

# [笔记][Effective Python]

Python

[笔记][Effective Python]

0. 前言等

1. 用 Pythonic 方式来思考

第 1 条：确认自己所用的 Python 版本

第 2 条：遵循 PEP8 风格指南

第 3 条：了解 bytes、str 与 unicode 的区别

第 4 条：用辅助函数来取代复杂的表达式

第 5 条：了解切割序列的办法

第 6 条：在单次切片操作内，不要同时指定 start、end 和 stride

第 7 条：用列表推导来取代 map 和 filter

第 8 条：不要使用含有两个以上表达式的列表推导

## 0. 前言等

设定只允许通过关键字形式来指定的参数。

jpch89：这里应该说的是 `keyword-only arguments`，强制关键字参数。

用 `zip` 函数来同时迭代两个列表。

jpch89：这个怎么用不太会。

元类 `metaclass`

动态属性 `dynamic attribute`

最小惊讶原则 `rule of least surprise`

范例代码：<https://github.com/bslatkin/effectivepython>

原书勘误表：<https://github.com/bslatkin/effectivepython/issues>

官网：<http://www.effectivepython.com/>

## 1. 用 Pythonic 方式来思考

`import this`

---

## 第 1 条：确认自己所用的 Python 版本

- `python --version`
- `python3 --version`
- 或者：

```
import sys
print(sys.version_info)
print(sys.version)
```

结果为：

```
sys.version_info(major=3, minor=6, micro=6, releaselevel='final',
serial=0)
3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)]
```

jpch89：还有一种方法是在控制台输入 `python -V`

---

## 第 2 条：遵循 PEP8 风格指南

8 号 Python 增强提案，又叫 PEP 8

Python Enhancement Proposal #8

<http://www.python.org/dev/peps/pep-0008>

**空白** `whitespace` 会影响代码的含义。

- 使用**四个空格** `space` 缩进，不要用制表符 `tab`
- 每行字符数小于等于 **79**
- 对于占据多行的表达式，首行之后的其余各行都要再缩进 **4** 个空格
- 文件中的函数与类之间要用**两个空行**隔开
- 同一个类中，各方法用**一个空行**隔开
- 使用下标来获取列表元素、调用函数或者给关键字参数赋值时，**不要**在两旁添加空格
- 变量赋值的时候等号两边要各有一个空格

**命名**：不同的命名风格体现不同角色。

- 函数、变量、属性用小写字母，以下划线连接
- 受保护的实例属性，以单下划线开头

- 私有的实例属性，以双下划线开头
- 类与异常，大驼峰式命名
- 模块级别的常量，全用大写字母，以下划线连接
- 类中的实例方法 `instance method`，首个参数应该叫做 `self`，以表示该对象本身
- 类方法 `class method`，首个参数应该叫做 `cls`，以表示该类本身

## 表达式和语句

- 不要把否定词放在前面，要写 `if a is not b` 而不是 `if not a is b`
- 不要用长度检测来判定为空 ( `if len(somelist) == 0` )，而是用 `if not somelist` (空值为 `False`)
- 判定非空用 `if somelist`
- 不要写单行的复合的 `if`、`for`、`while`、`except` 语句，要拆开来写
- `import` 总是放在文件开头
- 使用 `from bar import foo` 来导入 `bar` 包中的 `foo` 模块，即要写成绝对名称，而不是相对名称 (不能写 `import foo`)

jpch89：个人理解绝对名称即绝对路径，相对名称即相对路径。这里应该假定主文件和被导入的模块文件位于同一个包中。

- 如果一定要用相对名称，明确写出来 `from . import foo`
- `import` 按顺序划分为**标准库模块**、**第三方模块**以及**自用模块**。在每一部分，各 `import` 语句按字母顺序排列。

`Pylint` <http://www.pylint.org/>  
源码检测工具

## 第 3 条：了解 bytes、str 与 unicode 的区别

`Python 3` 中有两种字符串：

- `str` 类型：它的实例包含 `Unicode` 字符
- `bytes` 类型：它的实例包含原始的字节

`Python 2` 中也有两种字符串：

- `unicode` 类型：它的实例包含 `Unicode` 字符
- `str` 类型：它的实例包含原始的字节

二进制数据 -> `Unicode` 字符：`encode` 方法编码

`Unicode` 字符 -> 二进制数据：`decode` 解码

编码和解码的工作要放在界面最外围，核心部分用 `Unicode` 字符串。  
并且让程序可以接受多种文本编码，而保证输出一种编码形式 (最好是 `UTF-8`)。

要针对 Python 3 和 Python 2 编写两个辅助 helper 函数

Python 3

- 总是返回 str
- 总是返回 bytes

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of str

def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of bytes
```

Python 2

- 总是返回 unicode
- 总是返回 str

```
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value # Instance of unicode

def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value # Instance of str
```

ASCII 码相关知识补充

ASCII第一次以规范标准的形态发表是在1967年，最后一次更新则是在1986年，至今为止共定义了128个字符，其中33个字符无法显示（这是以现今操作系统为依归，但在DOS模式下可显示出一些诸如笑脸、扑克牌花式等8-bit符号），且这33个字符多数都已是陈废的

控制字符，控制字符的用途主要是用来操控已经处理过的文字，在33个字符之外的是95个可显示的字符，包含用键盘敲下空白键所产生的空白字符也算1个可显示字符（显示为空白）。

注意点：

- 在 Python 2 中，如果 str 只包含 7 位 ASCII 字符，那么 unicode 和 str 就成了同一种类型。但是在 Python 3 中这两者绝对不等价，空字符串也不行。
  - 可以用 + 连接 str 与 unicode
  - 可用等价于不等价操作符进行比较
  - 格式字符串中可以用 '%s' 等形式代表 unicode 实例
  - 可以用 str 与 unicode 给函数传参
- 在 Python 3 中 open 函数默认使用 UTF-8 编码格式系统本地的编码格式来操作文件，不能直接往里面写二进制数据 f.write(os.urandom(10))，会报错。在 Python 2 中文件操作的默认编码格式是二进制形式。同时适配 Python 2 和 Python 3 的写法是：

```
with open('/tmp/random.bin', 'wb') as f:  
    f.write(os.urandom(10))
```

读二进制文件也是一样，要用 rb，可以适配 Python 3 和 Python 2。

jpch89：这句话表述错误，在 Python 3 中，open 函数以文本模式操作文件时，如果不指定编码，该编码是依赖于平台的。

以下内容来自官网：<https://docs.python.org/3/library/functions.html#open>

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever locale.getpreferredencoding() returns), but any text encoding supported by Python can be used. See the codecs module for the list of supported encodings.

os.urandom(size) 随机生成 size 个字节的二进制数据。

文件句柄 file handle 其实是一种标识符或指针。

## 第 4 条：用辅助函数来取代复杂的表达式

jpch89：本条体现了两点 Python 之禅。

- Beautiful is better than ugly.
- Simple is better than complex.

```

from urllib.parse import parse_qs

# 解析查询字符串 query string
my_values = parse_qs('red=5&blue=0&green=',
                    keep_blank_values=True)
# print(repr(my_values)) # 原书写法
print(my_values) # 返回的是字典，直接这样写就行了
# >>>
# {'red': ['5'], 'blue': ['0'], 'green': ['']}

# 查询字符串中的参数可能有：多个值和空白 blank 值。
# 有些参数则没有出现。
# 使用 get 方法可以不报错的从字典中取值。
print('Red: ', my_values.get('red'))
print('Green: ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))

print('-' * 50)
# 需求：当查询的参数没有出现在查询字符串中
# 或者参数的值为空白的时候
# 可以返回 0
# 思路：空值和零值都是 False
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print('Red: %r' % red)
print('Green: %r' % green)
print('Opacity: %r' % opacity)

print('-' * 50)
# 需求：最后要用到的是整数类型
# 思路：类型转换
red = int(my_values.get('red', [''])[0] or 0)
# 这种长表达式的写法看上去很乱！

# 改进1：使用 Python 2.5 添加的三元表达式
red = my_values.get('red', [''])
red = int(red[0]) if red[0] else 0

# 改进2：使用跨行的 if/else 语句
green = my_values.get('green', [''])
if green[0]:
    green = int(green[0])
else:
    green = 0

# 改进3：频繁使用的逻辑，需要封装成辅助函数
def get_first_value(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:

```

```
        found = int(found[0])
    else:
        found = default
    return found
```

## 第 5 条：了解切割序列的办法

切片 `slice` 操作，可以对内置的 `list`、`str`、`bytes` 进行切割。

jpch89：还有元组 `tuple` 也支持切片操作。

对于实现了 `__getitem__` 和 `__setitem__` 这两个特殊方法的 `Python` 类，也支持切片操作。

### 基本写法

```
somelist[start:end]
```

- `start` 起始索引（包括）
- `end` 结束索引（不包括）

如果从列表开始切片，`start` 要留空，不用写 `0`。

```
assert a[:5] == a[0:5]
```

如果一直要取到末尾，`end` 要留空，写了多余。

```
assert a[5:] == a[5:len(a)]
```

负索引表示从末尾往前数。

切片时偏移量可以越界，但是索引取值时，偏移量不能越界，会抛出 `IndexError` 异常。

使用负变量作为 `start` 切片，比如 `somelist[-n:]`，当 `n >= 1` 的时候结果是取末尾 `n` 个元素，但是当 `n = 0` 的时候，则会得到原列表的拷贝。

切片返回的是一个新的列表，**对新列表进行修改，不会影响原列表。**

```
a = [1, 2, 3, [4, 5, 6], 7, 8]
b = a[2:]
```

```
print('更改前')
print('a为: ', a)
print('b为: ', b)
```

```
b[1] = [111, 222]
print('-' * 50)
print('更改后: ')
print('a为: ', a)
```

```
print('b为: ', b)

"""
更改前
a为:  [1, 2, 3, [4, 5, 6], 7, 8]
b为:  [3, [4, 5, 6], 7, 8]
-----
更改后:
a为:  [1, 2, 3, [4, 5, 6], 7, 8]
b为:  [3, [111, 222], 7, 8]
"""
```

赋值语句左侧切片出来的内容会自动替换成右侧的内容。  
左右两侧内容的长度不需要相等。

```
a = [1, 2, 3, [4, 5, 6], 7, 8]
print('更改前')
print('a为: ', a)
print('-' * 50)
a[:3] = [0]
print('更改后')
print('a为: ', a)

"""
更改前
a为:  [1, 2, 3, [4, 5, 6], 7, 8]
-----
更改后
a为:  [0, [4, 5, 6], 7, 8]
"""
```

`b = a[:]` 得到的是对原列表的拷贝

```
a = [1, 2, 3, [4, 5, 6], 7, 8]

# b = a[:] 得到的是对原列表的拷贝
b = a[:]
```

*# 不会抛出 AssertionError*  
*# 因为 a 和 b 值相等*  
*# 但是位于内存的不同位置*

```
assert b == a and b is not a
```

`b = a` 不会在内存中新建对象。`b` 与 `a` 的内存地址是一样的。  
左侧使用留空形式的切片操作，会使用右侧的值替换原列表的全部内容。

```
a[:] = ['new', 'content']
```



```

a = [1, 2, 3, [4, 5, 6], 7, 8]

# b = a 不会在内存中新建一个对象
b = a
# 所以 a 就是 b
assert a is b
print('更改前')
print('a为: ', a)
# 左侧使用留空形式的切片操作，会使用右侧的值替换原列表的全部内容。
a[:] = ['a', 'b', 'c']
print('-' * 50)
print('更改后')
print('a为: ', a)
"""
更改前
a为:  [0, [4, 5, 6], 7, 8]
-----
更改后
a为:  ['a', 'b', 'c']
"""

```

## 第 6 条：在单次切片操作内，不要同时指定 start、end 和 stride

```
somelist[start:end:stride]
```

`stride` 为步进值，假设为 `n`，就是每 `n` 个元素里面取 `1` 个。

- 提取索引值为偶数的元素

```
evens = a[1::2]
```

- 提取索引值为奇数的元素

```
odds = a[::2]
```

使用 `-1` 作为步进值，其他留空，可以对序列进行逆序操作。  
但是对于以 `utf-8` 编码的字节串来说，逆序之后再转码会报错。

```

w = '谢谢你'
x = w.encode('utf-8')
print(x)
y = x[::-1]
# 报错
# UnicodeDecodeError
z = y.decode('utf-8')

```

负步进值从末尾向前选取。

为了容易理解和美观：

- 不要一起写 `start`、`end` 和 `stride`
- 就算非要用 `stride`，尽量用正值，同时省略 `start` 和 `end`
- 如果三个都要用，可以**先步进切片，再范围切片**。或者**先范围切片，再步进切片**。  
两段式切片操作会多产生一份数据的浅拷贝，对执行时间和内存用量有一定影响。
- 可以考虑 `itertools` 模块的 `islice` 方法。该方法不允许对 `start`、`end` 和 `stride` 指定负值。

---

## 第 7 条：用列表推导来取代 `map` 和 `filter`

`list comprehension` 列表推导：用一份列表来制作另一份。

```
a = [1, 2, 3]
squares = [x ** 2 for x in a]
```

对于简单情况，列表推导要比内置的 `map` 函数更清晰，不用创建 `lambda` 函数：

```
a = [1, 2, 3]
squares = map(lambda x: x ** 2, a)
```

列表推导可以过滤原列表中的元素

```
a = [1, 2, 3]
even_squares = [x ** 2 for x in a if x % 2 == 0]
```

`filter` 结合 `map` 也可以有同样效果，不过更复杂。

```
a = [1, 2, 3]
alt = map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

也有字典 `dict` 推导和集合 `set` 推导。

---

## 第 8 条：不要使用含有两个以上表达式的列表推导