通用 Python 代码设计技巧

v1.04

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 冷知识，特性，代码设计技巧，编程经验，Python 编辑器特性等。许多人在学习完基础编程后不知道如何深入，本书也同时是针对此设计的 Python 进阶学习方向指南。

本书适合看完了 Python 基础教程(基础数据类型，文件操作，异常，类与面向对象编程)**而将要投入实践的程序员**。以了解基础以外还有什么东西可学，并提升代码设计技巧和编程习惯。

2.2 前言

为什么写这本书？

一些读者也许会有类似的经历，学习完 Python 的基础语法知识可能并不需要太久的时间，已有编程经验的程序员也许只要两三天，初学者花费数周也可以学习完。但是作者本人在学习完 Python 基础知识之后，却出现了一些更大的困难，主要体现在以下几个方面：

(1) 不知道如何优化自己写好的代码。

(2) 不知道如何应用已经学会的 Python 知识，可以应用在哪些领域。

(3) 担心在实现项目时，知识广度不够，以致于在不必要的地方过于钻牛角尖，或者重复造轮子。

(4) 担心在实现项目时，知识深度不够，以致于写出的代码质量较低，不够专业。

(5) 仍然看不懂一些优秀项目的源码，或者使用他人代码时无法看懂其实现过程，难以独自解决异常。

作者在此困境下，坚持不懈，进行了漫长的实践和学习，最终收获了一些经验心得，期间也走了不少弯路。作者在学习过程中，一开始记录了一些笔记和与友人的交流探讨，并分享给其他学习者，而后又实践了许多 Python 项目，作者在这一过程中逐渐将笔记体系化。希望其他 Python 爱好者能够从本书中获得裨益。

相信读完本书后，再结合一些项目实践，读者可以加深 Python 功底，自如地应对上述困难，全面提升 Python 编程能力，并写出优质的 Python 程序，也可应对 Python 问题的面试。另外，本书在各章节提供了大量优质参考资料的链接，以便读者深入学习。

本书最后一章提供了一些 Python 应用实例，以及一些优秀 Python 项目的链接，以提供读者 Python 的应用方向，相信读者在参阅后能有一定感悟，并能写出自己独特的优质应用。

2.3 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，因此作者会大量使用“简言之”一词，作出某些概念的通俗解释。在一些场合中，这也许会少量地丢失解释的严谨性或丰富性。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库之外，作者不会专门去介绍一些第三方库的具体用法。值得注意的是，第三方库的应用性强，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库的熟悉和使用(例如 tensorflow，sklearn 等不胜枚举)。第三方库具极重要的地位，如若涉及，请读者查阅相关官方文档或者其他工具书。

第三方库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，但作者建议”基本知识”应该和第三方库交叉性地进行学习。

2.4 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。故本文内容主要集中在基础，初级进阶，高级进阶三章，这是为了方便读者按顺序进行阅读，由浅入深。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 10 月完成此手册，使用的 Python 版本为 3.8.3，集成开发环境为 spyder4.0。)

读者可以运行：

```
import platform
print(platform.python_version())
```

以查看自己 Python 版本号。

因各种原因本书内容可能有误，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(https://legacy.Python.org/dev/PEPs/PEP-0008/)。

下面列举个别重要条目:

- 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)
- 为变量赋值时，等号两边要写一个空格。例：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，第三方库模块以及自用库模块。每一部分中应以模块字母顺序排列。
- 函数中给形参指定默认值时，等号两边不要有空格。例：写 def my_func(param0, param1='value')而不要写 def my_func(param0, param1 = 'value')。同样的在调用函数时，关键词实参的等号两边也不要有空格。例：写 my_func(param0, param1='value')而不要写 my_func(param0, param1 = 'value')。

……

注意 PEP8 提供了一种代码设计风格的参考，不必完全拘泥于 PEP8 等风格模式，总会有风格模式反而不适用你的程序，或者模棱两可的时候，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。但注意自动的 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约定俗成的风格。例如 import pandas as pd 其中的 pd 一般是约定的名称。

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，但其中不乏晦涩难读的文章，建议读者在有一定 Python 的实际编程经验后再去阅读。

3.1.2 注释的选择

长期工程要多写注释，短期工程或临时工程不必写太多的注释。

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。

基本的习惯是，对于绝大多数由 def 定义的函数至少写一行注释，说明一下函数的作用。对于逻辑过于复杂的代码块，应写一些注释，说明清楚理解难点以及该代码块的输入输出格式。对于存在多个解决方案的情形，可以写一些注释说明为什么选择此种解决方案。

3.1.3 多种注释或提示符

Python 中有很多种可用的注释，以下罗列 Python 中的各种注释及其使用方法，读者可以在各种不同情形下使用不同的注释：

- 单行注释：

这是 Python 中最常用、基本的一种注释。语法如下：
#comment。

- 多行注释：

Python 中的多行注释的方法是用多行字符串存储一段注释内容。语法如下：

"""comment"""或者'''comment'''，其中 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，常见于各种计算机高级语言。简言之，#TODO 表示待做工作，# FIXME 表示可能有问题的内容。在一些编辑器由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]分块注释符号：

分块注释符号是#%%。在 spyder 编辑器中，用"#%%"可实现代码分块，从而可以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"说明文字"。

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。这样可以很方便地启用或者停用这个代码块，把一个 1 改为 0 比删除添加一对"""符号容易得多。

另外有时这种注释还能起到独特的效果，尤其是如果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，而用多个"""号则可能引起多行注释符号逻辑冲突，但是用此法则可以很方便的实现。

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
    ...
```

(说明：设计代码时 flag 一词通常表示一种标志，用于后续判断。)

上述例子中，程序员可以通过修改 test_flags 列表中的 0，1 之来决定后续三个代码块是否执行。

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

[Python 3.5 功能]

Python 的类型提示功能常见于函数的参数列表，它用于说明一个函数各参数所要求的参数类型，以及返回值的类型，但并不是强制的，它只起提示作用。

在较大型的代码结构中，这种类型提示是很有用的，因为 Python 是一个十分灵活的语言，对数据类型要求不严格，这导致了许多时候我们无法通过阅读代码直接了解到某个函数需要的参数的数据类型是什么。此时函数编写者可以使用类型提示告诉代码使用者所期望的参数数据类型，从而减少或者避免后续代码运行种的异常情形。

语法例子：

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

事实上 Python 中有很丰富的类型提示功能，请详见 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)。(事实上很少人使用 Ellipsis 对象；你可以选择在代码中用…来代替 pass；另外你可以用 Ellipsis 对象作为某种特殊的数据，用于标记变量的状态。)

还有另一种情形也会用到省略号，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。

第三种情形则比较少见。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] = {}, ""
#生成一个无穷维的列表
a = [1]
a[0] = a
```

打印该容器，可见其输出形式带有省略号，表明这是一个无穷维的容器，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。

- 一般类型提示(又名变量标注 Variable Annotation 详见 PEP526)：

[pycharm 功能][Python3.6 新增]

pycharm 具有强大的联想功能(亦称自动补全功能)，但在一些场合中(例如变量经过了赋值)会无法判定一个变量的类别，那么 IDE 就不再能对这个变量作自动补全。

我们可以用手动指定的方式告诉 IDE 变量的类别，让 IDE 可以静态分析代码。其语法为 object_name : class_name，这相当于告诉 IDE object_name 的是 class_name 这个类的对象。

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。

当然不一定必须用”=”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。

总之在任何情况下，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、节约内存空间并减少代码量。

例如，当我们仅仅需要使用 **math** 模块的 **sin** 函数喝 **pi** 对象时，我们应写：

```
from math import sin

from math import pi

print(sin(pi / 2))
```

而不应写:

```
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。注意：如果你要使用后者方式导入所有对象，应该注释防范多个对象重名的风险，以免出现意外或者异常。

另外补充说明：实际上 from class_name import *并不是真的导入该模块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

编写程序时，有时候会出现一连串相似的全局变量，或者具有相近意义的全局变量，我们应该有意识地用列表来管理它们，以增加**可维护性**以及减少全局变量个数，一份代码中一般最好不要出现过多的全局变量。

(说明：可维护性简言之就是指程序员以后来修改这段代码时的时候比较方便，这种修改不一定是修复错误，也可能是扩展功能等。)

推荐写法示例：

```
suitspic = []

suitspic.append('s1.png')

suitspic.append('s2.png')

suitspic.append('s3.png')

suitspic.append('s4.png')
```

不推荐写法示例：

```
suitspic1 = 's1.png'

suitspic2 = 's2.png'

suitspic3 = 's3.png'

suitspic4 = 's4.png'
```

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。(简要说明：将变量结构化，层次化，找出各层框架，将框架本体作为类，框架的子项内容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库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或多人协作开发，应考虑使用 git 等版本控制工具进行维护。关于 git 的使用本书不详述，读者可以查阅相关资料。

3.1.8 增强剪切板

我们可以增强剪切板的功能以提高编程效率。具体来说，通常粘贴操作仅能粘贴剪切板中最晚的一项内容，经增强后，我们可以使用剪切板中的历史内容。

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，适合攥写文本或者编写代码一类的工作。

3.1.9 书签与长代码管理

[Pycharm 功能]

当写超长代码(作者经验结论：一个超过 5000 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。相比于 ctrl+F 搜索关键词的形式查找代码，它还支持跨文件跳转，对于超长代码或者多文件项目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或者符号)，然后 ctrl+F 搜索注释。此种方法是优点是简洁，直观。缺点是多数编辑器不直接支持跨文件搜索，更重要的缺点是程序员可能之后忘了之前使用了何种特殊注释，也不容易给其他人使用。

(2)：在代码头部(或其他地方)写一个所有 def foobar()(即：所有的显式函数)的汇总文档，并记住常用的 def foobar()，然后直接搜索函数名。程序员可以通过阅读汇总文档来复习函数名称和作用。

(说明：在代码中，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，理清代码的层次结构吗，将不同功能的代码封装至不同文件中，降低每份代码的长度。事实上，在实现中型以上的项目时，这是必须做的一件事，程序员应提前考虑这个问题，早做规划。

3.1.10 项目组织与文件命名规范

对于较大的项目而言，将所有文件放在同一个文件夹中显然会过于杂乱。此时需要将各类功能的文件分门别类地进行整理，本节介绍给文件夹命名的通识，方便组织项目文件以及理解其他人的项目结构。

常用的文件夹命名规范：

- resources
  存放数据型文件，例如 csv 文件等。不要在根目录直接存放此类资源文件。
- assets
  存放多媒体文件，例如 mp3, wav 文件等。在次级目录可进一步进行分类，例如用 images 文件夹来存放图片文件。
- src
  source 的缩写，存放源码文件。
- bin
  binary 的缩写，存放二进制文件。这里可以存放 Python 模块，可执行文件等。
- utils/tools
  存放工具型代码，简单理解为辅助函数，例如某种数据、格式的转换。

(提示：对于中大型项目，我们应该尽可能解耦，辅助函数也最好从主要代码中分离。)

- locale
  意为本地化，常用于国际化软件，存放语言配置文件或语言数据文件。用于翻译软件内语言(有时可能还要涉及界面排版等)。本地化有 i18n 和 L10n 两种形式。

- logs
  存放日志文件。

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，或者需要将代码排版成更合理的形式，程序员可以考虑利用这个特性来压缩行数。

上述代码中，其中第(1)(2)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(3)句表明，Python 中可以用分号分隔两个简单语句，并且分句可以同时写在同一行上；第(4)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。

例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`

交换两个变量的值 `a,b = b,a`

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))
```

运行结果如下：

```
1.6667

1.6666999999999998

False
```

简单分析代码即可看出，Python 的 float 存在一些精度问题。

解决方案：

(1)借助 Python 的 decimal 模块存储和操作小数，本书不再详细展开。

(2)判断相等时，不用 == 判断而是基于差的绝对值是否小于一个较小的数判断，例如使用 abs(1.6-1.7)<=1e-5。还可以直接利用 math 模块的 isclose 函数来直接实现这样的相等判断，它可以避免浮点数精度带来的误差。

另外补充一些其他关于 Python 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。

出于测试的目的我们运行如下代码：

(说明：timeit 是 Python 中用于测试小段代码运行时间的一个模块，使用简单。除此外我们还可以使用[IPython 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，则可以用 clockdeco 库的 clock 装饰器分步输出时间和调用过程。它们都是使用多次模拟运行的方法，基于大数定律的原理来估计运行时间。)

(提示：一般情况下，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 等类似的名称，因为阅读者无法直接了解变量意义。我们应该取名为有意义的名字。但本书的演示例子中，出于教学目的，变量名取名以简单为主，实际使用中应该有意识地取有意义的变量名字。)

(提示：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 后跟多项相连的 or 判断，也应考虑用 in 代替，可以使程序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
```

分析运行结果，容易发现用 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 是 Python 中一种强大的容器，其原理是哈希表，我们最关心的则是 set 知名的特点：元素互异。此特点可以辅助我们完成许多工作。

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。(除了上述用&、|、-符号完成集合运算，也可以用 union, intersection, difference 方法完成集合运算。)

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

print(target_bool)
```

运行结果如下：

```
True
```

结果表明列表 A 和列表 B 中的确含有两个相同的元素

另一例子：

对列表 A 进行去重。

```
A = [1,2,3,4,3,4,5,6]

A = list(set(A))

print(A)
```

运行结果如下：

```
[1, 2, 3, 4, 5, 6]
```

分析结果可以发现，尽管集合是无序的，此种去重方式实际上也是逐个元素按序进行判断的，对于保留原始列表的顺序具有一定的稳定性。

3.2.5 del 关键字

用 del 关键词可以删去一些变量，从而节约内存。

(补充：del 实际上是调用了对象的__del__魔法方法，__del__魔法方法类似于 C，C++中的析构函数。注意不要随意地重写__del__魔法方法，例如在其中新建元素或者没有正确删除对象等，会引起内存泄漏。)

简单例子：

```
import numpy as np

ID = 1

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

del ID
```

我们运行此段程序，目的是为了构造 data 数据，拿到了 data 数据后，不再需要之前的中间变量了，可以及时地进行删除。上述代码中的最后四行可以简写为 del n, x, y, ID。通过这一系列操作，就回收了这四个变量，这是一种程序员手动回收的方式，实现了很简单的内存管理和垃圾回收。在&4.2.5 节中进一步介绍了 Python 的垃圾回收机制。

3.2.6 变量内涵

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"(有 C++基础的读者也可以理解为"指针")，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。

这虽然是 Python 中的一项底层原理，但作者认为非常重要，因此也写在了基础章节，是否理解此内容很大地决定了 Python 程序员处理许多疑难问题的能力。

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。

(补充：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。)

(补充：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。)

(补充: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。)

(思考：请读者思考一个问题，在 Python 中，元组是不可变的，列表是可变的。那么我们能否修改一个元组中的列表，能否修改一个列表中的元组。提示：利用上一个补充的结论。)

在 Python 中，必须注意以下事实：

(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。如果不想传入一个列表，在函数中的操作会导致列表永久改变，传参时可以传入列表的浅拷贝 list[:]。

(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，容易产生意想不到的结果。

3.2.7 变量内涵深入理解：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。

(注意：浅拷贝和深拷贝需要首先导入 copy 模块，这里我们只导入 copy 模块下的 copy 函数和 deepcopy 函数。)

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，则在实际编程时可能会陷入"赋值陷阱"并产生很大的疑惑。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。这一块有一定难度，但作者认为非常重要，因此写在了基础章节。

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 的子对象的集合。

(提示：根据上一节"变量的内涵"中的说法，变量是对象的便利贴。这里 a 就是 obja 的便利贴。)

(补充：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 obja 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 obja 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 sobja 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]

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

分析此例中的代码，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
******************************
```

分析代码和打印标识结果(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 二维数组

上一节介绍了三种拷贝，它有一个重要应用案例：Python 二维数组的实现。二维数组常见且重要，Python 程序员应该学会如何正确地创建一个二维数组。

假设我们要创建一个 4 行 6 列，元素全为 None 的二维数组。使用二维列表来实现二维数组。

推荐的写法如下：

```
mylist = [[None for i in range(4)] for j in range(6)]
```

我们运行 print(mylist)打印这个二维列表，

运行结果为:

```
[[None, None, None, None],
 [None, None, None, None],
[None, None, None, None],
```

[None, None, None, None],

[None, None, None, None],

[None, None, None, None]]

初学者容易犯的错误是这样构造二维列表：

mylist2 = [[None] * 4] * 6

我们运行 print(mylist2)打印这个二维列表，

运行结果为:

[[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, None]]

分析运行结果，可以发现两段代码的运行结果是完全一致的，但是实际上第二种二维列表的实现是错误的。如果不理解三种拷贝的原理，很容易被迷惑。

我们将分别将两个列表的最后一个元素修改为空字符串：

mylist[-1][-1] = ""

mylist2[-1][-1] = ""

然后打印两份二维列表 print("mylist =",mylist),print("mylist2 =",mylist2)，观察运行结果：

mylist =

[[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, None],

[None, None, None, '']]

mylist2 =

[[None, None, None, ''],

[None, None, None, ''],

[None, None, None, ''],

[None, None, None, ''],

[None, None, None, ''],

[None, None, None, '']]

上述代码的目标是使得最后一个元素变为空字符串，分析运行结果可以发现第一个二维列表正确实现了预想的效果。但是第二个二维列表没有正确实现预想的效果，它将每一个子列表的最后元素变为了空字符串。

之所以会出现此种意外，是因为第二种实现形式中的[[None] * 4] * 6 的语法是对[None] * 4 进行五次浅拷贝，然后将五个副本和原来的列表存储在主列表中。此种语法构造的二维列表，看起来有 6 个子列表，实际上是 1 份列表的多份浅拷贝。

浅拷贝的特性就是其中一份副本子对象改变后，其他副本的子对象会进行同时改变。作为对比，在第一种实现形式中是用列表推导式实现的复制，这种复制其实是"依次创建"内容相同的列表，相当于深拷贝，不会出现第二种实现形式中的浅拷贝带来的意外。

上述推荐写法 mylist = [[None for i in range(4)] for j in range(6)]还有另一种等价的、简化的实现形式，可以写为 mylist = [[None] * 4 for j in range(6)]。并且此种实现方式也是完全标准的，不会出现浅拷贝带来的意外。

为加深理解，读者可以自己思考一下为什么上述的实现方式虽然也用了[] * n 的形式，作了浅拷贝，但是它是没有问题的。(提示：从三种实现形式的细微差别入手，结合浅拷贝的内涵进行思考。)

3.2.9 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。)

(补充：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
```

简言之，在变量名前加星号，则其可以指代不确定数量的变量，arg 变量名可以换为其他名称。

3.2.10 修改元组中的对象

有 Python 基础知识的用户应该熟悉：元组是不可变的对象。本书之前的章节中已经讲过，元组的不可变性是指元组中的对象的标识不可变，而不是内容不可变。

某些情况下，会出现奇怪的事情。

运行如下代码：

```
my_tuple = (["a"],["b"])

print(id(my_tuple[0]) % 10000) #  查看元组中第一个元素地址的后四位
```

打印结果为：

```
7040
```

运行：

```
my_tuple[0] += "x"
```

程序抛出了异常：

```
TypeError: 'tuple' object does not support item assignment
```

运行：

```
print(my_tuple)

print(id(my_tuple[0]) % 10000)
```

打印结果为：

```
(['a', 'x'], ['b'])
7040
```

可以注意到，虽然程序抛出了异常，但列表内容依然改变了。这是因为对于列表而言 += 运算符相当于先调用 extend 方法，然后进行赋值。

调用 extend 方法不会影响列表的标识，可以用它在元组中改变列表内容。赋值会尝试改变列表的标识，这是不被元组允许的，所以程序会抛出异常，但此时列表的内容已经改变。

3.2.11 了解 is 和==的区别

is 和==的区别主要有两个方面：

(1)：

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。

(补充：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，表示变量的内存地址。)

而 == 比较两边是否“相等”，是值相等的概念，不同的对象值相等的意义也不尽不同的。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。

(补充：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 &3.4.1 节。)

(2)：

判断一个变量是不是 None 对象，推荐用 is 而不要用 == 。这种情况下 is 的效率高于 == 。在一段 Python 程序中，全局只有一个 None。

举一例说明：

```
N1 = None
N2 = None
print("N1 == N2 ? {}".format(N1 == N2))
print("N1 is N2 ? {}".format(N1 is N2))
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 is N2 ? 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
```

分析结果，既可以用==也可以用 is 来判断一个对象是否为 None。两个列表所包含的内容相同时，用==判断结果为 True，但用 is 判断结果为 False。这表明它们是两个地址不同，内容相同的列表。另外注意到 L1 is L3 判定为 True，表明赋值拷贝的变量只是原来变量的*引用*。

注意到两个"I see sunshine"字符串用 is 相比较时，判定为真，但两个"I see sunshine!"字符串用 is 相比较时，判定为假。下文将给出此种奇怪的现象的解释。

补充解释：涉及字符串(或者一些其他内置对象)的比较时，会涉及一个 Python 的***驻留(interning)***问题，这是一个比较深刻的问题，读者只需要了解其概念。

字符串***驻留***指 Cpython 在编译时，为优化资源，某些情况下不新建新的不可变对象实体，而是会使用已经存在的对象的行为。有些字符串会被驻留，但是加上感叹号后，编译系统认为这是一个罕见的字符串，并不驻留，此时两个相同内容的字符串才会存储在不同的内存地址。

Python 关于整数的驻留的规则是：-5 到 256 的整数会发生驻留现象，257 及以上的数据不会发生驻留现象。

关于具体哪些字符串会发生驻留，这是一个复杂的实现细节问题，没有官方文档，本书不再对此详细介绍，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。

3.2.12 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，也很常用。

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。

(补充：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，或者更为直接地，检查这个对象是否有 __call__() 魔法方法)。

要充分理解生成器的原理，需要掌握迭代器的知识。

生成器是一种实现了惰性迭代的迭代器。

(补充：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。详见本书&5.6.2 节内容。)

(补充：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。

(补充：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"。)

接下来给出一个生成器的简单案例：

```
def simple_generator():
    num,count = 1,0
    while count <= 10:
        num *= 2
        count += 1
        yield num
    return num
gen = simple_generator()
for i in gen:
    print("i =",i)
```

程序输出结果：

```
i = 2
i = 4
i = 8
i = 16
i = 32
i = 64
i = 128
i = 256
i = 512
i = 1024
i = 2048
```

itertools 库中提供了一些生成器函数，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, foo[], len(foo)等相应语法或者相应函数时由 Python 解释器自动调用的方法。魔法方法的名称以双下划线开头，双下划线结尾，不同的魔法方法名称具有不同功能，其名称是固定的。程序员可以通过编辑魔法方法定制类的功能，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。

(提示：Python 中一切皆为对象，类也是对象，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，可以让代码更加精简，或者实现 func 的复用，也可以在程序运行时动态地修改类的特性。)

这里举几个重要魔法方法的应用案例，运行以下代码：

```
simple_box = list(range(10))

print(simple_box)

print("__class__ of simple box is",simple_box.__class__)

#注意这里的细微差别：这一行是__class__()，不同于上一行的__class__

print("__class__() of simple box is",simple_box.__class__())

print("__str__ of simple box is",simple_box.__str__())

print("__repr__ of simple box is",simple_box.__repr__())
```

运行结果如下：

```
#直接 print 打印的结果：

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

#__class__属性是：

class of simple box is <class 'list'>

#__class__()魔法方法返回的是：

__class__ of simple box is []
```

#__str__()魔法方法返回的是：

__str__ of simple box is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

#__repr__ ()魔法方法返回的是：

__repr__ of simple box is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

分析代码和运行结果：

__class__用于返回对象的类对象(Python 中类也是一种对象)，__class__不是魔法方法，是对象的属性，simple_box 是一个列表对象，它的__class__属性就是 list 这样的类对象。

由于类对象是一种***可调用对象***，它有__call__魔法方法，__class__()实际上是__class__.__call__()的语法糖，而一些类对象的无参数调用就是创建一个预设的对象(有些类对象必须有参数才能创建对象)。

运行：[].__class__.__call__()

输出结果：[]

继续分析上述案例，我们发现 simple_box.__str__()的返回结果和 simple_box.__repr__()的返回结果在此例中是一致的，这两种魔法方法都是返回对象的"内容消息"。它们有一个区别，__repr__是在调用 print()函数时自动使用的，用于打印对象的内容。__str__则是把对象直接输入在 IPython 的 console 中运行时自动使用的，用于打印对象的调试性消息。

我们构建自定义类时，修改这些魔法方法就可以实现如何展现、以何种格式展现、展现哪些自定义的类的信息了，还可以加入自定义逻辑，这展现了 Python 极其灵活的特点。

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。这也许会花费不短的时间，但是魔法方法的应用非常广泛，所以学习常用魔法方法依然是值得的。

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。但运算符重载也需谨慎使用，滥用运算符重载也可能会遇到不可预估的缺陷。

3.4.2 dir 函数

预设函数 dir()可以用于打印出某个实体的所有属性和方法，常用于查询。

此函数十分有用，在使用他人的代码或者第三方库时，如果不清楚某个对象有哪些方法可以使用，就可以用 dir 函数来打印其所有属性和方法，十分快捷。

这是一个案例(依赖 sklearn 库)，运行如下代码：

```
from sklearn.decomposition    import PCA

dir(PCA)
```

运行结果为：

```
['__abstractmethods__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
```

'__reduce__',

'__reduce_ex__',

'__repr__',

'__setattr__',

'__setstate__',

'__sizeof__',

'__str__',

'__subclasshook__',

'__weakref__',

'_abc_impl',

'_check_n_features',

'_fit',

'_fit_full',

'_fit_truncated',

'_get_param_names',

'_get_tags',

'_more_tags',

'_repr_html_',

'_repr_html_inner',

'_repr_mimebundle_',

'_validate_data',

'fit',

'fit_transform',

'get_covariance',

'get_params',

'get_precision',

'inverse_transform',

```
 'score',
 'score_samples',
 'set_params',
 'transform']
```

这样我们就知道了 PCA 类有哪些属性和方法可以使用，其中包括许多双下划线开头和结尾的魔法方法，程序员最好能够提前知道这些魔法方法的功能。除了魔法方法以外的名称一般不会设置简写，可以通过其名称大概猜出其大致功能。如果要进一步了解某个方法，例如 fit 方法的资料，可以打印该函数的__doc__属性，运行如下的代码：

```
print(PCA.fit.__doc__)
```

程序运行结果为：

```
Fit the model with X.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            Training data, where n_samples is the number of samples
            and n_features is the number of features.

        y : Ignored

        Returns
        -------
        self : object
            Returns the instance itself.
```

这样就打印了 PCA 的 fit 方法的文档。

3.5 代码提速

程序员写出高效代码，主要需要三类知识。第一类是程序逻辑的优化，主要依靠程序员的数据结构功底，决定了程序效率，是很重要的部分。第二类是对项目系统，结构的设计，这主要针对大型项目，依赖于程序员提前规划。第三类是具体语言的细节优化。本节主要介绍第三类知识。

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
```

不过实际中这个差别一般区别是不大的，主要花费的时间是在于业务逻辑上。但它提示了我们另一件事，Python 中很多基于布尔变量的操作，可以用整型变量代替完成。0 和 1 的拼写最简单，代码量更低一点，这是一种偷懒的做法。

另外读者需要了解 isinstance(True, 1)会返回真，以免用到时被程序迷惑。

如果读者如果想进一步了解 bool 是 int 的子类这个问题的底层原理，请参考问题：https://stackoverflow.com/questions/8169001/why-is-bool-a-subclass-of-int/8169049#8169049。

3.5.3 一些功能等效写法

Python 之禅中有一段话：

There should be one-- and preferably only one --obvious way to do it.

同样一个问题最好仅有一种解决方案。

读者也许在实际编程过程中遇到要实现一个简单的效果，但可以由多种写法的情形，也许不知道如何选择，选择哪一种更好。本小节将罗列部分常见功能的等效写法，给予简单分析，帮助程序员写出更好的代码。

- 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，你可以考虑用元组 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(此项差异较小)。
- 字符串的 join 函数，用于拼接字符串时效率很高。例如用" ".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"今晚的" + "月色" + "真美"或者格式化字符串等方式更高。但有些情况下使用加号的形式代码可能更加清晰。
- 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。)
- 我们可以用 str(L)来将一个列表转为其对应的字符串。
- 将多项式进行因式分解或提取公因式以减少计算量。数值分析、计算机组成原理等学科中有一个经典计算案例：

  $$ax^2 + bx + c = (ax + b)x + c$$

  等号左边的算式可以转为等号右边的算式，从而减少了计算量，这就是著名的秦九韶算法。

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

如果每次局部测试都很耗费时间，这会严重损伤程序员的开发效率和意愿。

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。前面两种情况中代码的耦合性天然地比较低，所以相对好处理。第三种情况就有点严重，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。如果没有做好分离，每次测试业务逻辑都要弹出窗口，也许还需要进行一些输入操作，对于测试而言展开一个窗口是过于费时的。

(说明：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，可以考虑使用 Python 的自动化测试框架，以实现对代码持续全面的测试，例如 pytest 库、Unittest 库、Robot Framework 库。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)。测试框架基本功能的学习并不困难，读者不需要花费很多精力就可以学会。

Unittest 库是 Python 标准库中自带的单元测试框架，pytest 是 Python 的另一个第三方单元测试库，用起来比 Unittest 更加方便，适合初学者学习。

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事物的集合。

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”。类不仅仅是天然的一些物体集合这样一个客观的概念，更应该是程序员所要操作或者研究的一组对象，是一个主观的概念。

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可以视为一个用"类"打包的整体。

举这个例子是为了展现这种打包的思想，另外一个使用案例是，假设你的脚本有一些数据需要存储到本地(或上传等)，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，例如直接 pickle 这个类就可以实现数据压缩和解压。

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。注意，这里所说的杂乱一方面体现在格式不整齐，这个可以通过代码换行解决，例如曲奇饼的数据可以写为：

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

这样就基本解决了格式不整齐的问题。但是杂乱更重要的体现在第二个方面，即数据太多，有些数据具有一定的重复性和冗余性，数据没有明确重点，读者不知道该关注哪一部分。

我们应该构造一种树形数据结构存储此类数据，用类的继承的思想来解决这类问题是很合适的。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(本小节只是给出一个简单示例，实际可能还有各种各样的食物类别)。再定义 Fruit 类继承 Natural_Food 类，再去定义 apple 苹果

类(最细颗粒度的数据也要用类维护，这样即使后续要增加细分的苹果类型，也容易维护)。

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' # 物品名称，此处的属性是抽象的


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。如果我们要添加甜品颗粒度的食物类别，只需要关注甜品和炖菜的区别，不用关注天然食品和加工食品之间的差别。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)。

这里再补充一个知识点，python 实际上在 3.7 版本新增了一个 dataclasses 类，它更适合用于作为存储数据的类。这里不再展开介绍，有兴趣的读者可以阅读官方文档(https://docs.python.org/3/library/dataclasses.html)。

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！')
```

动态打印常用于打印进度条，但程序员也可以用它实现各种自定义效果，例如用于展现数学模型实现的中间过程。

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。

动态导入的一种使用场景为：Python 解释器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。尤其是例如 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，只保留第一次的导入结果。

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，它可以绕开是否已有这个模块的检查，保证一定导入。动态导入也可以用于在程序运行时实时地改变一些模块，根据 PEP302，这是很不推荐的一种做法。

我们用一个变量来接收动态导入的模组。

其语法的使用示例如下：

```
math = __import__( "math ")

print(math.pi)

np = __import__( "numpy ")

print(np.pi)
```

4.2.3 数值中的下划线

[Python3.6 新增]

我们可以用下划线增强数值的可读性，运行如下的代码：

```
value = 5_000_000_000

print(value)
```

运行结果如下：

```
5000000000
```

4.2.4 海象表达式

[Python3.8 新增]

海象表达式英文名 **Assignment Expressions**，意为赋值表达式，应该算作一种语法糖，作用是让代码简洁。(关于海象表达式的提出详见 **PEP 572**)

其符号为 :=，意义很简单，即把右边的值赋值给左边的变量，那么和赋值符号 == 有什么区别呢？区别在于用 := 赋值的变量可以立即使用(在当前语句中被 **python** 解释器解释并当作变量处理)。

使用例子：

```
data = [1] * 8

if (datasize := len(data)) >= 5:

    print(f"数据量为{datasize}.")
```

输出结果：

```
数据量为 8.
```

4.2.5 垃圾回收机制

Python 解释器具有自动回收的功能。Python 相比于其他高级语言，例如 C++，Java 等，它的回收功能已经经过了很大的简化和自动化，使程序员可以在大多数情况下不必关心变量的回收内存管理。

在某些情况下，我们可能仍然需要理解其运作方式，因为尤其是在类似于循环引用的情况中，预设的垃圾回收机制可能导致*内存泄漏*。

(说明：内存泄漏可以简单理解为废弃的变量在内存中没有删除干净，且无法被垃圾回收机制检测到，导致占用内存越来越多，可能会导致程序崩溃或者使用久了越来越卡。)

(提示：Python 垃圾回收机制是一个较大的话题，本书没有足够的篇幅介绍其全貌，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 = []`，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，我们可以用 sys 库的 sys.getrefcount(obj)函数来查看某个对象的引用计数，一个对象的引用计数为 0 时才能被自动回收。循环引用中的对象的引用计数不会变为 0。

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。一个简单的应用案例如下：

```
import gc

g_num = 3

gc.collect(g_num)
```

这样就执行了 3 代*分代回收*机制的垃圾回收，gc.collect 默认采用 2 代的回收机制。

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。

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，语法为分若干行写若干个装饰器，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))。多个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。

(注意：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。)

python 中除了有函数装饰器，还有类装饰器，其语法和作用形式是类似的，这里不再赘述。

4.3.2 含参装饰器

含参装饰器指这样的一类装饰器，使用此类装饰器时，还可以传入自定义的参数，改变装饰器的部分功能。

实现含参装饰器，通常是需要一个装饰器工厂函数(即一个返回装饰器的函数)，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(在本例中就是 decorate 函数)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 计时装饰器

计时装饰器是装饰器的一个经典案例，如何写出一个装饰器，函数经过该装饰器装饰后，便具有了自动计算函数运行时间的功能(这是实时运行时间，不同于 timeit 的多次模拟估算运行时间)。只要调用经过此装饰器装饰的函数，程序就可以自动输出函数的运行时间等信息。

读者可以先思考并尝试自己实现，然后阅读和运行以下代码，对比运行结果。

```
import time
import functools
# time_measuring 就是要实现的装饰器，它接收一个函数。func 就是原函数。
def time_measuring(func):
    #用 functools.wraps 装饰内部函数以避免 time_measuring 装饰器修改函数的名称和文档。
    @functools.wraps(func)
    #装饰器是通用的，所以内部函数接受任意类型的参数，所以用两个可变参数，分别表示所有的位置函数和关键词参数。
    def inner_func(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        #计算用时
        time_cost = time.time() - start_time
        #打印用时，函数都有__name__ 这个属性，表示函数的名称
        print("Function \"" + func.__name__ + "\" costs %.2f"%(time_cost) + " seconds.")
        return result
    # 返回经过增强的函数。
    return inner_func
```

```
@time_measuring
def canPartition(nums): # 这只是一个用于测试上述计时器装饰器的例子，这是一个需要耗费一定时间的算法程序。函数用于判断能否把某个列表拆分成两个总和相等的子列表，这是个 0-1 背包算法问题，读者无需关注实现逻辑。
    s = sum(nums)
    if s % 2 == 1:
        return False
    t = s // 2
    l = len(nums)
    dp = [[0] * (t+1) for i in range(l+1)]
    # dp[x][y] 表示用前 x 个元素是否可以找到和为 y 的子集
    # 0<=x<=l 0<=y<=t
    dp[0] = [1] + [0] * t
    for i in range(1,l+1):
        ele = nums[i-1]
        for j in range(t+1):
            if j - ele >= 0:
                dp[i][j] = dp[i-1][j] | dp[i-1][j-ele]
            else:
                dp[i][j] = dp[i-1][j]
    return bool(dp[-1][-1])
# 测试案例：判断 1,2,...,200 能否分为两个总和相等的子列表
work_list = list(range(200))
result = canPartition(work_list)
print(result)
```

运行结果为：

```
Function "canPartition" costs 0.56 seconds.
```

```
True
```

我们直接调用函数，分析结果发现，函数经过装饰器作用后，调用时就可以同时打印其运行时间，且函数的使用和原来相同。这便完成了一个计时装饰器。

4.3.4 利用装饰器管理函数或类

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
```

本例中的 manage_process 装饰器返回的是函数本身，它的作用是使全局的列表添加被作用的函数，通过这种形式要从列表删去函数也比较容易，去掉函数前的装饰器就好了。相比于将所有函数当作全局变量，然后直接用一个列表进行管理更方便。因为

列表很容易变得很长，很长的列表想要删改中间的一个特定元素比较繁琐，并且可能需要反复切换代码页面，如果用装饰器管理就不存在这些问题。

我们也可以用类装饰器来实现用装饰器管理类，实现过程是类似的。

4.3.5 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，表示这个函数中星号后面的参数是强制的关键字参数。

在此给出一个此语法的简单例子，假设在开发游戏时要实现一个函数，其功能是让玩家获得一定量的金币:

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

以下为不合法的调用:

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

4.3.6 强制的位置参数

[Python3.8 新增]

这是 Python3.8 的新特性，类似于上一个小节介绍的强制的关键词参数，我们用"/"标记强制的位置参数，在"/"前的参数必须以位置参数的形式传参，而不能用关键词参数的形式传参。这里给出一个例子：

```
def myfunc(p1, p2, /, p3, p4, *, p5, p6):
    Pass
```

在本例中，调用 myfunc 函数时参数 p1 和 p2 必须是位置参数，参数 p5 和 p6 必须是关键词参数，p3 和 p4 可以是位置参数或者关键词参数。

4.3.7 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。程序员无法直接调用私有函数。

(补充：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 等语言所不同的是 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对变量进行名称改写。)

私有变量一种常见的使用场景为：

有时候我们可以会写出一个类，其中有大量的函数。我们可以将一些函数写为私有函数，相当于告诉别人这个是中间步骤，无需调用，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。程序员仍然可以直接使用以单下划线开头的私有变量。

4.3.8 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)
```

这里的 **ff** 相当于是冻结了 **a,b,c** 三个参数后的另一个函数了。

4.4 类与对象

4.4.1 isinstance 和 type 的区别

type(entity) == "class_name"和 isinstance(entity,"class_name")都常用于判断 entity 是否属于"class_name"类。两种判定方法是区别是，isinstance(entity,"class_name")这个判定方法中，当 entity 属于"class_name"类时，依然会返回真。由于在编写程序时，类常常要通过继承和被继承的方式进行拓展，用 isinstance(entity,"class_name")这个判定方法会具有更好的可拓展性，所以一般推荐使用 isinstance(entity,"class_name")进行判断某对象是否属于某类。

type(entity) == "class_name"是一种更加严格的判断。

阅读下面的例子：

```
class A():
    pass


class B(A):
    Pass


b = B()
print("b is instance of B?",isinstance(b, B))
print("b is instance of A?",isinstance(b, A))
print("type(b) == B?",type(b) == B)
print("type(b) == A?",type(b) == A)
```

程序运行结果如下：

```
b is instance of B? True
b is instance of A? True
type(b) == B? True
type(b) == A? False
```

分析运行结果可以发现，type 的判断更为严格。isintance 是对否属于该类以及其父类的判断。

4.4.2 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了效率。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。但作者再次强调：读者需要注意，使用此技巧时类最好要满足两个条件：(1)类的属性是事先已知且不变，(2)类多次用于创建实例，因为此技巧在提升效率的同时会降低代码的可扩展性。

4.4.3 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。

另外注意：Python 中的 super()函数并不是取"父类"，也是按照__mro__的顺序进行依次取类，这可能会在编写多继承时碰到。

假设当前类继承的多个超类中都有 foobar 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。因此，需要定义继承顺序，但普通的广度优先搜索(bfs)和深度优先搜索(dfs)都存在不能完美解决的情形。

(补充：dfs 无法解决菱形继承顺序，bfs 无法解决叉状继承顺序。读者可以阅读 http://blog.sina.com.cn/s/blog_45ac0d0a01018488.html 博客以了解详细知识。)

Python 中通过 C3 算法(一种*拓扑排序*算法)解决该问题。Python 中类分为新式类和经典类两种，Python3 中按预设方法创建的类就是新式类，新式类用 C3 算法搜索继承顺序。 __mro__ 属性保存了 C3 算法的结果，也就是最终的方法解析顺序。

*拓扑排序*是一个图论中的概念，用于将一个有向无环图中的所有节点排成一个线性序列，任意两个不同的节点存在先后次序。

C3 算法具体实现过程比较复杂，这里不详细介绍，读者可以查阅相关资料。

4.4.4 __init__和__new__的区别

这个问题常见于 Python 工程师面试中。

新建一个对象__init__和__new__魔法方法都需要调用一次，区别在于__init__是一个针对对象的概念，__new__是一个针对类的概念。__new__是用来控制对象的生成过程的，常用于元类编程，一般不需要修改。__init__是用来初始化对象的，应用很广泛。

新建对象时 Python 解释器会先调用__new__再调用__init__，如果__new__不返回对象，则不会调用__init__方法。

一个简单的例子，运行如下代码：

```
class Person():
    def __new__(cls,*args,**kwargs):
        print("__new__ called.")
        return super(Person, cls).__new__(cls)

    def __init__(self,name,info = None):
        print("__init__ called.")
        self.name = name
        self.info = info if info is not None else {}

Alice = Person("Alice")
```

程序运行结果：

```
__new__ called.
__init__ called.
```

分析结果可以看到，__new__先于__init__被调用。

4.4.5 __getattr__和__getattribute__的区别

这个问题常见于 Python 工程师面试中。

__getattr__用于当访问对象的属性访问不到时，就调用对象的__getattr__魔法方法，并将该方法的返回值代题对象的属性。__getattr__的应用十分广泛，一个很简单的应用如下：

```
class Person():
    def __init__(self,name,info = None):
        self.name = name
        self.info = info if info is not None else {}
```

```
    def __getattr__(self,key):
        return self.info[key]

info = {"age":10,"gender":0}
Alice = Person("Alice",info)
print("Alice's age is {}.".format(Alice.age))
```

程序运行结果为：

```
Alice's age is 10.
```

简单分析代码，可以发现虽然我们没有定义 Alice 的 age 属性，但是可以调用 Alice.age 不会报异常。此时 Python 解释器会先判断 Alice 有没有 age 属性，发现没有该属性后，就会调用 Alice 的__getattr__魔方方法，将方法返回值作为属性值。

__geattribute__方法也是在访问属性时调用的，与__getattr__方法不同的是，无论有没有那个属性，能否访问到那个属性，都会调用__geattribute__方法。可见__geattribute__方法是一个通用性质的入口，一般编程不建议重写__geattribute__方法，以免出错。更深入的话题涉及 Python 的***属性描述符***和***属性查找过程***。

4.4.6 鸭子类型和协议

Python 的魔法方法引出了 Python 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？假设我自定义了一个类 myclass，并为其定义了__iter__魔法方法或者__getitem__魔法方法中的一个，那么它就实现了可迭代对象的***协议***。它可以像列表一样被迭代，即可以用 for i in myclass()的语法来遍历元素，不需要 myclass 必须是列表，这样就使程序设计灵活，本例中可迭代对象就是一个***鸭子类型***。

这又引出了 Python 中的一个概念，称之为***协议***。在 Python 中，我们完成某些接口之后，Python 解释器就可能自动帮我们实现相应功能，常用的协议有迭代器协议(详见&5.6.2)、上下文管理器协议等。

4.4.7 上下文管理器

上下文管理器是 Python 中的一个高级特性，它是使用 with 关键词的对象。

在 Python 中以写入的形式打开文件的基本流程如下(假定同级目录下有 file.txt)：

```
f = open('file.txt', 'w')
f.write('contents for writing!')
```

这里的 f 是一个文件句柄对象。

运行上述代码，则脚本文件会在同目录下生成'file.txt'文件，并写入测试字符串。此种写法会产生内存泄漏，因为没有关闭已经打开了的文件句柄，应该再补上一句 f.close()。

上述写法也可以不用写 close，改用 with 的语法来写，代码如下：

```
with open('file.txt', 'w') as f:
    f.write('contents for writing!')
```

这里的 f 也是一个文件句柄对象，由 open('file.txt', 'w')创建。这种写法即便在打开文件时遇到异常，也能够正确地自动关闭文件。

至此复习了 with 的基本用法。

Python 中可以用 with 实现上下文管理器，用 with 操作文件时，之所以保证一定能关闭文件，是因为文件句柄有__exit__魔法方法，在__exit__中定义了关闭文件。

with 的逻辑是打开时调用__enter__魔法方法，退出时调用__exit__魔法方法，这是 Python 的上下文管理器协议。

接下来我们尝试构建一个自定义的文件句柄工具：

```
class SimpleHandle:
    def __init__(self, name, mode):
        self.name = name
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.name, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
```

```
        if self.file:
            self.file.close()

with SimpleHandle('file.txt', 'w') as f:
    f.write('contents!')
```

这就实现了一个文件句柄工具，在 **with** 语句中可以自动在退出时调用__**exit**__魔方方法，从而实现自动关闭文件，防止内存溢出。文件句柄工具只是 **with** 语句用法的基本案例，用户可以加入自己的逻辑，实现其他功能(例如读写、连接数据库等)。

上下文管理器还可以利用 **Python** 预设的 **contextlib** 库进一步简化。用 **contextlib** 的 **contextmanager** 装饰器就可以将一个函数变为一个上下文管理器，功能和用类实现上下文管理器是一致的。

将上述的代码文件句柄工具改写为如下：

```
import contextlib

@contextlib.contextmanager
def simple_handle(name, mode):
    try:
        f = open(name, mode)
        yield f
    finally:
        f.close()

with simple_handle('file.txt', 'w') as f:
    f.write('hello world')
```

同样可以实现和之前一样的功能，**@contextlib.contextmanager** 的用法是固定的，函数必须是生成器，**yield** 之前的内容等价于__**enter**__中的内容，**yield** 之后的内容等价于__**exit**__中的内容。

执行顺序为，调用 **simple_handle** 生成器函数，首先运行 open(name, mode)打开文件，这等价于之前用类实现上下文管理器时__**enter**__中的内容，然后生成器返回文件句柄对象。**with** 语句执行完后，**finally** 中的代码块就会执行。

更多资料请读者阅读文档 https://docs.python.org/3/library/contextlib.html。

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%。

4.5.2 numba.jit 装饰器

Python 程序运行效率低是因为 CPython 的解释器运行效率低。numda 模块的 jit 装饰器用于将一个函数译成机器语言，机器语言不需要编译，从而提高运行效率。

首先运行下列计算代码，并测试其运行时间：

```
import time

# 计算斐波那契数列，演示例子，读者无需关注其逻辑细节。

def computation():

    a,b = 0,1

    for i in range(200000):

        a,b = b,a+b

    return a


start = time.time()

computation()
```

```
span = time.time() - start
print(span)
```

运行结果如下：

```
0.3360774517059326
```

我们用 jit 装饰器增强函数，再运行，并测试运行时间：

```
from numba import jit
import time

@jit
def computation():
    a,b = 0,1
    for i in range(200000):
        a,b = b,a+b
    return a

start = time.time()
computation()
span = time.time() - start
print(span)
```

程序运行结果如下(略去首次运行)：

```
0.047023773193359375
```

可以看到经过 jit 装饰器增强后，函数运行时间从 0.34 秒减少到了 0.05 秒，加速效果显著。此方法具有通用性，几乎任何函数都可以使用此法进行加速。

5 高级进阶

5.1 底层实现

一般情况下我们不需要关心 Python 的底层实现，因为 Python 是一门注重功能实现而不是一门注重性能的编程语言。这里讨论一部分比较重要的 Python 底层实现，用于增加阅读者的知识丰富度，为实现更复杂，更庞大的体系时可以提供更深远、合理的方案。

5.1.1 列表的实现

Python 的列表是一种元素个数可变的***顺序表***，可以加入和删除元素，在各种操作中维持已有元素的顺序。元组拥有和列表类似的性值，这里只讨论列表的实现，元组是类似的。

(补充：顺序表是一个数据结构中的概念，数据以顺序结构存储在电脑内存中。)

随着 Python 版本的更新以下两段文字所描述的内容在实际中可能发生了变化，所以请读者注意最新的 Python 更新并以实际结果为准。

顺序表要求能容纳任意多的元素，就必须能更换元素存储区。要想在更换存储区时 list 对象的标识(id)不变，只能采用**分离式实现技术**。

(补充：**分离式实现技术**是一种顺序表的实现技术。)

在 Python 的官方系统中 list 实现采用了如下的实际策略：在建立空表（或很小的表）时，系统分配一块能容纳 8 个元素的存储区；在执行插入操作（insert 或 append 等）时，如果元素区满就换一块 4 倍大的存储区。但如果当时的表已经很大，系统将改变策略，换存储区时容量加倍。这里的“很大”是一个实现确定的参数，目前的值是 50000。引入后一个策略是为了避免出现过多空闲的存储位置。如前所述，通过这套技术实现的 list，尾端加入元素操作的平均时间复杂度是 O(1)。

5.1.2 字典的实现

Python 的字典和集合都是通过哈希表（又称为散列表）实现的，**哈希**是一种很重要的计算机技术,可以做到快速搜索和去重的效果。

(补充：哈希(hash)，亦称散列。简单理解即指将输入映射到另一个(通常是更小的)空间上输出，因此可能引起**冲突**。好的哈希函数应是接近一一对应的，减少冲突的且充分利用输出空间的。)

字典最常用的操作是查询，哈希表可以做到 O(1)时间的查询，为什么呢？我们还是要开辟连续的存储空间（即**顺序表**）进行存储，我们在存储数据就可以将输入的哈希

值直接作为存储的地址。这样在查询时，我们只需要计算一次输入的哈希值（计算哈希值是一步运算，复杂度为 O(1)）就得到了其存储的位置，从而将查询问题转化为了一个顺序表的访问问题，而顺序表的访问的复杂度是 O(1)。

这里还会涉及一些细节问题，例如哈希值越界（大于了开辟空间的长度上限）怎么办？哈希冲突了怎么办？其实这些问题有多种解决方案，这里介绍常见的解决方案。如果哈希值越界，但开辟空间的长度上限只有 x，那么将哈希值除以 x 取余即可。如果哈希冲突了，则在响应的存储位置开辟一个链表，将新元素以链表的形式进行存储，由于链表的访问需要 O(n)的复杂度，所以在极端最坏的情况(所有哈希值都冲突的情况)下，哈希表的查询复杂度可以从 O(1)变为 O(n)。

(补充：哈希表的链表加顺序表实现是一个很经典的哈希表实现思想。)

5.1.3 排序的实现

常见的排序算法有十种，即俗称的十大排序，分别是冒泡排序，选择排序，插入排序，快速排序，归并排序，希尔排序，堆排序，桶排序，计数排序和基数排序。十大排序是算法的基本功，蕴含着很多重要的算法基础，这不是本书要讲的重点，故本书将不再对其详细介绍。（如果读者不了解相关的知识那么最好先学习十大排序以学习本节的知识）。

本文将简单介绍 Python 的排序算法，蒂姆排序（Tim 排序），它不属于十大经典算法，但这是一种实践中被证明了的目前最好的排序算法。这种排序算法结合使用了归并排序和插入排序技术，时间复杂度是 O(nlogn)(基于比较的排序算法的复杂度下界是 O(nlogn))。蒂姆排序具有**适应性**(又称为**稳定性**)，也就是说如果数据中部分元素是已经排序好的，那么蒂姆排序可以利用这部分信息，从而复杂度更小。蒂姆排序的空间复杂度是 O(n)(基于比较的排序算法的复杂度下界是 O(1))。所以蒂姆排序并不是理论上最优的排序算法，但迄今为止，理论上最优的排序算法(O(nlogn)的时间复杂度，O(1)的空间复杂度同时具有适应性)还没有被找到。

蒂姆排序基本过程如下：

(1)考察待排序序非严格单调上升或严格单调下降的片段，反转其中严格下降片段。

(2)采用插入排序，对连续出现的几个特别短的上升序列排序，使整个序列变成一系列单调上升的纪录片段，每个片段长于某个特定值。

(3)通过归并产生更长的片段，控制这一归并过程，保证片段的长度尽可能的均匀，归并中采用一些策略，尽可能减少临时空间的使用。

5.1.4 随机数的实现

随机数是一个密码学中的重要概念，也许读者已经了解到计算机中产生的随机数都是**伪随机数**而不是真随机数。

常用的随机数产生器有 PMMLCG，Mersenne Twister 等。 我们这里讨论 Python 的 random 模块， Python 的 random 模块产生随机数是基于 os 库的 os.urandom() 产生的。os.urandom 函数可以生成指定字节的随机 bytes 对象。 那么 os.urandom 是基于什么原理产生随机数的呢，实际上更底层的原理是在 Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul., 8, 3-30.这篇论文中的。读者可以阅读该论文了解其随机数生成原理，本文将不再对其展开介绍。

python 官方提示我们通过 random 生产的这种随机数并不足够安全，如果想用于安全或者加密的目的，应该参考 python 的 secrets 模块[python3.6 新增模块]。

(关于本节内容可以拓展阅读 python 的官方文档 https://docs.python.org/3/library/random.html)

5.2 序列容器

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。(提示：如果读者具备数据结构的知识，阅读本章内容会更为顺畅)。

5.2.1 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且你在意程序的性能，应该考虑用 array，此时 array 的性能比 list 高很多，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))
```

关于 array.array 的详细资料，读者可以参考资料 https://docs.python.org/3/library/array.html。

5.2.2 collections.deque 双向队列

Python 中的列表天然地实现了栈的数据结构，可以利用 append 和 pop 方法(复杂度为 O(1))在表尾添加或者删除数据。但是 list 不能实现队列，想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。

关于 collections.deque 的详细资料，读者可以参考资料 https://docs.python.org/3/library/collections.html?#collections.deque。

5.2.3 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》。

5.2.4 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。

5.3 散列表容器

5.3.1 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。根据作者的经验，**这是一个极实用的对象**。

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。

5.3.2 collections.OrderedDict 有序字典[Python 3.1 新增]

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。

5.3.3 collections.Counter 计数器[Python 3.1 新增]

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。

5.4 常用模块、包或装饰器

5.4.1 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。其中最实用的一个函数为 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。)

5.4.2 Userdict 模块

我们写自定义类时，如果想要继承 dict 类，用常用的语法写则代码如下：

```
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。

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

5.4.3 logging 模块

Python 的 logging 模块为我们提供了日志管理功能，适用于中大型项目。在中大型项目中，日志对于开发和维护十分重要，如果只是依赖 print()简单地打印信息，功能显得单薄。

logging 可以输出不同严重程度的消息，并且程序员可以设置消息等级，使信息输出更加清晰可控。严重等级分别为：CRITICAL(致命), ERROR(错误), WARNING(警告), INFO(信息), DEBUG(调试), NOTESET(提示)。

默认是 WARNING(警告)的日志级别，只会输出严重程度大于或等于 WARNING(警告)的消息。

基本的使用案例如下：

```
import logging

logging.debug("a debug message")

logging.info("a info message")

logging.warning("a warning message")

logging.error("a error message")

logging.critical("a critical message")
```

这样就输出了三条日志信息：

```
WARNING:root:a warning message

ERROR:root:a error message

CRITICAL:root:a critical message
```

这里输出结果的 root 表示该信息是由谁给出的。

logging.basicConfig(**kwargs)是一个重要函数，我们可以用用它来设置要记录的日志级别、日志格式、日志输出位置、日志文件的打开模式等内容。

执行以下代码：

```
import logging
#设置了文件路径和日志输出级别
logging.basicConfig(filename='test.log', level=logging.DEBUG)
logging.debug("a debug message")
logging.info("a info message")
logging.warning("a warning message")
logging.error("a error message")
logging.critical("a critical message")
```

由于设置了文件路径，在 Python 控制台中不再输出消息了。在 Python 脚本文件的同目录下，生成了 test.log 的日志文件，日志信息如下：

```
DEBUG:root:a debug message
INFO:root:a info message
WARNING:root:a warning message
ERROR:root:a error message
CRITICAL:root:a critical message
```

关于 logging 模块的更多高级用法，请读者参考 logging 模块的官方文档 https://docs.python.org/3/library/logging.html。

5.4.4 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。

5.4.5 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，这种方法是"实例方法"。

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"类方法"，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"静态方法"，那么这个函数不需要传入 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)。

5.4.6 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。

5.4.7 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"选择排序"、"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"，但显然这样做比较复杂，不够 pythonic。还有一种方法是利用 pandas 库里的 sort_values 方法，但这需要将数据存入 pandas.Dataframe 再依赖 pandas 实现，灵活性较低，不易拓展功能。

这里我们利用 Python 预设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，数字实现了从小到大的排序。数字的英文名和中文名虽然不能直接比较，但也随数字进行了同时排序。

5.4.8 functools.lru_cache()装饰器

[Python 3.2 新增]

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。

(补充：所谓备忘，指的是将函数的运行结果储存起来，在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，不建议使用这个装饰器。

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，如不传入默认值为 128)，表示缓存的空间上限，传入 None 则表示无上限(从这里也可以看出 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。

5.4.9 functools.singledispatch 装饰器

[Python 3.4 新增]

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。

所谓函数重载，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，或使用模板，Java 中的泛型)。调用函数时，编译器会自动根据传入的参数的类型、数量来决定调用哪一个函数。

Python 虽然是一门动态类型语言，其语法极为灵活，函数传入的参数是不需要指明类型的，这就无法实现函数重载。但是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。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。乍一看代码变复杂很多，但新的代码是用函数重载的形式实现的。新代码更加扁平，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。

Python 之禅中告诉了我们 **Flat is better than nested**，扁平胜于嵌套。

5.5 习惯与通识

5.5.1 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。

5.6 类，对象和特性

5.6.1 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。简单理解就是，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。

(补充：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，只要用 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)。

(提示：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。

5.6.2 迭代器

Python 的迭代器是一个重要概念，迭代器是一种可以用于遍历数据容器。

首先请读者明确一件事，可迭代对象和迭代器是两个不同的概念，许多初学者会误认为 list 是一种迭代器，实际上 list 只是可迭代对象但并不是迭代器。

我们进行代码试验，从 collections.abc 模块中导入可迭代对象和迭代器。

运行如下代码：

```
from collections.abc import Iterable, Iterator
print(isinstance([], Iterable))
print(isinstance([], Iterator))
```

程序输出结果：

```
True
False
```

这就表明了列表是是可迭代对象但并不是迭代器。

事实上可迭代对象和迭代器是两种协议，可迭代对象指实现了__iter__魔法方法的对象，迭代器指实现了__iter__和__next__两种魔法方法的对象。

我们运行下列代码以验证结论：

```
from collections.abc import Iterable, Iterator
simple_list = [1,2,3]
simple_iter = iter(simple_list)
print("  -- simple_list -- ")
print(hasattr(simple_list, "__iter__"))
print(hasattr(simple_list, "__next__"))
print(isinstance(simple_list, Iterable))
print(isinstance(simple_list, Iterator))
print("  -- simple_iter -- ")
print(hasattr(simple_iter, "__iter__"))
print(hasattr(simple_iter, "__next__"))
print(isinstance(simple_iter, Iterable))
print(isinstance(simple_iter, Iterator))
```

程序运行结果为：

```
  -- simple_list --
True
False
True
False
  -- simple_iter --
True
```

```
True
True
True
```

iter()函数用于将一个可迭代对象变为迭代器。可迭代对象实现了__iter__魔法方法，迭代器实现了__iter__和__next__两种魔法方法。

了解可迭代对象和迭代器的区别之后，我们再手动实现一个迭代器，这边给出一个"图像迭代器"例子，需求是要遍历图像的每一个像素：

```
from collections.abc import Iterator
class Image(Iterator):
    #这是一个演示例子，图像像素存储在 list 中，像素以数值形式表现。
    def __init__(self,pixels):
        """
        Parameters
        ----------
        pixels : list
            pixels of the image.
        """
        self.pixels = pixels
        self._index = 0

    #__iter__这里是一种固定写法，返回 self 即可，也可以做一些初始化工作
    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self.pixels):
            pixel = self.pixels[self._index]
        else:
```

```
            self._index = 0
            #当元素不够迭代时 raise StopIteration 是一种固定写法，StopIteration
            可以被 for 循环识别
            raise StopIteration
        self._index += 1
        return pixel

simple_image = Image([0,1,2,3,4,5,6,7,8])
print("pixel =",next(simple_image))
print("pixel =",next(simple_image))
print("pixel =",next(simple_image))
print("pixel =",next(simple_image))
for i in simple_image:
    print("pixel =",i)
```

程序输出结果为：

```
pixel = 0
pixel = 1
pixel = 2
pixel = 3
pixel = 4
pixel = 5
pixel = 6
pixel = 7
pixel = 8
```

这样就实现了一个图像迭代器，本演示例子比较简单，也可以直接用列表实现，用迭代器显得多余。实际开发中可能面临更加复杂的情形，用迭代器会简单高效很多，迭代器的好处是 1：用来逐个取出元素，下一个元素可以依赖之前的元素。2：不用存储全部的元素，节约内存。

我们可以用 next 函数或者 for 的语法迭代其中的像素。用 for 循环迭代时会调用一次__iter__魔法方法，解下来每次迭代是调用__next__魔法方法来取元素。

5.6.3 元类

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

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，通过元类编程，我们可以不用 class 关键词也可以动态地创建一些类。但是绝大多数情况下我们不需要用到元类编程，许多 Python 高手跟我们反复强调了这一点，本节介绍元类仅仅用于知识拓展。

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。推荐阅读专著 *Fluent Python* 的最后一章。

5.6.4 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，所以又被称为微线程，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，很适合类似于爬虫，游戏，异步 I/O，数据读取或者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) #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。

5.6.5 异步编程

异步编程是 python 一种很实用的特性，可以通过 python 的 asyncio 库可以实现异步编程。在 python3.7 正式加入了 async 和 await 关键词，可以用于实现异步编程，简化了一些 asyncio 的语法，这里将仅简单介绍 async 和 await 关键词的用法，更高级的用法请读者阅读 asyncio 库的官方文档(https://docs.python.org/zh-cn/3/library/asyncio.html)。

这里我们用 task 函数模拟一个 1 秒的工作任务，pipeline 函数模拟流水线任务，它会依次执行三个 task 工作任务，运行如下代码：

```
import time


def task():
    print("task start")
    time.sleep(1)
    print("task done")
    print('%.2f' % time.time())


def pipeline():
    task()
    task()
    task()


print("流水线作业开始。")
pipeline()
print("流水线作业完成。")
```

运行上述代码，得到运行结果如下：

```
流水线作业开始。
task start
task done
```

```
1623666704.66
task start
task done
1623666705.68
task start
task done
1623666706.68
流水线作业完成。
```

查看运行结果可以推断总共 task 运行了三次，每次间隔 1 秒，总共用时 3 秒。将上述代码改为异步的写法，代码改写为：

```
import time
import asyncio

async def task():
    print("task start")
    await asyncio.sleep(1)
    print("task done")
    print('%.2f' % time.time())
    return None

def pipeline():
    asyncio.create_task(task())
    asyncio.create_task(task())
    asyncio.create_task(task())

print("流水线作业开始。")
pipeline()
print("流水线作业完成。")
```

得到运行结果如下：

流水线作业开始。

流水线作业完成。

task start

task start

task start

task done

1623682309.62

task done

1623682309.62

task done

1623682309.62

对比和分析两次代码的运行，读者可以理解一下异步的工作机理。在异步程序中，异步函数运行到 **await** 语句时异步函数会挂起，此时异步函数会跳出，继续执行之前的线程，直到 **await** 语句运行结束然后继续执行异步函数。通常 **await** 语句需要执行耗费时间的任务，例如读取数据，网络请求等。

所以在第二段代码中，程序表现为三段协程是异步从而“几乎同时”地执行，这就避免了因部分任务的等待拖累整个程序。

6 PYTHON 应用实例

Python 的应用领域实际上是很广泛的，根据 Python 官方网站的案例，主要运用于以下几个领域：Web 开发，图像界面开发，科学与数值计算，软件开发，系统管理。

Python 的应用很广泛，读者在平时遇到的各种计算机的问题，都可以尝试用 Python 进行解决，同时进行深入学习。例如编写批量操作文件(重命名，压缩，解压，转移等)，批量处理图片，批量处理工作表，桌面管理，爬虫，鼠标键盘宏，科学计算，人工智能，自制专属的小软件(音乐软件，背单词软件，日历，浏览器)等等。

初学者一定不要害怕，可以先从互联网上搜索他人的讨论，从一些代码仓库网站寻找他人的初步结果，学习他人的代码开始，了解此类问题的基本逻辑，再进行改进或者创造。

6.1 算法题

算法题主要用于锻炼程序员的逻辑思维，以及练习基本语法，本身不会涉及许多高级语法。熟练掌握算法原理对任何程序员是很重要的，这是写出高质量代码的基础。求职时，许多编程相关岗位中，算法题也是必考的。

值得注意的是，Python 的运行效率低于 C, C++, Java 等语言，同样一道算法题用 Python 来计算速度是这些语言的 50%左右，Python 的优势在于代码灵活，开发效率高。

6.1.1 字符串相加

数字在计算机中的大小是有限的，但字符串可以很长。大数的四则运算要基于字符串加减乘除来完成。在本题中希望实现一个函数，传入两个字符串，返回两个字符串表示的整数相加后的数(也是字符串形式)，参考代码如下：

```
#用 def 显式定义函数，要传入的 num1 和 num2 是数字意义的字符串变量，这里使用了&3.1.2 中提到的"函数标注"功能，使代码更易读。
def addStrings(num1: str, num2: str):
    # 以变量 r 创建一个列表，r 是 result 的简称，表示用于存储结果
    r = []
    length1, length2 = len(num1), len(num2)
    max_length = max(length1, length2)
    # 以变量 p 创建一个整数，存储进位数。
```

```
    p = 0
    for i in range(1,max_length+1):
        # 分别用 a 和 b 获取两个字符串的倒数第 i 位，如果当前循环次数小于等于字符串长度，设为 0(三元表达式)。
        a = int(num1[-i]) if i <= length1 else 0
        b = int(num2[-i]) if i <= length2 else 0
        # 如果 a + b + p >= 10，表示需要进位。
        if a + b + p >= 10:
            # t 是 temp 的简称，表示临时变量。如果进位将 t 是 a,b,p 之和减去 10，同时 p 设为 1。
            t = a + b + p - 10
            p = 1
        else:
            # 如果不进位将 t 是 a,b,p 之和，同时 p 设为 0。
            t = a + b + p
            p = 0
        # 将临时变量存入结果列表。列表是栈，添加在尾部效率较高，最后进行一次反转即可。
        r.append(str(t))

    # 走完了循环，进位还没有消耗，就要在列表末尾再添加一个 1，相当于在数字的首位添加一个 1。
    if p == 1:
        r.append("1")

    # 返回结果。r 是一个列表，需要反转一次，再用空字符串进行 join 以得到列表拼接而成的字符串。
    return "".join(reversed(r))
```

我们运行一个简单的测试用例 addStrings("999","1")，程序输出结果：

```
'1000'
```

在此测试用例上，函数正确地实现了功能，读者可以尝试运行其他测试用例。

6.1.2 最大正方形

给定一个由 0 和 1 组成的矩阵，求出矩阵中只包含 1 的最大正方形的面积。

示例：

输入：

```
matrix = [["0","1","1","1"],
["1","0","1","1"],
["1","0","1","0"]]
```

输出：

```
4
```

只包含 1 的最大正方形在矩阵右上角。

此题是一个比较经典的二维动态规划问题，实际中也有可能碰到类似问题。如果只是暴力循环每一个元素，逐次判断，复杂度是令人无法接受的，用合适的算法原理可以让程序高效地计算出答案。参考代码如下：

(补充：动态规划的英文名是 dynamic programming，简写是 dp，这个词及其简写在算法题中很常见。)

```
def maximalSquare(matrix: list(list())):
    # 获取矩阵的长和宽。
    n, m = len(matrix), len(matrix[0])
    # ans 是 answer 的简写，用于存储答案。
    # 定义状态转移矩阵。因为每个元素都一定会被更新，新元素只会用到已经更新过的值，这里可以直接设定为 matrix 的拷贝。copy 是列表的方法。
    # 如果不用 matrix.copy()的形式，可以写为[[0] * m for i in range(n)]。&3.2.8 节中介绍了如何正确创建一个二维数组。
    # 此题中状态表示以此位置为右下角的只含有 1 的最大正方形的边长。
```

```
dp = matrix.copy()
# ans 是 answer 的简写，用于存储答案。
ans = 0
# 对矩阵下标进行一次线性扫描，迭代所有下标。
for i in range(n):
    for j in range(m):
        # 根据下标取出当前迭代的元素。
        element = matrix[i][j]
        # 如果当前元素为 0，则将对应位置的状态也设为 0，且不需要进行后续判断。
        if element == '0':
            dp[i][j] = 0
        else:
            # 当前元素为 1。
            # 如果 i >= 1 and j >= 1，表明当前元素不在边界位置。
            if i >= 1 and j >= 1:
                # 状态 = 1 + 左边，左上和上方的状态中的最小值。这里是 dp 的核心逻辑，请读者仔细体会。
                dp[i][j] = 1 + min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
            # 否则是在边界位置，且元素为 1，直接将状态设为 1 即可，不可能是其他数字了。
            else:
                dp[i][j] = 1
        # 选出最大边长，每个元素跟之前的最大值进行一次比较。
        # 这是对 ans = dp[i][j] if dp[i][j] > ans else ans 的简写。
        ans = max(ans, dp[i][j])
# ans 是边长，函数返回面积。
return ans ** 2
```

我们运行一个简单的测试用例：

```
matrix = [["0","1","1","1"],
["1","0","1","1"],
["1","0","1","0"]]
print(maximalSquare(matrix))
```

程序输出结果如下：

```
4
```

在此测试用例上，函数正确地实现了功能，读者可以尝试运行其他测试用例。

6.2 爬虫案例

#to be continued

6.3 机器学习案例

#to be continued

6.4 纸牌游戏案例

#to be continued

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。

读者可以从 https://github.com/zjcm/Joyful-Python 获取本书电子版本。

参考文献

Effective Python – Brett Slatkin

Fluent Python – Luciano Ramalho

Python 高级编程 [中文译本] – Tarek Ziade

数据结构与算法 Python 语言描述 - 裘宗燕

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

Python 官方网站资料 https://www.python.org/dev/peps/ 等