测试一个默认值是None的变量或参数是否被设为其它值. 这个值在布尔语义下可能是false!
3. 永远不要用==将一个布尔量与false相比较. 使用 `if not x:` 代替. 如果你需要区分false和None, 你应该用像 `if not x and x is not None:` 这样的语句.
4. 对于序列(字符串, 列表, 元组), 要注意空序列是false. 因此 `if not seq:` 或者 `if seq:` 比 `if len(seq):` 或 `if not len(seq):` 要更好.
5. 处理整数时, 使用隐式false可能会得不偿失(即不小心将None当做0来处理). 你可以将一个已知是整型(且不是len()的返回结果)的值与0比较.

```
Yes: if not users:
         print 'no users'

     if foo == 0:
         self.handle_zero()

     if i % 10 == 0:
         self.handle_multiple_of_ten()
```

```
No:  if len(users) == 0:
         print 'no users'

     if foo is not None and not foo:
         self.handle_zero()

     if not i % 10:
         self.handle_multiple_of_ten()
```

6. 注意'0'(字符串)会被当做true.

过时的语言特性

Tip

尽可能使用字符串方法取代字符串模块. 使用函数调用语法取代apply(). 使用列表推导, for循环取代filter(), map()以及reduce().

定义:
当前版本的Python提供了大家通常更喜欢的替代品.

结论:
我们不使用不支持这些特性的Python版本, 所以没理由不用新的方式.

```
Yes: words = foo.split(':')

     [x[1] for x in my_list if x[2] == 5]

     map(math.sqrt, data)    # Ok. No inlined lambda expression.

     fn(*args, **kwargs)
```

```
No:  words = string.split(foo, ':')

     map(lambda x: x[1], filter(lambda x: x[2] == 5, my_list))

     apply(fn, args, kwargs)
```

词法作用域(Lexical Scoping)

Tip

推荐使用

定义:

嵌套的Python函数可以引用外层函数中定义的变量, 但是不能够对它们赋值. 变量绑定的解析是使用词法作用域, 也就是基于静态的程序文本. 对一个块中的某个名称的任何赋值都会导致Python将对该名称的全部引用当做局部变量, 甚至是赋值前的处理. 如果碰到global声明, 该名称就会被视作全局变量.

一个使用这个特性的例子:

```
def get_adder(summand1):
    """Returns a function that adds numbers to a given number."""
    def adder(summand2):
        return summand1 + summand2

    return adder
```

(译者注: 这个例子有点诡异, 你应该这样使用这个函数: `sum = get_adder(summand1)(summand2)`)

优点:

通常可以带来更加清晰, 优雅的代码. 尤其会让有经验的Lisp和Scheme(还有Haskell, ML等)程序员感到欣慰.

缺点:

可能导致让人迷惑的bug. 例如下面这个依据 PEP-0227 的例子:

```
i = 4
def foo(x):
    def bar():
        print i,
    # ...
    # A bunch of code here
    # ...
    for i in x:  # Ah, i *is* local to Foo, so this is what Bar sees
        print i,
    bar()
```

因此 `foo([1, 2, 3])` 会打印 `1 2 3 3` , 不是 `1 2 3 4` .

(译者注: x是一个列表, for循环其实是将x中的值依次赋给i.这样对i的赋值就隐式的发生了, 整个foo函数体中的i都会被当做局部变量, 包括bar()中的那个. 这一点与C++之类的静态语言还是有很大差别的.)

结论:

鼓励使用.

函数与方法装饰器

Tip

如果好处很显然, 就明智而谨慎的使用装饰器

定义:

用于函数及方法的装饰器 (也就是@标记). 最常见的装饰器是@classmethod 和 @staticmethod, 用于将常规函数转换成类方法或静态方法. 不过, 装饰器语法也允许用户自定义装饰器. 特别地, 对于某个函数 `my_decorator` , 下面的两段代码是等效的:

```
class C(object):
   @my_decorator
   def method(self):
       # method body ...
```

```
class C(object):
    def method(self):
        # method body ...
    method = my_decorator(method)
```

优点:

优雅的在函数上指定一些转换. 该转换可能减少一些重复代码, 保持已有函数不变(enforce invariants), 等.

缺点:

装饰器可以在函数的参数或返回值上执行任何操作, 这可能导致让人惊异的隐藏行为. 而且, 装饰器在导入时执行. 从装饰器代码的失败中恢复更加不可能.

结论:

如果好处很显然, 就明智而谨慎的使用装饰器. 装饰器应该遵守和函数一样的导入和命名规则. 装饰器的python文档应该清晰的说明该函数是一个装饰器. 请为装饰器编写单元测试.

避免装饰器自身对外界的依赖(即不要依赖于文件, socket, 数据库连接等), 因为装饰器运行时这些资源可能不可用(由 `pydoc` 或其它工具导入). 应该保证一个用有效参数调用的装饰器在所有情况下都是成功的.

装饰器是一种特殊形式的"顶级代码". 参考后面关于 Main 的话题.

线程

Tip

不要依赖内建类型的原子性.

虽然Python的内建类型例如字典看上去拥有原子操作, 但是在某些情形下它们仍然不是原子的(即: 如果__hash__或__eq__被实现为Python方法)且它们的原子性是靠不住的. 你也不能指望原子变量赋值(因为这个反过来依赖字典).

优先使用Queue模块的 `Queue` 数据类型作为线程间的数据通信方式. 另外, 使用threading模块及其锁原语(locking primitives). 了解条件变量的合适使用方式, 这样你就可以使用 `threading.Condition` 来取代低级别的锁了.

威力过大的特性

Tip

避免使用这些特性

定义:
Python是一种异常灵活的语言, 它为你提供了很多花哨的特性, 诸如元类(metaclasses), 字节码访问, 任意编译(on-the-fly compilation), 动态继承, 对象父类重定义(object reparenting), 导入黑客(import hacks), 反射, 系统内修改(modification of system internals), 等等.

优点:
强大的语言特性, 能让你的代码更紧凑.

缺点:
使用这些很"酷"的特性十分诱人, 但不是绝对必要. 使用奇技淫巧的代码将更加难以阅读和调试. 开始可能还好(对原作者而言), 但当你回顾代码, 它们可能会比那些稍长一点但是很直接的代码更加难以理解.

结论:
在你的代码中避免这些特性.