华章程序员书库

Python 代码整洁之道：编写优雅的代码

Clean Python: Elegant Coding in Python

[美] 苏尼尔・卡皮尔（Sunil Kapil）著

连少华　译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Python 代码整洁之道：编写优雅的代码 /（美）苏尼尔 · 卡皮尔（Sunil Kapil）著；连少华译 . —北京：机械工业出版社，2020.9
（华章程序员书库）
书名原文：Clean Python: Elegant Coding in Python

ISBN 978-7-111-66587-8

I. P…　II. ①苏…　②连…　III. 软件工具 - 程序设计　IV. TP311.561

中国版本图书馆 CIP 数据核字（2020）第 178782 号

本书版权登记号：图字　01-2020-2372

Python 代码整洁之道：编写优雅的代码

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：李美莹　　责任校对：李秋荣
印　　刷：北京瑞德印刷有限公司　　版　　次：2020 年 10 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：13
书　　号：ISBN 978-7-111-66587-8　　定　　价：79.00 元

客服电话：（010）88361066　88379833　68326294　　投稿热线：（010）88379604
华章网站：www.hzbook.com　　读者信箱：hzit@hzbook.com

The Translator's Words 译 者 序

自 1991 年 Python 诞生以来，到现在将近 30 年了。如今，Python 已经被很多领域的专业人士广泛使用，亦有相当多的小学生开始学习 Python 编程，可见其被接受的程度非常高！由于其学习门槛低、语法简单、易学易用等特性，Python 已经被诸多领域广泛使用，如金融工程、人工智能、数据分析、科学计算、自动化测试等，这些领域中既有专业的软件开发人员也有非专业的软件开发人员。随着时间的推移，Python 有可能会发展成一门基础学科，所以，学好 Python 是在一些领域生存发展的必备技能。

我翻译的第一本书是《C++ 代码整洁之道》，已经发现身边的一些公司和培训机构都有购买，大家的反响还是很不错的，网上也有不少的评论（当然，那本书是讲 C++ 相关的知识）。这本书是我翻译的第二本书，希望这本书也和第一本书一样能够被广大读者所接受。

自我学习 Python 以来，与之前学习过的 C++、C#、Java、Golang、Node.js 等编程语言相比，Python 给我的感觉是：入门容易（小学生都可以使用 Python 写程序），深入难（在工作中发现很多自称熟悉 Python 的人，不知道生成器是什么、迭代器是什么、Python 中有哪些数据结构），可见选择一本好的教材是多么重要。虽然网上有很多关于 Python 的视频，但结合我自身的经验，不建议通过视频学习 Python，因为投入产出比不高，视频中讲解的内容往往是过时的，会给初学者带来较大的困惑。

如果你有幸购买了本书，并且是 Python 爱好者，那么强烈建议你仔细地阅读本书的每一个章节。这些章节之间虽没有必然的联系，但还是建议你按顺序阅读。当然，本书

的内容有些难度，在阅读本书前，建议你对 Python 的基础知识先有一定的了解，否则可能会看不太明白，因为本书没有从 Python 的基础语法讲起。

市面上常见的 Python 有两大版本：一个版本是 Python 2.x 系列，目前已经停止维护，但还有一些公司在使用；另一个版本是 Python 3.x 系列，本书中所讲的都是 Python 3.x 系列的知识。据不完全了解，绝大部分公司的新项目都已经基于 Python 3.x 进行开发了，所以尽快掌握 Python 3.x 吧！ Python 2.x 和 Python 3.x 两大系列差异较大，其中的原因与 Python 之父的性格有较大关系，想知道详细情况的读者可在网上自行查找资料。

阅读本书前，建议先了解以下基础知识：

1）如何安装 Python 3.x 最新版本的解释器，截止到本书译完，最新的稳定版本是 Python 3.8，更多信息请参见 https://www.python.org/downloads/。安装一款自己喜欢的 IDE 开发工具，初学者一定不要在记事本中写代码，切记，切记！！！

2）Python 的基础知识——关键字、控制语法、常用数据结构等。

3）了解如何使用 IDE（如 pycharm、Sublime Text、VIM/Emacs），建议写一些 Python 的练习代码，如读写文本文件、九九乘法表、数据结构的使用等。

4）对软件开发人员而言，最好具有面向对象开发的基础，并且知道一些基本的原则，如单一职责原则、开闭原则、里氏替换原则、依赖倒置原则、迪米特法则等。

5）对测试人员而言，最好了解 unittest 和 pytest 框架。

通过阅读本书，你将学到以下主要内容：

1）如何编写整洁的 Python 代码。

2）Python 的数据结构及特点。

3）Python 中的函数、类和模块（模块在很多书中没有提及或只是简单提及，本书有着较详细的讲解）。

4）装饰器、生成器、迭代器和上下文管理器的作用和使用场景。

5）Python 3.x 中的一些新特性，如 async 及协程、类型标注等。

6）调试和单元测试的一些工具。

这本书的英文原版我初步浏览后就被深深吸引，所以着手进行了翻译。由于平时工作比较忙，所以只能利用业余时间进行翻译。翻译过程中也遇到了许多困难，即便针对

一些有争议的术语、内容等查阅了大量的资料，翻译完成以后又进行了仔细的推敲和校对，但受限于译者的水平，稿中仍然难免存在疏忽、遗漏，甚至翻译错误或不准确的地方。读者在阅读过程中发现了任何质量问题，都可以向译者或出版社反馈。我的初衷是帮助想学好 Python 的人，希望这本书能够给你的学习带来促进。

在此，我非常感谢为翻译本书做出或多或少贡献的人，他们是盛斌、谢威和杨丽丽。在这里感谢他们积极地参与到翻译工作中，也感谢他们对我的认可和支持。

我现在担任 CSDN C/C++ 大版的版主和 C++ 小版的版主，受限于精力，就没有再担任 Python 版块的版主，你可以在 CSDN 网站和我私下交流。

感谢出版社给予我无比的信任和翻译的机会，也感谢读者选择了本书！希望本书的内容及译文没有让读者失望。

连少华

2020 年 6 月于深圳

前　言 *Preface*

Python 是当今最流行的语言之一。相对较新的领域如数据科学、人工智能、机器人和数据分析，以及传统的专业如 Web 开发和科学研究等，都在拥抱 Python。对于用 Python 这样的动态语言编写代码的程序员来说，确保代码的高质量和无错误变得越来越重要。作为一名 Python 开发人员，你希望确保正在构建的软件能够让用户满意，而不会超出预算或无法发布。

Python 是一种简单的语言，但是很难写出好的代码，因为目前可以教我们写出更好的 Python 代码的资源并不多见。

目前 Python 中缺乏的是代码一致性、模式以及开发人员对良好 Python 代码的理解。对于每个 Python 程序员，良好的 Python 代码都有不同的含义。出现这种情况的原因可能是 Python 被用于如此多的领域，以至于开发人员很难就特定的模式达成一致。另外，Python 没有像 Java 和 Ruby 那样有关于整洁代码的书籍。已经有人尝试编写这类书籍，但这样的尝试比较少，而且坦率地说，它们的质量也不高。

本书的主要目的是为不同级别的 Python 开发人员提供技巧，以便他们能够编写更好的 Python 软件和程序。无论你在哪个领域使用 Python，本书都可以为你提供各种各样的技巧。本书涵盖了从基础到高级的所有级别的 Python 知识，并向你展示了如何使代码更符合 Python 的风格。

请记住，编写软件不仅是一门科学，而且还是一门艺术，本书将教你如何成为一名更好的 Python 程序员。

Acknowledgements 致　　谢

首先，我要感谢 Apress 的 Nikhil。Nikhil 于 2018 年 10 月联系我，并说服我与 Apress Media LLC 合作写书。然后，我要感谢 Apress 的助理编辑 Divya Modi，感谢她在我撰写期间给予的大力支持，以及在我繁忙的工作时间里给予我的耐心。另外，非常感谢 Apress 的开发编辑 Rita Fernando，她在评审过程中提供了宝贵的建议，使本书对 Python 开发人员更有价值。接下来，我要感谢 Sonal Raj 对每一章的严格审查，也发现了很多我从未发现的问题。

当然，我要感谢 Apress 的整个生产团队对我的支持。

最后（但并非不重要的），我要感谢我亲爱的无可替代的家人，特别是他们理解写一本书需要大量的时间。感谢我的母亲 Leela Kapil 和父亲 Harish Chandra Kapil 的鼓励和支持。我深爱的妻子 Neetu，非常感谢你在我写这本书时所给予的所有鼓励和支持，这让一切都不同了。你太棒了！

关于作者 *About the Author*

Sunil Kapil 在过去的 10 年中一直从事软件开发工作，用 Python 和其他几种语言编写代码，主要涉及 Web 和移动端服务的软件开发。他开发、部署并维护了被数百万用户喜爱和使用的各种项目，这些项目是与来自不同专业环境的团队合作完成的，涉及世界著名的软件公司。他也是开源的热情倡导者，并持续贡献 Zulip Chat 和 Black 等项目。他还与非营利组织合作，并以志愿者的身份为其软件项目做出贡献。

Sunil Kapil 经常在各种聚会和会议上讨论 Python。

你可以访问他的有关软件工程、工具和技术的网站。最重要的是，你可以通过电子邮件联系他或在社交媒体上关注他。

网站：https://softwareautotools.com/

E-mail: snlkapil@gmail.com

Twitter: @snlkapil (https://twitter.com/snlkapil)

LinkedIn: https://www.linkedin.com/in/snlkapil/

GitHub: https://github.com/skapil

About the Technical Reviewer 关于技术审校者

Sonal Raj (@_sonalraj) 是一位作家、工程师、导师，也是一名喜欢 Python 超过 10 年的粉丝。他曾在高盛（Goldman Sachs）就职，也曾在印度科技学院（Indian Institute of Science）担任研究员。他是金融科技行业不可或缺的一员，擅长构建交易算法和低延迟系统。同时他也是一名开源开发人员和社区成员。

Sonal 拥有信息技术和工商管理硕士学位。他的研究领域包括分布式系统、图形数据库和教育技术。他是英国工程技术学会（IET）的一名活跃成员，也是印度技术教育协会的终身会员。

他是 *Neo4j High Performance* 一书的作者，这是一本关于图形数据库 Neo4j 的功能和使用的书。他也是《面试要点》系列丛书的作者，这些丛书侧重于技术面试方法。Sonal 还在 People Chronicles Media 担任编辑，也是 *Journal of Open Source Software*（JOSS）的评论员，以及 Yugen 基金会的创始人之一。

目　录 *Contents*

第 1 章 Chapter 1

关于 Python 的思考

Python 和其他编程语言不同的地方在于它的简洁但不失深度。正因为简洁，谨慎地编写 Python 代码很重要，尤其是在大型项目中，很容易不小心写出复杂和臃肿的代码。Python 有一个称作“Python 之禅”的设计哲学，注重简洁而不是复杂[⊖]。

在这一章，你将学会如何使你的 Python 代码更加具有可读性和简洁性的常用准则。我将讲述一些众所周知的准则，当然也有一些可能是不太常见的。当你即将着手开发新项目或者正在开发某一个项目时，希望你知道这些准则，以便可以提高代码质量。

注意 在 Python 的世界里，遵循 Python 之禅的哲学能让你的代码变得更加 Python 化。Python 官方文档推荐了很多实践准则，可以使你的代码更加整洁以及更具有可读性。阅读 PEP8 规范将会帮助你理解为什么一些实践准则被推荐。

1.1 编写 Python 代码

Python 有一些官方文档，定义了编写 Python 化代码的最佳实践，叫作 PEP8 规范。

⊖ http://www.python.org/dev/pess/pep-0020/

这种风格规范会随着时间不断改进。你可以访问 https://www.python.org/dev/peps/pep-0008 了解更多信息。

在这一章，你将学习一些定义在 PEP8 中的一般实践，并且明白开发者如何从遵循这些一般实践中受益。

1.1.1 命名

作为一名软件开发者，我使用过多种开发语言，比如 Java、NodeJS、Perl、Golang。所有这些编程语言对变量、函数、类等都有命名规范。Python 同样也有推荐使用的命名规范。我将讨论一些在编写 Python 代码时需要遵守的命名规范。

1. 变量和函数

你应该使用小写字母命名变量和函数，并且用下划线分割单词，因为这会让你的代码更具可读性，如代码清单 1-1 所示。

代码清单1-1　变量命名

```
names = "Python"                          # variable name
job_title = "Software Engineer"           # variable name
                                          with underscore

populated_countries_list = []             # variable name
                                          with underscore
```

你应该在代码里使用非混淆（内置属性）的命名方法，即使用一个下划线或者两个下划线，如代码清单 1-2 所示。

代码清单1-2　非混淆的命名

```
_books = {}                   # variable name to define
                              private
__dict = []                   # prevent name mangling with
                              python in-build lib
```

当你不想让一个类的成员变量被外部访问时，应该使用一个下划线为前缀来命名变量。这仅仅是一个约定俗成的规定，Python 并没有强制规定以一个下划线为前缀来定义

私有化。

Python 对于函数也有同样的约定，如代码清单 1-3 所示。

代码清单1-3　通常的函数命名

```
# function name with single underscore
def get_data():
    ---
    ---

def calculate_tax_data():
    ----
```

同样的规则也被应用在私有方法和那些你想防止与 Python 内置函数出现名称混淆的方法，如代码清单 1-4 所示。

代码清单1-4　表示私有方法和防止名称混淆的函数命名

```
# Private method with single underscore
def _get_data():
    ---
    ---
# double underscore to prevent name mangling with other
in-build functions
def __path():
    ----
    ----
```

除了遵循这些命名规则外，重要的是使用具体的名称，而不是对函数或变量使用模糊的名称。

我们来考虑设计一个传入用户 ID 返回用户对象的函数，如代码清单 1-5 所示。

代码清单1-5　函数命名

```
# Wrong Way
def get_user_info(id):
    db = get_db_connection()
    user = execute_query_for_user(id)
    return user

# Right way
```

```
def get_user_by(user_id):
    db = get_db_connection()
    user = execute_user_query(user_id)
    return user
```

在这里，第二个函数 `get_user_by`，确保你使用相同的词汇来传递变量，它为函数给出了正确的上下文。第一个函数 `get_user_info` 就显得模棱两可，因为参数 `id` 不明确，它是用户表的索引 ID？还是付款的 ID 或者是其他含义的 ID？这种代码会对使用你的 API 的其他开发者造成困惑。为了解决这个问题，我在第二个函数改动了两个地方——函数名和传递的参数名，这让代码更加具有可读性。当阅读第二个函数时，你就能正确明白函数的含义以及函数的期望输出。

作为一名开发人员，你有责任仔细思考如何命名变量和函数，使代码对其他开发人员具有可读性。

2. 类

类的命名应该像其他编程语言一样使用大驼峰方法[⊖]。代码清单 1-6 演示了一个简单的例子。

代码清单1-6　类命名

```
class UserInformation:
    def get_user(id):
        db = get_db_connection()
        user = execute_query_for_user(id)
        return user
```

3. 常量

你应该全部使用大写字母来定义常量名，代码清单 1-7 演示了一个例子。

代码清单1-7　常量命名

```
TOTAL = 56
TIMOUT = 6
MAX_OVERFLOW = 7
```

⊖ 命名方法有很多种，比如匈牙利命名法、Pascal 命名法（又称之为大驼峰命名法）、小驼峰命名法等。——译者注

4. 函数和方法参数

函数和方法参数应该遵循与变量和方法名相同的规则。类方法使用 `self` 作为第一个关键字参数，而函数不使用，如代码清单 1-8 所示。

代码清单1-8　函数和方法参数

```
def calculate_tax(amount, yearly_tax):
    ----

class Player:
    def get_total_score(self, player_name):
        ----
```

1.1.2　代码中的表达式和语句

有时候为了节省代码行数或者让你的同事对你的代码印象深刻，你可能会用一种“巧妙”的方式来编写代码。然而，编写巧妙的代码是有代价的，它会对代码可读性和简洁性有一定的影响。让我们来看下代码清单 1-9 所示的代码片段，它是对嵌套字典进行排序。

代码清单1-9　对嵌套字典进行排序

```
users = [{"first_name":"Helen", "age":39},
         {"first_name":"Buck", "age":10},
         {"first_name":"anni", "age":9}
        ]
users = sorted(users, key=lambda user: user["first_name"].
lower())
```

这段代码存在的问题在哪呢？

你使用一行代码的 lambda 表达式对嵌套字典按照 `first_name`（名字字段）进行排序，使其看起来像是使用一种巧妙的方式对字典进行排序，而不是使用循环。

然而，第一眼看这段代码时不是那么容易理解，尤其是对新手而言，因为 lambda 语法比较奇特，并不是一种容易理解的概念。当然，在这里你通过使用 lambda 节省了代码行数，因为它允许你以一种巧妙的方式对字典进行了排序，然而这并没有让这段代码明了并具有良好的可读性。这段代码无法解决丢失键值或者字典是否合法等问题。

让我们使用一个函数来重写这段代码，并且尽量让代码更明了并具有可读性。这个函数会校验所有非期望值，并且编写起来更简单，如代码清单 1-10 所示。

代码清单1-10　以函数形式对字典排序

```
users = [{"first_name":"Helen", "age":39},
         {"first_name":"Buck", "age":10},
         {"name":"anni", "age":9}
        ]

def get_user_name(users):
"""Get name of the user in lower case"""
    return users["first_name"].lower()

def get_sorted_dictionary(users):
"""Sort the nested dictionary"""
    if not isinstance(users, dict):
        raise ValueError("Not a correct dictionary")
    if not len(users):
        raise ValueError("Empty dictionary")

    users_by_name = sorted(users, key=get_user_name)
    return users_by_name
```

正如你所见，这段代码校验了所有可能的非期望值，并且比起之前的一行代码风格更具有可读性。一行代码风格能节省代码行，但是给你的代码增加了复杂度。这并不意味着一行代码风格很糟糕，在这里我尽力去强调的点是如果一行代码风格让阅读代码很艰难，尽量避免使用。

当编写代码时，你必须慎重做这些决策。有时候一行代码让你的代码具有可读性，有时候则相反。

让我们再考虑一个例子，你正在尝试读取一个 CSV 文件，并且计算被这个 CSV 文件处理的行数。代码清单 1-11 的代码将告诉你为什么使代码具有可读性很重要，以及命名在使你的代码更具有可读性中如何扮演重要角色。

将代码分解成辅助函数有助于使复杂代码具有阅读性，并且在你的生产代码遇到特定错误时更方便调试。

代码清单1-11　读取一个CSV文件

```
import csv

with open("employee.csv", mode="r") as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
            print(f'\t{row["name"]} salary: {row["salary"]}'
                  f'and was born in {row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

上述代码在 `with` 语句里做了很多事情。为了让它更具有可读性，你可以把处理 `salary`（薪酬）的逻辑从 CSV 文件中剥离到一个不同的函数中，这样也能减少出错。当很多逻辑聚集在很少行的代码里时是很难调试的，所以你应该在定义函数时确保目标明确、界限明了。所以，在代码清单 1-12 中我们把它再细分一点。

代码清单1-12　以更具可读性的代码读取一个CSV文件

```
import csv

with open('employee.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    process_salary(csv_reader)

def process_salary(csv_reader):
"""Process salary of user from CSV file."""
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} salary: {row["salary"]}')
        line_count += 1
    print(f'Completed {line_count} lines.')
```

这里编写了一个辅助函数而不是在 `with` 语句写下所有的逻辑。这让读者很清楚函数 `process_salary` 实际是干什么的。如果你想要处理特定的异常或者想要从一个 CSV 文件读取更多数据，为了遵循单一职责原则，可以把这个函数再细分。

1.1.3 拥抱 Python 编写代码的方式

PEP8 中有一些对写出整洁并且更具有可读性的代码的可以遵循的建议。让我们看下其中的一些准则。

1. 更偏向使用 join 而不是内置的字符串连接符

每当你考虑代码性能时，使用 `"".join()` 方法（如 `"".join([a, b])`）而不是使用内置的字符串连接符（如 `a += b` 或者 `a = a + b`）。`"".join()` 方法保证在各种 Python 实现中连接操作耗时较少。

这是因为当你使用 `join` 时，Python 对已经连接的字符串只分配一次内存，但是当你使用连接符连接字符串时，Python 不得不为每一次连接字符串分配新的内存，因为 Python 字符串是不可更改的。如代码清单 1-13 所示。

代码清单1-13　使用join方法

```
first_name = "Json"
last_name = "smart"
# Not a recommended way to concatenate string
full_name = first_name + "  " +  last_name
# More performant and improve readability
" ".join([first_name, last_name])
```

2. 在判断是否为 None 时考虑使用 is 和 is not

在判断是否为 None 时，始终使用 is 或者 is not。在如下情况下，把这一点记在脑海里：

```
if val:                    # Will work when val is not None
```

在上述代码中，确保记住你是要把 `val` 当作 `None`，而不是其他容器类型，例如 `dict`（字典）或者 `set`（集合）。让我们更加深入理解下这样的代码在哪种情况会让人感到惊讶。

在先前的代码中，如果 `val` 是一个空字典，`val` 会被当作是 false，它不是预想的那样，所以编写这种代码时一定要小心。

不要这样：

```
val = {}
if val:                 # This will be false in python context
```

相反，为了让你的代码更少地引发错误，应尽可能编写明确含义的代码（即显式地编写代码）。

而是应该这样：

```
if val is not None:       # Make sure only None value will be false
```

3. 更偏向使用 is not 而不是 not ... is

`is not` 和 `not ... is` 没有什么差别。然而，`is not` 语法比起 `not ... is` 更具有可读性。

不要这样：

```
if not val is None:
```

而是应该这样：

```
if val is not None:
```

4. 绑定标识符时考虑使用函数而不是 lambda 表达式

当把 lambda 表达式赋值给一个特定标识符时，考虑使用函数。`lambda` 是 Python 实现一行代码操作的一个关键字，然而比起使用 `def` 关键字编写一个函数，使用 `lambda` 可能不是一个好的选择。

不要这样：

```
square = lambda x: x * x
```

而是应该这样：

```
def square(val):
    return val * val
```

`def square(val)` 函数对象比起泛型 `lambda` 更有助于字符串表示和回溯。这种类型的使用减少了 `lambda` 的实用性。考虑在较大的表达式中使用 `lambda`，这样就不会影响代码的可读性。

5. 与 return 语句保持一致

如果函数有返回值，确保任何执行此函数的地方都返回该值。确保在函数退出的所有地方都有返回表达式是很好的做法。但是如果一个函数只需要执行一个操作而不用返回一个值，Python 默认返回 None。

不要这样：

```
def calculate_interest(principle, time rate):
    if principle > 0:
        return (principle * time * rate) / 100

def calculate_interest(principle, time rate):
    if principle < 0:
        return
    return (principle * time * rate) / 100
```

而是应该这样：

```
def calculate_interest(principle, time rate):
    if principle > 0:
        return (principle * time * rate) / 100
    else:
        return None

def calculate_interest(principle, time rate):
    if principle < 0:
        return None
    return (principle * time * rate) / 100
```

6. 更偏向使用 "".startswith() 和 "".endswith()

当你需要检查前缀和后缀时，考虑使用 `"".startswith()` 和 `"".endswith()` 而不是切片。slice 对切片字符串是一个非常有用的方法，在切片大字符串或执行字符串操作时可能会获得更好的性能。然而，在做诸如检查前缀或者后缀这种简单的事情的时候，请使用 startswith 或者 endswith，因为这会让读者很清楚你正在对字符串前缀或后缀进行检查。换言之，这将使你的代码更具有可读性和更整洁。

不要这样：

```
data = "Hello, how are you doing?"
if data[:5] == "Hello":
```

而是应该这样：

```
data = "Hello, how are you doing?"
if data.startswith("Hello")
```

7. 比较类型时更偏向使用 isinstance() 方法而不是 type()

当你对两个对象类型进行比较时，考虑使用 `isinstance()` 而不是 `type()`，因为 `isinstance()` 对子类返回 true。考虑这样一个场景：你传递的数据结构是类似于 `ordereddict` 的 `dict` 的子类，因为特定类型的数据结构，`type()` 会失败，然而 `isinstance()` 会识别出它是 `dict` 的子类。

不要这样：

```
user_ages = {"Larry": 35, "Jon": 89, "Imli": 12}
type(user_ages) == dict:
```

而是应该这样：

```
user_ages = {"Larry": 35, "Jon": 89, "Imli": 12}
if isinstance(user_ages, dict):
```

8. 比较 Boolean 值的 Python 化方法

在 Python 里，有很多比较 Boolean 值的方法

不要这样：

```
if is_empty = False
if is_empty == False:
if is_empty is False:
```

而是应该这样：

```
is_empty = False
if is_empty:
```

9. 为上下文管理器编写显式代码

当你使用 `with` 语句编写代码时，考虑使用函数来处理不同于资源获取和释放的其他任何操作。

不要这样：

```
class NewProtocol:
    def __init__(self, host, port, data):
        self.host = host
        self.port = port
        self.data = data

    def __enter__(self):
        self._client = Socket()
        self._client.connect((self.host,
                              self.port))
        self._transfer_data(data)

    def __exit__(self, exception, value, traceback):
        self._receive_data()
        self._client.close()

    def _transfer_data(self):
        ---

    def _receive_data(self):
        ---

con = NewProtocol(host, port, data)
with con:
    transfer_data()
```

而是应该这样：

```
#connection
class NewProtocol:
    def __init__(self, host, port):
        self.host = host
        self.port = port
    def __enter__(self):
        self._client = socket()
        self._client.connect((self.host,
                              self.port))

    def __exit__(self, exception, value, traceback):
        self._client.close()

    def transfer_data(self, payload):
       ...
    def receive_data(self):
       ...

with connection.NewProtocol(host, port):
    transfer_data()
```

在第一段代码中，Python 的方法 `__enter__` 和 `__exit__` 除了打开和关闭连接之外还执行了其他操作。最好显式地编写不同的函数来执行除了打开和关闭连接的操作。

10. 使用 Lint 工具来格式化代码

代码 linter 是一种格式化代码的重要工具。一个项目中保持一致的代码格式很有价值。

Lint 工具主要帮你解决以下问题：

- 语法错误。
- 结构化未使用过的变量或者向函数传递正确的参数。
- 指出违背 PEP8 规范的地方。

作为开发人员，Lint 工具使你的工作效率更高，因为它们通过在运行时查找问题来节省大量时间。Python 提供很多 Lint 工具，一些工具处理 lint 的特定部分，比如文档字符串（docstring）风格的代码质量，以及流行的 Python Lint 工具，比如 Flak8/Pylint 检查 PEP8 所有规则和 mypy 等专门用于检查 Python 类型标注的工具。

你可以将它们集成到你的代码中，也可以使用一个包含标准检查的代码，以确保你遵循 PEP8 风格规范。其中最著名的是 Flake8 和 Pylint。无论你选用哪一个工具，都要确保它符合 PEP8 的规则。

在 Lint 工具里可以找到一些特征：

- 遵循 PEP8 规则。
- 导入模块顺序。
- 命名（变量、函数、类、模块、文件等的 Python 命名）。
- 循环导入。
- 代码复杂度（通过分析代码行数、循环和其他参数来校验函数的复杂度）。
- 拼写检查。
- 文档字符串风格的检查。

运行 Lint 工具有不同的方式：

- 编码时使用 IDE。
- 在提交时使用预提交工具。
- 通过使用 Jenkins、CircleCI 等进行持续集成。

注意 有一些常见的实践肯定会提高你的代码质量。如果你想最大限度地利用好 Python 的良好实践，请查看 PEP8 官方文档。同样，阅读 GitHub 上的良好代码将帮助你明白如何编写更好的 Python 代码。

1.2 使用文档字符串

在 Python 中，文档字符串是使代码文档化最有效的方法。

文档字符串通常写在方法、类和模块的第一个语句。一个文档字符串是其对象的 `__doc__` 的特殊属性。

Python 官方推荐使用三个双引号来编写文档字符串。你可以在 PEP8 官方文档找到这些准则。让我们来简单讨论在 Python 代码里编写文档字符串的一些最佳实践，如代码清单 1-14 所示。

代码清单1-14　函数里的文档字符串

```
def get_prime_number():
    """Get list of prime numbers between 1 to 100."""
```

Python 推荐一种编写文档字符串的特殊方法。在本章，我们接下来会讨论编写文档字符串的不同的方法，然而这些不同方法遵循一些共同的规则。Python 定义如下规则：

- 即使字符串适合一行，也使用三重引号。当你想扩展时，这个规则很有用。
- 三重引号中的字符串前后不应有任何空行。
- 文档字符串中的语句应该使用句点（.）作为结束符。

类似地，可以使用 Python 多行文档字符串规则来编写多行文档字符串。在多行上

编写文档字符串是用更具描述性的方式注释代码。而不是写在每行的注释中，你可以在 Python 中利用 Python 多行文档字符串为代码编写描述性的文档字符串。这也有助于其他开发人员在代码中找到文档，而不是参考那些冗长乏味的文档，如代码清单 1-15 所示。

代码清单1-15　多行文档字符串

```
def call_weather_api(url, location):
    """Get the weather of specific location.

    Calling weather api to check for weather by using weather api
    and location. Make sure you provide city name only, country and
    county names won't be accepted and will throw exception if not
    found the city name.

    :param url:  URL of the api to get weather.
    :type url: str
    :param location:  Location of the city to get the weather.
    :type location: str
    :return: Give the weather information of given location.
    :rtype: str

    """
```

这里有一些地方要注意。

- ❑ 第一行是函数和类的简要描述。
- ❑ 最后一行末尾有个句点。
- ❑ 文档字符串中的简要描述和摘要之间有一行空白。

使用 Python 3 类型标注（typing）模块编写上述函数，如代码清单 1-16 所示。

代码清单1-16　使用类型标注的多行文档字符串

```
def call_weather_api(url: str, location: str) -> str:
    """Get the weather of specific location.

    Calling weather api to check for weather by using weather api
    and location. Make sure you provide city name only, country and
    county names won't be accepted and will throw exception if not
    found the city name.
    """
```

如果你在 Python 代码里使用了类型标注，就没必要写参数信息了。

正如我提到的文档字符串的种类很多，多年来，从多种方式引入了新的文档字符串的编写方法。虽然没有更好或推荐的方法来编写文档字符串，但是，请确保在整个项目中对文档字符串使用相同的风格，以便它们具有一致的格式。

有四种不同的方式来编写文档字符串。

- 下面是谷歌的文档字符串例子：

```
"""Calling given url.

Parameters:
    url (str): url address to call.

Returns:
    dict: Response of the url api.
"""
```

- 下面是一个重新构造的文本示例（Python 官方文档推荐）：

```
""" Calling given url.

:param url: Url to call.
:type url: str
:returns: Response of the url api.
:rtype: dict
"""
```

- 下面是 NumPy/SciPy 文档字符串的例子：

```
""" Calling given url.

Parameters
----------
url : str
    URL to call.

Returns
-------
dict
    Response of url.
"""
```

- 下面是一个 Epytext 例子：

```
"""Call specific api.

@type url: str
@param file_loc: Call given url.
@rtype: dict
@returns: Response of the called api.
"""
```

1.2.1　模块级文档字符串

应该在文件的顶部放置一个模块级文档字符串来简要描述模块的用法。这些注释同样应该放在 `import` 之前。模块文档字符串应该关注模块的目标，包括模块里的所有方法和类，而不是描述一个特定的方法或者类。如果你认为某个方法或类在使用该模块之前需要在较高的级别上让客户端知道，则仍然可以简要地描述该方法或类，如代码清单 1-17 所示。

代码清单1-17　模块文档字符串

```
"""
This module contains all of the network related requests. This
module will check for all the exceptions while making the
network calls and raise exceptions for any unknown exception.
Make sure that when you use this module, you handle these
exceptions in client code as:
NetworkError exception for network calls.
NetworkNotFound exception if network not found.
"""

import urllib3
import json

....
```

当你为模块编写文档字符串时应该考虑做如下事情：

- 写一个简要的描述模块的目的。
- 如果你想详述任何有助于读者了解模块的内容，像在代码清单 1-15 一样，你可以增加异常信息，但是要注意不要太详细。
- 考虑模块文档字符串作为提供模块描述信息的方式，而不是所有函数或者类的操作细节。

1.2.2　使类文档字符串具有描述性

类文档字符串主要用于简要描述类的使用及其目标。让我们看看一些示例如何编写类文档字符串，如代码清单 1-18 所示。

代码清单1-18　单行文档字符串

```
class Student:
"""This class handle actions performed by a student."""

    def __init__(self):
        pass
```

这个类只有一行文档字符串，它简要描述了 Student 类。如前所述，确保遵循一行的规则。让我们考虑代码清单 1-19 所示的类的多行文档字符串。

代码清单1-19　多行类文档字符串

```
class Student:
    """Student class information.

    This class handle actions performed by a student.
    This class provides information about student full name,
    age, roll-number and other information.

    Usage:
    import student

    student = student.Student()
    student.get_name()

    >>> 678998
    """

  def __init__(self):
      pass
```

上述类文档字符串是多行的；我们编写了更多关于学生类的用法，以及如何使用它。

1.2.3　函数文档字符串

函数文档字符串可以写在函数之后或函数之前。函数文档字符串主要关注描述函数的操作，如果你没有使用 Python 类型标注，那么也可以考虑包含参数，如代码清单 1-20 所示。

代码清单1-20　函数文档字符串

```
def is_prime_number(number):
    """Check for prime number.
```

```
    Check the given number is prime number or not by checking
    against all the numbers less the square root of given number.

    :param number:  Given number to check for prime.
    :type number: int
    :return: True if number is prime otherwise False.
    :rtype: boolean
    """
        ...
```

1.2.4　一些有用的文档字符串工具

有很多用于 Python 的文档字符串工具。文档字符串工具通过将文档字符串转换为 HTML 格式的文档文件来帮助文档化 Python 代码。这些工具还通过运行简单的命令而不是手动维护文档来帮助你更新文档。从长远来看，让它们成为开发流程的一部分，会使它们更加有用。

下面是一些有用的文档工具，每个文档工具都有不同的目标，因此你选择哪一个将取决于你的使用场景。

- Sphinx：http://www.sphinx-doc.org/en/stable/

  这是 Python 最流行的文档工具。这个工具将自动生成 Python 文档。它可以生成多种格式的文档文件。

- Pycoo：https://pycco-docs.github.io/pycco/

  这是为 Python 代码生成文档的快速方法。此工具的主要功能是并排显示代码和文档。

- Read the docs：https://readthedocs.org/

  这是开源社区中一个流行的工具，它的主要功能是为你构建版本和托管文档。

- Epydocs：http://epydoc.sourceforge.net/

  该工具基于 Python 模块的文档字符串生成 API 文档。

使用这些工具使长期维护代码更加容易，并帮助你保持代码文档的一致格式。


注意 文档字符串是 Python 的一个很好的特性，它可以使编写代码文档变得非常容易。尽早开始在代码中使用文档字符串将确保当你的项目使用数千行代码时，你不需要花费太多时间。

1.3 编写 Python 的控制结构

控制结构是所有编程语言的基本部分，对于 Python 也是如此。Python 有很多方法可以编写控制结构，但是有一些最佳实践可以保持 Python 代码的整洁。在本节中，我们将研究这些控制结构的 Python 最佳实践。

1.3.1 使用列表推导

列表推导是一种编写代码的方法，以类似于 Python `for` 循环的方式解决现有问题，然而它允许在有或没有 `if` 条件的情况下在列表中做到这一点。Python 中有多种方法可以从一个列表派生出另一个列表。Python 中实现这一点的主要工具是 `map`（映射）和 `filter`（筛选）方法。但是，建议使用列表推导的方法，因为与其他选项（如 `map` 和 `filter`）相比，列表推导更具可读性。

在本例中，你尝试使用 `map` 的版本查找数字的平方：

```
numbers = [10, 45, 34, 89, 34, 23, 6]
square_numbers = map(lambda num: num**2, num)
```

以下是列表推导版本：

```
square_numbers = [num**2 for num in numbers]
```

让我们看看另一个例子，在这个例子中，对所有的真值使用一个过滤器，以下是 `filter` 的版本：

```
data = [1, "A", 0, False, True]
filtered_data = filter(None, data)
```

以下是列表推导版本：

```
filtered_data = [item for item in filter if item]
```

正如你可能已经注意到的，列表推导版本比 `filter` 和 `map` 版本可读性强得多。

Python 官方文档也建议你使用列表推导而不是 `filter` 和 `map`。

如果 `for` 循环中没有复杂的条件或复杂的计算，则应考虑使用列表推导。但是如果你在一个循环中做很多事情，为了可读性，最好还是使用循环。

为了进一步说明在 `for` 循环中使用列表推导的意义，让我们看一个需要从字符列表中识别元音的示例。

```
list_char = ["a", "p", "t", "i", "y", "l"]
vowel = ["a", "e", "i", "o", "u"]
only_vowel = []
for item in list_char:
    if item in vowel:
        only_vowel.append(item)
```

以下是一个使用列表推导的例子：

```
[item for item in list_char if item in vowel]
```

如你所见，与使用循环相比，使用列表推导时，此示例的可读性更强且代码行更少。此外，循环还有额外的性能开销，因为每次都需要将项追加到列表中，而在列表推导中不需要这样做。

类似地，与列表推导相比，调用 `filter` 和 `map` 函数时需要额外的开销。

1.3.2　不要使用复杂的列表推导

你还需要确保列表推导不太复杂，否则会妨碍代码的可读性且更容易出错。

让我们考虑使用列表推导的另一个例子。列表推导对于一个条件下最多两个循环是可行的。除此之外，它可能会妨碍代码的可读性。

下面是一个例子，你希望在其中转置这个矩阵：

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]
```

把它转换成如下：

```
matrix = [[1,4,7],
          [2,5,8],
          [3,6,9]]
```

使用列表推导，你可能会如下编写代码：

```
return [[ matrix[row][col] for row in range(0, height) ] for
col in range(0,width) ]
```

这里的代码是可读的，而且使用列表推导是有意义的。你甚至可能希望以更好的格式编写代码，例如：

```
return [[ matrix[row][col]
          for row in range(0, height) ]
          for col in range(0,width) ]
```

当你具有如下的多个 `if` 条件时，可以考虑使用循环而不是列表推导：

```
ages = [1, 34, 5, 7, 3, 57, 356]
old = [age for age in ages if age > 10 and age < 100 and age is
not None]
```

上述代码中，很多事情都发生在一起，这很难阅读而且容易出错。在这里使用 `for` 循环而不是使用列表推导可能是一个好主意。

你可以考虑按如下方式编写此代码：

```
ages = [1, 34, 5, 7, 3, 57, 356]
old = []
for age in ages:
    if age > 10 and age < 100:
        old.append(age)
print(old)
```

如你所见，这有更多的代码行，但它更具有可读性而且也更整洁。

因此，一个好的经验规则是从列表推导开始，当表达式开始变得复杂或可读性开始受到妨碍时，转换为使用循环。

注意 明智地使用列表推导可以改进代码，但是过度使用列表推导可能会妨碍代码的可读性。因此，在处理复杂的语句时，遇到一个以上的条件或循环时，尽量不要使用列表推导。

1.3.3　应该使用 lambda 吗

在 lambda 对表达式有帮助的地方使用 lambda，而不是用它替换函数的使用。让我们考虑代码清单 1-21 中的示例。

代码清单1-21　使用不带赋值的lambda

```
data = [[7], [3], [0], [8], [1], [4]]
def min_val(data):
"""Find minimum value from the data list."""
    return min(data, key=lambda x:len(x))
```

在这里，代码使用 lambda 作为函数来查找最小值。但是，建议你不要将 lambda 用作如下匿名函数：

```
min_val = min(data, key=lambda x: len(x))
```

这里，使用 lambda 表达式用于 `min_val` 函数体计算，将 lambda 表达式作为函数编写会重复 `def` 的功能，这违反了 Python 以一种且只有一种方式做事的哲学。

PEP8 文档中提到了关于 lambda：

始终使用 `def` 语句，而不是将 lambda 表达式直接绑定到名称的赋值语句。

推荐：

```
def f(x): return 2*x
```

不推荐：

```
f = lambda x: 2*x
```

第一种形式表示结果函数对象的名称是“f”，而不是泛型“<lambda>”。这通常对于回溯和字符串表示更有用。使用赋值语句消除了 lambda 表达式相对于显式 def 语句所能提供的唯一好处（即它可以嵌入更大的表达式中）。

1.3.4　何时使用生成器与何时使用列表推导

生成器和列表推导的主要区别在于，列表推导将数据保存在内存中，而生成器则不这样做。

在下列情形下使用列表推导：

- 当需要多次遍历列表时。
- 当需要列出方法来处理生成器中不可用的数据时。
- 当没有大量的数据可以重复，并且你认为把数据保存在内存中不是问题时。

假设你希望从文本文件中获取文件行，如代码清单 1-22 所示。

代码清单1-22　从文本当中读取文件

```
def read_file(file_name):
"""Read the file line by line."""
    fread = open(file_name, "r")
    data = [line for line in fread if line.startswith(">>")]
    return data
```

在这里，文件可能太大，以至于列表中有很多行可能会影响内存并使代码变慢。因此，你可能需要考虑在列表上使用生成器。请参见代码清单 1-23 中的示例。

代码清单1-23　使用生成器从文档中读取文件

```
def read_file(file_name):
"""Read the file line by line."""
    with open(file_name) as fread:
        for line in fread:
            yield line

for line in read_file("logfile.txt"):
    print(line.startswith(">>")
```

在代码清单 1-23 中，不是使用列表推导将数据推入内存，而是一次读取每一行并执行操作。但是，可以传递列表推导以执行进一步的操作，查看它是否找到所有以 >> 开头的行，而生成器需要每次运行来查找以 >> 开头的行。

两者都是 Python 的优秀特性，如前所述使用它们会使代码具有良好的性能。

1.3.5　为什么不要在循环中使用 else

Python 循环中有一个 `else` 子句。基本上，你可以在代码中的 `for` 或 `while` 循环

之后使用 else 子句。只有当从循环中正常退出时，代码中的 else 子句才会运行。如果使用中断关键字[⊖]退出循环，则不会进入代码的 else 子句部分。

一个带有循环的 else 子句会有点混乱，这使得许多开发人员避免使用这个特性。考虑到正常流中 if/else 条件的性质，这是可以理解的。

让我们看下代码清单 1-24 中的简单示例，代码试图在列表上循环，并在循环的后面有一个 else 子句。

代码清单1-24　有for循环的else子句

```
for item in [1, 2, 3]:
    print("Then")
else:
    print("Else")

Result:
     >>> Then
     >>> Then
     >>> Then
     >>> Else
```

乍一看，你可能认为它应该只打印三个 Then 子句，而不是 Else，因为在 if/else 块的正常情况下，这将被跳过。这是一种看待代码逻辑的自然方法，然而，这种假设在这里是不正确的。如果使用 while 循环，这会变得更加混乱，如代码清单 1-25 所示。

代码清单1-25　有while循环的else子句

```
x = [1, 2, 3]
while x:
    print("Then")
    x.pop()
else:
    print("Else")
```

⊖　如 break、raise 等。——译者注

结果如下：

```
>>> Then
>>> Then
>>> Then
>>> Else
```

在这里，while 循环一直运行到列表不为空，然后运行 else 子句。

在 Python 中有这个功能是有原因的。一个主要的用例可以是在 for 和 while 循环之后有一个 else 子句，以便在循环结束后执行一个额外的操作。

让我们考虑代码清单 1-26 的例子。

代码清单1-26　有break的else子句

```
for item in [1, 2, 3]:
    if item % 2 = 0:
        break
    print("Then")
else:
    print("Else")
```

结果如下：

```
>>> Then
```

但是，有更好的方法来编写这段代码，而不是在循环之外使用 else 子句。可以将 else 子句与 break 条件在循环中一起使用，也可以不使用 break 条件。但是，不使用 else 子句也可以有多种方法来达到同样的目的。你应该在循环中使用 break 条件而不是 else 子句，因为这样会有被其他开发人员误解代码的风险，而且也很难一眼就理解代码。见代码清单 1-27。

代码清单1-27　使用break而不使用else子句

```
flag = True
for item in [1, 2, 3]:
    if item % 2 = 0:
        flag = False
        break
    print("Then")
if flag:
    print("Else")
```

结果如下：

```
>>> Then
```

这样的代码使阅读和理解变得更容易，并且在阅读代码时不易混淆。`else` 子句是编写代码的有趣方法，但是它可能会影响代码的可读性，因此避免使用它可能是解决问题的好方法。

1.3.6　为什么 range 函数在 Python 3 中更好

如果你使用过 Python 2，那么你可能使用过 `xrange`。在 Python 3 中，xrange 重命名为 `range`，并带有一些额外的特性。`range` 类似于 `xrange` 并生成一个迭代器。

```
>>> range(4)
range(0, 5)          # Iterable
>>> list(range(4))
[0, 1, 2, 3, 4]      # List
```

Python 3 的 `range` 函数中有一些新特性。与列表相比，`range` 的主要优点是它不将数据保存在内存中。与列表、元组和其他 Python 数据结构相比，`range` 表示一个不可变的可迭代对象，它总是占用较小且相同的内存量，而不管 `range` 的大小，因为它只存储 `start`、`stop` 和 `step` 值，并根据需要计算值。

有两件事情可以使用 `range` 做到，但是使用 `xrange` 却做不到。

- 你可以对两个 `range` 数据做比较。

```
>>> range(4) == range(4)
True
>>> range(4) == range(5)
False
```

- 你可以切片。

```
>>> range(10)[2:]
range(2, 10)
>>> range(10)[2:7, -1)
range(2, 7, -1)
```

`range` 有很多新特性，你可以从 https://docs.python.org/3.0/library/functions.html#range 查看更多详情。

另外，当需要处理代码中的数字而不是数字列表时，可以考虑使用 `range`，因为它比列表快得多。

在处理数字时，还建议在循环中尽可能多地使用迭代器，因为 Python 为你提供了一个类似 `range` 的方法来轻松地完成它。

不要这样：

```
for item in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(item)
```

而是应该这样：

```
for item in range(10):
    print(item)
```

第一个循环在性能上要昂贵得多，如果这个列表刚好足够大，那么由于内存状况和从列表中弹出的数字，会使代码运行变得很慢。

1.4 引发异常

异常能够报告代码中的错误。在 Python 中，异常是由内置模块处理的。对异常机制有较好的理解非常重要，理解何时何地使用它们不会使代码易出错。

异常可以毫不费力地暴露代码中的错误，因此不要忘记在代码中添加异常。异常帮助 API 或库的使用者了解代码的限制，以便他们在使用代码时也可以使用良好的错误机制。在代码的正确位置引发异常可以极大地帮助其他开发人员理解你的代码，并使第三方客户在使用你的 API 时感到满意。

1.4.1 习惯引发异常

你可能想知道在 Python 代码中何时何地引发异常。

我通常更喜欢在发现当前代码块的基本假设为 false 时抛出异常。当代码失败时，在 Python 中更倾向使用异常。即使你有一个连续的错误，你也可以只引发一个异常。

让我们考虑一下，你需要在代码清单 1-28 中让两个数字相除。

代码清单1-28　数字相除引发异常

```
def division(dividend, divisor):
"""Perform arithmetic division."""
    try:
        return dividend/divisor
    except ZeroDivisionError as zero:
        raise ZeroDivisionError("Please provide greater than 0
        value")
```

正如你在这段代码中看到的，每当你假设代码中可能有错误时，都会引发异常。这有助于在调用这段代码时，当出现 ZeroDivisionErrov 时产生异常，并以不同的方式进行处理。请参见代码清单 1-29。

代码清单1-29　不引发异常的除法

```
result = division(10, 2)

What happens if we return None here as:

def division(dividend, divisor):
"""Perform arithmetic division."""
    try:
        return dividend/divisor
    except ZeroDivisionError as zero:
        return None
```

如果调用者不处理调用 division（dividened，divisor）方法失败的情况，即使代码中有 ZeroDivisionError，也会在发生异常时从 division（dividened，divisor）方法返回 None，则当代码大小增加或需求增加时，将来很难诊断变化。在出现任何故障或异常时最好避免使用 division（divident，divisor）函数返回 None，使调用者更容易了解在函数执行期间失败的原因。当我们引发异常时，我们让调用者预先知道输入值不正确以及需要提供正确的值，并避免隐藏的 bug。

从调用者的角度来看，获取异常比返回值更方便，返回值是 Python 风格，用于指示出现故障。

Python 的信条是“请求宽恕比请求许可更容易”。这意味着你不必事先检查以确保不会得到异常，相反，如果你获取异常，你可以处理它。

基本上，你希望确保每当你认为代码中存在失败的可能性时都会引发异常，以便调用类能够优雅地处理它们，而不会感到意外。

换句话说，如果你认为你的代码无法合理运行，并且还没有找到答案，请考虑抛出异常。

1.4.2 使用 finally 来处理异常

在 Python 中，`finally` 块中的代码总是会被运行。`finally` 关键字在处理异常时非常有用，特别是在处理资源时。无论是否引发异常，都可以使用 `finally` 来确保关闭或释放文件或资源。即使没有捕获异常或没有要捕获的指定的异常，也是这样的。见代码清单 1-30。

代码清单1-30 finally关键字的使用

```
def send_email(host, port, user, password, email, message):
"""send email to specific email address."""
try:
    server = smtlib.SMTP(host=host, port=port)
    server.ehlo()
    server.login(user, password)
    server.send_email(message)
finally:
    server.quite()
```

上述代码中，使用 `finally` 处理异常有助于清理服务器连接中的资源，以防在登录或发送电子邮件时发生某种异常。

可以使用 `finally` 关键字来编写关闭文件的代码块[⊖]，如代码清单 1-31 所示。

代码清单1-31 使用finally关键字关闭文件

```
def write_file(file_name):
"""Read given file line by line"""
    myfile = open(file_name, "w")
    try:
```

⊖ 这并不是一种好方法，可以使用 `with` 语句来达到关闭文件的目的。——译者注

```
        myfile.write("Python is awesome")          # Raise
                                                   TypeError
    finally:
        myfile.close()             # Executed before TypeError
                                   propagated
```

上述代码中，你正在处理关闭 `finally` 块中的文件。无论是否有异常，`finally` 中的代码将始终运行并关闭该文件。

因此，当你希望执行某个代码块而不管是否存在异常时，你应该选择使用 `finally` 来执行。使用 `finally` 将确保明智地处理资源，此外还会让代码更整洁。

1.4.3　创建自己的异常类

当你正在创建一个 API 或库，或者正在处理一个希望定义自己的异常，以与项目或 API 一致的项目时，建议创建自己的异常类。这在诊断或调试代码时有极大帮助，同时也有助于使代码更整洁，因为调用者会知道抛出错误的原因。

假设在数据库中找不到用户时不得不引发异常，你想要异常类的名称反映错误的含意，异常类 `UserNotFoundError` 本身就解释了异常及其含义。

你可以在 Python 3+ 中定义自己的异常类，如代码清单 1-32 所示。

代码清单1-32　创建特定异常类

```
class UserNotFoundError(Exception):
"""Raise the exception when user not found."""
    def __init__(self, message=None, errors=None):
        # Calling the base class constructor with the parameter
        it needs
        super().__init__(message)
        # New for your custom code
        self.errors = errors

def get_user_info(user_obj):
"""Get user information from DB."""
    user = get_user_from_db(user_obj)
    if not user:
        raise UserNotFoundException(f"No user found of this id:
        {user_obj.id}")
```

```
get_user_info(user_obj)
>>> UserNotFoundException: No user found of this id: 897867
```

需要确保在创建自己的异常类时，异常类具有良好的边界并且是可描述的。确保只在找不到用户的地方使用 `UserNotFoundException`，并且能通知调用者在数据库中没有找到用户信息。为自定义的异常设置一组特定的边界可以更容易地诊断代码。当查看代码时，确切地知道为什么代码会抛出那个异常。

还可以为带有命名的异常类定义更广泛的作用域，但是该名称应该表示它处理特定类型的情况，如代码清单 1-33 所示。代码清单 1-33 显示了 `ValidationError`，你可以将其用于多个验证案例，但是它的范围是由与验证相关的所有异常定义的。

代码清单1-33　创建与范围相关的自定义的异常类

```
class ValidationError(Exception):
"""Raise the exception whenever validation failed.."""
    def __init__(self, message=None, errors=None):
        # Calling the base class constructor with the parameter
        it needs
        super().__init__(message)
        # New for your custom code
        self.errors = errors
```

与 `UserNotFoundException` 相比，`ValidationError` 的异常范围更广泛，可在你认为验证失败或没有有效输入时引发。但是，边界仍然由验证上下文定义，确保你知道异常的范围，并在该异常类的范围中发现错误时引发异常。

1.4.4　只处理特定的异常

当你捕获异常时，建议你仅仅捕获特定异常，而不是使用 `except:clause`(except：子句)。

```
except: or except Exception will handle each and every
exception, which can cause your code to hide critical bugs or
exceptions which you don't intend to.
```

（except：或者 except Exception 将处理所有的异常，它会导致代码隐藏不想隐藏的关键错误或异常。）

让我们看下下面的代码片段，它使用 try/catch 块中的 except 子句调用函数 get_even_list。

不要这样：

```
def get_even_list(num_list):
"""Get list of odd numbers from given list."""
    # This can raise NoneType or TypeError exceptions
    return [item for item in num_list if item%2==0]

numbers = None
try:
    get_even_list(numbers)
except:
    print("Something is wrong")

>>> Something is wrong
```

这类代码隐藏了一个异常，比如 NoneType 或 TypeError，这显然是代码中的一个 bug，客户端应用程序或服务将很难弄清楚它们为什么会收到“有问题”这样的消息。相反，如果你用正确的消息引发一个特定类型的异常，API 客户端会明确知道发生了什么问题。

在代码中使用 except 时，Python 内部将其视为 BaseException。有一个特定的异常非常有帮助，特别是在更大的项目代码里。

而是应该这样：

```
def get_even_list(num_list):
"""Get list of odd numbers from given list."""
    # This can raise NoneType or TypeError exceptions
    return [item for item in num_list if item%2==0]

numbers = None
try:
    get_even_list(numbers)
except NoneType:
    print("None Value has been provided.")
except TypeError:
    print("Type error has been raised due to non sequential
    data type.")
```

处理特定异常有助于调试或诊断问题。调用者将立即知道代码失败的原因，并强制

你添加代码来处理特定的异常。这也使得你的代码在调用时不易出错。

根据 PEP8 文档，在处理异常时，如下情况应使用 except 关键字：

- 当异常处理程序打印或记录回朔时，至少用户会意识到在这种情况下发生了错误。
- 当代码需要做一些清理工作，但却让异常随着 raise 向上传播时，try…finally 是处理这种情况的更好的方法。

注意 处理特定异常是编写代码时的最佳实践之一，特别是在 Python 中，因为这将帮助你在调试代码时节省大量时间。同时，它将确保你的代码快速失败，而不是在代码中隐藏 bug。

1.4.5 小心第三方的异常

在调用第三方 API 时，了解第三方库引发的所有异常非常重要。了解所有类型的异常可以帮助你调试问题。

如果你认为一个异常不太适合你的场景，可以考虑创建自己的异常类。在使用第三方库时，如果要根据应用程序错误来重命名异常名称或在第三方异常中添加新消息，可以创建自己的异常类。

让我们看看代码清单 1-34 中的 botocore 客户端库。

代码清单1-34 创建一个自定义异常类

```
from botocore.exceptions import ClientError

ec2 = session.get_client('ec2', 'us-east-2')
try:
    parsed = ec2.describe_instances(InstanceIds=['i-badid'])
except ClientError as e:
    logger.error("Received error: %s", e, exc_info=True)
    # Only worry about a specific service error code
    if e.response['Error']['Code'] == 'InvalidInstanceID.NotFound':
        raise WrongInstanceIDError(message=exc_info, errors=e)

class WrongInstanceIDError(Exception):
```

```
"""Raise the exception whenever Invalid instance found."""
    def __init__(self, message=None, errors=None):
        # Calling the base class constructor with the parameter
        it needs
        super().__init__(message)
        # New for your custom code
        self.errors = errors
```

这里考虑两件事：

- 当在第三方库中发现特定错误时，添加日志会使调试第三方库中的问题更容易。
- 在这里，你定义了一个新的错误类来定义自己的异常。你可能不想为每个异常都这样做，但是如果你认为创建一个新的异常类会使你的代码更整洁、更具有可读性，那么请考虑创建一个新的类。

有时很难找到正确的方法来处理第三方库 /API 抛出的异常。至少了解一些由第三方库抛出的常见异常，这样会让你在遇到 bug 时更容易定位问题所在。

1.4.6　try 最少的代码块

每当处理代码中的异常时，请尽量将 try 块的代码保持在最少。这使其他开发人员更清楚代码的哪一部分可能会抛出错误。拥有最少代码或可能在 try 块中抛出异常的代码也有助于更轻松地调试问题。没有用于异常处理的 `try/catch` 块可能会稍微快一些，但是如果不处理异常，则可能会导致应用程序失败。所以，有个好的异常处理会使你的代码无错误，并可以在线上环境中节省成本。

让我们看一个例子。

不要这样：

```
def write_to_file(file_name, message):
"""Write to file this specific message."""
    try:
        write_file = open(file_name, "w")
        write_file.write(message)
        write.close()
    except FileNotFoundError as exc:
        FileNotFoundException("Please provide correct file")
```

如果仔细查看前面的代码，你会发现有可能出现不同类型的异常，FileNotFound 或 IOError。

可以在一行上使用不同的异常，也可以在不同的 try 块中编写不同的异常。

而是应该这样：

```
def write_to_file(file_name, message):
"""Write to given file this specific message."""
    try:
        write_file = open(file_name, "w")
        write_file.write(message)
        write.close()
    except (FileNotFoundError, IOError) as exc:
        FileNotFoundException(f"Having issue while writing into
        file {exc}")
```

即使在其他行上没有出现异常的风险，也最好按如下方式在 try 块中编写最少代码。

不要这样：

```
try:
    data = get_data_from_db(obj)
    return data
except DBConnectionError:
    raise
```

而是应该这样：

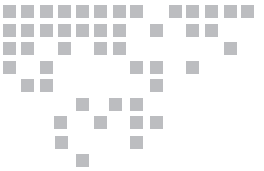
```
try:
    data = get_data_from_db(obj)
except DBConnectionError:
    raise
return data
```

这使得代码更整洁，并清楚地表明只有在访问 get_data_from_db 方法时，才会出现异常

1.5 小结

在本章中，学习了一些常见的实践，这些实践可以帮助你提高 Python 代码的可读性和简洁性。

此外，异常处理是用 Python 编写代码的最重要部分之一。对异常有很好的理解有助于维护应用程序。在大型项目中尤其如此，因为应用程序的不同运转部分由不同的开发人员处理，所以你有更多机会遇到各种生产问题。在代码的正确位置出现异常可以节省大量时间和金钱，特别是在调试生产中的问题时。日志记录和异常是任何成熟的软件应用程序中最重要的两个部分，因此提前对它们进行规划并将它们作为软件应用程序开发的核心部分将有助于编写更易于维护和可读的代码。



Chapter 2 第 2 章

数 据 结 构

数据结构是组成任何编程语言的基本模块。掌握好数据结构可以节省大量时间，并且使用它们可以使代码易于维护。Python 有许多用于存储数据的数据结构，了解在什么时候使用哪种数据结构，会对内存、易用性和代码性能方面有比较大的差异。

本章首先介绍一些常见的数据结构，并解释在代码中如何使用它们。之后还将介绍在特定情况下使用这些数据结构的优点。接下来，你可以思考一下在 Python 中使用字典的重要性。

2.1 常用数据结构

Python 中有许多主要的数据结构。在本节中，你将了解最常见的数据结构。对数据结构概念有较好的理解会对编写代码有很大的帮助。巧妙地使用数据结构可以提高代码的性能，减少 bug。

2.1.1 使用集合

集合（set）是 Python 中的基本数据结构，它也是最容易被忽视的。使用集合的主要好处是速度快。那么，让我们来看看集合的其他特性：

- 集合元素不能重复。
- 不支持索引访问集合里的元素。
- 集合使用散列表之后，可以在 O(1) 时间内访问元素。
- 集合支持一些常见的操作，如列表的切片和查询。
- 集合可以在插入元素时对元素进行排序。

考虑到这些约束条件，当你不需要数据结构中的通用功能时，可以使用集合，这将使你的代码在访问数据时速度更快。代码清单 2-1 展示了使用集合的示例。

代码清单2-1　使用集合访问数据

```
data = {"first", "second", "third", "fourth", "fifth"}
if "fourth" in data:
    print("Found the Data")
```

集合是使用散列表实现的，因此每当一个新项添加到集合中时，该项在内存中的位置由散列的对象确定。这就是为什么散列在访问数据时性能非常好。如果你有数千个元素，并且需要经常访问这些元素，那么使用集合的速度会更快，而不要使用列表。

接下来看另一个例子（代码清单 2-2），其中使用到的集合非常有用，它能够保证你的数据不会重复。

代码清单2-2　使用集合去重

```
data = ["first", "second", "third", "fourth", "fourth",
"fifth"]
no_duplicate_data = set(data)
>>> {"first", "second", "third", "fourth", "fifth"}
```

集合可用作字典的键，也可以使用集合用作其他数据结构的键，如列表（list）。

让我们思考代码清单 2-3 的示例，从数据库生成一个字典，其中 ID 值作为键，键的值为用户名字和姓氏。

代码清单2-3　集合用作姓氏和名字

```
users = {'1267':{'first': 'Larry', 'last':'Page'},
         '2343': {'first': 'John', 'last': 'Freedom'}}
```

```
ids = set(users.keys())
full_names = []
for user in users.values():
    full_names.append(user["first"] + "" + user["last"])
```

这将给出一组 ID 和一个全名列表。如你所见，集合可以从列表派生。

注意 集合是很有用的数据结构，当需要经常访问数字列表中的项时，并且在 O(1) 时间内访问这些项，可以考虑使用集合。在下次需要使用数据结构时，建议在考虑使用列表或元组之前优先考虑一下集合能不能满足需求。

2.1.2 返回和访问数据时使用 namedtuple

namedtuple 从根本上说是一个带有数据名称的元组。namedtuple 包含元组的全部特性，但也有一些元组没有的额外特性。使用 namedtuple 可以很容易创建轻量级对象类型。

namedtuple 使你的代码更加具有 Python 特色。

1. 访问数据

使用 namedtuple 访问数据可以提高代码的可读性。如果你想创建一个类，使其值在初始化后不会被更改。你可以创建一个类似代码清单 2-4 所示的类。

代码清单2-4 不可变的类

```
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

point = Point(3, 4, 5)
point.x
point.y
point.z
```

如果你不想改变 Point 类里面的值，并且更喜欢使用 namedtuple 编写，这将会

提高代码的可读性，如代码清单 2-5 所示。

代码清单2-5 namedtuple实现

```
Point = namedtuple("Point", ["x", "y", "z"])
point = Point(x=3, y=4, z=5)
point.x
point.y
point.z
```

正如你所见，此处代码的可读性比使用普通类的好，且代码行数更少。因为 namedtuple 使用的内存和元组一样，因此性能也和元组一样。

你可能很好奇为什么不使用 dict 代替 namedtuple，因为 namedtuple 更容易编写。

无论是否被命名，元组都是不可变的。namedtuple 通过使用名称访问而不是索引访问，使访问数据更加方便。namedtuple 还有一个严格的限制，即字段名必须是字符串。此外，namedtuple 不执行任何散列操作，因为它生成一个类型（type）。

2. 返回数据

通常你会以元组的形式返回数据。然而，你应该考虑使用 namedtuple 来返回数据，因为它使代码在没有太多上下文的情况下更具有可读性。我甚至建议，当数据从一个函数传递到另一个函数时，应该考虑是否可以使用 namedtuple，因为它使代码更具有 Python 特色和可读性。让我们思考代码清单 2-6。

代码清单2-6 将函数的值以元组的形式返回

```
def get_user_info(user_obj):
    user = get_data_from_db(user_obj)
    first_name = user["first_name"]
    last_name = user["last_name"]
    age = user["age"]
    return (first_name, last_name, age)

def get_full_name(first_name, last_name):
    return first_name + last_name

first_name, last_name, age = get_user_info(user_obj)
full_name = get_full_name(first_name, last_name)
```

那么，这些函数有什么问题呢？问题在返回值。值得注意的是，从数据库中获取数据之后，将返回用户的 first_name、last_name 和 age 的值。考虑到需要将这些值传递给其他函数，如 get_full_name 函数，当读取代码时，正在传递的这些值会给你带来视觉影响。如果像这样有更多的值需要传递时，很难想象在遵循代码规则的情况下会有多困难。如果将这些值封装到数据结构中，以此来提供上下文而不编写额外的代码，结果可能会更好。接下来，让我们使用 namedtuple 重写这段代码，这将会更有意义。如代码清单 2-7 所示。

代码清单2-7　使用namedtuple返回函数的值

```
def get_user_info(user_obj):
    user = get_data_from_db(user_obj)
    UserInfo = namedtuple("UserInfo", ["first_name", "last_
    name", "age"])

    user_info = UserInfo(first_name=user["first_name"],
                         last_name=user["last_name"],
                         age=user["age"])

    return user_info

def get_full_name(user_info):
    return user_info.first_name + user_info.last_name
user_info = get_user_info(user_obj)
full_name = get_full_name(user_info)
```

使用 namedtuple 编写代码会给出上下文，而不需要在代码中提供额外的信息。user_info 作为 namedtuple 给出了额外的上下文，而没有在函数 get_user_info 中返回时显式设置。因此，使用 namedtuple 可以使代码在长期运行状态下更加具有可读性和可维护性。

如果你需要返回 10 个值，通常情况下，可以考虑把返回的值放入 tuple 或 dict。当数据移动时，这两种数据结构都不是非常具有可读性。元组不会为 tuple 中的数据提供任何上下文或名称，并且 dict 不具有不可变性。当你不想在第一次赋值后更改数据时，这会对你产生约束。namedtuple 填补了这两个空白。

最后，如果想要将 namedtuple 类型转化为 dict 类型或者将列表转化为 namedtuple 类型，namedtuple 将为你提供简便的方法。因此，使用它们非常灵活，

下次创建不可变的数据或返回多个值的类时，可以考虑使用 namedtuple 以提高代码的可读性和可维护性。


注意 在你认为对象表示法会使你的代码更符合 Python 风格和更具备可读性的地方，你应该使用 namedtuple 而不是 tuple。当有多个值需要在上下文传递时，我通常会考虑使用 namedtuple。在这些情况下，namedtuple 可以满足要求，因为它使代码更具有可读性。

2.1.3 理解 str、Unicode 和 byte

了解 Python 语言中的一些基本概念将帮助你成为一个开发人员，同时会让你在处理数据方面成为一个更好的程序员。具体来讲，在 Python 中，简单地理解 str、Unicode 和 byte 能够帮助你在工作中很好地处理数据。由于 Python 内置库与其简单性，使代码在处理数据或处理与数据相关时更加简洁。

你可能已经知道，str 在 Python 中代表字符串类型，如代码清单 2-8 所示。

代码清单2-8 给str赋予不同的值

```
p = "Hello"
type(p)
>>> str

t = "6"
type(t)
>>> str
```

Unicode 几乎为所有语言中的每个字符串都提供了唯一的标识，如下所示：

```
0x59 : Y
0xE1 : á
0x7E : ~
```

Unicode 分配给字符的数字称为代码点（code point）。那么，Unicode 的作用是什么呢？

Unicode 的作用是为几乎所有语言的每个字符提供一个唯一的 ID，不论什么语言，

都可以对任何字符使用 Unicode 代码点。通常，Unicode 的格式是首位有 U，紧接着是至少 4 个十六进制数字。

因此，需要记住的是，Unicode 所做的一切是为每个字符分配一个名为代码点的数字 ID，这样就有了一个明确的指引。

将任何字符串映射到位模式，此过程称为编码（encoding）。这些位模式用于计算机的内存或磁盘上，可以通过多种方式对字符进行编码，最常见的方法就是 ASCII、ISO-8859-1 和 UTF-8。

Python 解释器使用 UTF-8 进行编码。

接下来，让我们简要地谈谈 UTF-8。UTF-8 将所有的 Unicode 字符映射到长度为 8、16、24 或 32 的位模式，与它们相应的是 1、2、3 或 4。

例如，Python 解释器将 a 转换为 01100001，并将 å 转换为 11000011 01011111 (0xC3 0xA1)。因此很容易理解 Unicode 为何如此有用。


注意 在 Python 3 中，所有字符串都是 Unicode 字符序列。所以，你不应该考虑将字符串编码为 UTF-8 或者将 UTF-8 解码为字符串。你仍然可以使用字符串编码方法将字符串转换为字节并将字节转换回字符串。

2.1.4 谨慎使用列表，优先使用生成器

迭代器非常有用，特别是用来处理大量数据。我见过一些代码，人们使用列表来存储数据序列，但是它会存在内存泄漏的风险，从而影响系统的性能。

让我们来思考代码清单 2-9。

代码清单2-9 使用列表返回素数

```
def get_prime_numbers(lower, higher):
    primes = []
    for num in range(lower, higher + 1):
        for prime in range(2, num + 1):
```

```
            is_prime = True
            for item in range(2, int(num ** 0.5) + 1):
                if num % item == 0:
                    is_prime = False
                    break

        if is_prime:
            primes.append(num)
print(get_prime_numbers(30, 30000))
```

这样的代码存在什么问题呢？第一，可读性低；第二，存在内存泄漏的风险，因为你正在向内存中存储大量的数据。如何使这段代码在可读性和性能方面更好？

在这里可以考虑使用生成器，它使用 `yield` 关键字来生成数字，并且可以使用它们作为迭代器来弹出值。让我们使用迭代器重写这个示例，如代码清单 2-10 所示。

代码清单2-10 对素数使用生成器

```
def is_prime(num):
    for item in range(2, int(math.sqrt(num)) + 1):
        if num % item == 0:
            prime = False
    return prime

def get_prime_numbers(lower, higher):
    for possible_prime in range(lower, higher):
        if is_prime(possible_prime):
            yield possible_prime
        yield False

for prime in get_prime_numbers(lower, higher):
    if prime:
        print(prime)
```

这段代码的可读性和性能都很好。另外，生成器会在无意中迫使你考虑重构代码。在这里，返回列表中的值会使代码更加冗余，而生成器避免了这个问题。

我观察到一个很常见的情况就是，迭代器在从数据库获取数据时非常有用，并且你不知道要获取多少行。当你试图将这些值保存在内存中时，可能启用内存工作模式。相反，尝试使用迭代器，它将立即返回一个值，然后转到下一行给出下一个值。

当你需要通过 ID 访问数据库以获取用户的年龄和姓名时，你知道数据库中 ID 的索

引和数据库中用户的总个数，即 1 000 000 000。大多数情况下，我发现在一些代码里面，开发人员试图使用列表从块（chunk）中获取数据。代码清单 2-11 给出了一个很好的例子。

代码清单2-11　访问数据库并将结果存储在列表中作为块

```
def get_all_users_age(total_users=1000):
    age = []
    for id in total_users:
        user_obj = access_db_to_get_users_by_id(id)
        age.append([user.name, user.age])
    return age

total_users = 1000000000
for user_info in range(total_users):
    info = get_all_users_age()
    for user in info:
        print(user)
```

在这里，你试图通过访问数据库来获取用户的年龄和姓名。但是，当系统中没有足够的内存时，这种方法可能不是很好。因为你正在选择一个你认为可以让内存安全（即内存足够大）的数字来存储用户信息，然而并不能保证这一点。Python 提供了一个生成器作为解决方案，来避免这些问题，并在代码中解决这些问题。你可以考虑重写这些代码，如代码清单 2-12。

代码清单2-12　使用迭代器方法

```
def get_all_users_age():
    all_users = 1000000000
    for id in all_users:
        user_obj = access_db_to_get_users_by_id(id)
        yield user.name, user.age

for user_name, user_age in get_all_users_age():
    print(user_name, user_age)
```

注意　生成器在 Python 中是非常有用的特性，因为生成器能够使代码具有可读性。同时，生成器还会强制开发者考虑代码的可读性。

2.1.5　使用 zip 处理列表

当有两个列表需要并行处理时，可以考虑使用 `zip`。它是 Python 的内置函数，并且非常高效。

假设数据库的用户表中有一个用户名字段和薪酬字段，你希望将它们合并到另一个列表中，并将其作为所有用户的列表返回。函数 `get_users_name_from_db` 和 `get_users_salary_from_db` 提供了一个用户列表和相应的用户薪酬，你怎么能把它们合并起来呢，其中一种方法如代码清单 2-13 所示。

代码清单2-13　合并列表

```
def get_user_salary_info():
    users = get_users_name_from_db()
    # ["Abe", "Larry", "Adams", "John", "Sumit", "Adward"]

    users_salary = get_users_salary_from_db()
    #  ["2M", "1M", "60K", "30K", "80K", "100K"]

    users_salary = []
    for index in len(users):
        users_salary.append([users[index], users_salary[index]])

    return users_salary
```

有没有更好的方法来解决这个问题呢？ Python 有一个名为 `zip` 的内置函数，可以轻松地处理这部分，如代码清单 2-14 所示。

代码清单2-14　使用zip

```
def get_user_salary_info():
    users = get_users_name_from_db()
    # ["Abe", "Larry", "Adams", "John", "Sumit", "Adward"]

    users_salary = get_users_salary_from_db()
    #  ["2M", "1M", "60K", "30K", "80K", "100K"]

    users_salary = []
    for usr, slr in zip(users, users_salary):
        users_salary.append(usr, slr)

    return users_salary
```

如果你有大量的数据，此处可以考虑使用生成器，而不是使用列表存储。`zip` 可以很容易并行处理合并两个列表，因此使用 `zip` 可以更高效地完成这些工作。

2.1.6 使用 Python 的内置函数

Python 有很多非常棒的内置库，考虑到这些库很多，因此本章不能全部列举出来。接下来将介绍一些基本的数据结构库，这些库可以对代码产生重大影响并提高代码的质量。

1. collections

这是使用最广泛的库之一，同时也有很有用的数据结构。特别是 `namedtuple`、`defaultdict` 和 `orderddict`。

2. csv

使用 `csv` 读取和写入 CSV 文件。`csv` 不是在读取文件时编写自己的方法，因此可以节省大量的时间。

3. datetime 和 time

毫无疑问 `datetime` 和 `time` 是最常用的两个库。事实上，你可能已经遇到过它们。如果没有遇到，在不同的场景中熟悉这些库中可用的不同方法是有益的，尤其是在处理计时问题时。

4. math

`math` 库中有很多很有用的方法，用来执行从基础到高级的数学计算。在寻找第三方库来解决数学问题之前，可以尝试看看这个库中是否已经有解决的方法。

5. re

`re` 库没有可以用来代替正则表达式解决问题的方案。事实上，`re` 是 Python 语言中

最好的库之一。如果你非常了解正则表达式，你可以变着花样使用 re 库。在使用正则表达式时，它提供了一些方法，可以更容易地执行一些比较困难的操作。

6. tempfile

这是一个很好的内置库，用来创建一次性临时文件。

7. itertools

在这个库中最有用的方法是排列和组合。但是，如果你想深入研究它，你将会发现使用迭代工具可以解决很多计算机问题。它还有一些很有用的函数，如 dropwhile、product、chain 和 islice。

8. functools

如果你是喜欢函数式编程的开发人员，functools 库非常适合你。它有许多函数，可以帮助你以一种更实用的方式来思考代码。最常见的部分就在这个库里。

9. sys 和 os

当你要执行一些特殊的操作系统或者 OS 级别的操作，可以使用 sys 库和 os 库。sys 库和 os 库提供了一些方法，使操作系统能够做出许多令人惊奇的事情。

10. subprocess

subprocess 库可以帮助你在系统上毫不费力地创建多个进程。该库易于使用，它创建多个进程并使用多个方法来处理它们。

11. logging

没有良好的日志记录功能，任何大型项目都不可能成功。Python 的 logging 库可以帮助你轻松地将日志添加到系统中。它有不同的方式输出日志，如控制台、文件和网络。

12. json

JSON 是通过网络和 API 传递信息的实际标准。Python 的 `json` 库非常适用于处理不同的场景。`json` 库接口易于使用，并且其文档也非常好用。

13. pickle

在日常编程中你可能不会使用它，但每当需要序列化和反序列化 Python 对象时，没有比 `pickle` 更好的库。

14. __future__

这是一个伪模块，它支持与当前解释器不兼容的新语言特性。因此，如果你想用未来的版本，你可能需要考虑在代码中使用它们。如代码清单 2-15 所示。

代码清单2-15　使用__future__

```
import __future__ import division
```

注意　Python 拥有丰富的库，可以为你解决许多问题。了解它们是弄清楚它们能为你做什么的第一步。熟悉 Python 内置库将帮助你长期运行。

现在，你已经在 Python 中探索了一些最常见的数据结构，让我们深入了解一下 Python 中最常见的数据结构之一：字典。如果你正在编写专业的 Python 代码，你肯定会使用字典，让我们更进一步了解它们。

2.2　使用字典

字典是 Python 中最常用的数据结构之一。字典是一种可以更快地访问数据的方法。Python 有很简洁的内置字典库，这也使它们更便于使用。在本节中，你将更紧密地接触字典中一些有用的特性。

2.2.1 何时使用字典与何时使用其他数据结构

当你需要映射数据时，可以考虑在代码中使用字典作为数据结构。如果你正在存储需要映射的数据，并且想要快速访问它，那么使用字典就是最明智的选择。但是，你不想所有的数据存储都使用字典。因此，例如一个例子，考虑到一种情况，当你需要一个类的额外机制或需要一个对象时，或者当你需要数据结构中的数据具有不可变性，可以考虑使用 `tuple` 或 `namedtuple`。考虑构建代码时所需的特定数据结构。

2.2.2 collections

`collections` 是 Python 中很有用的模块之一，也是高性能的数据结构。`collections` 具有许多接口，这些接口对于执行具有 `dictionary` 的不同任务是非常有用的。因此，让我们看看 `collections` 中一些主要的工具。

1. Counter

`Counter` 提供了一种便利的方法用来计算相同的数据，如代码清单 2-16 所示。

代码清单2-16 Counter

```
from collections import Counter

contries  = ["Belarus", "Albania", "Malta", "Ukrain",
"Belarus", "Malta", "Kosove", "Belarus"]
Counter(contries)
>>> Counter({'Belarus': 2, 'Albania': 1, 'Malta': 2, 'Ukrain':
1, 'Kosove': 1})
```

`Counter` 是 `dict` 的一个子类，它是一个有序集合，其中元素存储为字典的键，计数存储为键的值。这是计算值个数的最有效方法之一。`Counter` 有许多有用的方法，如 `most_common()`，顾名思义，返回最常见的元素及其个数。如代码清单 2-17 所示。

代码清单2-17 Counter中的most_count()方法

```
from collections import Counter

contries  = ["Belarus", "Albania", "Malta", "Ukrain",
"Belarus", "Malta", "Kosove", "Belarus"]
```

```
contries_count = Counter(contries)
>>> Counter({'Belarus': 2, 'Albania': 1, 'Malta': 2, 'Ukrain':
1, 'Kosove': 1})
contries_count.most_common(1)
>>> [('Belarus', 3)]
```

其他方法如 `elements()` 返回一个迭代器，其中元素的重复次数与计数次数相同。

2. deque

如果要创建队列和栈，可以考虑使用 `deque`。它允许从左到右追加值，而且 `deque` 还以相同的 O(1) 性能从任何一侧支持线程安全和内存效率高的追加（`append`）和弹出（`pop`）操作。`deque` 还有以下方法：`append(x)` 从右侧追加 x，`appendleft(x)` 从左侧追加 x，`clear()` 清除所有的元素，`pop()` 从右侧弹出元素，`popleft()` 从左侧弹出元素，`reverse()` 反转元素。让我们看看其中一些案例，如代码清单 2-18 所示。

代码清单2-18　deque

```
from collections import deque

# Make a deque
deq = deque("abcdefg")

# Iterate over the deque's element
[item.upper() for item in deq]
>>> deque(["A", "B", "C", "D", "E", "F", "G"])

# Add a new entry to right side
deq.append("h")
>>> deque(["A", "B", "C", "D", "E", "F", "G", "h"])
# Add an new entry to the left side
deq.appendleft("I")
>>> deque(["I", "A", "B", "C", "D", "E", "F", "G", "h"])

# Remove right most element
deq.pop()
>>> "h"

# Remove leftmost element
deq.popleft()
>>> "I"
# empty deque
deq.clear()
```

3. defaultdict

defaultdict工作方式与dict类似，因为它是dict的子类。用函数default_factory初始化defaultdict，该函数不带参数，并为不存在的键提供默认值。defaultdict不会像dict那样引发KeyError错误。任何不存在的键都会获得default_factory函数返回的值。

让我们看一个简单的例子，如代码清单2-19所示。

代码清单2-19 defaultdict

```
from collections import defaultdict

# Make a defaultdict
colors = defaultdict(int)

# Try printing value of non-existing key would give us default
values
colors["orange"]
>>> 0
print(colors)
>>> defaultdict(int, {"orange": 0})
```

4. namedtuple

collecions模块中使用最多的工具之一就是namedtuple。它是一个具有指定字段和固定长度的tuple子类。在代码中任何使用元组的地方都可以使用namedtuple。namedtuple是一个不可变的列表，可以更容易地读取代码和访问数据。在前面我已经讨论过namedtuple，所以请参考这个结果来了解它的更多信息。

5. ordereddict

使用ordereddict可以按特定顺序获取键。dict没有将给你的顺序作为插入顺序，这是ordereddict的主要特性。在Python 3.6+中，dict还具有在默认情况下，按插入顺序排序的特性。

让我们看一个例子，如代码清单2-20所示。

代码清单2-20 ordereddict

```
from collections import ordereddict

# Make a OrderedDict
colors = OrderedDict()

# Assing values
colors["orange"] = "ORANGE"
colors["blue"] = "BLUE"
colors["green"] = "GREEN"

# Get values
[k for k, v in colors.items()]
>>> ["orange", "blue", "green"]
```

2.2.3 有序字典、默认字典、普通字典

在前面的章节里已经涉及了一些关于字典的主题。现在，让我们仔细看看一些不同类型的字典。`OrderedDict` 和 `DefaultDict` 字典类型是 `dict` 类（普通字典）的子类，并添加了一些特性使它们与 `dict` 区别开来。然而，它们具有与普通字典相同的特性。在 Python 中使用这些字典类型是有原因的，接下来将讨论如何使用这些不同的字典来更好地利用这些库。

从 Python 3.6 开始，`dict` 按插入顺序排序，这实际上降低了 `OrderedDict` 的效率。让我们一起讨论一下 Python 3.6 版本之前的 `OrderedDict`。在将它们插入字典时，`OrderedDict` 会给出有序的值。有时候你可能希望代码可以以有序的方式访问数据，此时可以考虑使用 `OrderedDict`。与字典相比，`OrderedDict` 没有任何额外的成本，所以性能方面两者是相同的。

假设你希望在第一次引入编程语言时存储，可以使用 `OrderedDict` 在创建年份之前获取该语言的信息，如代码清单 2-21 所示。

代码清单2-21 OrderedDict

```
from collections import OrderedDict

# Make a OrderedDict
language_found = OrderedDict()
```

```
# Insert values
language_found ["Python"] = 1990
language_found ["Java"] = 1995
language_found ["Ruby"] = 1995

# Get values
[k for k, v in langauge_found.items()]
>>> ["Python", "Java", "Ruby"]
```

有时候，在访问或者插入字典的键时，你想给键分配默认值。在普通的字典中，如果键不存在，就会出现 KeyError。然而 defaultdict 会为你创建默认键，参见代码清单 2-22。

代码清单2-22　defaultdict

```
from collections import defaultdict

# Make a defaultdict
language_found = defaultdict(int)

# Try printing value of non-existing key
language_found["golang"]
>>> 0
```

在这里，当你调用 defaultdict 并尝试访问不存在的 golang 键时，内部 defaultdict 将调用函数对象（在 language_found 容器中是 int 函数）。这是在构造函数中传递的函数对象。它是一个可调用的对象，它包括函数和对象类型。所以，你传递的 int 和 list 是 defaultdict 中的函数。当你尝试访问不存在的键，它会调用已传递的函数，将其返回的值指定为新键的值。

正如你所了解，字典是 Python 中的键值集合。很多高级库，如 defaultdict 和 OrderedDict 正在字典的基础上构建，以添加一些没有额外成本的新特性。毫无疑问，dict 会稍微快一点，但是大多数情况下都会有忽视的差别。因此，在为这些问题编写自己的解决方案时，请考虑使用 defaultdict 和 OrderedDict。

2.2.4　使用字典的 switch 语句

Python 没有 switch 关键字。然而，Python 有许多特性以一种更整洁的方式使用

switch 的功能。你可以利用 dictionary 来创建一个 switch 语句，而且当你根据特定的标准有多个选项可供选择时，你也应该考虑以这种方式编写代码。

思考一个根据特定国家的税收规则计算每个县的税收的系统。有多种方法可以做到这一点，然而，拥有多个选项的最困难部分是在代码中不添加多个 if-else 条件。让我们看看如何更优雅地使用字典来解决这个问题。参见代码清单 2-23。

代码清单2-23　使用字典的switch语句

```
def tanzania(amount):
    calculate_tax = <Tax Code>
    return calculate_tax

def zambia(amount):
    calculate_tax = <Tax Code>
    return calculate_tax

def eritrea(amount):
    calculate_tax = <Tax Code>
    return calculate_tax

contry_tax_calculate = {
      "tanzania": tanzania,
            "zambia": zambia,
      "eritrea": eritrea,
}
def calculate_tax(country_name, amount):
    country_tax_calculate["contry_name"](amount)

calculate_tax("zambia", 8000000)
```

在这里，你只需要使用字典来计算税额，与使用特有的 switch 语句相比，这会使代码更加优雅而且可读性更强。

2.2.5　合并两个字典的方法

假设你有两个想合并的字典。与以前的版本相比较，在 Python 3.5+ 中这样做要简单得多。合并任何两个数据结构都是很棘手的，因为在合并数据结构时，你需要特别注意内存的使用和数据的丢失。如果你使用额外的内存来保存合并的数据结构，考虑到字典中数据的大小，你应该要了解系统的内存限制。

数据丢失也是一个关注点。你可能会发现，某些数据由于特定的数据结构限制而丢失，例如在字典中，你不能设置重复的键。因此，当你在字典之间执行合并操作时，请记住这些事情。

在 Python 3.5+ 中，你可以这样做，如代码清单 2-24 所示。

代码清单2-24 在Python 3.5+中合并字典

```
salary_first = {"Lisa": 238900, "Ganesh": 8765000, "John":
3450000}
salary_second = {"Albert": 3456000, "Arya": 987600}
{**salary_first, **salary_second}
>>> {"Lisa": 238900, "Ganesh": 8765000, "John": 345000,
"Albert": 3456000, "Ary": 987600}
```

然而，在 Python 3.5 之前的版本，你可以通过一些额外的工作来完成这个任务。请参见代码清单 2-25。

代码清单2-25 在Python 3.5之前的版本中合并字典

```
salary_first = {"Lisa": 238900, "Ganesh": 8765000, "John":
3450000}
salary_second = {"Albert": 3456000, "Arya": 987600}
salary = salary_first.copy()
salary.update(salary_second)
>>> {"Lisa": 238900, "Ganesh": 8765000, "John": 345000,
"Albert": 3456000, "Ary": 987600}
```

Python 3.5+ 有 PEP 448，它建议扩展使用 *（迭代解包装运算符）和 **（字典解包装运算符）。这肯定会提高代码的可读性。这不仅仅适用于字典，也适用于 Python 3.5 版本之后的列表。

2.2.6 优雅地打印字典

Python 有一个名为 `pprint` 的模块，因此你可以很好地打印内容。你需要导入 `pprint` 包来执行操作。`pprint` 允许你在打印任何数据结构时提供缩进选项。缩进将应用于你的数据结构，如代码清单 2-26 所示。

代码清单2-26　使用pprint打印字典

```
import pprint

pp = pprint.PrettyPrinter(indent=4)
pp.pprint(colors)
```

对于嵌套更多且有大量数据的复杂字典，这可能不能像预期的那样工作。为此你可以考虑使用 JSON，如代码清单 2-27 所示。

代码清单2-27　使用json打印字典

```
import json

data = {'a':12, 'b':{'x':87, 'y':{'t1': 21, 't2':34}}}
json.dumps(data, sort_keys=True, indent=4)
```

2.3　小结

数据结构是所有编程语言的核心。正如你在阅读本章所了解的，Python 提供了许多用于存储和操作数据的数据结构。Python 提供了各种形式的数据结构工具，用于对不同类型的对象或数据集执行各种操作。作为 Python 开发人员，了解不同类型的数据结构非常重要，这样你可以在编写应用程序时作出正确的决定，特别是在资源密集型的应用程序中。

我希望本章对你了解 Python 中一些最有用的数据结构有所帮助。熟悉不同类型的数据结构及其不同的运转状态会使你成为更好的开发人员，因为你可以在工具包中使用不同类型的工具。