通用 Python 代码设计技巧

1 目录

2 序

2.1 目标读者

本书是 Python 介于基础和进阶之间的书，主要包含通用的 Python 豆知识(黑科技)，代码设计技巧，编程经验，Python 编辑器特性等。

本书适合看完了 Python 基础教程(列表，字典，文件操作，异常，类与面向对象编程等)而将要投入实践的程序员(尤其是需要自己设计代码的人)。以了解基础以外还有什么东西可学，从而强化代码设计技巧。

2.2 宗旨

作者整理了一些基础教程里可能没有的通用知识点。有些技巧可能比较简单，但是不容易查到的，本书也会介绍。

作者希望能抛砖引玉，建立一个尽量完整的“检索表”，使读者了解一些有效的或者有趣的通用工具和 Python 特性。出于效率，易读性和抛砖引玉的考虑，作者不会对具体知识作过于深入的分析或详尽的描绘。作者将特别注意易读性和易理解性，作者会大量使用“简言之”一词。在一些场合中，这也许会少量地丢失严谨性或丰富性。如果读者感兴趣或者实际遇到了相应问题请查阅更详尽的资料。

本书的内容定位是 Python 通用知识，除了少量特别好用的或者很通用的第三方模块之外，作者不会专门去介绍一些第三方模块的具体用法。值得注意的是，第三方模块的应用性较强，很多实际场合中的工作也许最需要的恰恰是对某几个第三方模块的熟悉和使用(例如 tensorflow, sklearn, Flask, Django, Pyside2 等不胜枚举)。第三方模块、第三方框架具极重要的地位，如若涉及，请读者查阅相关官方文档或者其他工具书。

(作者不写这类资料是因为这类资料已经有很多，并且第三方模块的世界是极其庞大的。本书的定位在于”基本知识”和内置模块的理解和运用，但作者推荐”基本知识”应该和第三方模块交叉性地进行学习。)

2.3 结构和排版

本书结构原则：越基本的，越重要的东西越往前排，越复杂的，越不容易用到的往后排。重要性差不多的那些，再分门别类放在一起。每一节的内容不会太多，致力于使读者可以在较少的时间内获取最多的有用信息。

一些排版约定：

- (括号内的内容)表示补充性的文字。
- ***既加粗又斜体的内容***表示这是一个专有名词。
- 带有下划线的内容表示强调。
- **加粗的内容**表示强调。
- 紫色的内容表示代码。
- 蓝色的内容表示程序的输出结果。
- 绿色的内容表示代码中的注释。
- 红色的内容表示本书中的注释。

(作者于 2021 年 3 月完成此手册，使用的 Python 版本为 3.8，集成开发环境为 spyder4.0)

因各种原因内容可能有误，还请读完讨论、纠正一二。

作者的联系方式可在本书最后一页找到。

3 基础

3.1 习惯与通识

3.1.1 PEP8 设计风格

PEP8 是一种推荐的 Python 代码设计风格，整份文档详见官方指南(legacy.Python.org/dev/PEPs/PEP-0008/)。

下面列举个别条目:

- 使用空格表示缩进，而不要使用制表符；(制表符可能会在代码传播过程中变形)
- 写 my_list = []而不要写 my_list=[]或者 my_list = []；
- 如果 my_list 是一个列表，写 if my_list: 而不要写 if len(my_list) > 0: 来判断 my_list 不是空列表。除了集合外的其他容器也是类似的。且我们可以同时在多种意义上检验 my_object 是否不为空(空列表，空元组，空字符串，None 等等都是某种意义上的空，但注意空集不是“空”的！)；
- 文件中的 import 应划为三部分，分别表示标准库模块，第三方模块以及自用模块。每一部分中应以模块字母顺序排列。

……

注意 PEP8 提供了一种代码设计风格的参考，不要完全拘泥于 PEP8 等风格模式，总会有风格模式反而不适用你的程序，或者模棱两可的情况，此时可以灵活应对。

许多编辑器带有 auto-PEP8 功能，例如 pycharm，经过插件增强的 Jupyter 等，注意自动的 PEP8 只能解决部分风格问题(例如它不会将 if len(mylist) > 0: 改为 if mylist:)，因此我仍然推荐读者详细了解 PEP8。

除了 PEP8，我们可能还需要遵循一些额外的风格。例如 import pandas as pd 其中的 pd 一般是约定的。

PEP 的全称是 Python Enhancement Proposals，一个常见的翻译是 Python 增强方案。PEP 提案总共有几百篇，阅读这些文章可以深化 Python 的理解，其中不乏晦涩难读的文章。

3.1.2 注释

长期工程要多写注释，短期工程或临时工程不要写太多的注释。

如果你认为目前的脚本文件以后可能会回顾复用，一定要多写注释。

3.1.3 多种注释或提示符

Python 中多种可用的注释，以下罗列 Python 中的各种注释及其使用方法：

- 单行注释：

#comment。

- 多行注释：

"""comment"""或者'''comment'''，comment 可以跨行。

- 特殊注释#TODO comment 和#FIXME comment：

这两种注释符具有特定含义，简言之，#TODO 表示待做工作，# FIXME 表示可能有问题的内容，在一些编辑器它们所定义的注释具有特殊的颜色或高亮，从而有更特殊的提醒效果。

另外当代码较长时，可以在代码中穿插#TODO 注释和#FIXME 注释，那么后续你能利用大多数编辑器具有的 ctrl+F 查询功能直接定位#TODO 和#FIXME 以快速查看你要做的事。

- [spyder 功能]分块注释：

#%%。在 spyder 编辑器中，可实现代码分块，从而可以容易地进行例如运行单元格等某些操作。

- 基于 if 的多行注释：

有时，我们是要用注释停用或启用一段代码而不是真正的"注释"。

鉴于 python 的 if 控制流不需要一组大括号对应，就有了一种做法。可以在一个代码块的前一行添加一个缩进量适合的 if 0:，然后我们可以随时把 0 切换为 1，方便地启用或者停用这个代码块，有时这能起到独特的效果。

例如你需要嵌套式多行注释来停用代码时，有些编辑器的嵌套式多行注释操作可能比较繁琐，而此法更方便。

基于此思路，还可以实现基于一个列表控制多个代码块的组合停用(组合启用)。

```
test_flags = [1,1,0]
if test_flags[0]:
    #run some codes
    …
if test_flags[1]:
    #run some codes
    …
if test_flags[2]:
    #run some codes
    …
```

(设计代码时 flag 一词通常表示一种标志，用于后续判断。)

- 函数中的类型提示(type hints)(又名函数标注 function annotation)：

例如：

```
def mf(num: int, method: int) -> str:
```

用于说明一个函数各参数所要求的类型，以及返回值的类型。它只起提示作用。

事实上 Python 中有很丰富的类型提示功能，详见 typing 模块官方文档 https://docs.python.org/3/library/typing.html#generics。

- 省略号或者 ellipsis 类：

Python 可以用 ellipsis 类的 Ellipsis 对象来做 type hints，它在代码中的形式是一个省略号(Python3)。(很少人用这个；你可以选择用…来代替 pass，效率更低但也许更好看)

还有另一种情形也会用到，即 a[1:5, …]，它表示多维切片，意为第一个维度选取 2-6 个元素，从第二个维度开始都选取所有元素。

还有另一种情形也会用到。我们首先通过以下代码生成无数层嵌套的字典或者列表：

```
#生成一个无穷维的字典
a, b = a[b] = {}, 5
```

```
#生成一个无穷维的列表
foo = [1,2]
foo[1:1] = [foo]
```

打印该容器，其输出形式带有省略号，要对此类容器进行切片，则一定要使用省略号的多维切片形式。

- [pycharm 功能]基于冒号的一般类型提示(又名变量标注 variable annotation)：

pycharm 具有强大的联想功能(亦称自动补全功能)，但在某些场合中(例如变量经过了赋值)会无法判定一个变量的类别，那么编辑器就不再能对这个变量作自动补全。

我们可以手工指定告诉编辑器变量的类别，例如我们可以在代码中加入这样一条 Banana : Fruit 类别注释表明 Banana 是 Fruit 类的对象，这样后续我们可以基于 Fruit 中的属性和方法对 Banana 作自动补全。

- 续行符：

使用例子：

```
Your_Name = \
input()
```

续行符的用法很简单，即用于分割一行较长的代码，但不可以用续航符把一个词拆开。在 Python 中，如果换行发生在一组括号之间，那么可以省略换行符。

3.1.4 分割长文本

如果你的代码很长。

可以在合适的位置增加代码

```
…
#==============================
…
```

以分块代码，使你的代码结构更清晰。

如果你的打印结果很长，

可以在合适的位置增加代码。

```
    ...
print("=" * 30)
    ...
```

使你的打印结果更清晰。

当然不一定非得用"="号，你可以用你喜欢的符号或字母。甚至你进行代码分块时，可以首尾分别用不同的代码线，可能会更清晰。在任何情况下，我们应该有意识地让代码本身和代码输出结果的排版更清晰。

3.1.5 导入单个方法或者对象

如果你确定你的程序中只会用到某个模块的单个方法或者对象，那么应该仅仅导入这个特定的方法或者对象而不是导入整个模块。这个习惯可以增加程序的效率以及减少代码量(尤其是需要多次使用该方法时)。

例如应该写：

```
from math import sin
from math import pi
print(sin(pi / 2))
```

而不要写:

```
import math
print(math.sin(math.pi / 2))
```

如果你会多次用到某个模块的方法和对象，可以用 import math 导入模块或者 from math import * 导入模块的所有对象(实际上并不是真的所有对象而是那个模块的__all__变量定义的对象)。

3.1.6 管理全局变量

用列表来管理一连串相似的全局变量，以增加可维护性以及减少全局变量个数。

(可维护性简言之就是以后回过头来改的时候比较方便，这个改不一定是修复错误。)

推荐写法示例：

```
suitspic = []
suitspic.append(pygame.image.load('s1.png').convert_alpha())
```

```
suitspic.append(pygame.image.load('s2.png').convert_alpha())
suitspic.append(pygame.image.load('s3.png').convert_alpha())
suitspic.append(pygame.image.load('s4.png').convert_alpha())
```

不推荐写法示例：

```
suitspic1 = (pygame.image.load('s1.png').convert_alpha())
suitspic2 = (pygame.image.load('s2.png').convert_alpha())
suitspic3 = (pygame.image.load('s3.png').convert_alpha())
suitspic4 = (pygame.image.load('s4.png').convert_alpha())
```

如果你的全局变量特别多(例如上百个)，你应该考虑借助*面向对象编程*的设计方式——简言之，将变量结构化，找出一层层框架，将框架作为类，变量作为对象或者对象的属性值。

3.1.7 共享礼仪与版本控制

如果你的程序不是私用的，则应有一个习惯：在抬头或者程序外部说明你使用的 Python 的版本号。必要情况下应说明一些复杂模块的版本号和你的编程环境。

如果你的程序不是临时的，对程序文件的命名应有一个习惯：如从 file0.00 开始命名，每次修改时备份并递增版本号，以防止文件混乱(关于版本号的命名规则本书不详述)。

如果你的程序需要长期维护，应考虑使用 git 进行维护(关于 git 的使用本书不详述)。

3.1.8 增强剪切板

为了增强剪切板的功能。对于 Mac 系统，可以使用 Paste、Elfron 等软件。对于 Windows 系统，可以用 WIN+V 组合键打开 Windows 自带的剪切板管理软件。

运用熟练后你会发现这是一个很好的工具。

3.1.9 [Pycharm 功能]书签

写超长代码(作者经验结论：超过 10000 行的代码文件)时，不断地翻找某块曾经写过的代码是非常费时费力的。我们可以利用 Pycharm 的书签功能，在重要的，常常需要修改和回顾的区域加上书签。我们可以在 Pycharm 中用 shift+F11 组合键打开书签列表，然后跳转到指定书签的位置(可以跨文件)。对于超长代码而言这是一个非常好用的功能。

除此之外还有以下替代方案解决这个问题：

(1)：写一些特别的注释(代码本身中不会出现的词)，然后 ctrl+F 搜索注释。

(2)：代码头部写一个所有 def foobar(): 的汇总文档，并记住常用的 def foobar(): ，直接搜索函数名。

(变量名取为 foobar 或者 foo 表示它是一个一般性的名字，无特别意义。)

所以一个 def foobar(): 内部尽量不要写得太长(作者经验结论：不超过 500 行)，否则找到这个函数后，在内部找又是很麻烦的。

(3)：模块化编程，封装代码，降低每份代码的长度。

3.2 特性

3.2.1 单行结构

Python 中可以实现将一些多行结构压缩成单行结构，以下语句都是合法的：

```
(1)if 1 + 1 == 2: print("Correct!")

(2)for i in range(5): print(i)

(3)print("I am ", end = ""); print("bored.")

(4)x = 3; print("even " if 3 % 2 == 0 else "odd")
```

一般而言，不推荐这些写法，因为可读性不好。但如果需要节约行数，可以考虑利用这个特性来做一些事。其中第一二句表明，if 或者 for 语句后的内容可以直接写在这一行后面；第三句表明，Python 中可以用分号分隔两个语句，并且分句可以同时写在同一行上；第四句语句中分号右边的形式称为*三元表达式*。

另外还有一些单行结构，

如同时导入多个模组 import gc, copy, traceback

同时进行多项变量赋值 a,b,c = [],[],[] 或 x,y = 1,2

有时候这类结构能节约代码量和代码行数。

3.2.2 float 陷阱

首先运行如下代码:

```
print(0.3333 + 0.6667 + 0.6667)

print(0.6667 + 0.6667 + 0.3333)
```

```
print((0.3333 + 0.6667 + 0.6667) == (0.6667 + 0.6667 + 0.3333))
```

结果如下：

```
1.6667

1.6666999999999998

False
```

分析代码，容易看出，Python 的 float 存在一些精度问题。

解决方案：借助 decimal 模块存储和操作小数，本书不再详细展开。

顺便说一些关于小数的知识：

Python 中小数 3.0 可以写为 3.，而小数 0.3 可以写为.3。

Python 中可以用 float('inf')创建一个无穷大的小数。

3.2.3 充分利用 in 运算符

range()函数是 Python 中一个很好的用于循环的函数，但并不总是最好的，循环次数是固定的情况下：for i in [0,1,2]: 的写法比 for i in range(3): 更清晰且更高效。

(timeit 是 Python 中用于测试小段代码运行时间的一个简单模块，除此外我们还可以使用装饰器计算时间，或者用[Juptyer 功能]%timeit -o 魔术命令等来计算函数运行时间。对于递归形式的函数实现，还可以用 clockdeco 库的 clock 装饰器分步输出时间和调用过程。)

出于测试的目的我们运行如下代码：

(提示：一般而言，我们应避免将变量名取名为 A,B,C 或将函数名取名为 f1,f2，而应该取名为有意义的名字(也许是比较长的名字或者由多个单词由下划线相连的名字)。本书的演示例子中，变量名取名还是以简单为主，实际使用中尽量取有意义的名字。)

(Python3 中变量取名可以是中文，但我不推荐这么做。)

```
import timeit

from timeit import Timer

def f1():

    for i in range(3): pass

def f2():

    for i in [0,1,2]: pass

if __name__=='__main__':
```

```
    t1=Timer("f1()","from __main__ import f1")
    t2=Timer("f2()","from __main__ import f2")
    print (t1.timeit(10000))
    print (t2.timeit(10000))
```

结果如下：

```
0.008436999999958061
0.0038281999998162064
```

打印结果分别表示函数 f1 和 f2 的估计运行时间，可以发现用 in 的写法更高效。

另外如果 if 后跟较多的 or 项，也应考虑用 in 代替，可以更高效并且节约代码量(也更清晰)。出于测试的目的我们运行如下代码：

```
import timeit
from timeit import Timer
def f1():
    i = 0
    if i == 1 or i == 2 or i == 5 or i == 6: pass
def f2():
    i = 0
    if i in [1,2,5,6]: pass
if __name__=='__main__':
    t1=Timer("f1()","from __main__ import f1")
    t2=Timer("f2()","from __main__ import f2")
    print (t1.timeit(10000))
    print (t2.timeit(10000))
```

结果如下：

```
0.0045046000000183994
0.003779600000143546
```

可以发现用 in 的写法更高效并且节约了代码量。

3.2.4 充分利用 set 对象

set 的特点是元素互异，对于集合而言&、|、-分别表示取交集、取并集、差集运算，基于此可以完成许多工作。

例子：

判断列表 A 和列表 B 中是否有两个相同的元素。

```
A = [1,2,3,4]

B = [3,4,5,6]

target_bool = len(set(A) & set(B)) == 2 #用 target_bool 存储列表 A 和列表 B 中是否有两个相同的元素这个真值。
```

另一例：

对列表 A 进行去重。

```
A = list(set(A))
```

3.2.5 del 关键字与垃圾回收机制

del 关键词可以删去一些变量，从而节约内存。

简单例子：

```
import numpy as np

ID = 1

np.random.seed(ID)

n=100

x=np.linspace(0,100,n)

y=(ID % 3+1)*x**(ID % 4)+(ID % 2+1)*x+(ID % 100)

y=y+np.random.normal(0,(y[-1]-y[1])/20,n)

data=np.transpose(np.array([x,y]))

del n

del x

del y
```

Python 垃圾回收机制是一个较大的话题，下文将描述一部分其中的重要内容，对原理感兴趣的读者可以了解变量的*弱引用*等概念，阅读相关资料（http://docs.python.org/3/library/weakref.html）。

通常函数的调用会创建许多局部变量，这些局部变量会在运行完毕后被回收。另一种情况是执行 `x = 0` 然后再执行 `x = 1`，那么第一个 `x` 也会被回收。

如果涉及循环引用，例如：

```
listA, listB = [], []

listA.append(listB)

listB.append(listA)
```

那么它们就无法被默认的*引用计数*机制自动回收，很可能会导致*内存泄漏*，为了解决这个问题，我们应该使用 gc 库(gc 是 garbage collection 的简称)来增强垃圾回收机制。本文希望读者能形成这个问题意识，关于如何具体处理此问题的具体细节，读者可以查阅 gc 库的官方文档（https://docs.python.org/3/library/gc.html）等相关资料，本书不展开详述。

3.2.6 变量的内涵

变量的一种流传的理解是"装对象的盒子"，但更好的理解是把 Python 中的变量理解为"对象的便利贴"，对象是先于变量存在的，并且一个对象可以贴多个"便利贴"。这种认知的转变可以很好地改善对 Python 中变量的理解。

同时注意在 Python 代码中：列表，元组等容器保存的是对象的标签，而不是对象本身。

(Python 中，变量分为两种类型，可变的和不可变的。例如集合，列表，字典是可变的，字符串，元组等是不可变的。)

(通常不可变的对象也是可散列的对象，可变的对象也是不可散列的对象，但这个不是一定的。一个对象是否可以散列取决于它是否拥有__hash__魔法方法，是否可变取决于能否在不改变该对象标识的情况下改变该对象的值。)

(注意元组中存储了对象，元组的不可变性指的是元组中存储的对象的标识(即物理内存)不可变，对象的值可以变化。)

在 Python 中，须注意以下事实：

(1)如果一个自定义函数函数以一个可变的变量作为参数，例如 list，那么在函数中的所有对该变量的操作都会导致该变量的永久改变，而不仅仅是在函数体内部的局部改变。

(2)我们应该避免将一个可变的变量作为函数的默认参数(例如 def f(p = [])):，这意味着多次(不指定该参数的情形下)调用该函数时会共用一个可变对象，很容易出 bug。

3.2.7 赋值陷阱与三种拷贝

Python 中存在三种拷贝方式，称之为直接赋值，浅拷贝和深拷贝。其实现方式很简单，例如我们有一个列表 mylist。

(浅拷贝和深拷贝需要首先导入 copy 模块，这里我们只要导入 copy 模块下的 copy 函数和 deepcopy 函数就可以。)

三种拷贝方式的代码实现过程如下：

```
from copy import copy, deepcopy

mybackup_a = mylist #直接赋值

mybackup_b = copy(mylist) #浅拷贝

mybackup_c = deepcopy(mylist) #深拷贝
```

初学者如果不懂 Python 的变量的内涵以及 Python 的三种拷贝，则很可能会陷入"赋值陷阱"并产生疑惑。这是因为初学者往往容易把一般的用等号做的"直接赋值"理解为深拷贝，从而可能引起一些迷惑性很强的漏洞。理解 Python 的三种拷贝是很有必要的，可以让我们避免因拷贝带来的问题。

以下对三种拷贝进行辨析，均假定 a 是要拷贝的对象的变量，b 是拷贝后的对象的变量,obja 是要拷贝的对象，sobja 是所有 oa 的子对象。

(根据上一节"变量的内涵"中的说法，变量是对象的便利贴。)

(所谓子对象，可以简单地这样理解，例如一个列表中包含一个字符串，那么那个字符串就是这个列表对象的子对象。)

- 直接赋值：

b = a 就是用 b 对 a 进行直接赋值，其意义是让 b 这个变量贴到 obja 对象上，也就是说变量 a 和变量 b 同时指向 obja 这一个对象。

此时如果对 a 作一些变化操作，那么也会影响 b。

读者可以运行如下的示例代码：

```
a = [1]

b = a

a.append('1')
```

```
print("b =",b)
```

打印结果为:

```
b = [1, '1']
```

请注意一件事，a.append('1')可以称之为对 a 做了改变。如果让 a 赋值给另一个变量，如 a = ['foo']，这不能叫 a 做了改变，只能称之为 a 指向给了一个新的变量，原来的 objа 对象没有受到影响，那么不会影响 b。**这一点必须理解清楚，是 Python 中很重要的一个特性，**作者在此再给出一个简单例子说明此现象，代码如下：

```
a = 3

b = a

a += 1

print("a =",a)

print("b =",b)
```

打印结果为：

```
a = 4

b = 3
```

类似于上述 a 和 b 都指向[1]列表的例子，此例中 a 和 b 都指向整数 3 这个对象，但为什么这里改了 a 的值，b 不会改变？因为实际上整数是"不可变"的对象，a += 1 的意义为 a = a + 1，这并不是说改变了 a 的值，而是把 a 贴给了 a + 1 这个新的对象。整数 3 没有发生改变，所以 b 也不可能发生改变。

- 浅拷贝

b = copy(a)就是用 b 对 a 做浅拷贝，其意义为建立一个 objа 的新的副本，并让 b 指向这个新的副本(记为 objb，其所有子对象为 sobjb)，所以 b 和 a 指向的是不同的对象。

但浅拷贝不会对子对象建立副本，所以 sobjb 和 sobjа 是完全相同的。

读者可以运行如下的示例代码：

```
from copy import copy,deepcopy

a = [['foo'],5]

b = copy(a)

print("a =",a)
```

```
print("b =",b)
print("*" * 30)
a.append(6)
print("a =",a)
print("b =",b)
print("*" * 30)
a[0].append('foobar')
print("a =",a)
print("b =",b)
print("*" * 30)
```

打印结果为:

```
a = [['foo'], 5]
b = [['foo'], 5]
******************************
a = [['foo'], 5, 6]
b = [['foo'], 5]
******************************
a = [['foo', 'foobar'], 5, 6]
b = [['foo', 'foobar'], 5]
******************************
```

从此例中我们可以看到，对 **a** 作改变不会影响 **b**，它们指向独立的对象，但是对 **a** 的子对象做操作却会使 **b** 的子对象同步变化，这便是浅拷贝的特点。

- 深拷贝：

b = deepcopy(a)就是用 **b** 对 **a** 做深拷贝，其意义和浅拷贝类似，但不仅仅是新建 **obja** 的副本，而是连 **obja** 的子对象也一并进行复制了。无论是深拷贝还是浅拷贝，得到的对象与原来的对象的标识都不同。

运行如下的示例代码：

```
from copy import copy,deepcopy

a = [['foo'],5]

b = copy(a)

c = deepcopy(a)

d = a

print("*" * 30)

print("ida 的尾号=",id(a) % 10000)

print("idb 的尾号=",id(b) % 10000)

print("idc 的尾号=",id(c) % 10000)

print("idd 的尾号=",id(d) % 10000)

print("*" * 30)

print("id(a[0]) 的尾号=",id(a[0]) % 10000)

print("id(b[0]) 的尾号=",id(b[0]) % 10000)

print("id(c[0]) 的尾号=",id(c[0]) % 10000)

print("id(d[0]) 的尾号=",id(d[0]) % 10000)

print("*" * 30)
```

打印结果为：

```
******************************

ida 的尾号= 3552

idb 的尾号= 4384

idc 的尾号= 1280

idd 的尾号= 3552

******************************

id(a[0]) 的尾号= 6896

id(b[0]) 的尾号= 6896

id(c[0]) 的尾号= 112

id(d[0]) 的尾号= 6896
```

****

通过打印标识(两个对象的标识相同，它们是相同的变量)，我们可以从此例中看出三种 Python 拷贝方式的差别。

3.2.8 交换值与元组拆包

在其他语言中交换 A,B 的值可能需要三条语句如 C=A, A=B, C=A;

Python 中的写法：A,B = B,A;

多个值交换也可以类似写 A,B,C = C,A,B;

上述代码隐含着元组拆包的原理。

元组拆包指的是，对于 python 中的任何可迭代对象，我们可以通过一些特定的句法直接获取该对象其中的部分子对象。这个概念直接说明比较难以理解，请结合以下例子进行理解。

```
my_object = (1, 2, 3, 4, 5)

X, _, _, _, Y = my_object
```

(提示：在一些编程语言中，_ 表示占位符，视为我们不关心的变量。)

(引用 *Fluent Python* 中的说法：如果做国际化软件，_ 可能不是理想的占位符，否则它是一个很好的占位符，文档 https://docs.python.org/3/library/gettext.html 提到了这一点。)

那么上述代码实际上就完成了 X=1, Y=5，也就是我们成功地将元组的第一个值和最后一个值赋值给了 X 和 Y。

并且 X, _, _, _, Y = my_object 有简化写法如下：

```
X, *args, Y
```

简单之，在变量名前加星号，则其可以指代不确定数量的变量。

3.2.9 了解 is 和==的区别

主要有两个方面：

一：

is 比较两边是否是同一个对象，是内存地址相等的概念，判断占用的内存地址是否一样，或者说两个对象的标识是否一样，我们可以用 id()函数获取一个对象的标识。

(标识是任何对象的一项生命周期内恒定不变的，一一对应的属性，是一个整数。)

而 == 比较两边是否“相等”，是值相等的概念，不同的对象“相等”的意义其实是不同的。 a == b 本质上是调用了所属类的__eq__()魔法方法，换句话说 a == b 是 a.__eq__(b)的语法糖。

(魔法方法是 Python 中的一种方法，它是类和对象的一项属性，详见本书 3.4 节)

涉及字符串(或者一些其他内置对象)的比较时，还会涉及一个 Python 的***驻留(interning)***问题，这是一个比较复杂的且深刻的实现细节问题，没有官方文档，本书不对此展开描述，有兴趣的读者可以查阅相关资料(***Fluent Python*** 第八章最后一节提到此细节。)。

二：

判断一个变量是不是 None 对象，用 is 而不要用 == 。通常而言 is 的效率高于 == 。

举一例说明：

```
N1 = None
N2 = None
N3 = N1
print("N1 == N2 ? {}".format(N1 == N2))
print("N1 == N3 ? {}".format(N1 == N3))
print("N2 == N3 ? {}".format(N2 == N3))
print("N1 is N2 ? {}".format(N1 is N2))
print("N1 is N3 ? {}".format(N1 is N3))
print("N2 is N3 ? {}".format(N2 is N3))
print("N1 == None ? {}".format(N1 == None))
print("N1 is None ? {}".format(N1 is None))
L1 = [None]
L2 = [None]
L3 = L1
print("L1 == L2 ? {}".format(L1 == L2))
print("L1 == L3 ? {}".format(L1 == L3))
```

```
print("L2 == L3 ? {}".format(L2 == L3))
print("L1 is L2 ? {}".format(L1 is L2))
print("L1 is L3 ? {}".format(L1 is L3))
print("L2 is L3 ? {}".format(L2 is L3))
S1 = "I see sunshine"
S2 = "I see sunshine"
print("S1 == S2 ? {}".format(S1 == S2))
print("S1 is S2 ? {}".format(S1 is S2))
S1plus = "I see sunshine!"
S2plus = "I see sunshine!"
print("S1plus == S2plus ? {}".format(S1 == S2))
print("S1plus is S2plus ? {}".format(S1 is S2))
R1 = 3.
R2 = 3.
print("R1 == R2 ? {}".format(R1 == R2))
print("R1 is R2 ? {}".format(R1 is R2))
I1 = 257
I2 = 257
print("I1 == I2 ? {}".format(I1 == I2))
print("I1 is I2 ? {}".format(I1 is I2))
```

结果如下：

```
N1 == N2 ? True
N1 == N3 ? True
N2 == N3 ? True
N1 is N2 ? True
N1 is N3 ? True
N2 is N3 ? True
```

N1 == None ? True

N1 is None ? True

L1 == L2 ? True

L1 == L3 ? True

L2 == L3 ? True

L1 is L2 ? False

L1 is L3 ? True

L2 is L3 ? False

S1 == S2 ? True

S1 is S2 ? True

S1plus == S2plus ? True

S1plus is S2plus ? False

R1 == R2 ? True

R1 is R2 ? True

I1 == I2 ? True

I1 is I2 ? False

3.2.10 特殊的 else

Python 中的 else 有一种独特的用法，即可以在一段循环结构后面加 else 结构。其意义不作一般 else 的"否则"意义理解，而是"恰好与否则有些相反"地理解为，如果循环结构内的内容正常执行完，就执行 else 块里的内容，如果因为 break 跳出了循环结构，就不执行 else 块里的内容。

一个简单示例如下：

```
for i in [1,2,3]:
    print('for run', i)
else:
    print('else run')
```

其输出结果为：

```
for run 1
```

```
    for run 2

    for run 3

    else run
```

根据 *Effective Python* 所述，如无特别理由，我们不应利用这个特性去写代码，容易写出令人迷惑的代码。根据 *Fluent Python* 所述，在一些情况下，我们可以利用这种结构去实现一些效果或者简化代码量。

3.3 函数与对象

本书中的“函数”和“方法”是同义的。当谈到“函数与对象”时，这个对象指的是***可调用对象***。当谈到“类与对象”时，这个对象指的是类的***实例化***产生的具体对象。

3.3.1 推导式

推导式又名解析式，包括列表推导式，字典推导式，集合推导式和生成器推导式等。这是 Python 中的一种**特别方便且特别常用**的工具，并且它在很多时候还有实际效率的提升。

(如你首次接触此概念，可以首先学习”列表推导式”。)

关于推导式的资料有许多，因此本册不再详细叙述。

学会列表推导式后我们应该有意识地多主动使用这个工具。但是请不要滥用这个工具(如超过两层嵌套的推导式或者借由推导式的中间过程去完成你的工作等)，这样代码可读性会很差。

3.3.2 函数的可变参数

Python 的函数可以加可变参数，有时候这是极为方便的一种用法。

语法很简单，在函数的位置参数前加上一个星号或者两个星号即可，例如:

```
def my_function(para1, *para2, **para3):

    pass
```

出于风格一致性的目的，一般可以让一个星号和两个星号的可变参数命名为 ***args** 和 ****kwargs** 。

然后 **args** 和 **kwargs** 会变成函数的两个局部变量，分别以列表的形式和字典的形式。

一个示例:

```
def hasitem(**kwargs): # 判断是否拥有指定条目
    target_id = -1
    target_label = "no"
    require_valid = 0
    if 'target_id' in kwargs: # 如果判断时对目标条目序列号有要求 …
    if 'target_label' in kwargs: # 如果判断时对目标条目标签有要求
        …
    if 'require_valid' in kwargs: # 如果判断时对目标条目有效性有要求
        …
    …#实现判断过程
    return 0
```

这个例子里可以看到 **kwargs** 相当于变成了函数的一个局部变量，调用这个函数的时候可能是这样写的 hasitem(target_id = 100, require_valid = 1)，那么调用函数时 **kwargs** 就是这样的一个字典 {target_id:100, require_valid:1} 的局部变量。

(作者的经验结论：很多时候自定义类的 __init__ 魔法方法可以考虑用可变参数的形式来设定，因为我们可能经常需要拓展自定义类的 __init__ 方法或者 __init__ 里有非常多的参数，此时用可变参数可以极大地增强可维护性以及减少视觉杂讯。)

3.3.3 enumerate 函数与 zip 函数

enumerate 函数与 **zip** 函数是 **Python** 中一种很方便的循环工具。

简言之，**zip** 函数可以将多个长度相同的列表 **L1, L2, L3** 按次序逐个“缝合”得到一个新列表。

(还有一种特殊用法，即 **zip**(*变量名)，它相当于逆向的 **zip** 操作。)

enumerate 函数则将一个列表 **L** 与其元素序号逐个“缝合”得到一个新列表，**特别常用**。

以例子展示之：

```
lst1 = ['A','B','C','D']
lst2 = ['E','F','G','H']
lst3 = [1,2,3,4]
```

```
print("1 ========")
for I in enumerate(lst1):
    print(I)
print("2 ========")
for I in zip(lst1,lst2):
    print(I)
print("3 ========")
zipped_object = zip(lst1,lst2)
for I in zip(*zipped_object):
    print(I)
print("4 ========")
for I in zip(lst1,lst2,lst3):
    print(I)
print("5 ========")
zipped_object_2 = zip(lst1,lst2,lst3)
for I in zip(*zipped_object_2):
    print(I)
```

打印结果为：

```
1 ========
(0, 'A')
(1, 'B')
(2, 'C')
(3, 'D')
2 ========
('A', 'E')
('B', 'F')
('C', 'G')
```

('D', 'H')

3 ========

('A', 'B', 'C', 'D')

('E', 'F', 'G', 'H')

4 ========

('A', 'E', 1)

('B', 'F', 2)

('C', 'G', 3)

('D', 'H', 4)

5 ========

('A', 'B', 'C', 'D')

('E', 'F', 'G', 'H')

(1, 2, 3, 4)

(zip 函数或者 enumerate 函数产生的新列表每个元素都是元组，这不影响循环的使用，因为 python 可以自动对元组拆包。)

3.3.4 生成器

使用 yield 关键词的函数或方法是生成器函数。调用生成器函数返回的是生成器对象。

生成器是一个可调用对象，我们可以用 Python 内置的 callable() 函数判断一个对象是否是可调用对象。

当我们需要惰性地实现一个迭代器时，生成器适合我们的要求。

(什么是迭代器？迭代器即实现了 __iter__ 无参数方法的对象。简单理解为：列表，字典等对象是常见的迭代器，可用于作循环操作；可以用 next(this_iterator)来获取下一个迭代值。)

(什么是惰性？简言之，每次实际调用这个对象时，才会执行该对象的生成数据的函数，则它是惰性的。[x**0.5 for x in range(100)]不是惰性的，因为它会一步生成 100 个数据。)

(一个常见的实际场景为：预读取一个很长的一篇文章，则用生成器是适合的。如果用迭代器，整篇文章都要进入内存，容易引起内存溢出。但生成器的用处不止于此，还可用于按规则生成序列，或达到类似递归的效果等。)

要理解生成器函数，关键是要理解”每次调用生成器函数会返回一个生成器”，如果多次调用则返回多个，且多个生成器之间是相互独立的。

通过 next(this_iterator)调用一个生成器，”生成”一个值。

对于一个特定的生成器，它和函数的区别在于：函数是运行至 return 然后返回值结束该函数；生成器是运行至 yield 然后”生成”值并结束本次运行，下次运行时会接着上次 yield 的位置继续向下运行，运行到下一个 yield 时，”生成”新的值并结束本次运行，如果没有找到新的 yield，则会抛出 StopIteration 异常。

(yield 有两个常见意思：让步和生产。Python 的生成器中的 yield 同时体现了这两种意思，即生产数据，同时生产完毕后暂停为下一次生产”让步”。)

itertools 库中提供了许多生成器函数，读者可以了解之以避免重复造轮子。关于 itertools 库的详细内容可见文档（https://docs.python.org/3/library/itertools.html）。

3.4 类与对象

3.4.1 魔法方法

Python 的魔法方法(又名魔术方法，特殊方法等)是一个极其强大的面向对象编程的工具。其严谨意义不详述。简言之，一个对象的魔法方法是不需要主动调用，而是 Python 解释器遇到+, /, ==, del, in, for, print, object[], len(foo)等自带语法时自动调用的方法。有了魔法方法可以使程序变简单。

例如假设 x 是一个自定义类，如果你写了 for i in x: 这实际会多次调用 iter(x)，而这个实际上又是调用了 x.__iter__()方法。

(一个意识：类的魔法方法并不是固定不变的，有时候我们可以动态地修改一些自定义类的魔法方法。例如：我们可以写完一个类 Myclass, 然后紧接着去写 Myclass.__len__() = func (这里的 func 可以是一个函数)。这样 __len__ 的实现过程就写在 Myclass 外部了，目的是让代码更加精简，或者实现 func 的复用性等其他效果。)

Python 有近百种默认的魔法方法(仍在增加中)，读者可以查阅完整的 Python 语言参考手册中关于魔法方法的资料(https://docs.python.org/3/reference/datamodel.html)，其中可以重点关注__iter__, __repr__, __str__, __bool__ , __mul__, __len__, __setitem__, __getitem__, __new__, __del__, __eq__, __call__ 等比较重要的魔法方法。

通过修改__add__, mul__, __eq__一类的魔法方法，我们可以实现***运算符重载***(即改变这个类的加号，乘号，等于号操作的意义)。如果使用得当，这个特性会让你的代码更加灵活易读。但滥用运算符重载可能会遇到不可预估的缺陷。

魔法方法自然地引出程序设计中的一个概念，称之为***鸭子类型***，鸭子类型的原文描述是"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"这句话旨在强调我们不用关注鸭子类型本身是什么，而是应该关于它是如何工作的，具有哪些属性。

这个思想在 Python 中如何体现的呢？就是魔法方法了，比如我自定义了一个类 myclass，并为其定义了__iter__魔法方法或者__get__魔法方法，那么它就可以像列表一样被迭代，也即可以用 for i in myclass()的语法，不需要 myclass 真的是列表，这样就使程序设计灵活。

3.5 代码提速

3.5.1 用**作乘方运算

Python 中 a**b 表示计算 a 的 b 次幂得到的数值，我们还可以利用 pow(a,b)和 math 模块里的 pow(a,b)函数作等价的计算。可以验证三种方法中，a**b 有最好的效率，所以一般而言推荐直接用**作乘方运算即可。

3.5.2 用 while 1:而不用 while True:

Python 写一个死循环，while 1:的效率比 while True:高一些，这是因为 Python 中的布尔值在进行判断或运算时，它会首先转为 int 类型, bool 是 int 的子类。

运行如下代码就可以发现在 Python 中 bool 类型是 int 类型的子类：

```
issubclass(type(True),type(1))
```

结果为：

```
True
```

不过实际中这个一般区别是不大的，因为主要花费的时间是在于业务逻辑上。

3.5.3 一些等价替换写法

- a = list(range(N)) 总是比 a = [x for x in range(N)] 快。

(注意：如果你确定你的这个列表是固定的，以后不会再改，可以考虑用元组 a = tuple(range(N))，会比列表高效许多。)

- n * m == 0 大多数情况下比 n == 0 or m == 0 快(差异较小)。
- 用" ".join["今晚的", "月色", "真美"]的形式来拼接一系列字符串，效率相对于用加号等方式较高。

- 有时候我们可能获得了两个点的坐标(x1,y1), (x2,y2)，然后我们需要判断两点之间的距离是否大于一个特定的值 distance。我们很可能用两点间距离公式去做这个问题，但考虑到 Python 计算平方比计算开二次根平均快 20%左右，我们可以将判定式子两边平方以提升效率。
  原来的代码：((x1-x2)**2-(y1-y2)**2)**0.5 > distance

  替换代码: (x1-x2)**2-(y1-y2)**2 > distance ** 2

- -(x+1) 可以替换为 ~x，波浪号表示按位取反，效率更高。(这里的 x 指整数，~x 实际上是 x.__invert__()的语法糖。)

- 用 str(L)来将一个列表内的元素转为字符串型，而不要用[str(x) for x in L]。前者的代码更简洁且更高效。

4 初级进阶

4.1 习惯与通识

4.1.1 避免局部测试时执行全局计算量

局部测试通常是高频的，应该设法避免局部测试时执行繁重的全局计算量，例如一个繁重的数据预处理，或者读取一个大文件，或者一个图形界面(窗口)的执行等。

前面两种情况代码耦合性天然低一些，相对好处理。尤其是第三种情况，即做一个大型图形的程序时，我们总应在设计阶段提前考虑好测试问题，让测试和窗口可分离，因为对于测试而言展开一个窗口是过于费时的。

(我们应该“将显示层与逻辑层”分离，这体现了***高内聚，低耦合***的设计准则。)

如果工程量比较小，可以不考虑以上问题。

如果工程量比较大，测试频率很高，应考虑使用 Python 自动化测试框架，例如 pytest 模块、unittest 模块、Robot Framework 模块(测试框架是一项很有效的工具，但本书不再对此展开叙述)。要入门测试框架(pytest)，可以阅读这本参考书 *Python Testing with pytest* - Brian Okken，要进一步熟悉测试框架，可以阅读官方文档（https://docs.pytest.org/en/latest/plugins.html）。

4.1.2 利用类传值

对于类，最基本的理解是，它是一系列具有某些特点的物的集合。

这种理解有助于理解类的基本概念，但进一步来说有时候反而会阻碍我们对于类的灵活使用。因为，我们实际上是可以把任何我们想要打包操作的一个整体视为一个类，一种简单的说法是”利用类传值”。

一个传值的简单的例子是要构建一个图形化界面，我们自定义了一个父级窗口，又为窗口定义了一些按钮，复选框，滚动条之类的子组件。由于子组件需要与父级窗口联动，新建子组件时，我们可能要在子组件的__init__函数里传入父级窗口这个对象，以收取父级窗口的各种数据并与之关联。那么在这里父级窗口整理可以视为一个用”类”打包的整体，其中包含了各种各样的数据可以一并通过一个变量传给子组件的初始化函数。

当然上述例子是一个简单示例，我们实际做图形化窗口时总是基于一些第三方模块(题外话：作者推荐用 Pyside2 这个模块去做 GUI)。在这些第三方模块里面那些窗口一般都已经是一些做好了的类了。作者举这个例子是为了展现这种打包的思想，例如假设你的脚本有一些数据需要存储到本地(或上传至服务器等)，则你在设计代码时，

应提前规划好，**把这些要存储的东西打包放到一个”类”里面**，这样写出来的代码可维护性会好很多。

4.1.3 利用类维护程序的状态

在本书的 3.1.6 小节中曾提到，应有系统地管理全局变量的意识(这样做可以理解为维护程序的状态)，当时是用列表来管理全局变量。这里则另外强调，我们应该用类维护程序的状态。

普遍推荐用类去维护程序的状态，而不是用列表或者字典。其原因是类的可维护性强。我们设计程序时，如果发现用字典去存储数据存在多层复杂嵌套，就应该迁移到类上，但这一步可能有繁琐的工作量。由于程序的状态很有可能需要不断扩展，所以适合在最早的时候就用类维护程序的状态，就不用考虑之后迁移至类的问题了。

4.1.4 利用类编写嵌套式信息

所谓嵌套式信息，指数据中具有大量重复性的，互相嵌套的关系。

例如现在要编写并存储七种食物的数据:

```
apple = { 'Name':'苹果', 'Price':5, 'Charges':1 , 'Description':'新鲜苹果',
'Type':'FOOD', 'Eat':10, 'Drink':8,'Shelf':15 * 24 * 60, 'Labels':['Natural', 'Fruit']}

lemon = { 'Name':'柠檬', 'Price':8, 'Charges':1, 'Description':'可食用水果',
'Type':'FOOD', 'Eat':8, 'Drink':6,'Shelf':30 * 24 * 60, 'Labels':['Natural', 'Fruit']}

banana ={'Name':'香蕉', 'Price':5, 'Charges':5, 'Description':'可食用水果',
'Type':'FOOD', 'Eat':12, 'Drink':6,'Shelf':6 * 24 * 60, 'Labels':['Natural', 'Fruit']}

cake = {'Name':'蛋糕', 'Price':50, 'Charges':1, 'Chargeable':True ,
'Description':"", 'Type':'FOOD', 'Eat':65, 'Drink':10, 'Processed':2,'Shelf':4 * 24 * 60,

'Labels':['Baking']},

donut:{ 'Name':'甜甜圈', 'Price':9, 'Charges':1, 'Chargeable':True,
'Description':"", 'Type':'FOOD', 'Eat':10, 'Drink':1, 'Processed':3,'Shelf':3 * 24 * 60,

'Labels':['Baking']}

cookie:{ 'Name':'曲奇饼', 'Price':3, 'Charges':1, 'Chargeable':True,
'Description':"", 'Type':'FOOD', 'Eat':4, 'Processed':2,'Shelf':90 * 24 *
60,'Labels':['Baking']}

boiled_fish = { 'Name':'水煮鱼', 'Price':10, 'Charges':1, 'Chargeable':True,
'Description':"", 'Type':'FOOD', 'Eat':10, 'Drink':5, 'Processed':1,'Shelf':3 * 24 *
60,'Labels':['Stew']}
```

#Shelf 表示保质期，Charge 表示一份该物品有多少件, Eat 和 Drink 表示食物充饥和解渴的能力，Processed 表示物品的加工程度(非天然程度)。

可以看到，如果用字典去存储这些数据，则代码量很大，且显得杂乱。

我们应该用类的继承的思想去处理这个问题，首先定义一个最大的 Item 物品类，然后定义 Food 食物子类(本小节只是给出一个简单示例，实际可能还有食物以外的各种各样的物品)，然后定义 Natural_Food 天然食品类和 Processed_Food 加工食品两个子类。再定义 Fruit 类去继承 Natural_Food 类，再去定义 apple 苹果类(最细的数据也用类维护是为了如果后续要增加细分的苹果类型，容易修改)。

对上述的改写示例如下(没有完全还原，仅还原了部分物品，其余是类似的)：

```
class Item():
    def __init__(self):
        self.name = 'default_name'


class Food(Item):
    def __init__(self):
        self.charge = 1
        self.type = 'FOOD'


class Fruit(Food):
    def __init__(self):
        self.description = '可食用水果'
        self.labels = ['Natural', 'Fruit']


class Dessert(Food):
    def __init__(self):
        self.description = ''
        self.labels = ['Natural', 'Fruit']


class Apple(Fruit):
```

```
    def __init__(self):
        self.name = '苹果'
        self.price = 5
        self.eat = 10
        self.drink = 10
        self.description = '新鲜的有机苹果快来买呀！'
        self.shelf = 15 * 24 * 60


class Banana(Fruit):
    def __init__(self):
        self.name = '香蕉'
        self.price = 5
        self.eat = 12
        self.drink = 8
        self.charge = 5
        self.shelf = 6 * 24 * 60
```

这种数据更为清晰。并且如果我们想要添加新的水果，只需关注水果之间的差别即可，不需要关注水果和甜品之间的差别了，这就可以极大地减少工作量(本例子尚为简单，有时我们可能需要编写和存储大量这样的数据)。

4.2 特性

4.2.1 动态打印

当我们 print 一个字符串的时候，可以在这个字符串最前面加上\r，表示覆盖当前这一行的内容，从而实现动态打印(如打印一个进度条)。

请读者运行如下代码以查看动态打印效果：

```
import time
print('=====第一个例子=====')
for i in range(11):
```

```
        time.sleep(0.2)
        print('\r 加载中：{0}{1}%'.format('■'*i,(i*10)), end='')
print("")
print('加载完成！')
print('=====第二个例子=====')
for i in range(20):
        time.sleep(0.1)
        print('\r 加载中：{0}{1}{2}'.format('-'*i,'>','-'*(19-i)), end='')
print("")
print('加载完成！')
```

4.2.2 动态导入

动态导入即利用__import__来导入模块而不是用 import。

一种使用场景为：许多 IDE 在导入模块的时候会检查是否已经有该名称的模块，如果有则不导入。例如 Jupyter 编辑器，如果不重启 Jupyter 内核，则多次导入一个模组是无效的。但如果这个模组本身是变化的，或者是要编辑的自定义模组，就需要用__import__来动态导入。

我们用一个变量来接收动态导入的模组。

其基本的使用示例如下：

```
math = __import__( "math ")
print(math.pi)
np = __import__( "numpy ")
print(np.pi)
```

4.3 函数与对象

4.3.1 装饰器

装饰器是一个以函数为参数，以函数为返回值的函数。装饰器是函数*闭包*的*语法糖*。

(闭包是一种***高阶函数***——可以接受函数作为参数或者返回函数的函数。可以简单地把闭包理解为生产函数的工厂函数。)

(语法糖即不增加代码功能，仅让代码更简洁的工具)

装饰器可以用于增强函数的功能，Python 中有一些预设的装饰器，例如 @classmethod, @abstractmethod 等等。如果你的代码频繁地涉及增强函数的功能，自定义的装饰器是一个很好的选择，或者你有多个函数公用同样的前置操作，则也可以考虑使用装饰器。

装饰器的用法是加一个@符号在装饰器名前，然后在下方写需要增强的函数名，例如：

```
@classmethod
targetfunction
```

上述代码等价于:

```
targetfunction = classmethod(targetfunction)
```

我们鼓励第一种写法，其写法更简洁易维护。

装饰器也可以叠加应用，语法为分两行写两个装饰器，再于下方写函数。

语法示例如下：

```
@d1
@d2
def func():
    pass
```

这等价于 func = d1(d2(func))。装饰器的作用顺序遵循"就近原则"，此例中 func 函数先被 d2 装饰器增强，再被 d1 装饰器增强。

(值得注意的是，被装饰器装饰之后的函数已经是另外一个函数了，其函数名称，函数的__doc__属性等会发生改变，这可能会使得测试时产生意想不到的结果。解决方法：使用 functools 模块下的 wraps 装饰器增强函数。)

4.3.2 含参装饰器

实现含参装饰器，通常是需要一个装饰器工厂函数(即返回装饰器的一个函数)，为了解其语法结构，这里给出一个示例如下：

```
def deco_birth_plant(adj = 1):
    def decorate(func):
        print(adj)
        return func
    return decorate

#第一种用法
@deco_birth_plant()
def myfunc():
    pass

#第二种用法
@deco_birth_plant(adj = 5)
def myfunc():
    pass
```

(注意：在 Python 中，函数是一***等对象***，并且 func 表示函数本身，而一旦在函数后面加括号，即一旦写 func()就表示是 func 这个函数的返回结果。这是正确理解 Python 中的闭包，装饰器，协程的一个核心点。)

两种用法都会涉及括号，这是和无参数的装饰器不同的。加了括号表明我们不再是使用 deco_birth_plant 这个函数作为装饰器，而是使用 deco_birth_plant 函数的返回结果(也就是 decorate)作为装饰器，这一点对于含参装饰器的理解尤为重要。

4.3.3 利用装饰器管理函数

装饰器除了可以用于增强函数，还能用于管理函数

如果你要写一系列的函数，并且希望用一个列表管理这些函数，可以写这样的代码来实现：

```
processes = []
```

```
def manage_process(func):
    processes.append(func)
    return func

@ manage_process
def buy_item(item):
    pass

@ manage_process
def sell_item(item):
    pass

@ manage_process
def change_item(item):
    pass
```

该装饰器的闭包函数返回的是函数本身，它使全局变量添加作用函数，通过这种形式要从列表删去列表也比较容易(如果列表很长，删改中间的一个特定函数比较繁琐)。

4.3.4 强制的关键字参数

(本节假定读者已了解 Python 函数的位置参数和关键字参数的区别。)

Python 中有一种语法，你可以在一个函数的参数列表中插入一个单星号，表示这个函数。

在此给出一个作者之前用 Python 开发游戏时的一个此语法的简单例子(备注：Python 语言不适合开发游戏，需要造许多轮子):

```
def get_money(self, num, type= "gold ", *, automerge=True, sec):
"""one unit gets money
:param num: number of money :class:`int`
```

```
    :param type: type of money :class:`str`
    :param automerge: whether it merges automatically in player's item slot
    :param sec: auxiliary parameter, which item slot is working
    """
    ...
```

这里相当于 num 是位置参数, type 是关键字参数，而*后面的是强制关键字参数，那么在调用这个函数的时候就必须明确地指定 automerge 和 sec 参数

以下为合法的调用:

```
get_money(100,automerge=True,sec = 0) #意为获得 100 个金币
get_money(100,'silver',automerge=True,sec = 0) #意为获得 100 个银币
```

以下为不合法的调用:

```
get_money(100,'silver',True,sec = 0) #意为获得 100 个银币
```

4.3.5 私有变量

Python 中，命名一个函数时，可以以双下划线开头，表明这是一个私有函数。

(私有变量表明我们无法在当前类外调用或操作该变量，但是和 C++, Java 语言所不同的是 Python 中的私有变量是伪私有变量，即我们依然可以设法在类外调用一个类的私有函数，Python 中的私有变量实际上是名称改写机制。)

私有变量的一种使用场景如下：

有时候我们可以会写出一个类，其中有大量的函数。

那么我们用将一些函数写为私有函数，告诉别人这个是中间步骤，无须调用，也尽量不要去修改以免出现不可预估的问题。

类似的，名称以单下划线开头的变量也是私有变量，其意义是类似的。Python 中开头位置的单下划线和双下划线具有特别的含义，因此普通的变量应当避免这种命名风格。

单下划线和双下划线的区别是，Python 的解释器不会对单下划线的变量作特殊处理。

4.3.6 冻结参数

如果我们需要多次调用一个函数，且该函数的参数列表很长，那么代码看上去会很繁琐。如果其中的部分参数是相同的，那么我们可以利用 functools 库的 partial 对象冻结其中的部分参数，简化代码量。

以下是一个简单例子：

已有如下代码：

```
from functools import partial

from numpy import pi,e

def get_min(a,b,c,x,y,z):

    return min[a,b,c,x,y,z]
```

要调用五次这个函数，原写法：

```
get_min(a = pi, b = e, c = 5, x = 1, y = 2, z = 3)

get_min(a = pi, b = e, c = 5, x = 2, y = -0.5, z = 1.15)

get_min(a = pi, b = e, c = 5, x = 3, y = -3, z = 2)

get_min(a = pi, b = e, c = 5, x = 4, y = -4.5, z = 3.75)

get_min(a = pi, b = e, c = 5, x = 5, y = -6, z = 0.75)
```

替换写法：

```
ff = partial(get_min, a = pi, b = e, c = 5)

ff(x = 1, y = 2, z = 3)

ff(x = 2, y = -0.5, z = 1.15)

ff(x = 3, y = -3, z = 2)

ff(x = 4, y = -4.5, z = -5.75)

ff(x = 5, y = -6, z = -0.75)
```

4.4 类与对象

4.4.1 利用类的__slots__属性优化

如果我们要写一个自定义类，并且这个类的属性是事先已知的，且固定不变的(不是指属性值固定不变，而是属性的名单是固定不变的)，并且这个类将被用于创造较多

的实例，那么应该考虑使用 __slots__ 这个特殊的类属性来节约内存和提高代码速度。

其原理大致为，Python 默认情况下使用字典来存储对象的属性，但如果指定了 __slots__ 属性，那么就会用元组来存储属性，从而节约了空间并提升效率。所以指定了__slots__属性后，无法给对象增加__slots__里以外的属性(即属性的变量名必须在__slots__容器里，以字符串的形式存在)。

如果你的自定义类没有任何属性，可以写__slots__ = ()。

本书给出一个简单的例子：

```
import timeit
from timeit import Timer
class Testclass_a():
    def __init__(self, name, rank):
        self.name = name
        self.rank = rank

class Testclass_b():
    __slots__ = ['name','rank']
    def __init__(self, name, rank):
        self.name = name
        self.rank = rank

def f1():
    for i in range(1000):
        new_object = Testclass_a('', 0)

def f2():
    for i in range(1000):
        new_object = Testclass_b('', 0)
```

```
if __name__=='__main__':
    t1=Timer("f1()","from __main__ import f1")
    t2=Timer("f2()","from __main__ import f2")

    print (t1.timeit(10000))
    print (t2.timeit(10000))
```

运行结果如下：

```
8.204243599999245
6.892281599999478
```

这个简单例子展示了用 __slots__ 特殊的类属性指定变量后效率更高。

4.4.2 多继承与__mro__属性

__mro__ 是一个特殊的类属性，mro 的全称为 Method Resolution Order，即方法解析顺序。 __mro__ 的属性是一个元组，其中包含了从当前类到最大的 object 类，按方法解析顺序罗列出各个超类。

运行如下代码：

```
class X():pass
class Y():pass
class A(X, Y):pass
class B(Y):pass
class C(A, B):pass
class D(C, X, B):pass

print ("=== C.__mro__ ===")
print (C.__mro__)
print ("=== D.__mro__ ===")
print (D.__mro__)
```

运行结果为：

```
=== C.__mro__ ===

(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.X'>, <class
'__main__.B'>, <class '__main__.Y'>, <class 'object'>)

=== D.__mro__ ===

(<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.X'>, <class '__main__.B'>, <class '__main__.Y'>, <class 'object'>)
```

我们可以从 __mro__ 属性中看到类的继承顺序。

如果当前类继承的多个超类中都有 foobar 这个函数，那么我们当前类调用 foobar 函数时会不稳定。这个问题称为多继承的*二义性*问题。Python 中通过*拓扑排序*算法解决该问题， __mro__ 属性可以为我们展示拓扑排序算法的结果，也就是最终的方法解析顺序。

4.5 代码提速

4.5.1 节约查询方法的时间

我们可以节约从模块中查询方法的时间，字面意思理解之即可。请阅读以下这个简单例子体会它：

```
import math

def f1():

    for i in range(100):

        math.sqrt(i)

def f2():

    a = math.sqrt

    for i in range(100):

        a(i)
```

可以验证 f2 的运行时间约为 f1 的 67%。

5 高级进阶

5.1 非散列表容器

Python 的列表，字典和集合是很强大的容器，如不考虑效率和代码量，它们可以完成绝大多数的工作。

但在某些特定场合，我们应该使用一些针对性的容器提升效率。本节将介绍 Python 中的各种容器和使用场景。

5.1.1 array.array 数组

首先我们通过 from array import array 导入 array 对象。

如果你的列表只用于存储数值，则 array 有较好的表现，并且 array 对象的 frombytes 和 tofile 方法支持更高效的读取和写入操作(可能有显著提升，读取文件很慢时应该考虑)。

array 数组类型不支持 sort()方法，但是支持 sorted()方法，如果需要排序，可以使用以下代码:

```
a = array.array(a. typecode, sorted(a))
```

5.1.2 collections.deque 双向队列

对于一个列表，可以利用 append 和 pop 方法(复杂度为 O(1))实现栈的数据结构，但是要在列表的第一个元素之前添加一个元素或者要删除列表的第一个元素是比较费时的(复杂度为 O(n))。

如果你的数据容器涉及比较频繁的从两端添加删除元素操作，应该考虑使用 collections 库的 deque 对象，它是一种安全和高效的双向队列。

5.1.3 numpy.ndarray 高阶数组

如果你的列表维数很高，则应考虑使用 NumPy 库的 ndarray 对象作为你的数据容器。NumPy 库提供了大量的针对高维数组的方法。

关于 NumPy 库的使用本书不再展开，读者可以参阅 Wes McMinney 的著作《利用 Python 进行数据分析》。

5.1.4 collections.namedtuple

collections 库中的 namedtuple 对象可译为"具名元组"或"可命名元组"等(本小节将采用"具名元组"),其功能和元组类似,但每个元组的值都有名字且以属性的形式储存。

具名元组的使用和本章中介绍的其他容器不太一样,nametuple 本身并不是一个数据结构或容器,而是一个***类型工厂函数***。我们要首先用 nametuple 新建一个生产具名元组类,再用类去生产具名元组。

一个简单示例如下:

```
from collections import namedtuple

card_factory = namedtuple('Singleton', ['rank', 'suit']) #Singleton 译为单张卡牌,这里相当于用 nametuple 类型工厂函数自定义了一个类。

card_a = card_factory(2, "club")

card_b = card_factory(3, "spade")

print("card_a is",card_a)

print("card_b is",card_b)

print("suit of card_a is",card_a.suit)

print("suit of card_b is",card_b.suit)

print("type of card_a is",card_a.__class__) #打印卡牌 a 的变量类型。
```

代码运行结果如下:

```
card_a is singleton(rank=2, suit='club')

card_b is singleton(rank=3, suit='spade')

suit of card_a is club

suit of card_b is spade

type of card_a is <class '__main__.Singleton'> （这里显示了卡牌 a 的变量类型为 Singleton）
```

此例中,我们首先用 nametuple 注册了一个名为"Singleton"的类,然后用 Singleton 实例化了两个对象 card_a, card_b, 其意义为两张卡牌(梅花 2 和黑桃 3)。可以用 ***对象名.属性名*** 的语法获取卡牌的属性。

5.2 散列表容器

5.2.1 collections.defaultdict 默认字典

用 Python 的字典进行增删改查操作时，如果使用了这个字典不存在的 key，那么会抛出异常，我们可以通过 get 方法来避免这个问题，但更方便和更高效的方法是用 collections 库的 defaultdict 对象。

defaultdict 对象拥有 default_factory 这个实例属性，当查找的 key 不存在时，default_factory 会返回默认值，也许我们需要用这个默认值继续工作，总之 defaultdict 对象可以让我们少写许多 get()方法。

5.2.2 collections.OrderedDict 有序字典

一般而言类似于字典这样的散列表是无序的，而 collections 库中的 OrderedDict 容器是一种"有序的"字典。它保留了字典的功能，同时因为有序可以对其作索引操作。

OrderedDict 容器的语法和默认字典的语法很相似。

5.2.3 collections.Counter 计数器

collections 库中的 Counter 对象给我们提供了一种可以用于计数的字典，它拥有字典的所有功能。

新建 Counter 时，可以直接传入一个列表或者字符串作为参数，则 Counter 在建立时会自动对各元素进行计数。

一个简单示例如下：

```
from collections import Counter

c = Counter([1,1,2,3,4,4,4])

print("c =",c)
```

上述代码运行结果如下：

```
c = Counter({4: 3, 1: 2, 2: 1, 3: 1})
```

这个 c 具有字典的全部功能，例如作索引 c[4]可以得到 3。

5.3 常用模块或装饰器

5.3.1 bisect 模块

对有序序列对象操作可以考虑用 bisect 模块，它使用了二分法完成大部分的工作。其中最常用的一个函数为 bisect.insort，其作用为，为序列插入一个值，插入的时候自动地保证该序列是有序的。

一个简单的示例如下：

```
from bisect import insort

data = [5,8,10,11,17]

insort(data,9)

print("data =",data)
```

程序输出结果为：

```
data = [5, 8, 9, 10, 11, 17]
```

另一个比较好用的工具是 bisect. bisect 函数，它用于查询一个值在序列中的位置。注意和 index 略有不同的是，bisect 是从 1 而不是 0 开始计数的。

一个简单的使用示例如下：

```
from bisect import bisect as bs

data = [5,8,10,11,17]

print(bs(data,10))
```

程序输出结果为：

```
3
```

bisect 是使用二分原理进行操作的(复杂度为 O(log(n)))，所以其效率会高于列表的 insert 和 index 操作(复杂度为 O(n))。

(潜约定：在描述时间复杂度中，如无特别说明，log 表示以 2 为底的对数。)

5.3.2 Userdict 模块

我们写自定义类时，如果想要继承 dict 类，用常用的语法写，代码如下：

```
class Foo(dict):

    …
```

不过这样写是有问题的，一般不推荐直接继承 Python 的 dict 类，可能会引起一些不可预估的 bug。collections 模块提供了 Userdict 类，用于替代我们的继承 dict 时的超类。

Userdict 类的功能和 dict 是完全相同的，区别在于，dict 是用 C 语言(CPython)写的，Usedict 用纯 Python 的方式(PyPy)实现了 dict 的效果。

Userdict 类的意义在于让用户继承写子类。

(在 Python2 中 Userdict 不是在 collections 模块下的而是在 Userdict 这个模块下的。)

5.3.3 property 装饰器

Python 的 property 装饰器是 Python 中在面向对象编程时，很实用的一个装饰器。它是 Python 的预设三大装饰器之一(分别为@property, @classmethod, @staticmethod)。

@property 的作用是把一个自定义的 class 类中的函数包装为这个类的属性，其属性名为该函数名。这样我们可以用获取属性的语法(例如用一个英文句号.的语法)来调用该函数及获取其调用结果，可以让使用程序变得方便。

一个简单示例如下：

```
class Apple():
    def __init__(self):
        self.name = ""

    @property
    def size(self):
        return 1.0

this_apple = Apple()
print("size of this apple is",this_apple.size)
```

其代码运行结果如下：

```
size of this apple is 1.0
```

值得注意的是，通过**@property** 定义的属性是只读的，我们可以获取这个属性但不可以改变这个属性。如果想要 **set** 属性，需要再额外定义函数并加上 **属性名.setter** 这样的装饰器作用该函数。一种推荐的写法如下：

```
class Apple():
    def __init__(self):
        self.name = ""
        self._size = 1.0

    @property
    def size(self):
        return 1.0

    @size.setter
    def size(self, real):
        if real < 0:
            raise ValueError("InVaild Value For Apple.size")
        print("apple",self.name,"has been changed to",real)
        self._size = real

this_apple = Apple()
this_apple.name = "001"
print("size of this apple is",this_apple._size)
this_apple.size = 1.1
print("size of this apple is",this_apple._size)
```

其代码运行结果如下：

```
size of this apple is 1.0
apple 001 has been changed to 1.1
size of this apple is 1.1
```

注意这段带有 set 功能的代码的逻辑与上一段只读的纯 property 稍有不同，我们需要用一个不同(于 size)名字的_size 来保存"苹果"的"尺寸"属性。通常这样做是有必要的，其目的为防止在@size.setter 装饰器下的函数中设置 apple 的 size 属性时，再次调用函数本身从而陷入死循环。

5.3.4 classmethod 装饰器和 staticmethod 装饰器

当我们需要调用一个自定义类中的某个方法的时候，需要进行***实例化***(实例化指创建一个该 class 的对象)，再用 **对象名.方法名** 的方式调用该方法。

有些时候我们不需要用这个对象来做其他的事，这种情况下语法就显得臃肿，Python 预设的 classmethod 装饰器用于改善这个问题。

经过@classmethod 装饰的函数表示它是一个"类函数"，不需要实例化即可调用该函数，也不需要再写 self 参数了，但要写一个表示类本身的 cls 参数。"类函数"中我们仍然可以使用这个类的所有属性和方法(注意：如果用该方法对 cls 的属性作修改，也就是对类属性产生影响，从而会对全部该类的对象产生影响)。

经过@staticmethod 装饰的函数表示它是一个"静态函数"，那么这个函数既不需要传入 self 参数或者传入 cls 参数便可以使用。用 **类名.方法名** 或者 **对象名.方法名** 的方式调用该方法。静态函数里边因为没有 cls，就不可以直接访问类属性了(仍然可以通过 **类名.属性** 的方式访问)。

5.3.5 自定义排序

假设我们要对一个数组进行排列，规定偶数总是大于奇数，如果同为偶数或者奇数，则按数值进行判断。如何在 Python 中比较简单地完成？

这里给出一种方法：利用 functools 标准库中的 cmp_to_key 对象，其意义为排序规则，其用法为作为参数传给 sort 函数。

```
data = [4,7,3,11,0,5,-8,3,9]

import functools


def compare_with_number(x, y):
    if x % 2 == 0 and y % 2 == 1:
        return 1
    elif x % 2 == 1 and y % 2 == 0:
        return -1
```

```
    if x > y:
        return 1
    elif x < y:
        return -1
    else:
        return 0

data.sort(key = functools.cmp_to_key(compare_with_number))
print("data =",data)
```

程序输出结果如下：

```
data = [3, 3, 5, 7, 9, 11, -8, 0, 4]
```

这样就完成了按自定义的规则进行排序。下一节介绍了一个常用案例"同时排序"，它是自定义排序的一种更复杂的应用。

5.3.6 同时排序

考虑"同时排序"问题：现有一张 9 行 3 列的二维表，我们想要对第一列进行排序，同时保证每行的内容不变。

我们可以通过"快速排序"等排序方法手动实现 sort 细节，然后对多个列表进行"同步操作"，但显然这样是不 pythonic 的。

我们可以利用 Python 的自定义排序完成这个任务。具体使用过程如下：

```
column_1 = [4,7,3,11,0,5,-8,3,9]
column_2 =
['four','seven','three','eleven','zero','five','negative_eight','three','nine']
column_3 = ['四','七','三','十一','零','五','负八','三','九']

import functools
data = [column_1,column_2,column_3]

def transpose_data(data): # 转置一张二维列表。
```

```
        return [[data[i][j] for i in range(len(data))] for j in range(len(data[0]))]


    def compare_with_number(x, y): # 自定义的排序标准，按每行的第一个元素来判断。
        if x[0] > y[0]:
            return 1
        elif x[0] < y[0]:
            return -1
        else:
            return 0


    data_t = transpose_data(data) # 转置数据
    data_t.sort(key = functools.cmp_to_key(compare_with_number)) # 排序
    data = transpose_data(data_t) # 再次转置数据
    column_1, column_2, column_3 = data
    print("column_1 =",column_1)
    print("column_2 =",column_2)
    print("column_3 =",column_3)
```

程序输出结果如下：

```
column_1 = [-8, 0, 3, 3, 4, 5, 7, 9, 11]
column_2 = ['negative_eight', 'zero', 'three', 'three', 'four', 'five', 'seven', 'nine', 'eleven']
column_3 = ['负八', '零', '三', '三', '四', '五', '七', '九', '十一']
```

这样就实现了对三列的”同时排序”。

5.3.7 functools.lru_cache()装饰器

Python 的 functools 标准库中的 lru_cache 装饰器常称为缓存装饰器。它的作用是对函数增加***备忘(memorization)***功能。

(所谓备忘，指的是将函数的运行结果储存起来，再下次输入相同的参数的时候不再执行函数而是直接返回备忘中的结果。这在递归编程和动态规划中很常用。)

这可以(在额外占用一些空间资源的情况下)增加函数的效率，很适合那些需要经常调用且输入参数可能多次重复的函数(例如计算斐波那契数列等)。如果的输入参数是不容易重复的(例如：用户的两次访问时间的间隔)，则此装饰器会失灵。

functools.lru_cache()装饰器可以传入两个参数，分别是 maxisize 和 typed。

第一个传入整数(根据官方文档，最好传入 2 的幂，如不传入默认为 128)，表示缓存的空间上限，传入 None 则表示无上限(从这里可以看出 Python 中 None 这个对象对于一些函数是有特殊意义的)。

第二个参数传入布尔值，默认为 False，表示不同类型但同值(即 X == Y 判定为真，但 X.__class__ == Y.__class__判定为假)的变量会被视为相同的参数对待。如果设为 True，那么不同类型但同值的变量会被视为不同的参数对待。

5.3.8 functools.singledispatch 装饰器[Python 3.8 功能]

functools 模块中的 singledispatch 装饰器的作用是为了实现类似于 C ++中的函数重载效果。

所谓重载，简言之，就是有多个函数共用一个函数名称，但是函数传入的参数类型不同(在 C ++中传入参数需要预先指定类型和个数，或指明函数模板和泛型)。调用函数时，电脑会自动根据传入的参数的类型和数量来决定调用哪一个函数。

在 Python 中，函数传入的参数是不需要指明类型的。我们有时候可能需要根据传入参数的类型来做相对应的不同的任务。

一个很简单的示例如下：

```
def print_square_value(obj):
    if isinstance(obj, float):
        print(obj ** 2)
    elif isinstance(obj, str):
        print(float(obj) ** 2)
```

这段代码定义了一个函数，其功能很简单，输出一个浮点数的平方值，或者判断如果输入的是字符串，则转为浮点数再输出其平方值(只是展示这个装饰器的简单示例，不再加其他处理)。上述代码可以用 singledispatch 装饰器改写为如下形式：

```
from functools import singledispatch
```

```
@singledispatch
def print_square_value(obj):
    pass


@print_square_value.register(float)
def _(number):
    print(number ** 2)


@print_square_value.register(str)
def _(msg):
    print(float(msg) ** 2)
```

改写之后的代码和之前的代码是等价的，也是实现了 print_square_value 这个函数。乍一看的话，代码变复杂很多，但新的代码的好处在于它是更扁平的，尤其是如果 case 很多的情况下，可以控制各个分函数的代码行数，这种结构比一个行数很多的大函数容易维护且更清晰。

5.4 习惯与通识

5.4.1 用字节码分析程序

dis 模块(CPython 的内置模块)的 dis 对象可以用于追踪一个函数在 CPU 中的运行轨迹，我们通过这个工具分析一个函数或者一些语句的实现。

一个简单示例如下：

```
from dis import dis
dis(lambda x:x+1)
```

程序运行结果为：

```
101           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

这便是 lambda x:x+1 的字节码，阅读字节码可以让我们详细了解代码的工作过程。关于如何理解字节码本书不再展开，读者可以阅读官方文档 https://docs.python.org/zh-cn/3.7/library/dis.html#opcode-STORE_FAST 和讨论资料 https://opensource.com/article/18/4/introduction-python-bytecode。

关于 dis 模块的详细资料读者可以阅读文档 http://docs.python.org/3/library/dis.html。

5.5 类与对象

5.5.1 抽象基类

抽象基类是用于给用户实现*接口*时作为超类使用的类。

(简言之，我们写自定义类的时候可以继承一些抽象基类，以实现一些效果。)

(在 Python 中，*接口*是一种不能实例化的类，用于让自定义类继承以实现一些接口中的方法和*协议*。而*协议*就是指实现了一些魔法方法，例如实现了 __getitem__ 魔法方法和 __len__ 魔法方法就是实现了序列的协议，那么就可以用于 for 循环，如果又实现了 __setitem__ 方法，就可以应用于 random.shuffle。)

在 Python 中我们一般不需要自己定义抽象基类，而且用 collections.abc 模块中提供的一些抽象基类即可（在 number 和 io 模块中也有一些抽象基类）。

(abc 是 abstract class 的简称，即抽象类。)

一个最常用的例子是 collections.abc 模块中的 Sequence 抽象基类，意为序列，继承了 Sequence 的自定义类可以实现类似于 list 的效果。其意义在于我们只需要实现 __getitem__ 和 __len__ 这两个魔法方法，抽象基类就可以自动为我们实现相关的方法，从而允许我们作索引等操作。

本小节将以 Sequence 作为例子展示抽象基类的用法。

```
from collections.abc import Sequence


class Arithmetic_Progression(Sequence): # 实现一个等差数列
    def __init__(self):
        self.name = ''
        self.length = 100
        self.offset = 1 # 首项
```

```
        self.difference = 1 # 公差

    def __getitem__(self,index):
        return self.difference * (index - 1) + self.offset

    def __len__(self):
        return self.length

a = Arithmetic_Progression()
a.offset = 1.5
a.difference = 1.5
print('a[1] =',a[1])
print('a[2] =',a[2])
print('9 是数列 a 的第',a.index(9),'项')
print('67.5 是数列 a 的第',a.index(67.5),'项')
```

运行结果如下：

```
a[1] = 1.5
a[2] = 3.0
9 是数列 a 的第 6 项
67.5 是数列 a 的第 45 项
```

在本例中定义了一个等差数列的序列容器，然后定义了 __getitem__ 和 __len__ 两个魔法方法，如果只是这样的话还不能让序列容器实现 index 方法。

通过继承 Sequence 抽象基类，就可以由 Sequence 的*协议*自动实现 index 方法而不需要写额外的代码。在这里 index(num)就是告诉我们 num 是等差数列中的第几项。

关于 collections.abc 模块的其他使用方法，可以阅读官方资料 https://docs.python.org/3/library/collections.abc.html。

5.5.2 元类

元类是 Python 中深奥的知识，绝大多数情况下我们不需要用到。——Tim Peters。

元类是生产类的类，Python 类元编程中的高级工具，在 Python 中，类是一等对象，这意味着我我们可以不用 class 关键词也可以动态 地创建一些类。

其基本的用法是利用 type 类动态地新建类，通常我们是把 type 视为函数，让它返回一个对象的类型，实际上 type 是一个类，且这个类的实例可以是全新的类。type 的三个参数分别是 name, bases 和 dict。其意义分别是新建类的类名称，新建类的所有基类(即父类)和新建类的属性名和值。

注意用 type()类新建的类是无法*序列化*的(即无法用 pickle 模块对其进行压缩和解压操作)。

元类是一个很大的话题，本书仅对其作了最简单的介绍。感兴趣的读者可以阅读相关资料以深入了解。推荐阅读 *Fluent Python* 一书最后一章。

5.5.3 协程(coroutine)

协程的语法类似于生成器，协程是指在一个函数中，用到 x = yield 这样的语法，则该函数称为协程函数，调用该函数则会生成一个协程。

协程的作用类似于线程，但它是比线程更小的执行单元，在协程之间切换的速度远快于在线程之间切换的速度。协程经常用于函数之间的切换，很适合类似于游戏，异步 I/O 等事件触发型使用场景(触发事件时创建协程)。

协程具有四种工作状态：

- 等待开始执行。'GEN_CREATED'。
- 解释器正在执行。'GEN_RUNNING'。
- 在 yield 表达式处暂停。'GEN_SUSPENDED'。
- 执行结束。'GEN_CLOSED。

可以用 inspect.getgeneratorstate 函数来获取一个协程的当前状态。

假定我们已经有了 test_coro 这个协程函数，使用一个协程的基本流程如下：

(1) 创建一个协程。my_coro = test_coro()。这一步将创建一个 my_coro 协程，且 my_coro 的状态为'GEN_CREATED'。

(2) 预激协程。next(my_coro)。这一步将让 my_coro 协程运行至第一个 yield 处，协程的状态为'GEN_SUSPENDED'。这里的 next(my_coro)有一种等价替换写法为 my_coro.send(None)。

(3) 激活协程。用 my_coro.send()来激活协程，send 函数中可以填各种参数而不仅仅是 None，则 send 函数的参数会赋值给 yield 左边的变量(对于 x = yield 而言就是 x)。如果是 x = yield y 的写法，则协程可以同时返回出 y 的值。也可以用 next(my_coro)激活协程，相当于参数为 None。

(4) 协程终止。如果协程运行到最后一个 yield，则会自动终止。否则可以用 my_coro.close()进行手动关闭。

为了说明这个比较复杂的流程，以下给出一个协程的简单例子：

```
def test_coro(str1):
    print('INNER >>> str1 = ' + str1)
    str2 = yield str1
    print('INNER >>> str1 = ' + str1)
    print('INNER >>> str2 = ' + str2)
    str3 = yield "".join([str1,' ',str2])
    print('INNER >>> str1 = ' + str1)
    print('INNER >>> str2 = ' + str2)
    print('INNER >>> str3 = ' + str3)
    str4 = yield "".join([str1,' ',str2,' ',str3])

def out(): print(msg) #用于输出测试消息

from inspect import getgeneratorstate as gg
my_coro = test_coro('where there')
msg = gg(my_coro);out()
msg = next(my_coro);out()
msg = gg(my_coro);out()
```

```
msg = my_coro.send('is a will');out()
msg = gg(my_coro);out()
msg = my_coro.send('there is a way');out()
msg = gg(my_coro);out()

msg = '=== END ===';out()
```

其输出结果为:

```
GEN_CREATED
INNER >>> str1 = where there
where there
GEN_SUSPENDED
INNER >>> str1 = where there
INNER >>> str2 = is a will
where there is a will
GEN_SUSPENDED
INNER >>> str1 = where there
INNER >>> str2 = is a will
INNER >>> str3 = there is a way
where there is a will there is a way
GEN_SUSPENDED
=== END ===
```

我们可以从此例中理解协程的工作流程。可以注意到此例中协程直至最后都没有成为'GEN_CLOSED'状态，因为协程是在终止之后，才会变为'GEN_CLOSED'状态，如果是自动终止的(对于此例而言，如果再激活一次就会自动终止)，还会抛出 StopIteration 异常。

关于本书

如您有任何提议，可发邮件至 zjc_m@outlook.com 或 1115436971@qq.com 联系作者。

本书仍在更新中。

读者可以从 https://github.com/zjcm/Joyful-Python 获取最新版本。

参考文献

(其中带有黄色高亮的书推荐读者作拓展阅读)

Effective Python – Brett Slatkin

Fluent Python – Luciano Ramalho

Python 高级编程 [中文译本] – Tarek Ziade

利用 Python 进行数据分析 [中文译本] – Wes McKinney

problem solving with algorithms and data structures using python – Bradley N.Miller David L.Ranum

*What the f*ck Python [中文译本] - https://github.com/leisurelicht/wtfpython-cn*

设计模式：可复用面向对象软件的基础 - Erich Gamma 等

Python 游戏编程入门 - Jonathan S.Harbour

Deep Learning with Pytorch – Eli Stevens 等

Flask Web 开发实战：入门、进阶与原理解析 - 李辉

广大互联网社区中的经验分享文章，“Functional Programming HOWTO” https://docs.python.org/3/howto/functional.html ，Python 官方网站资料(尤其是 PEPs) https://www.python.org/dev/peps/ 等