优化措施

引子(目标)

Python之禅

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.**
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

中文翻译:

```
Python 之禅，蒂姆·彼得斯 (Tim Peters)
美丽总比丑陋好。
显式优于隐式。
简单胜于复杂。
复杂总比复杂好。
扁平比嵌套好。
稀疏比密集好。
可读性很重要。
特殊情况不足以违反规则。
虽然实用胜过纯粹。
错误不应该悄无声息地过去。
除非明确沉默。
面对模棱两可的情况，拒绝猜测的诱惑。
应该有一种——最好只有一种——显而易见的方法来做到这一点。
尽管除非您是荷兰人，否则这种方式一开始可能并不明显。
现在总比没有好。
尽管从来没有比现在更好。
如果实现很难解释，那就不是一个好主意。
如果实现很容易解释，那可能是个好主意。
命名空间是一个非常棒的想法——让我们做更多这样的事情吧！
```

较为口语化的翻译:

```
优美 > 丑陋
明确 > 隐晦 （1）
简单 > 复杂
复杂 > 繁复 （2）
扁平 > 嵌套
稀疏 > 拥挤（3）
可读性很重要（4）
固然实用性比洁癖更重要，
人们所谓的“特例”也往往没有真特殊到需要无视上述规则的程度
除非必要，否则不要无故忽视异常（5）
如果遇到模棱两可的逻辑，请不要自作聪明地瞎猜。
应该提供一种，且最好只提供一种，一目了然的途径
当然这是没法一蹴而就的，除非你是荷兰人（6）
诚然，做 总好过 不做。
但，不假思索地闷头蛮干 还不如 压根别做（动手之前要细思量）
倘若你的实现很难解释，它一定不是个好主意
倘若你的实现一目了然，它可能是个好主意（7）
命名空间大法好，同志们要多多搞！
```

标注表:

1. 该引入的包显式地一条条罗列出来，不要合并；不要用星号；不要在方法里藏意想

不到的的副作用，等等等等。还一个例子，有一种著名的软件设计原则 Convention over Configuration（约定优于配置）如果做得不谨慎，比如你约定的规则并不是真的业界惯例，就会违背这条。

2. StackOverflow上针对这句话的提问: 必要的复杂总是难免的，繁复啰嗦的代码却是不可接受的。你可以做很多事，很复杂的事，但是不能啰嗦，更不能难以理解。复杂不是罪，但是代码需要更有逻辑、更有机的组织。简而言之，Simple > Complex > Complicated > Chaotic。（另外，以上内容仅限Python语境，不同语境下对Complex和Complicated的定义可能会有所不同）
3. 有人喜欢写很长的one-liner 比如：`lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qso`
一行流快速排序 这样固然可以炫技，但是也很难懂啊。让其他人读不懂的代码不是优雅的代码
4. 写这篇文章的动机之一就是看到有人把 Readability counts 翻译成可读性计数
5. 实操中很多人不注意 catch 完就 log 一下就不管了，很快啊，这样好么？这样不好。软件界一般都讲 Let it fail，学名为 Fail-fast 法则。简而言之就是整个项目周期中越早暴露的问题，其修复成本越低。等到你的项目上线了结果出来各种诡异的bug你会毫无头绪，结果只能去翻长长的日志。所以我劝各位，不要再犯这样的聪明，小聪明。
6. 本文作者 Tim Peters 解释说这里的荷兰人指的是 Python 的作者 Guido van Rossum 吹捧 gvanrossum 的彩虹屁：等同于“你个荷兰佬他娘的还真是个天才”
7. 贯穿整个 PEP 20 的核心就是一句话“你的代码是给别人读的！”。从这个角度而言，难以理解、难以维护的代码，即便是“高性能”，也肯定不是好代码；但是反过来，一目了然的逻辑也不代表就一定是好代码。编程可太难了

Pythonic

Pythonic 就是很 Python 的 Python 代码。比如「这很知乎」一般用来表示「知乎社区特有的现象」，「这很百度」表示「百度公司特有的行为」，「很 Python 」的代码就是 「 Pythonic 」。也就是说，Python 很明显区别于其它语言的（~~优雅~~）写法。

比如，在 C-Like 语言中一般这样交换两个数字 a，b：

```
int t = a;
a = b;
b = t;
```

而在 Python 中，一般这样写：

```
a, b = b, a
```

类似地，Python 一般这样判断数字在某个区间：

```
0 < a < 10
```

当处理文件时，一般不必 try-except 来安全地善后，而是通过 context manager：

```
with open(path, mode) as fp:
    ...  # do something
```

再比如 `for-else`、`try-else` 等等等等很多，诸如此类 Python 特有的，称之为特性也好、语法糖也罢的东西，就是 Pythonic。

其实，只要认真阅读一遍 PEP 8，并尽量遵守，代码就足够 Pythonic 了。

Pythonic 小例子

变量交换

```
# Bad
tmp = a
a = b
b = tmp

# Pythonic
a,b = b,a
```

列表推导

```
# Bad
my_list = []
for i in range(10):
    my_list.append(i*2)

# Pythonic
my_list = [i*2 for i in range(10)]
```

单行表达式

虽然列表推导式由于其简洁性及表达性，被广受推崇。

但是有许多可以写成单行的表达式，并不是好的做法。

```
# Bad
print 'one'; print 'two'

if x == 1: print 'one'

if <complex comparison> and <other complex comparison>:
    # do something

# Pythonic

print 'one'
print 'two'

if x == 1:
    print 'one'

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

带索引遍历

```
# Bad
for i in range(len(my_list)):
    print(i, "-->", my_list[i])

# Pythonic
for i,item in enumerate(my_list):
    print(i, "-->",item)
```

字符串拼接

```
# Bad
letters = ['s', 'p', 'a', 'm']
s=""
for let in letters:
    s += let

# Pythonic
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
```

真假判断

```
# Bad
if attr == True:
    print 'True!'

if attr == None:
    print 'attr is None!'

# Pythonic
if attr:
    print 'attr is truthy!'

if not attr:
    print 'attr is falsey!'

if attr is None:
    print 'attr is None!'
```

真假值对照表：

类型	False	True
布尔	False(与0等价)	True(与1等价)
字符串	""(空字符串)	非空字符串，例如 " ", "blog"
数值	0, 0.0	非0的数值，例如：1, 0.1, -1, 2
容器	[], (),	至少有一个元素的容器对象，例如：[0],(None,),['']
None	None	非None对象

访问字典元素

```
# Bad

d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']    # prints 'world'
else:
    print 'default_value'

# Pythonic

d = {'hello': 'world'}

print d.get('hello', 'default_value') # prints 'world'
print d.get('thingy', 'default_value') # prints 'default_value'

# Or:
if 'hello' in d:
    print d['hello']
```

操作列表

```
# Bad

a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)

# Pythonic

a = [3, 4, 5]
b = [i for i in a if i > 4]
# Or:
b = filter(lambda x: x > 4, a)
Bad

a = [3, 4, 5]
for i in range(len(a)):
    a[i] += 3

# Pythonic

a = [3, 4, 5]
a = [i + 3 for i in a]
# Or:
a = map(lambda i: i + 3, a)
```

文件读取

```
# Bad

f = open('file.txt')
a = f.read()
print a
f.close()

# Pythonic

with open('file.txt') as f:
    for line in f:
        print line
```

代码续行

```
# Bad
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say “I’m going to sleep.”"""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function

# Pythonic
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say “I’m going to sleep.”"
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

显式代码

```
# Bad
def make_complex(*args):
    x, y = args
    return dict(**locals())

# Pythonic
def make_complex(x, y):
    return {'x': x, 'y': y}
```

使用占位符

```
# Pythonic
filename = 'foobar.txt'
basename, _, ext = filename.rpartition('.')
```

链式比较

```
# Bad
if age > 18 and age < 60:
    print("young man")

# Pythonic

if 18 < age < 60:
    print("young man")

# 理解了链式比较操作，那么你应该知道为什么下面这行代码输出的结果是 False
>>> False == False == True
False
```

三目运算

这个保留意见。随使用习惯就好。

```
# Bad
if a > 2:
    b = 2
else:
    b = 1
#b = 2

# Pythonic

a = 3

b = 2 if a > 2 else 1
#b = 2
```

or赋值

某些情况下，`or`可以替换`if else`达到代码简化的作用，比如在变量赋值时，并且支持链式调用

```python
# 基本用法
v = p1 or p2
# v = p1 if p1 else p2

# 链式用法, 直到找到为真的值来赋值, 否则赋值最后的值
v = p1 or p2 or p3 or default

# if p1:
#     v = p1
# elif p2:
#     v = p2
# elif p3:
#     v = p3
# else:
#     v = defualt
```

Python3 新特性

*和**解构

```
# Pythonic

a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]

a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4

# 解构字典和列表
print(*[1], *[2], 3, *[4, 5])

def fn(a, b, c, d):
    print(a, b, c, d)

fn(**{'a': 1, 'c': 3}, **{'b': 2, 'd': 4})

# 也可以解构元祖，集合set等
*range(4), 4

[*range(4), 4]

{*range(4), 4, *(5, 6, 7)}

{'x': 1, **{'y': 2}}
```

字符串格式化

```
# 早期 - python 2

print("name:%s, age:%s, sex: %s," %('张三', 18, '男'))

# 后来 - python 2

print("name:{}, age:{}, sex:{}".format('张三', 18, '男'))
print("name:{name}, age:{age}, sex:{sex}".format(name='张三', age=18, sex='男'))

# 现在 - python3
name, age, sex = ('张三', 18, '男') # 自动解包
print(f"name:{name}, age:{age}, sex:{sex}")
```

f字符串支持用=

自动记录表达式和调试文档

增加 = 说明符用于 f-string。形式为 `f'{expr=}'` 的 f-字符串将扩展表示为表达式文本，加一个等于号，再加表达式的求值结果。

```
# 基本用法
>>> import datetime
>>> user = 'eric_idle'
>>> member_since = datetime.date(1975, 7, 31)
>>> f'{user=} {member_since=}'
"user='eric_idle' member_since=datetime.date(1975, 7, 31)"

# 通常的 f-字符串格式说明符 允许更细致地控制所要显示的表达式结果：
>>> delta = datetime.date.today() - member_since
>>> f'{user=!s}  {delta.days=:,d}'
'user=eric_idle  delta.days=17,436'

# = 说明符将输出整个表达式，以便详细演示计算过程：
>>> from math import cos, radians
>>> theta = 30
>>> print(f'{theta=}  {cos(radians(theta))=:.3f}')
theta=30  cos(radians(theta))=0.866
```

字典新增合并运算

python 3.9新增特性

合并 (`|`) 与更新 (`|=`) 运算符已被加入内置的 `dict` 类。它们为现有的 `dict.update` 和 `{**d1, **d2}` 字典合并方法提供了补充。

```
>>> x = {"name": "张三", "sex": "男"}
>>> y = {"name": "张小三", "sex": "男"}
>>> print(x | y)  # 取并集并覆盖
{'name': '张小三', 'sex': '男'}
>>> print(y | x)  # 取并集并覆盖
{'name': '张三', 'sex': '男'}
```

数字加下划线

数字中可以使用下划线增强可读性

```python
print(1_000_000_000_000_00)
print(0x_FF_FF_FF_FF)

# 字符串格式化也支持下划线_选项
```

类型标注

参考官网: https://docs.python.org/zh-cn/3/library/typing.html

变量标注

```python
# 需要一部分IDE支持
primes: list[int] = []

captain: str  # Note: no initial value!

class Starship:
    stats: dict[str, int] = {}

primes.append('sda')  # 这里会类型检查不通过
print(primes)
```

类型别名

把类型赋给别名，就可以定义类型别名。

```python
# Vector 和 list[float] 相同，可互换
Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# 通过类型检查; 一个浮点数的列表组合 类型为 Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名主要适用于简化复杂的类型签名

```python
from collections.abc import Sequence  # 序列泛型

ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# 静态类型检查器会将前一个类型签名视为与当前类型签名完全相同。
def broadcast_message(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str, str]]]) -> None:
    ...
```

标准库

csv

参考标准库: https://docs.python.org/zh-cn/3/library/csv.html

`CSV` (Comma Separated Values) 格式是电子表格和数据库中最常见的输入、输出文件格式。

`csv` 模块实现了 `CSV` 格式表单数据的读写

```
# 基本读写
import csv

# csv.reader 读取
with open('eggs.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
    for row in spamreader:
        print(', '.join(row))

# csv.writer 写入
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

# 字典读写
import csv

# csv.DictReader 读取字典格式
with open('names.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row['first_name'], row['last_name'])

# csv.DictWriter 写入字典格式
with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

io

参考官网: https://docs.python.org/zh-cn/3/library/io.html

io 模块提供了 Python 用于处理各种 I/O 类型的主要工具。三种主要的 I/O类型分别为: 文本 I/O, 二进制 I/O 和 原始 I/O。

文本IO

内存中文本流作为 StringIO 对象使用：

```
# 文本I/O预期并生成 str 对象。
f = io.StringIO("some initial text data")

# 主要用于生成小的文本文件（例如：csv），便于处理，而非创建一个临时文件（磁盘IO）
```

二进制 I/O

二进制I/O（也称为**缓冲I/O**）预期 bytes-like objects 并生成 bytes 对象。

```
# 存储二进制数据
f = io.BytesIO(b"some initial binary data: \x00\x01")

# 多常用于图片、验证码等。
```

collections 常用容器

ChainMap

一个 ChainMap 将多个字典或者其他映射组合在一起，创建一个单独的可更新的视图。如果没有 maps 被指定，就提供一个默认的空字典，这样一个新链至少有一个映射。

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']

# python3 新语法 针对字典 的 | 运算符也可以
```

Counter

一个计数器工具提供快速和方便的计数

```
>>> from collections import Counter
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

defaultdict

返回一个新的类似字典的对象。`defaultdict` 是内置 `dict` 类的子类。它重载了一个方法并添加了一个可写的实例变量。其余的功能与 `dict` 类相同因而不在此文档中写明。

```
>>> from collections import defaultdict
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

namedtuple

命名元组赋予每个位置一个含义，提供可读性和自文档性。它们可以用于任何普通元组，并添加了通过名字获取值的能力，通过索引值也是可以的。

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])  # 使用位置或关键字参数实例化
>>> p = Point(11, y=22)                      # 像普通元组 (11, 22) 一样可索引
>>> p[0] + p[1]
33
>>> x, y = p                                 # 像普通元组一样解构
>>> x, y
(11, 22)
>>> p.x + p.y                                # 也可按名称访问字段
33
>>> p
Point(x=11, y=22)                            # 以可读的name=value形式作为__repr__的输出
```

内置函数

map

filter

any

all

zip

在多个迭代器上并行迭代，从每个迭代器返回一个数据项组成元组

```
# 有时候经常遇到sql执行之后返回这样的值
>>> keys = ('a', 'b', 'c')
>>> values = [[1,2,3], [4,5,6], [7,8,9]]
>>> list(map(dict, map(lambda items: zip(keys, items), values)))
[{'a': 1, 'b': 2, 'c': 3}, {'a': 4, 'b': 5, 'c': 6}, {'a': 7, 'b': 8, 'c': 9}]
```

sorted

sum

range

getattr

setattr

项目实战

1. 导包顺序
2. 变量赋值以及获取
3. 代码空格

4. 重复代码提取
5. 常量提取
6. 标准库函数和工具

其他

导出带目录的PDF，的扩展：Markdown Preview Enhanced