

尚学堂·百战程序员

[www.itbaizhan.cn](http://www.itbaizhan.cn)

WROX  
A Wiley Brand

Professional Python

# Python高级编程



[美] Luke Sneeringer  
宋云剑 刘磊

著  
译

清华大学出版社

# 尚学堂 百战程序员

学习编写更出色的代码，让工作流程更加

平滑

www.itbaizhan.cn

Python是一门强大且快速增长的动态编程语言。虽然它提供了很多用于编写高级、简洁、可维护代码的工具，但是对于这些技术却一直没有一个清晰的解释。《Python高级编程》为介绍所有Python功能提供了一个入口。每个概念的完整阐述、关于应用程序的讨论、动手指南都将教会你设计更好的架构，以及编写使得应用程序更加健壮与高效的代码。如果你已经熟悉Python，并期望使用它提供的所有功能，那本书正是为你准备的。

## 主要内容

- ◆ 涵盖所有语言的功能，包括函数、如何应用装饰器、上下文管理器与生成器
- ◆ 介绍Python的类与对象模型、元类、类工厂以及抽象基类
- ◆ 验证如何操纵Unicode字符串，以及Python2与Python3字符串的区别
- ◆ 提供Python2与Python3差异的深入讲解，并阐述如何编写跨版本的代码
- ◆ 探讨单元测试、命令行界面工具以及新的异步编程库

## 作者简介

Luke Sneeringer是一位经验丰富的Python开发人员，他曾经为诸如FeedMagnet、May Designs以及Ansible在内的多家公司设计、架构和创建多个Python应用程序。他经常作为演讲嘉宾出席Python会议。

Sneeringer的书并没有像同类书那样讲述有关Python的方方面面，而是先着重阐述核心功能，并辅之以恰当的示例，然后才讲述对于这些核心功能所需了解的知识以及如何使用它们。总之，如果你未来会花更多时间在Python语言上，那么本书将是又一本绝佳的Python书籍。

MagPi, February 2016

清华大学出版社数字出版网站

**WQBook**   
www.wqbook.com

 **wrox**<sup>TM</sup>  
A Wiley Brand

ISBN 978-7-302-45285-0



9 787302 452850 >

定价：49.80元

尚学堂.百战程序员  
www.itbaizhan.cn

# Python 高级编程

[美] Luke Sneeringer 著

宋云剑 刘 磊 译

清华大学出版社

北京

# 尚学堂·百战程序员

www.itbaizhan.cn

Luke Sneeringer

Professional Python

EISBN: 978-1-119-07085-6

Copyright © 2016 by John Wiley & Sons, Inc., Indianapolis, Indiana

All Rights Reserved. This translation published under license.

Trademarks: Wiley, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Access is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2016-1650

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

## 图书在版编目(CIP)数据

Python高级编程 / (美)卢克·斯内因格(Luke Sneeringer)著；宋云剑，刘磊译。—北京：清华大学出版社，2016

书名原文：Professional Python

ISBN 978-7-302-45285-0

I. ①P… II. ①卢… ②宋… ③刘… III. ①软件工具—程序设计 IV. ①TP311.56

中国版本图书馆 CIP 数据核字(2016)第 246895 号

责任编辑：王军于平

封面设计：牛艳敏

版式设计：思创景点

责任校对：成凤进

责任印制：何芊

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈：010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者：三河市君旺印务有限公司

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185mm×260mm 印 张：16.5 字 数：412 千字

版 次：2016 年 11 月第 1 版 印 次：2016 年 11 月第 1 次印刷

印 数：1~3500

定 价：49.80 元

---

产品编号：067812-01

## 译者序

Python 是一门适用面极广的语言，被广泛应用于 Web 开发、网络开发、系统开发、科学计算、机器学习、数据分析、数据可视化等领域，而近些年数据科学的火爆更为 Python 的流行添砖加瓦。但让 Python 如此流行的更重要原因是：Python 具有极好的可读性。

在国内，开发人员常常漠视可维护性、可测试性和质量。这种不幸局面的最终结果是项目协作困难、代码难以维护、修改风险极高。尽管 Python 语言的设计者和 Python 社区都非常重视编写干净、可维护的代码，但是仍然很容易出现相反的局面。我相信造成这种结果主要是由于 Python 开发人员并没有对 Python 的高级特性进行深入了解，没有对 Python 编写的最佳实践进行学习。缺乏这些技能，或许可以编写出完成功能的代码，但编写完成优雅、可维护的代码就力不从心，就像学会少量文字可以让我们写字，却不能让我们写诗。

本书不是为 Python 初学者准备的，文中很多语法和使用情景都要求读者有相关的技术基础。比如第 I 部分关于函数的高级特性，如生成器、装饰符等；第 II 部分关于类的高级特性，如抽象类、类工厂，第 III 部分数据以及其他高级主题都需要你对 Python 基础有一定的理解。如果有其他语言的编程经验，也同样很有帮助。如果你有一定的编程经验，想要了解 Python 高级特性的缘起和内部工作机制，从而能够写出更优雅、更可读的代码，那么本书一定是你的不二之选。

在这里要感谢清华大学出版社的编辑，他们为本书的出版投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。

在本书翻译过程中，译者本着忠于原文的态度，在翻译过程中力求真实再现原文风貌，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。本书全部章节由宋云剑、刘磊翻译，参与本书翻译的还有邴奇、张毅、刘艳波、田冠华、李新、李宝安、毛瑞娟、刘轶宽。

译者

## 作者简介

Luke Sneeringer 曾为多家公司设计、架构、构建和贡献了大量 Python 应用程序，这些公司包括 FeedMagnet、May Designs 与 Ansible。此外，他经常作为讲师出席各种 Python 会议。他与妻子 Meagan 以及他们喜欢的猫和鱼共同住在美国德克萨斯州的奥斯汀。

## 技术编辑简介

Alan Gauld 是一位通过认证的开放组体系结构框架(The Open Group Architecture Framework, TOGAF)企业架构师，在电信与客户服务行业工作。他从 1974 年开始编程，从 1998 年开始使用 Python。他是两本有关 Python 书籍的作者。在业余时，他爱好徒步、摄影、旅行与音乐。

Elias Bachaalany 是一名计算机程序员、软件逆向工程师及技术作者。Elias 还与人合著了 *Practical Reverse Engineering* (Wiley, 2014) 与 *The Antivirus Hacker's Handbook* (Wiley, 2015)。在供职于 Hex-Rays S.A 期间，他改善了 IDA Pro 的脚本设施并为 IDAPython 项目贡献代码。

## 致 谢

如果没有编辑 Kevin Shafer 以及技术编辑 Alan Gauld 与 Elias Bachaalany，本书不会顺利出版。他们的努力使本书更加完善(并减少了其中的大量错误)。Wiley 出版社整个团队的杰出工作使得我的开始相对没有吸引力的手稿成为现在的图书。

尤其感谢 Jason Ford——我亲密的朋友和杰出的企业家，他给了我一份能够源源不断地找到乐趣的工作。他给了我专业性编写 Python 书籍的第一个机会，并使 Python 成为每天有趣的问题、有趣的辩论以及兴奋不已(嗯，兴奋的原因还包括薪水)的来源。

我还要对很多 Python 社区外的朋友表示感激，他们在过去这些年陪我一起工作或娱乐。有很多名字需要列举，我的良心不允许我遗漏任何一位：Mickie Betz、Frank Burns、David Cassidy、Jon Chappell、Diana Clarke、George Dupere、John Ferguson、Alex Gaynor、Jasmin Goedtel、Chris Harbison、Boyd Hemphill、Rob Johnson、Daniel Lindsley、Jeff McHale、Doug Napleone、Elli Pope、Tom Smith 与 Caleb Sneeringer。

感谢我的父母 Jim Sneeringer 与 Cheryl Sneeringer，他们教会我如何生活。

最后，如果没有一个完整的段落来感谢支持、奉献和爱我的妻子 Meagan，就不是完整的致谢。她说服我本书值得一写，并在写作过程的每一步都亲切地支持我。我非常幸福并且非常感激她陪伴在我生命中的每一天。

—Luke Sneeringer

# 尚学堂·百战程序员

欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！

[www.itbaizhan.cn](http://www.itbaizhan.cn)

## 前　　言

最近，Python 已经成为越来越多开发人员的首选语言。Python 的使用者遍布全球，他们将该语言用于各种目的。随着 Python 的广泛应用，越来越多的开发人员都在花时间编写 Python。

Python 稳定增长的原因在于它是一门强大的语言，但很多经验丰富的 Python 开发人员也仅仅了解该语言所能完成工作的冰山一角。

### 本书读者对象

本书的目标读者是那些已经使用 Python 工作并熟悉它，而且希望进一步学习的开发人员。本书假定读者已经完成过 Python 开发过程中的大多数基本任务(例如使用过 Python 交互式终端)。

如果你是希望寻求由中级到高级的 Python 语言功能的读者，就应该从头至尾阅读本书。

你也可能曾经使用过更高级的语言功能，或可能需要维护使用这些功能的代码，那么可以考虑将本书作为参考手册；或者处理具体实现代码时可以将本书作为索引，从而充实对高级功能的理解。

### 本书内容

本书涵盖所有最新版本的 Python(包括 Python 2 与 Python 3)。在编写本书时，最新的可用版本是 Python 3.4, Python 3.5 还是 beta 版。本书主要涵盖了 Python 2.6/2.7/3.3/3.4。本书提供的大多数代码既能在 Python 2 上运行，也能在 Python 3 上运行，Python 2 代码会特别标注出来。

此外，本书还占用一整章的篇幅深入讲解 Python 2 与 Python 3 的区别，其中提供了编写跨版本运行代码的建议，以及如何将代码移植到 Python 3。

本书主要集中在两个领域。第一个领域是语言功能本身。例如，本书用几章阐述关于 Python 类与对象模型工作机制的多个方面。第二个领域是作为标准库的一部分提供的模块。例如，本书用 3 章阐述了 3 个模块：asyncio、unittest 与 argparse。

## 本书结构

本书可以分为 4 个部分：

本书的前 3 章介绍 Python 中的函数。前两章分别阐述装饰器与上下文管理器，它们是用于修改或为增加功能封装函数的可重用方式。第 3 章则是关于生成器的内容，生成器是一种设计函数的方法，它可以使函数一次返回一个值，而不是提前创建一整列值并将其一次性返回。

第Ⅱ部分包含第 4~7 章，介绍 Python 类与语言对象模型。第 4 章阐述魔术方法，第 5 章和第 6 章分别阐述元类和类工厂，这是以强大方式构建类的两种方式。最后，第 7 章阐述抽象基类，解释 abc 模块，以及如何让类来声明实现的模式。

第Ⅲ部分包含第 8 章和第 9 章，介绍字符串与数据。第 8 章阐述如何在 Python 中使用 Unicode 字符串(与使用字节字符串相比较)，详细介绍了在 Python 2 与 Python 3 中字符串的区别。第 9 章则阐述正则表达式，包含了 Python 的 re 模块，以及如何编写正则表达式。

最后，第Ⅳ部分包含无法融入前三部分的所有内容。第 10 章深入介绍 Python 2 与 Python 3 的区别，以及如何编写能够兼容这两个平台的代码。第 11 章介绍单元测试，主要关注 unittest 模块。关于命令行界面(CLI)工具的第 12 章介绍有关 optparse 与 argparse 的内容，这两个 Python 模块用于编写命令行工具。第 13 章介绍 asyncio，它是一个新的异步编程库，在 Python 3.4 中被引入到 Python 标准库。第 14 章介绍代码风格。

## 使用本书的前提条件

首先，需要一台运行 Python 的计算机。

虽然在大多数章节中没有区别，但本书在方法上稍微有点以 Linux 为中心(在第 12 章区别最大)。本书的示例在 Linux 环境中运行，如果在 Windows 环境中运行，输出结果可能会有略微不同。

## 勘误表

尽管我们已经尽了各种努力来保证书中或代码中不出现错误，但是人无完人，错误在所难免。如果发现本书中的错误，如拼写错误或代码错误，请告诉我们，我们将不胜感激。通过发送勘误表，可以让其他读者避免几个小时的困惑，同时可以帮助我们提供更高质量的信息。

要在网站上找到本书的勘误表，请访问 <http://www.wrox.com> 网页，点击勘误表链接。在这个网页上，您能看到本书提供的和 Wrox 编辑发布的所有勘误表。

如果在本书勘误页面上找不到提供的勘误表，请访问 [www.wrox.com/contact/](http://www.wrox.com/contact/)

techsupport.shtml 网页，填写那个网页上的表格并且发给我们您发现的错误。我们将核对反馈信息，如果正确，我们将在本书的勘误表页面张贴该错误消息，并在本书后续版本中修正这一问题。

## P2P.wrox.com

要与作者和同行讨论，请加入 [p2p.wrox.com](http://p2p.wrox.com) 网站上的 P2P 论坛。这个论坛是一个基于 Web 的系统，可以张贴与 Wrox 图书相关的信息和相关技术，与其他读者和技术用户交流心得。该论坛提供电子邮件订阅功能，当论坛上有新贴发布时，可以给你传送你选择感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 <http://p2p.wrox.com> 上，你将发现许多不同的论坛，它们不仅有助于你阅读本书，而且还有助于你开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

- (1) 进入 [p2p.wrox.com](http://p2p.wrox.com)，单击 Register 链接。
- (2) 阅读使用协议，并单击 Agree 按钮。
- (3) 填写加入该论坛所需要的信息和自己希望提供的其他可选信息，单击 Submit 按钮。
- (4) 你会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

**注意：**不加入 P2P 就可以阅读论坛中的信息，但是必须加入论坛才能发布自己的信息。

加入论坛后，可以发布新消息和回复其他用户发布的信息。可以随时阅读这个网站上的信息。如果想收到特定论坛发来的新消息，单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

要想更多了解如何使用 Wrox P2P 的信息，一定要阅读 FAQ，了解有关论坛软件的工作情况，以及 P2P 和 Wrox 图书的许多常见问题的解答。要阅读 FAQ，只需要在任意 P2P 页面上单击 FAQ 链接即可。

## 目 录

### 第I部分 函数

第1章 装饰器 .....	3
1.1 理解装饰器 .....	3
1.2 装饰器语法 .....	4
1.3 在何处使用装饰器 .....	6
1.4 编写装饰器的理由 .....	6
1.5 编写装饰器的时机 .....	7
1.5.1 附加功能 .....	7
1.5.2 数据的清理或添加 .....	7
1.5.3 函数注册 .....	7
1.6 编写装饰器 .....	7
1.6.1 初始示例：函数注册表 .....	8
1.6.2 执行时封装代码 .....	9
1.6.3 装饰器参数 .....	16
1.7 装饰类 .....	20
1.8 类型转换 .....	23
1.9 小结 .....	25
第2章 上下文管理器 .....	27
2.1 上下文管理器的定义 .....	27
2.2 上下文管理器的语法 .....	28
2.2.1 with语句 .....	28
2.2.2 enter和exit方法 .....	28
2.2.3 异常处理 .....	29
2.3 何时应该编写上下文管理器 .....	30
2.3.1 资源清理 .....	30
2.3.2 避免重复 .....	31
2.4 更简单的语法 .....	38
2.5 小结 .....	39
第3章 生成器 .....	41
3.1 理解生成器 .....	41
3.2 理解生成器语法 .....	41

3.2.1 next函数 .....	43
3.2.2 StopIteration异常 .....	45
3.3 生成器之间的交互 .....	47
3.4 迭代对象与迭代器 .....	49
3.5 标准库中的生成器 .....	50
3.5.1 range .....	50
3.5.2 dict.items及其家族 .....	50
3.5.3 zip .....	51
3.5.4 map .....	52
3.5.5 文件对象 .....	52
3.6 何时编写生成器 .....	53
3.6.1 分块访问数据 .....	53
3.6.2 分块计算数据 .....	54
3.7 何时使用生成器单例模式 .....	54
3.8 生成器内部的生成器 .....	55
3.9 小结 .....	56

### 第II部分 类

第4章 魔术方法 .....	59
4.1 魔术方法语法 .....	59
4.2 可用的魔术方法 .....	60
4.2.1 创建与销毁 .....	61
4.2.2 类型转换 .....	63
4.2.3 比较 .....	65
4.3 其他魔术方法 .....	75
4.4 小结 .....	76
第5章 元类 .....	77
5.1 类与对象 .....	77
5.1.1 直接使用type .....	78
5.1.2 type链 .....	80
5.1.3 type的角色 .....	80
5.2 编写元类 .....	81

5.2.1 __new__方法.....	81	7.4.3 额外的抽象基类.....	124
5.2.2 __new__与__init__方法.....	81	7.5 小结 .....	124
5.2.3 元类示例.....	82	<b>第III部分 数 据</b>	
5.2.4 元类继承.....	82		
<b>5.3 使用元类.....</b>	<b>84</b>	<b>第8章 字符串与 Unicode.....</b> 127	
5.3.1 Python 3.....	85	8.1 文本字符串与字节字符串 .....	127
5.3.2 Python 2.....	85	8.2 包含非 ASCII 字符的字符串.....	132
5.3.3 需要跨版本执行的代码怎么办.....	85	8.2.1 观察区别 .....	132
5.3.4 跨版本兼容性在何时重要.....	86	8.2.2 Unicode 是 ASCII 的超集.....	133
<b>5.4 何时使用元类.....</b>	<b>87</b>	8.3 其他编码 .....	133
5.4.1 说明性类声明.....	87	8.4 读取文件 .....	135
5.4.2 类验证.....	88	8.4.1 Python 3 .....	135
5.4.3 非继承属性.....	90	8.4.2 Python 2 .....	137
<b>5.5 显式选择的问题.....</b>	<b>91</b>	8.4.3 读取其他源 .....	137
<b>5.6 meta-coding .....</b>	<b>92</b>	8.4.4 指定 Python 文件编码.....	137
<b>5.7 小结 .....</b>	<b>94</b>	<b>8.5 严格编码 .....</b>	<b>139</b>
<b>第6章 类工厂.....</b> 95			
6.1 类型回顾 .....	95	8.5.1 不触发错误 .....	139
6.2 理解类工厂函数 .....	96	8.5.2 注册错误处理程序 .....	140
6.3 决定何时应该编写类工厂 .....	98	<b>8.6 小结 .....</b>	<b>141</b>
6.3.1 运行时属性 .....	98	<b>第9章 正则表达式.....</b> 143	
6.3.2 避免类属性一致性问题 .....	102	9.1 使用正则表达式的原因 .....	143
6.3.3 关于单例模式问题的解答 .....	105	9.2 Python 中的正则表达式 .....	144
6.4 小结 .....	107	9.2.1 原始字符串 .....	144
<b>第7章 抽象基类 .....</b> 109			
7.1 使用抽象基类 .....	109	9.2.2 match 对象 .....	145
7.2 声明虚拟子类 .....	110	9.2.3 找到多个匹配 .....	145
7.2.1 声明虚拟子类的原因 .....	111	<b>9.3 基本正则表达式 .....</b>	<b>146</b>
7.2.2 使用 register 作为装饰器 .....	113	9.3.1 字符组 .....	146
7.2.3 __subclasshook__ .....	113	9.3.2 可选字符 .....	150
7.3 声明协议 .....	115	9.3.3 重复 .....	151
7.3.1 其他现有的方法 .....	115	<b>9.4 分组 .....</b>	<b>152</b>
7.3.2 抽象基类的价值 .....	118	9.4.1 零分组 .....	154
7.3.3 抽象属性 .....	120	9.4.2 命名分组 .....	155
7.3.4 抽象类或静态方法 .....	121	9.4.3 引用已经存在的分组 .....	156
7.4 内置抽象基类 .....	122	<b>9.5 先行断言 .....</b>	<b>157</b>
7.4.1 只包含一个方法的抽象基类 .....	122	<b>9.6 标记 .....</b>	<b>158</b>
7.4.2 可供集合使用的抽象基类 .....	123	9.6.1 不区分大小写 .....	158
		9.6.2 ASCII 与 Unicode .....	159

9.6.3 点匹配换行符	159	11.1.1 副本生态系统	183
9.6.4 多行模式	159	11.1.2 隔离的环境	184
9.6.5 详细模式	160	11.1.3 优点与缺点	185
9.6.6 调试模式	160	11.2 测试代码	185
9.6.7 使用多个标记	160	11.2.1 代码布局	186
9.6.8 内联标记	160	11.2.2 测试函数	186
9.7 替换	161	11.2.3 assert语句	188
9.8 已编译的正则表达式	162	11.3 单元测试框架	188
9.9 小结	163	11.3.1 执行单元测试	189
<b>第IV部分 其他高级主题</b>			
<b>第 10 章 Python 2 与 Python 3</b>	<b>167</b>	11.3.2 载入测试	192
10.1 跨版本兼容性策略	167	11.4 模拟	193
10.1.1 __future__ 模块	168	11.4.1 模拟函数调用	193
10.1.2 2to3	168	11.4.2 断言被模拟的调用	195
10.1.3 限制	170	11.4.3 检查模拟	197
10.1.4 six	170	11.4.4 检查调用	199
10.2 Python 3 中的变更	171	11.5 其他测试工具	199
10.2.1 字符串与 Unicode	171	11.6 小结	201
10.2.2 Print 函数	171	<b>第 12 章 CLI 工具</b>	<b>203</b>
10.2.3 除法	172	12.1 optparse	203
10.2.4 绝对与相对导入	173	12.1.1 一个简单的参数	203
10.2.5 “老式风格”类的移除	174	12.1.2 选项	205
10.2.6 元类语法	175	12.1.3 使用 optparse 的原因	212
10.2.7 异常语法	176	12.2 argparse	213
10.2.8 字典方法	178	12.2.1 本质	213
10.2.9 函数方法	179	12.2.2 参数与选项	214
10.2.10 迭代器	179	12.2.3 使用 argparse 的理由	220
10.3 标准库重定位	180	12.3 小结	221
10.3.1 合并“高效”模块	180	<b>第 13 章 asyncio 模块</b>	<b>223</b>
10.3.2 URL 模块	181	13.1 事件循环	223
10.3.3 重命名	181	13.2 协程	227
10.3.4 其他包重组	181	13.3 Future 对象与 Task 对象	229
10.4 版本检测	182	13.3.1 Future 对象	229
10.5 小结	182	13.3.2 Task 对象	230
<b>第 11 章 单元测试</b>	<b>183</b>	13.4 回调	231
11.1 测试的连续性	183	13.4.1 不保证成功	232
		13.4.2 幕后	232
		13.4.3 带参数的回调	233
		13.5 任务聚合	233

13.5.1 聚集任务.....	234	14.1.4 不要做重复工作.....	244
13.5.2 等待任务.....	235	14.1.5 让注释讲故事.....	245
13.6 队列.....	238	14.1.6 奥卡姆剃刀原则.....	245
13.7 服务器.....	240	14.2 标准.....	245
13.8 小结.....	242	14.2.1 简洁的规则.....	246
<b>第 14 章 代码风格 .....</b>	<b>243</b>	14.2.2 文档字符串.....	246
14.1 原则.....	243	14.2.3 空行.....	246
14.1.1 假定你的代码需要维护 .....	243	14.2.4 导入.....	247
14.1.2 保持一致性 .....	244	14.2.5 变量.....	247
14.1.3 考虑对象在程序中的存在方式， 尤其是那些带有数据的 对象.....	244	14.2.6 注释.....	248
		14.2.7 行长度.....	248
14.3 小结.....	249		

# 尚学堂·百战程序员

欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！

[www.itbaizhan.cn](http://www.itbaizhan.cn)

## 第一部分

### 函 数

---

- 第1章 装饰器
- 第2章 上下文管理器
- 第3章 生成器

# 尚学堂·百战程序员

欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！

[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 第 1 章

## 装 饰 器

装饰器是一个用于封装函数或类的代码的工具。它显式地将封装器应用到函数或类上，从而使它们选择加入到装饰器的功能中。对于在函数运行前处理常见前置条件(例如确认授权)，或在函数运行后确保清理(例如输出清除或异常处理)装饰器都非常有用。对于处理已经被装饰的函数或类本身，装饰器也很有用。例如，装饰器可以将函数注册到信号系统，或者注册到 Web 应用程序的 URI 注册表中。

本章将概要介绍什么是装饰器，以及装饰器如何与 Python 的函数和类交互。本章还列举了几个 Python 标准类库中常见的装饰器。最后，本章提供了编写装饰器并将其附加到函数和类上的指南。

### 1.1 理解装饰器

究其核心而言，装饰器就是一个可以接受调用也可以返回调用的调用。装饰器无非就是一个函数(或调用，如有`_call_`方法的对象)，该函数接受被装饰的函数作为其位置参数。装饰器通过使用该参数来执行某些操作，然后返回原始参数或一些其他的调用(大概以这种方式与装饰器交互)。

由于函数在 Python 中是一级对象，因此它们能够像其他对象一样被传递到另一个函数。装饰器就是接受另一个函数作为参数，并用其完成一些操作的函数。

实际上这很容易理解。考虑下面一个非常简单的装饰器。它仅仅为被装饰的调用点字符串附加了一个字符串。

```
def decorated_by(func):
    func.__doc__ += '\nDecorated by decorated_by.'
    return func
```

现在，考虑下面这个普通的函数：

```
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

函数的 docstring 是在第一行指定的字符串。假如在 Python 的 Shell 中对该函数运行 Help 命令，就能看到该字符串。下面是将装饰器应用到 add 函数的示例：

```
def add(x, y):
    """Return the sum of x and y."""
    return x + y
add = decorated_by(add)
```

以下是执行 help 命令得到的结果：

```
Help on function add in module __main__:

add(x, y)
    Return the sum of x and y.
    Decorated by decorated_by.
(END)
```

这里发生了什么？其实就是装饰器修改了 add 函数的 `_doc_` 属性，然后返回原来的函数对象。

## 1.2 装饰器语法

大多数时候开发人员使用装饰器来装饰函数，他们只对装饰过的最终函数感兴趣，而对于未装饰函数的引用最终就变得多余。

正因为如此(也是为了更整洁)，所以定义函数，给它赋一个特定的名称，然后立刻将装饰过的函数赋给相同的名称就不可取。

因此，Python 2.5 为装饰器引入了特殊的语法。装饰器的应用是通过在装饰器的名称前放置一个@字符，并在被装饰函数声明之上添加一行(不包含隐式装饰器的方法签名)来实现的。

下面来看一下如何优先将 `decorated_by` 装饰器应用到 `add` 方法：

```
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

再次注意，这里没有给`@decorated_by` 提供方法签名。假设该装饰器有一个单独的位置参数，而这个参数是装饰过的方法(在某些情况下，会看到一个带有其他参数的方法签名，该内容将在本章后面讨论)。

该语法允许在声明函数的位置应用装饰器，从而代码更容易阅读并且可以立即意识到应用了装饰器。可读性很重要。

## 装饰器应用的顺序

何时应用装饰器？使用@语法时，在创建被装饰的可调用函数后，会立刻应用装饰器。因此以上两个示例中所展示的将 decorated\_by 应用到 add 的方式几乎是一样的。首先创建 add 函数，然后立即使用 decorated\_by 将其封装起来。

需要注意的重要一点是，对某个可调用函数，可以使用多个装饰器(就像可以多次封装函数调用一样)。

但请注意，如果通过@语法使用多个装饰器，就需要按照自底向上的顺序来应用它们。起初觉得这违反直觉，但是恰恰说明了 Python 解释器实际所做的工作。

考虑下面这个应用了两个装饰器的函数：

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

首先发生的是由解释器创建 add 函数，然后应用 decorated\_by 装饰器。该装饰器返回了一个可调用函数(正如所有装饰器做的一样)，该函数被发送给 also\_decorated\_by 装饰器，also\_decorated\_by 也做了同样的事情，接下来结果被赋给 add 函数。

切记，装饰器 decorated\_by 的应用程序与下面的代码在语法上是相同的：

```
add = decorated_by(add)
```

前面两个装饰器示例与下面的代码在语法上相同：

```
add = also_decorated_by(decorated_by(add))
```

在这两种情况下，读取代码时首先读到装饰器 also\_decorated\_by。但是，装饰器的应用是自底向上的，这与函数的解析(由内向外)是相同的。工作也采用同样的原则。

在传统的函数调用情况下，解释器一定首先解析内部函数调用，以便有合适的对象或值发送给外部调用。

```
add = also_decorated_by(decorated_by(add)) # First, get a return value for
                                              # `decorated_by(add)`.
add = also_decorated_by(decorated_by(add)) # Send that return value to
                                              # `also_decorated_by`.
```

有了装饰器后，通常首先创建 add 函数。

```
@also_decorated_by
@decorated_by
def add(x, y):
```

```
"""Return the sum of x and y."""
return x + y
```

然后，调用装饰器`@decorated_by`，并将其作为`add`函数的装饰方法。

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

`@decorated_by`函数返回自己的可调用函数(在本例中，是`add`的修改版本)。该返回值在最后步骤发送给`@also_decorated_by`。

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

应用装饰器时需要记住一件重要的事情，即装饰器的应用是自底向上的。很多时候，顺序非常重要。

## 1.3 在何处使用装饰器

Python 标准库中包括很多包含装饰器的模块，并且很多常用工具和框架利用它们实现常用功能。

例如，如果要使一个类上的方法不需要这个类的实例，可以使用`@classmethod`或`@staticmethod`装饰器，它们是标准库的一部分。`mock`模块(用于单元测试，在 Python 3.3 以后被添加到标准库中)允许使用`@mock.patch`或`@mock.patch.object`作为装饰器。

一些常见工具也使用装饰器。`Django`(用于 Python 的常见 Web 框架)使用`@login_required`作为装饰器，允许开发人员指定用户必须登录才能查看一个特定页面，并且使用`@permission_required`应用更具体的权限限制。`Flask`(另一个常见的 Web 框架)使用`@app.route`充当指定的 URI 与浏览器访问这些 URI 时所运行的函数之间的注册表。

`Celery`(常见的 Python 任务运行工具)使用复杂的`@task`装饰器来标识函数是否为异步任务。该装饰器实际上返回`Task`类的实例，用来阐明如何使用装饰器制作一个方便的 API。

## 1.4 编写装饰器的理由

装饰器提供了一种绝妙的方式来告知，“在指定的位置，我想要这个指定的可重用的功能片段”。当装饰器编写得足够好时，它们是模块化且清晰明确的。

装饰器的模块化(可以很容易地从函数或类中使用和移除装饰器)使它们完美地避免重

复前置和收尾代码。同样，因为装饰器与装饰函数自身交互，所以它们善于在其他地方注册函数。

另外，装饰器是显式的。它们在所有需要它们的被调用函数中即席使用。因此这对于可读性很有价值，从而使调试也变得更方便。被应用的位置以及被应用的内容都非常明显。

## 1.5 编写装饰器的时机

在 Python 的应用程序和模块中有几个很好的有关编写装饰器的用例。

### 1.5.1 附加功能

大概使用装饰器最常见的理由是想在执行被装饰方法之前或之后添加额外的功能。这可能包括检查身份、将函数结果记录到固定位置等用例。

### 1.5.2 数据的清理或添加

装饰器也可以清理传递给被装饰函数的参数的值，从而确保参数类型的一致性或使该值符合指定的模式。例如，装饰器可以确保发送给函数的值符合指定类型，或者满足一些其他的验证标准(稍后将介绍相关示例，`@requires_ints` 装饰器)。

装饰器也可以改变或清除从函数中返回的数据。如果想让函数返回一个原生的 Python 对象(如列表或字典)，但是最终在另一端接收到序列化格式的数据(如 JSON 或 YAML)，这将是很有价值的用例。

有些装饰器实际上为函数提供了额外的数据，通常这些数据是附加参数的形式。`@mock.patch` 装饰器就是这种示例，因为它(除了原有参数之外)为函数提供了一个作为附加位置参数创建的 `mock` 对象。

### 1.5.3 函数注册

很多时候，在其他位置注册函数很有用——例如，在任务运行器中注册一个任务，或者注册一个带有信号处理器的函数。任何由外部输入或路由机制决定函数运行的系统都可以使用函数注册。

## 1.6 编写装饰器

装饰器仅仅是这样的函数：(通常)接受被装饰的可调用函数作为唯一参数，并且返回一个可调用函数(如前面几个普通示例所示)。

一个很重要的注意事项是：当装饰器应用到装饰函数时(而不是调用装饰器时)，会执行装饰代码本身。理解这一点至关重要，通过接下来的几个示例，会对其有清晰的理解。

### 1.6.1 初始示例：函数注册表

考虑下面这个简单的函数注册表：

```
registry = []
def register(decorated):
    registry.append(decorated)
    return decorated
```

`register`方法是一个简单的装饰器。它附加一个位置参数，该参数被装饰到注册表变量，然后返回未改变的装饰方法。任何接收 `register` 装饰器的方法都将把自身附加到 `registry`。

```
@register
def foo():
    return 3

@register
def bar():
    return 5
```

如果访问注册表，可以很容易遍历注册表并且在内部执行函数。

```
answers = []
for func in registry:
    answers.append(func())
```

`answers` 列表中此时包含了[3, 5]。这是因为函数是按顺序执行的，所返回的值被附加到 `answers`。

有几个在函数注册中有意义的用例。例如，将“钩子”添加到代码中，以便能够在关键事件前后执行自定义功能。这里有一个 `Registry` 类恰好能处理这种情况：

```
class Registry(object):
    def __init__(self):
        self._functions = []

    def register(self, decorated):
        self._functions.append(decorated)
        return decorated

    def run_all(self, *args, **kwargs):
        return_values = []
        for func in self._functions:
            return_values.append(func(*args, **kwargs))
        return return_values
```

在该类中值得注意的是 `register` 方法——即装饰器——仍然像以前一样工作。让绑定方法作为装饰器很好。它接收 `self` 作为第一个参数(就像其他绑定方法)，并且需要一个被装饰方法作为其额外的位置参数。

通过几个不同的注册表实例，可以拥有完全分离的注册表。甚至可以在多个注册表中注册同一个函数，如下所示：

```
a = Registry()
b = Registry()

@a.register
def foo(x=3):
    return x

@b.register
def bar(x=5):
    return x

@a.register
@b.register
def baz(x=7):
    return x
```

运行任意注册表的 run\_all 方法中的代码将得出如下结果：

```
a.run_all()    # [3, 7]
b.run_all()    # [5, 7]
```

注意，run\_all 方法能够接受参数，运行时将参数传入底层函数。

```
a.run_all(x=4)    # [4, 4]
```

## 1.6.2 执行时封装代码

这种装饰器非常简单，因为被装饰函数是在未经修改的条件下传递的。但是，执行被装饰方法时，可能希望运行额外的功能。为此，可以返回一个添加合适功能且(通常)在执行过程中调用被装饰方法的可调用函数。

### 1. 一个简单的类型检查

下面是一个简单装饰器，确保函数接收的所有参数都是整型，否则报错：

```
def requires_ints(decorated):
    def inner(*args, **kwargs):
        # Get any values that may have been sent as keyword arguments.
        kwarg_values = [i for i in kwargs.values()]

        # Iterate over every value sent to the decorated method, and
        # ensure that each one is an integer; raise TypeError if not.
        for arg in list(args) + kwarg_values:
            if not isinstance(arg, int):
                raise TypeError('%s only accepts integers as arguments.' %
                                decorated.__name__)
```

```
# Run the decorated method, and return the result.
return decorated(*args, **kwargs)
return inner
```

这里发生了什么？

装饰器自身是 `requires_ints`。它接收一个参数：`decorated`，即被装饰的可调用函数。装饰器唯一做的事情就是返回一个新的可调用函数，即本地函数 `inner`。该函数替代了被装饰方法。

通过声明一个函数并使用 `require_ints` 装饰它，可以看到实际结果：

```
@requires_ints
def foo(x, y):
    """Return the sum of x and y."""
    return x + y
```

注意运行 `help(foo)` 会得到的内容：

```
Help on function inner in module __main__:

inner(*args, **kwargs)
(END)
```

将名称 `foo` 赋给 `inner` 函数，而不是赋给原来被定义的函数。如果运行 `foo(3,5)`，将利用传入的这两个参数运行 `inner` 函数。`inner` 函数执行类型检查，然后运行被装饰的方法，仅仅是由于 `inner` 函数使用返回的被封装方法(`*args, **kwargs`)调用它，返回值是 8。如果缺少这个调用，被装饰方法将被忽略。

## 2. 保存帮助信息

用一个装饰器去细究函数的文本字符串或者截取 `help` 的输出并不可行。因为装饰器是用于添加通用和可重用功能的工具，所以相对模糊的注释是有必要的。而且一般来讲，如果使用函数的人尝试对函数执行 `help`，那么他只希望了解函数的核心信息，而不是关于 shell 的信息。

该问题的解决方案实际上是装饰器。Python 实现一个名为 `@functools.wraps` 的装饰器，将一个函数中的重要内部元素复制到另一个函数。

下面是同一个 `@requires_ints` 装饰器，但是它使用 `@functools.wraps` 来添加：

```
import functools

def requires_ints(decorated):
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        # Get any values that may have been sent as keyword arguments.
        kwarg_values = [i for i in kwargs.values()]

        # Iterate over every value sent to the decorated method, and
```

# www.itbaizhan.cn

```
# ensure that each one is an integer, raise TypeError if not.
for arg in args + kwarg_values:
    if not isinstance(i, int):
        raise TypeError('%s only accepts integers as arguments.' %
                        decorated.__name__)

    # Run the decorated method, and return the result.
    return decorated(*args, **kwargs)

return inner
```

装饰器本身几乎没有改变，除了在第二行增加了将@functools.wraps 装饰器应用到inner 函数。现在还必须导入 functools(在标准类库中)。还需要注意一些额外的语法。这个装饰器实际上使用了参数(稍后详细介绍)。

现在将装饰器应用到同一个函数，如下所示：

```
@requires_ints
def foo(x, y):
    """Return the sum of x and y."""
    return x + y
```

接下来就可以看到现在运行 help(foo)会发生什么：

```
Help on function foo in module __main__:

foo(x, y)
    Return the sum of x and y.
(END)
```

可以看到 foo 函数的文本字符串及其方法签名，这就是查看 help 时读到的内容。但是从原理上讲，仍然应用了@requires\_ints 装饰器，并且 inner 函数仍然在运行。

根据所运行的 Python 版本的不同，针对 foo 运行 help 得到的结果略微不同，尤其是在有关函数签名方面。之前的部分代表 Python 3.4 的输出结果。但是在 Python 2 中，提供的函数签名仍将是 inner(因此，参数为\*args 与\*\*kwargs 而不是 x 与 y)。

### 3. 用户验证

该模式的一个常见用例(即在运行装饰方法之前执行某种正确性检查)是用户验证。考虑一个期望将用户作为其第一个参数的方法。

用户应该是这个 User 与 AnonymousUser 类的实例，如下所示：

```
class User(object):
    """A representation of a user in our application."""

    def __init__(self, username, email):
        self.username = username
        self.email = email

class AnonymousUser(User):
```

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
"""An anonymous user; a stand-in for an actual user that nonetheless
is not an actual user.

"""

def __init__(self):
    self.username = None
    self.email = None

def __nonzero__(self):
    return False
```

这里的装饰器是一个强大的工具，用于隔离用户验证的样板代码。`@requires_user` 装饰器能够非常容易地验证你得到 User 对象，因此就不是一个匿名用户。

```
import functools

def requires_user(func):
    @functools.wraps(func)
    def inner(user, *args, **kwargs):
        """Verify that the user is truthy; if so, run the decorated method,
        and if not, raise ValueError.

        """
        # Ensure that user is truthy, and of the correct type.
        # The "truthy" check will fail on anonymous users, since the
        # AnonymousUser subclass has a `__nonzero__` method that
        # returns False.
        if user and isinstance(user, User):
            return func(user, *args, **kwargs)
        else:
            raise ValueError('A valid user is required to run this.')
    return inner
```

装饰器应用于常用的模板需求——验证一个用户已经登录到应用程序。当以装饰器的方式实现该功能时，程序更容易复用且更加容易维护，函数的装饰器应用程序看上去也更加简洁与明显。

注意，该装饰器仅仅能够正确包装函数或静态方法，如果包装绑定到类的方法，将会失败。这是由于装饰器忽略了将 `self` 发送给绑定方法作为第一个参数的期望。

## 4. 输出格式化

除了对输入到函数的参数进行检查，装饰器的另一用途是检查函数的输出。

当使用 Python 时，尽可能使用原生 Python 对象通常更加可行。但是你经常会希望得到序列化的输出格式(如 JSON 格式)。在所有相关函数的结尾手动将结果转换为 JSON 格式非常繁琐，也不是好主意。理想情况下，只有在必要时才应该使用 Python 结构，并且可能在序列化之前还需要应用其他样板代码(序列化或类似代码)。

装饰器为这一问题提供了完美且方便的解决方案。考虑下面这个接受 Python 输出结果并将其序列化为 JSON 格式的装饰器：

```
import functools
import json

def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        result = decorated(*args, **kwargs)
        return json.dumps(result)
    return inner
```

将`@json_output`装饰器应用到一个简单的函数，如下所示：

```
@json_output
def do_nothing():
    return {'status': 'done'}
```

在 Python shell 中执行函数，会得到如下结果：

```
>>> do_nothing()
{'status": "done"}
```

注意，你将得到包含有效 JSON 的字符串而不是字典。

装饰器的优美之处在于它的简洁。将其应用到函数后，一个返回 Python 字典、列表或其他对象的函数立刻就会变为返回序列化 JSON 格式的版本。

你可能要问，“为什么它如此有价值？”毕竟，为添加装饰器增加的一行代码仅仅移除了一行代码——调用`json.dumps`的代码。但是请考虑在应用程序的需求扩展时，装饰器所起的作用。

例如，如果希望捕获特定异常并以指定格式的 JSON 输出，而不是让异常冒泡并输出回溯该怎么办？正因为有了装饰器，所以可以非常容易地添加该功能：

```
import functools
import json

class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message

def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
```

# 尚学堂·百战程序员

第 I 部分 函数

www.itbaizhan.cn

```
"""
@functools.wraps(decorated)
def inner(*args, **kwargs):
    try:
        result = decorated(*args, **kwargs)
    except JSONOutputError as ex:
        result = {
            'status': 'error',
            'message': str(ex),
        }
    return json.dumps(result)
return inner
```

通过将错误处理作为参数传递给`@json_output` 装饰器，对于应用装饰器的任意函数都可以为其添加该功能。这就是装饰器的用武之地。对于代码的可移植性和重用性而言，它们是非常有用的工具。

现在，如果使用`@json_output` 装饰的函数抛出`JSONOutputError`，那么可以对该具体错误进行处理。函数示例如下：

```
@json_output
def error():
    raise JSONOutputError('This function is erratic.')
```

在 Python 解释器中运行`error` 函数的结果如下：

```
>>> error()
{'status": "error", "message": "This function is erratic."}
```

注意，只有`JSONOutputError` 异常类(及任意子类)可以接收此特殊处理。任何其他异常将被正常传递，并且生成一个回溯。考虑如下函数：

```
@json_output
def other_error():
    raise ValueError('The grass is always greener...')
```

当运行该函数时，将会得到期望的回溯，如下所示：

```
>>> other_error()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in inner
  File "<stdin>", line 3, in other_error
ValueError: The grass is always greener...
```

重用性和可维护性是装饰器价值的一部分。因为在整个应用程序中，装饰器被当成一个可重用的、普遍适用的概念(在本例中，是 JSON 序列化)。当存在适用该理念的新需求时，装饰器成为容纳该功能的容器。

从本质上讲，装饰器是避免重复的工具，它们的部分价值就是为代码的未来维护提供

钩子。

也可以不使用装饰器完成该功能。考虑下面这个需要登录用户的示例。编写一个实现该功能的函数并不难，只需要将该函数置于需要该功能的函数开头部分即可。该装饰器主要是一个语法糖，但语法糖也有价值。毕竟代码被读的次数远远多于被写的次数，而且很容易一眼就找到装饰器。

## 5. 日志管理

执行时封装代码的最后一个示例是通用的日志管理函数。考虑下面引发函数调用、计时然后将结果记录到日志的装饰器：

```
import functools
import logging
import time

def logged(method):
    """Cause the decorated method to be run and its results logged, along
    with some other diagnostic information.
    """
    @functools.wraps(method)
    def inner(*args, **kwargs):
        # Record our start time.
        start = time.time()

        # Run the decorated method.
        return_value = method(*args, **kwargs)

        # Record our completion time, and calculate the delta.
        end = time.time()
        delta = end - start

        # Log the method call and the result.
        logger = logging.getLogger('decorator.logged')
        logger.warn('Called method %s at %.02f; execution time %.02f '
                   'seconds; result %r.' %
                   (method.__name__, start, delta, return_value))

        # Return the method's original return value.
        return return_value
    return inner
```

当把装饰器应用到函数时，它会正常执行函数，但使用 Python 的 logging 模块在函数调用结束后会将相关信息记录到日志。现在，应用了装饰器的任意函数立刻就拥有了(最基本的)日志管理功能。

```
>>> import time
>>> @logged
... def sleep_and_return(return_value):
```

```
...     time.sleep(2)
...     return return_value
...
>>>
>>> sleep_and_return(42)
Called method sleep_and_return at 1424462194.70;
    execution time 2.00 seconds; result 42.
42
```

与前面的示例不同，该装饰器没有用显而易见的方式改变函数调用。不会出现将装饰器应用到一个函数后该函数返回不同结果这样的情况。在前面的示例中，如果检测不通过，则会引发异常或是修改返回结果；而本例中的装饰器并不明显，它默默完成后台工作，但在任何情况下都不会修改实际的返回结果。

## 6. 变量参数

值得注意的是，`@json_output` 和 `@logged` decorators 两个装饰器都提供了仅仅接受并以最小化检查、变量参数和关键字参数传递的 `inner` 函数。

这是一种重要的模式。它尤其重要的一种应用方式是很多装饰器被用于装饰普通函数以及类的方法。记住在 Python 中，在类中声明的方法接受一个额外的位置参数，按照惯例为 `self`。应用装饰器并不会改变这一点（这就是为什么前面示例中的 `quires_user` 装饰器无法应用于类中的绑定方法）。

例如，如果`@json_result` 用来装饰一个类的方法，调用 `inner` 函数并且它会接收类的实例作为第一个参数。实际上，这没有问题。在本例中，该参数只是 `args[0]`，然后它被顺利地传递给被装饰的方法。

### 1.6.3 装饰器参数

至此，一直没变的是所枚举的所有装饰器本身看上去并没有任何参数。正如所讨论的，有一个隐式参数——即被装饰的方法。

但是，有时让装饰器自身带有一些需要的信息，从而使装饰器可以用恰当的方式装饰方法十分有用。一个传递给装饰器的参数与一个在调用时传递给函数的参数的区别正在于此。传递给装饰器的参数只被处理一次，即在函数声明并被装饰时处理。与之相反，传递给函数的参数在该函数被调用时处理。

在前面的示例中，已经介绍了一个参数被发送给重复使用`@functools.wraps` 的装饰器。它接收一个参数——被封装的方法，该方法中的帮助和文档字符串以及类似的内容都将保存。

但是，装饰器有隐式的调用签名。它接收一个位置参数——被装饰的方法。那么，这又如何工作呢？

答案是这很复杂。回忆在运行时封装代码最基本的装饰器。这些装饰器在局部作用域声明一个内部方法后返回。这是由装饰器返回的可调用函数。该函数被赋值给函数名称。

接受参数的装饰器额外增加了一层封装。这是由于接受参数的装饰器并不是实际的装饰器，而是一个返回装饰器的函数，该函数接受一个参数(被装饰的方法)，然后装饰函数并返回一个可调用函数。

这听起来让人困惑。考虑下面的示例，其中对@json\_output 装饰器进行参数化，以允许输出结果缩进及按照键排序：

```
import functools
import json

class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message

def json_output(indent=None, sort_keys=False):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    def actual_decorator(decorated):
        @functools.wraps(decorated)
        def inner(*args, **kwargs):
            try:
                result = decorated(*args, **kwargs)
            except JSONOutputError as ex:
                result = {
                    'status': 'error',
                    'message': str(ex),
                }
            return json.dumps(result, indent=indent, sort_keys=sort_keys)
        return inner
    return actual_decorator
```

那么，这里发生了什么？为什么排序和缩进能够生效？

这是一个 json\_output 函数，它接受两个参数(indent 和 sort\_keys)，它返回另一个名为 actual\_decorator 的函数，该函数(顾名思义)意在用作装饰器。这是一个经典的装饰器——一个接受单独可调用函数(decorated)作为参数并返回一个可调用函数(inner)的可调用函数。

注意，对 inner 函数做了微小的修改，从而可以接受 indent 与 sort\_keys 参数。这些参数仿照 json.dumps 所接受的参数，因此调用 json.dumps 接受装饰器签名中提供给 indent 与 sort\_keys 的值，并在倒数第三行代码中将它们提供给 json.dump。

inner 函数是最终使用 indent 和 sort\_keys 参数的函数。这没有问题，因为 Python 的代码块的作用域规则允许这么做。即使在调用过程中为 inner 与 sort\_keys 参数的赋值不同也

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

没有问题，因为 inner 是一个局部函数(每次使用装饰器都返回不同的副本)。

json\_output 装饰器的应用如下所示：

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

现在如果运行 do\_nothing 函数，就会得到一个带有缩进和换行的 JSON 块，如下所示：

```
>>> do_nothing()
'{\n    "status": "done"\n}'
```

## 1. 为什么函数能被当成装饰器使用

如果 json\_output 不是装饰器，而是返回装饰器的函数，那么应用它的方式看起来为什么就像是在应用装饰器？在此，Python 解释器是如何使它生效的呢？

在此要详细解释的是操作顺序。这里的关键是操作顺序。更具体地说，函数在装饰器应用语法(@)之前调用(json\_output(indent=4))。因此，函数调用的结果被应用到装饰器上。

首先，就是解释器发现对 json\_output 函数的调用，然后解析该调用(注意，黑体字不包含@)：

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

json\_output 函数所做的就是定义另一个函数 actual\_decorator，然后返回它。该函数的返回值被提供给@，如下所示：

```
@actual_decorator
def do_nothing():
    return {'status': 'done'}
```

现在，运行 actual\_decorator。它声明了另一个局部函数 inner，然后返回它。如前所述，该函数被赋给名称 do\_nothing，也就是被装饰方法的名称。调用 do\_nothing 时，inner 函数被调用，执行被装饰的方法，然后输出带有合适缩进的 JSON 结果。

## 2. 调用签名很重要

当引入修改过的新函数 json\_output 时，意识到实际上你引入的是一个不可向后兼容的变更，这非常重要。

为什么？因为现在期望一个额外的函数调用。即使想要这个旧的 json\_output 函数行为，并且不需要任何可用参数的值，也仍然必须调用这个方法。

换言之，必须做如下操作：

```
@json_output()
def do_nothing():
    return {'status': 'done'}
```

注意圆括号，它们很重要，因为它们表明该函数已被调用(即使没有参数)，然后该结果被应用到@。

下面的代码与前面的代码不重复，也不等价：

```
@json_output  
def do_nothing():  
    return {'status': 'done'}
```

这里有两个问题。第一，代码在本质上是混乱的，因为如果习惯看到没有签名的装饰器，那么要求提供一个空的签名是违反直觉的。第二，如果旧的装饰器已经存在于应用程序中，则必须返回去编辑所有存在的调用。如果可能，应当尽量避免向后不兼容的修改。

在理想情况下，这个装饰器可以对3种不同类型的应用程序生效：

- @json\_output
- @json\_output()
- @json\_output(indent=4)

结果证明，通过基于装饰器所接受的参数修改装饰器的行为是可能的。记住，装饰器只是一个函数，具有其他函数的所有灵活性，它可以为了响应输入而完成所需要完成的工作。

考虑下面这个 json\_output 的更灵活的迭代：

```
import functools  
import json  
  
class JSONOutputError(Exception):  
    def __init__(self, message):  
        self._message = message  
  
    def __str__(self):  
        return self._message  
  
def json_output(decorated_=None, indent=None, sort_keys=False):  
    """Run the decorated function, serialize the result of that function  
    to JSON, and return the JSON string.  
    """  
    # Did we get both a decorated method and keyword arguments?  
    # That should not happen.  
    if decorated_ and (indent or sort_keys):  
        raise RuntimeError('Unexpected arguments.')  
  
    # Define the actual decorator function.  
    def actual_decorator(func):  
        @functools.wraps(func)  
        def inner(*args, **kwargs):  
            try:
```

```
result = func(*args, **kwargs)
except JSONOutputError as ex:
    result = {
        'status': 'error',
        'message': str(ex),
    }
return json.dumps(result, indent=indent, sort_keys=sort_keys)
return inner

# Return either the actual decorator, or the result of applying
# the actual decorator, depending on what arguments we got.
if decorated_:
    return actual_decorator(decorated_)
else:
    return actual_decorator
```

该函数尝试更加智能地判断当前它是否被作为装饰器使用。

首先，它确保不被以意料之外的方式调用。永远不要期望既可以接受被装饰的方法又可以接受关键字参数，因为装饰器在被调用过程中只能接受被调用方法作为唯一参数。

其次，它定义了 `actual_decorator` 方法，也就是(顾名思义)实际被返回或应用的装饰器。它定义的 `inner` 函数是由装饰器最终返回的函数。

最后，该装饰器会基于被调用的方式返回合适的结果：

- 如果设置了 `decorated_`，它将作为一个没有方法签名的纯装饰器被调用，它的职责是应用最终的装饰器并返回 `inner` 函数。再次注意，观察接受参数的装饰器实际上是如何生效的。首先，调用并解析 `actual_decorator(decorated_)` 函数，然后以 `inner` 作为唯一参数调用该函数的返回结果(结果必须是可调用函数，因为这是一个装饰器)。
- 如果没有设置 `decorated_`，那么这就是带有参数关键字的调用，并且函数必须返回一个实际的装饰器，该装饰器接受被装饰的方法并返回 `inner` 函数。该函数直接返回 `actual_decorator` 装饰器。然后 Python 解释器将它作为实际的装饰器应用(最终返回 `inner` 函数)。

为什么该技术很有价值？它允许维护已经应用的装饰器功能。这意味着不必更新每一处装饰器被应用的位置，但仍然可以获得在需要时添加参数的灵活性。

## 1.7 装饰类

记住，本质上来说装饰器是一个接受可调用函数的可调用函数，并返回一个可调用函数。这意味着装饰器可以被用于装饰类和函数(毕竟类本身也是可调用函数)。

装饰类有多种用途。它们十分有价值，因为正如函数装饰器那样，类装饰器可以与被装饰类的属性交互。类装饰器可以添加属性或将属性参数化，或是它可以修改一个类的 API，从而使类被声明的方式与实例被使用的方式不同。

你或许要问，“通过子类增加或修改一个类的属性是否合适”？通常答案是肯定的。但

是，在某些情况下另一种途径或许更加合适。例如，考虑一个会在应用程序中应用到多个类的通用功能，但该功能位于类层级的不同位置。

例如，考虑一个类的一个功能：每个实例都知道自身被实例化的时间，并按照创建时间排序。该功能对于很多不同的类都是通用的，实现方式是需要3个额外的属性——实例化的时间戳、`_gt_`方法和`_lt_`方法。

可以通过多种方式添加属性。下面是使用类装饰器的实现方式：

```
import functools
import time

def sortable_by_creation_time(cls):
    """Given a class, augment the class to have its instances be sortable
    by the timestamp at which they were instantiated.
    """
    # Augment the class' original `__init__` method to also store a
    # `__created` attribute on the instance, which corresponds to when it
    # was instantiated.
    original_init = cls.__init__

    @functools.wraps(original_init)
    def new_init(self, *args, **kwargs):
        original_init(self, *args, **kwargs)
        self.__created = time.time()
    cls.__init__ = new_init

    # Add `__lt__` and `__gt__` methods that return True or False based on
    # the created values in question.
    cls.__lt__ = lambda self, other: self.__created < other.__created
    cls.__gt__ = lambda self, other: self.__created > other.__created

    # Done; return the class object.
    return cls
```

在该装饰器中，首先保存了类的原始方法`_init_`的副本。你无须担心该类是否已有`_init_`方法。由于`object`对象包含`_init_`方法，因此该属性一定会存在。接下来，创建一个将会被赋值给`_init_`的新方法，该方法首先调用原始方法，之后完成一些额外工作，也就是将实例化时间戳赋值给`self._created`属性。

值得注意的是，该模式与前面在执行时封装代码的示例非常相似——创建一个封装另一个函数的函数，该函数的主要职责是执行被封装的函数，但同时也增加了一小部分其他功能。

值得注意的是，如果应用`@sortable_by_creation_time`装饰器的类定义了自己的`_lt_`与`_gt_`方法，那么该装饰器将会重写这两个方法。

如果类没有识别出`_created`属性被用于排序，那么该属性值自身并没有什么用处。因此，装饰器还加入了`_lt_`与`_gt_`魔术方法。这使`and`操作符基于这些方法的结果返回`True`

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

或 False。这同时也影响了 sorted 函数和其他类似函数的行为。

这是必须的，以使任意类的实例可以根据实例化时间排序。该装饰器可以应用到任何类，包括那些与其父类不相关的类。

下面是一个简单类的示例，该类可以根据实例的创建时间排序：

```
>>> @sortable_by_creation_time
... class Sortable(object):
...     def __init__(self, identifier):
...         self.identifier = identifier
...     def __repr__(self):
...         return self.identifier
...
>>> first = Sortable('first')
>>> second = Sortable('second')
>>> third = Sortable('third')
>>>
>>> sortables = [second, first, third]
>>> sorted(sortables)
[first, second, third]
```

记住，装饰器仅可以用于解决问题，这并不意味着装饰器就是解决问题的最恰当方法。

例如，对于本例来说，可以使用 mixin 或一个仅仅定义 `_init_`、`_lt_` 与 `_gt_` 方法的小类来实现同样的功能。使用 mixin 的简单方式如下所示：

```
import time

class SortableByCreationTime(object):
    def __init__(self):
        self._created = time.time()

    def __lt__(self, other):
        return self._created < other._created

    def __gt__(self, other):
        return self._created > other._created
```

可以使用 Python 的多重继承将 mixin 应用到类：

```
class MyClass(MySuperclass, SortableByCreationTime):
    pass
```

该方法与使用装饰器的方法相比有不同的优缺点。一方面，它不会直接重写由类或父类定义的 `_lt_` 与 `_gt_` 方法(当随后阅读代码时，很难发现装饰器已经重载了这两个方法)。

另一方面，很容易陷入这样的情况——由 `SortableByCreateTime` 提供的 `_init_` 方法没有执行。如果 `MyClass` 或 `MySuperclass` 或是任何 `MySuperclass` 的父类定义了 `_init_` 方法，那么将会执行父类的 `_init_` 方法。将类继承顺序反过来并不能解决该问题；改变继承顺序并不

会影响执行父类\_init\_方法这一结果。

与之相反，装饰器能够正确处理\_init\_方法，仅仅通过重写被装饰类的\_init\_方法，从而添加额外操作或是完全不修改\_init\_方法。

那么哪一种方法是正确的？这要视情况而定。

## 1.8 类型转换

至此，本章的讨论仅考虑了装饰器期望装饰函数并返回函数，或装饰器期望装饰类并返回类的情况。

但是，并没有必要保持这种关系的理由。装饰器的唯一需求是一个可调用函数接受一个可调用函数并返回一个可调用函数。没有要求必须返回同种类型的可调用函数。

一个更高级的装饰器用例实际上不这样做。尤其是，装饰器装饰一个函数，但返回一个类，这很有价值。当增加大量样板代码时，可以允许开发人员对于简单情况使用简单函数，而对于复杂情况则允许继承应用程序 API 中的类，在这些情况下，装饰器是非常有用的工具。

对此，在 Python 生态系统的流行任务执行器中使用的装饰器是 celery。celery 包提供的@celery.task 装饰器期望装饰一个函数，而该装饰器实际上会返回 celery 内部的 Task 类，而被装饰的函数在子类的 run 方法中被使用。

考虑下面一个使用类似方法的简单示例：

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')
    def identify(self):
        return 'I am a task.'
def task(decorated):
    """Return a class that runs the given function if its run method is
    called.
    """
    class TaskSubclass(Task):
        def run(self, *args, **kwargs):
            return decorated(*args, **kwargs)
    return TaskSubclass
```

这里发生了什么？装饰器创建了 Task 的一个子类并返回该类。该类是一个可调用函数并调用一个类创建该类的实例，返回该类的\_init\_方法。

这么做的价值在于为大量的扩展提供一个钩子。基本的 Task 类可以比 run 方法定义更

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

多的内容。例如，start方法或许可以异步执行任务。基本类或许也可以提供用于保存任务状态的方法。使用一个装饰器将一个函数替换为一个类，可以使开发人员只需要考虑所编写任务的实际内容，而装饰器会完成余下的工作。

可以通过接受一个该类的实例并执行它的identify方法来查看实际效果：

```
>>> @task
>>> def foo():
>>>     return 2 + 2
>>>
>>> f = foo()
>>> f.run()
4
>>> f.identify()
'I am a task.'
```

## 陷阱

这个特定方法会带来一些问题。尤其是，一旦任务函数被@task\_class装饰器装饰时，它就会变为一个类。

考虑下面以这种方式被装饰的简单任务函数：

```
@task
def foo():
    return 2 + 2
```

现在，尝试在解释器中直接执行该函数：

```
>>> foo()
<__main__.TaskSubclass object at 0x10c3612d0>
```

这是一件糟糕的事情。该装饰器以这样一种方式修改函数：如果开发人员执行该函数，它并不会完成任何人期望的工作。通常很难接受函数被声明为foo，并以复杂的foo().run()形式执行(而在本例中这样执行是必需的)。

解决该问题需要更多考虑装饰器与Task类的构建方式。考虑下面修订的版本：

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def __call__(self, *args, **kwargs):
        return self.run(*args, **kwargs)

    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')

    def identify(self):
        return 'I am a task.'
```

```
def task(decorated):
    """Return a class that runs the given function if its run method is
    called.
    """
    class TaskSubclass(Task):
        def run(self, *args, **kwargs):
            return decorated(*args, **kwargs)
    return TaskSubclass()
```

在此有一些关键区别。第一个区别是用于调用基类 Task 而新增的 `_call_` 方法。第二个区别(用于补充第一点)是`@task_class` 装饰器现在返回 `TaskSubclass` 类的实例而不是类本身。

能够接受该方案是由于对于装饰器来说唯一的要求是返回一个可调用函数，而 Task 新增的 `_call_` 方法意味着它的实例现在可以被调用。

为什么该模式很有价值？`Task` 类虽然非常简单，但是很容易看到如何将更多的功能添加进来，这对于管理和执行任务非常有用。

但是，该方法会在原始函数被直接调用时维持原始函数的功能，再次考虑被装饰的函数：

```
@task
def foo():
    return 2 + 2
```

现在，如果在解释器中执行该函数会返回什么？

```
>>> foo()
4
```

这正是你所期望的，这使该类成为更好的类与装饰器设计。在底层装饰器返回一个 `TaskSubclass` 实例。当该实例在解释器中被调用时，它的 `_call_` 方法被调用，该方法会调用 `run` 函数，从而调用原始函数。

你会发现，使用 `identify` 方法仍然会获得返回的实例：

```
>>> foo.identify()
'I am a task.'
```

现在有这样一个实例，直接调用它时，调用方法与原始函数并无不同。但是，该实例可以包含用于提供其他功能的其他方法和属性。

这个功能很强大，可以使开发人员编写一个可以容易且明显转换为类的函数，从而提供额外的函数调用方式或添加其他相关功能。这是一个有用的范例。

## 1.9 小结

对于编写可维护的、可读的 Python 代码而言，装饰器是极有价值的工具。装饰器的价值在于它是显式的且可重用的。装饰器提供了一种完美的方式来使用样板代码，一次编写

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

就可以将其用于多种不同情况。

这个有用的范例之所以能够生效，是因为 Python 的数据模型将函数与类作为一级对象提供，它们可以作为参数传递并可以像语言中的其他对象那样扩展。

另一方面，该模型也有缺点。尤其是装饰器的语法，虽然整洁且易于阅读，但使一个函数被封装到另一个函数这个事实变得模糊，从而给调试带来挑战。糟糕的装饰器或许会由于它们忽略了被封装可调用函数的本质而导致错误(例如，通过忽略绑定方法与被绑定函数之间的区别)。

此外，记住，与任何函数一样，解释器必须执行装饰器中的代码，这会对性能有影响。装饰器也不例外；留心使用装饰器的目的，这与留心编写其他代码的目的并无不同。

考虑使用装饰器作为一种封装不相关函数开头与结尾功能的方法。与之相似，装饰器是用于函数注册、发信号、某种情况下的类增强以及其他功能的强大工具。

第 2 章中讨论了上下文管理器，这是将需要在整个程序中一小段需要复用的功能以高效便捷的方式划分出来的方法。

# 第 2 章

## 上下文管理器

上下文管理器是装饰器的近亲。与装饰器类似的是，它们都是包装其他代码的工具。然而，装饰器包装用于定义的代码块(如函数或类)，而上下文管理器可以包装任意格式的代码块。

在大多数方面，上下文管理器与装饰器的作用等价(并且通常情况下很多项目所提供的 API 都允许使用其中任意一种方式，这部分内容将在本章后面讨论)。

本章将会介绍和解释上下文管理器的概念，将展示如何以及何时使用它，并列举几种处理上下文代码段中引起的异常的方式。

### 2.1 上下文管理器的定义

上下文管理器是一个包装任意代码块的对象。上下文管理器保证进入上下文管理器时，每次代码执行的一致性；当退出上下文管理器时，相关的资源会被正确回收。

值得注意的是，上下文管理器一定能够保证退出步骤的执行。如果进入上下文管理器，根据定义，一定会有退出步骤。即使内部代码抛出了异常，这点也成立。事实上，如果退出步骤处理异常合适，那么上下文管理器的退出代码为处理这类异常提供了一个机会(虽然并不强制要求这么做)。

因此，上下文管理器的功能类似于执行 `try`、`except` 和 `finally` 关键字。通常，这也是一种封装需要被重复使用的 `try-except-finally` 结构的有效机制。

上下文管理器或许被用到最多的就是——作为确保资源被正确清理的一种方式。

## 2.2 上下文管理器的语法

考虑一个适用上下文管理器的常见用例——打开文件。通过使用 Python 中的内置函数 `open` 打开文件。打开一个文件后，关闭文件就是你的责任了，如下所示：

```
try:  
    my_file = open('/path/to/filename', 'r')  
    contents = my_file.read()  
finally:  
    my_file.close()
```

使用 `finally` 子句确保无论发生什么，`my_file` 文件都将被关闭。假如读取文件时发生错误，或者其他地方出现问题，`finally` 子句仍然会执行，而 `my_file` 文件会关闭。

### 2.2.1 with 语句

那么，如何使用上下文管理器完成同样的功能——打开文件并确保其被正确关闭呢？上下文管理器在 Python 2.5 中引入，该版本新增了一个关键字：`with`。使用 `with` 语句可以进入上下文管理器。

碰巧，Python 的内置函数 `open` 也能作为上下文管理器使用。这段代码与你之前看到的完全相同：

```
with open('/path/to/filename', 'r') as my_file:  
    contents = my_file.read()
```

从本质上讲，实际上是 `with` 语句对其后代码进行求值(在本例中，就是调用 `open` 函数)。该表达式会返回一个对象，该对象包含两个特殊方法：`_enter_` 和 `_exit_`(稍后对其进行详细解释)。`_enter_` 方法返回的结果会被赋给 `as` 关键字之后的变量。

值得注意的是，在 `with` 后的表达式结果没有被赋给所谓的变量，这很重要。实际上，返回值没有赋给任何对象，只有 `_enter_` 的返回值会被赋给该变量。

简单性是使用上下文管理器的重要原因。然而更为重要的是，记住用于异常处理和清理的代码有时非常复杂，并且在不同的地方应用也非常麻烦。与装饰器相同的是，使用上下文管理器的关键原因在于避免代码重复。

### 2.2.2 enter 和 exit 方法

记住，`with` 语句的表达式的作用是返回一个遵循特定协议的对象。具体来说，该对象必须定义一个 `_enter_` 方法和一个 `_exit_` 方法，且后者必须接受特定参数。

除了传统的 `self` 参数，`_enter_` 方法不接受任何其他参数。当对象返回时该方法立即执行，然后如果有 `as` 变量 (`as` 子句是可选项)，返回值将被赋给 `as` 后面使用的变量。一般来说，`_enter_` 方法负责执行一些配置。

另一方面，`__exit__`方法带有3个位置参数(不包括传统的`self`参数)：一个异常类型、一个异常实例和一个回溯。如果没有异常，这3个参数全被设置成`None`，但如果在代码块内有异常发生，则参数被填充。

考虑下面这个简单的类，该类的实例被用作上下文管理器：

```
class ContextManager(object):
    def __init__(self):
        self.entered = False

    def __enter__(self):
        self.entered = True
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        self.entered = False
```

该上下文管理器并没有做什么工作。它只是返回自身和设置其`entered`变量，在进入时设置为`True`，退出时设置为`False`。

通过在 Python shell 中查看这个上下文管理器可以观察到这一点。如果创建一个新的`ContextManager`实例，将发现如预料的那样，它的`entered`值是`False`。

```
>>> cm = ContextManager()
>>> cm.entered
False
```

如果使用相同的`ContextManager`实例作为上下文管理器，观察它的`entered`属性会先变成`True`，然后在退出时再次变成`False`。

```
>>> with cm:
...     cm.entered
...
True
>>> cm.entered
False
```

如果在其他地方不需要使用`ContextManager`实例，可以用`with`语句将其实例化。该方法可行的原因在于它的`__enter__`方法只返回了它本身。

```
>>> with ContextManager() as cm:
...     cm.entered
...
True
```

### 2.2.3 异常处理

上下文管理器必须定义`__exit__`方法，该方法可以选择性地处理包装代码块中出现的异常，或者处理其他需要关闭上下文管理器状态的事情。

如前所述，`__exit__`方法必须定义3个位置参数：异常类型(本章中称为`exc_type`)、异常实例(在此称为`exc_instance`)以及回溯选择(在此称为`traceback`)。如果上下文管理器中的代码没有发生异常，则所有3个参数的值都为`None`。

如果`__exit__`方法接收一个异常，就有处理这个异常的义务。从根本上讲，这个方法有3个可选项：

- 可以传播异常(因为会在`__exit__`完成后再次抛出异常)。
- 可以终止异常。
- 可以抛出不同的异常。

可以通过让一个`__exit__`方法返回`False`实现异常的传播，或者通过让`__exit__`返回`True`终止异常。另外，如果`__exit__`抛出一个不同的异常，它将代替异常被发送出去。

本章将通过示例详细介绍这3个选项。

## 2.3 何时应该编写上下文管理器

有几个编写上下文管理器的常见理由。一般来说，这些情况都涉及确保某种资源以一种期望的方式被初始化和反初始化，或尽力去避免重复。

### 2.3.1 资源清理

打开和关闭资源(如文件或数据库连接)是编写上下文管理器的重要因素之一。确保出现异常时正确关闭资源往往很重要，这样能够避免最终随着时间的推移而产生很多的僵尸进程。

上下文管理器的优势就在于此。通过在`__enter__`方法中打开资源并返回它，可以保证`__enter__`方法能执行，同时也能在异常出现之前关闭这个资源。

考虑下面这个打开PostgreSQL数据库连接的上下文管理器：

```
import psycopg2

class DBConnection(object):
    def __init__(self, dbname=None, user=None,
                 password=None, host='localhost'):
        self.host = host
        self.dbname = dbname
        self.user = user
        self.password = password

    def __enter__(self):
        self.connection = psycopg2.connect(
            dbname=self.dbname,
            host=self.host,
            user=self.user,
```

```

    password=self.password,
)
return self.connection.cursor()

def __exit__(self, exc_type, exc_instance, traceback):
    self.connection.close()

```

在上下文管理器中，可以针对数据库执行查询操作并检索结果。

```

>>> with DBConnection(user='luke', dbname='foo') as db:
...     db.execute('SELECT 1 + 1')
...     db.fetchall()
...
[(2,)]

```

但是，只要上下文管理器存在，分配给 db 的数据库指针就会被关闭，然后再次查询也就无法成功。

```

>>> with DBConnection(user='luke', dbname='foo') as db:
...     db.execute('SELECT 1 + 1')
...     db.fetchall()
...
[(2,)]
>>> db.execute('SELECT 1 + 1')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
psycopg2.InterfaceError: cursor already closed

```

这里发生了什么？上下文管理器创建了一个 psycopg2 连接对象并且返回另一个指针，开发人员可以使用该指针与数据库交互。尽管如此，当上下文管理器存在时，保证连接是处于关闭状态仍很重要。

如上所述，这一点很重要，因为占用数据库的连接不仅消耗内存，而且在应用主机和数据库主机上它们也会打开文件或端口。此外，有些数据库也有最大连接数的阈值。

还请注意，与之前的示例不同，这个上下文管理器不仅仅在 `_enter_` 方法的结尾返回自身，还返回一个数据库的指针。这个示例很有用，但它，执行的仍然是上下文管理器的 `_exit_` 方法。

大多数与数据库相关的框架都会处理数据库连接的打开或关闭操作，但原则依然是：如果打开一个资源，一定要确保正确地关闭它，此时上下文管理器是一个十分优秀的工具。

### 2.3.2 避免重复

当提到避免重复时，异常处理是最为常见的。上下文管理器能够传播和终止异常，这使得最适合将它与 `except` 子句放到同一个地方定义。

#### 1. 传播异常

`_exit_` 方法只是向流程链上传播异常，这是通过返回 `False` 实现的，根本不需要与异

# 尚学堂·百战程序员

第I部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

常实例交互。考虑下面的上下文管理器：

```
class BubbleExceptions(object):
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        if exc_instance:
            print('Bubbling up exception: %s.' % exc_instance)
        return False
```

在上下文管理器中运行普通代码块(不抛出异常)将不会做什么特别的事情。

```
>>> with BubbleExceptions():
...     5 + 5
...
10
```

另一方面，该代码块实际上抛出了一个异常：

```
>>> with BubbleExceptions():
...     5 / 0
...
Bubbling up exception: integer division or modulo by zero.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
```

在此有两件值得注意的重要事情。第一个输出行(以 Bubbling up exception: integer...开头)由`__exit__`方法自身产生。它对应于`__exit__`方法第二行中的`print`语句。这意味着`__exit__`方法的确运行了且已完成。因为该方法返回了`False`，所以被首先发送给`__exit__`的异常只是被重新抛出了。

## 2. 终止异常

如前所述，`__exit__`方法拥有的另一个选项就是终止它所接收的异常。下面的上下文管理器终止所有可能发送给`__exit__`方法的异常(但是永远也不要这样做)：

```
class SuppressExceptions(object):
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        if exc_instance:
            print('Suppressing exception: %s.' % exc_instance)
        return True
```

这段代码的大部分与之前的`BubbleExceptions`类的代码相似，主要区别在于现在`__exit__`方法返回的是`True`而不是`False`。

下面这个普通的示例代码保持不变，没有引发任何异常：

```
>>> with SuppressExceptions():
...     5 + 5
...
10
```

但如果想要引发异常，可以做一些处理，所看到的结果将会不同：

```
>>> with SuppressExceptions():
...     5 / 0
...
Suppressing exception: integer division or modulo by zero.
```

注意，首先且最明显的事情是回溯消失了。由于异常被`_exit_`方法处理了(终止)，因此程序没有引发异常，继续执行。

第二件要注意的事情就是没有返回任何值。当进入解释器时，尽管表达式`5+5`返回了`10`，但引发异常的表达式`5/0`根本不会显示值。异常在计算该值的过程中引发，从而触发`_exit_`的运行。实际上永远不会返回任何值。另外，值得注意的是任何出现在`5/0`后面的代码都不会再执行。

但如你所料，在上下文块内定义的异常处理器都会在上下文块结束之前处理。上下文块内被认为已经处理的异常不会发送给`_exit_`方法。

考虑下面的示例：

```
with SuppressExceptions():
    try:
        5 / 0
    except ZeroDivisionError:
        print('Exception caught within context block.')
```

如果运行该例，将会输出消息“Exception caught within context block.”，并且不会将异常发送给`_exit_`。

尽管传播异常相对比较简单，但是对于终止异常应当总是小心。终止太多的异常会使调试代码极其困难。简单地终止所有异常与一个`try`块的功能基本上是等价的，如下所示：

```
try:
    [do something]
except:
    pass
```

可以这样说，这种情况是很不明智的。

不过`_exit_`方法还是可以有条件地终止或处理异常，因为它们提供了异常的实例和类型，以及完整的回溯。事实上，异常处理是完全可以自定义的。

### 3. 处理特定异常类

一个简单的异常处理函数`_exit_`可以仅检查异常是否是特定异常类的实例，执行任何

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

必要的异常处理，并根据是否获得其他类型的异常类返回 True(或返回 False)。

```
class HandleValueError(object):
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        # Return True if there is no exception.
        if not exc_type:
            return True

        # If this is a ValueError, note that it is being handled and
        # return True.
        if issubclass(exc_type, ValueError):
            print('Handling ValueError: %s' % exc_instance)
            return True

        # Propagate anything else.
        return False
```

如果使用该上下文管理器并且在代码块内引发 ValueError，则会看到相应的输出，之后异常终止。

```
>>> with HandleValueError():
...     raise ValueError('Wrong value.')
...
Handling ValueError: Wrong value.
```

与之类似，如果使用上下文管理器但引发的是不同类的异常(如 TypeError)，那么抛出该异常并输出回溯。

```
>>> with HandleValueError():
...     raise TypeError('Wrong type.')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Wrong type.
```

就本身而言，这并没有太大价值。毕竟，这仅仅是一个更直观的 try 子句的替代品。

```
try:
    [do something]
except ValueError as exc_instance:
    print('Handling ValueError: %s' % exc_instance)
```

上下文管理器有价值的应用之一是必须在 except 子句中完成的工作很重要，且必须要在应用程序中多处复用的情况。上下文管理器不仅封装了 except 子句，还同时封装了它的内容。

#### 4. 不包括的子类

如何完成类或实例的检查也可以更加灵活。例如，假如想要捕获一个给定的异常类，但不希望显式地捕获它的子类。在传统的 except 代码块中不能这样做(也不该这样做)，但是上下文管理器就能处理这样的极端情况，如下所示：

```
class ValueErrorSubclass(ValueError):
    pass

class HandleValueError(object):
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        # Return True if there is no exception.
        if not exc_type:
            return True

        # If this is a ValueError (but not a ValueError subclass),
        # note that it is being handled and return True.
        if exc_type == ValueError:
            print('Handling ValueError: %s' % exc_instance)
            return True

        # Propagate anything else.
        return False
```

注意，现在对 HandleValueError 上下文管理器稍加修改。通过使用==来检查其类型，而不是使用之前示例中更为传统的 issubclass 来进行检查。这意味着，对 ValueError 的处理与之前一样，只是管理器不再处理 ValueError 的子类，例如前面定义的 ValueErrorSubclass 类：

```
>>> with HandleValueError():
...     raise ValueErrorSubclass('foo bar baz')
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
__main__.ValueErrorSubclass: foo bar baz
```

#### 5. 基于属性的异常处理

上下文管理器可以根据异常的类型来决定是否处理异常(这也是 except 子句必须做的工作)，与此类似，它还可以根据异常的属性来决定是否处理异常。

考虑下面这个用于方便运行 shell 命令的函数，并使用一个引发异常的类来响应 shell 错误：

```
import subprocess
```

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
class ShellException(Exception):
    def __init__(self, code, stdout='', stderr=''):
        self.code = code
        self.stdout = stdout
        self.stderr = stderr

    def __str__(self):
        return 'exit code %d - %s' % (self.code, self.stderr)

def run_command(command):
    # Run the command and wait for it to complete.
    proc = subprocess.Popen(command.split(' '), stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE)
    proc.wait()

    # Get the stdout and stderr from the shell.
    stdout, stderr = proc.communicate()

    # Sanity check: If the shell returned a non-zero exit status, raise an
    # exception.
    if proc.returncode > 0:
        raise ShellException(proc.returncode, stdout, stderr)

    # Return stdout.
    return stdout
```

这样一个函数(与异常类)非常易于使用。下面的代码尝试使用 rm 命令删除一个不存在的文件：

```
run_command('rm bogusfile')
```

执行这行代码将会如期望那样生成 ShellException 回溯。

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in run_command
  _main_.ShellException: exit code 1 - rm: bogusfile: No such file or directory
```

处理这些异常时发生了什么？处理任何常见的 ShellException 异常很简单，但是设想一个情景，收到一个 ShellException 异常但只希望对特定的退出代码进行处理。此时上下文管理器是一种可行的解决方式。

例如，假设想要移除一个文件，但该文件已经被删除的情况也能接受(就该例的目的而言，忽略 os.remove 的存在)。在这种情况下，返回值是 0 也可以接受，表明文件已被成功移除；返回值是 1 时，表明文件已经不存在。另一方面，退出代码为 64 则表示仍然存在问题，因为存在某种类型的使用错误。这仍然会引发该异常。

下面是一个上下文管理器，基于代码返回 ShellException 实例：

```

class AcceptableErrorCodes(object):
    def __init__(self, *error_codes):
        self.error_codes = error_codes

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_instance, traceback):
        # Sanity check: If this is not an exceptional situation, then just
        # be done.
        if not exc_type:
            return True

        # Sanity check: If this is anything other than a ShellException,
        # then we do not actually know what to do with it.
        if not issubclass(exc_type, ShellException):
            return False

        # Return True if and only if the ShellException has a code that
        # matches one of the codes on our error_codes list.
        return exc_instance.code in self.error_codes

```

该示例代码实际上引入了一种新的模式。当上下文管理器被初始化时，它接收允许的错误代码。在本例中，AcceptableErrorCodes 接受任何整型参数，并且这些参数用于确定实际上可接收的是哪些错误码。

如果希望通过 AcceptableErrorCodes 上下文管理器移除一个不存在的文件，它会顺利执行。

```

>>> with AcceptableErrorCodes(1):
...     run_command('rm bogusfile')
...

```

但是，该上下文管理器不会是盲目地处理所有 ShellException 异常，考虑下面实际上误用 rm 的示例：

```

>>> with AcceptableErrorCodes(1):
...     # -m is not a switch available to rm (at least in Mac OS X).
...     run_command('rm -m bogusfile')
...
Traceback (most recent call last):
File "<stdin>", line 3, in <module>
File "<stdin>", line 11, in run_command
__main__.ShellException: exit code 64 - rm: illegal option -- m
usage: rm [-f | -i] [-dPRrvW] file ...
        unlink file

```

因此，为什么会引起回溯？因为退出代码是 64(在 Mac OS X 系统中，这可能随正在使用的操作系统的不同而有所不同)，并且告诉上下文管理器只能容忍的退出代码为 1。因此 \_\_exit\_\_ 返回了 False，像往常一样，抛出异常。

## 2.4 更简单的语法

到目前为止，所探索的大多数上下文管理器实际上都非常简单。虽然它们都是完整构造的类，但它们唯一的真实目的是提供更直观的、线性的`_enter_`与`_exit_`功能。

该结构极其强大。它允许进行非常复杂的创建并且上下文管理器可以完成大量的自定义逻辑。但是，很多上下文管理器非常简单，创建类并手动定义`_enter_`和`_exit_`可能显得有点大材小用。

在处理简单情况时，有一个更简单的方法。Python 的标准库提供了一个用于装饰简单函数的装饰器，该装饰器会将其转换为一个上下文管理器类。

该装饰器就是`@contextlib.contextmanager`。以该装饰器装饰的函数在函数执行期间返回单个值(yield 语句将在第 3 章的生成器中详细介绍)。

考虑`AcceptableErrorCodes`类作为一个单独、更直观函数的样子：

```
import contextlib

@contextlib.contextmanager
def acceptable_error_codes(*codes):
    try:
        yield
    except ShellException as exc_instance:
        # If this error code is not in the list of acceptable error
        # codes, re-raise the exception.
        if exc_instance.code not in codes:
            raise

        # This was an acceptable error; no need to do anything.
        pass
```

该函数最终与类完成完全相同的工作(值得注意的是代码`pass`行仅仅为了教学目的——明显并不需要它)

```
>>> with acceptable_error_codes(1):
...     run_command('rm bogusfile')
```

类似的，上下文管理器仍然会检查错误代码，并且只会拦截特定错误代码。

```
>>> with acceptable_error_codes(1):
...     run_command('rm -m bogusfile')
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
File "<stdin>", line 11, in run_command
__main__.ShellException: exit code 64 - rm: illegal option -- M
usage: rm [-f | -i] [-dPRrvW] file ...
        unlink file
```

这个简单的语法(仅仅声明一个函数体为 `yield` 的函数，并使用 `@contextlib.contextmanager` 装饰该函数)用于创建最简单的上下文管理器绰绰有余，且使代码更加易于阅读。需要它提供的强大功能时，请创建一个上下文管理器，否则可以使用一个带有装饰器的函数。

## 2.5 小结

上下文管理器提供了确保资源被正确处理的优秀方式，并且能够将需要在程序中多个不同位置重复使用的异常代码封装到一个位置。

与装饰器一样，上下文管理器是用于采纳“只做一次”原则的工具，除非迫不得已需要在多处重复代码。装饰器用于封装命名函数与类，而上下文管理器更适用于封装任意代码段。

第3章将讨论生成器，在需要每个值时，生成器可以在遍历时逐个产生值，而不必提前计算整个值集。

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



第 3 章  
生 成 器

生成器处理值序列时允许序列中的每一个值只在需要时才计算，而不是像传统列表那样，一定要提前计算列表中所有的值。

在恰当的地方使用生成器能节省大量内存，因为大的数据集没必要完全存入内存。与之相似，生成器能够处理一些无法由列表准确表示的序列形式。

本章说明什么是生成器，以及在 Python 中使用生成器的语法，还介绍了一些 Python 标准库中提供的普通生成器。

### 3.1 理解生成器

生成器是一个函数，它并不执行并返回一个单一值，而是按照顺序返回一个或多个值。生成器函数执行直到被通知输出一个值，然后会继续执行直到再次被通知输出值。这会持续执行直到函数完成或生成器之上的迭代终止。

如果完全没有终止生成器的显式要求，生成器可以表现为一个无限序列。这本身没有问题。在发生这种情况时，代码的任务就是在恰当的时候从生成器上的迭代序列中跳出来（例如，使用 `break` 语句）。

### 3.2 理解生成器语法

通常，生成器函数的特征就是在函数内部有一个或多个 `yield` 语句，而不是 `return` 语句。在 Python 2 中，`yield` 语句和 `return` 语句不能在同一个函数中共存。但是，在 Python 3 中两者可以同时存在（稍后将会详细介绍）。

像 `return` 语句一样，`yield` 语句命令函数返回一个值给调用者。但是与 `return` 语句不同的是，`yield` 语句实际上不会终止函数的执行。执行会暂时停顿直到调用代码重新恢复生成

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

器，在停止的地方再次开始执行。考虑下面这个非常简单的生成器：

```
def fibonacci():
    yield 1
    yield 1
    yield 2
    yield 3
    yield 5
    yield 8
```

这个生成器表示斐波那契数列的开始部分(换言之，就是队列中每个整数的值都是它前面两个整数值的和)。如你所见，在 Python 交互终端中，通过使用简单的 for...in 循环实现对生成器的迭代。

```
>>> for i in fibonacci():
...     print(i)
...
1
1
2
3
5
8
Obviously, this particula
```

显而易见，这种特殊的生成器能更好地表示为一个普通的 Python 列表。但是，下面的生成器不会返回 6 个斐波那契数字，而是返回一个无穷的斐波那契数列：

```
def fibonacci():
    numbers = []
    while True:
        if len(numbers) < 2:
            numbers.append(1)
        else:
            numbers.append(sum(numbers))
            numbers.pop(0)
        yield numbers[-1]
```

这个生成器将会输出一个无穷的斐波那契数列。在之前显示的交互式终端中使用 for...in 只会输出数字，这个终端上的显示越来越多，很快就变得很长(换句话说，屏幕被占满了)。

**注意：**出于好奇，在 Python 3.4 终端中我尝试多运行几分钟，看看数字超过整数的最大值需要多长时间。但是在大约 5 分钟后，我变得烦躁不安并试图通过键盘终止运行。该计算本身很可能迅速地达到 sys.maxsize 的值，但是终端的 I/O 就慢得多。

# www.itbaizhan.cn

不像之前的 fibonacci 函数，该生成器不能更好地表示为一个简单的 Python 列表。事实上，试图将其表示为一个简单的 Python 列表不仅不明智，而且不可能。Python 列表不能存储无穷的数值序列。

## 3.2.1 next 函数

在不使用 for...in 循环的情况下可以向生成器请求一个值。有时可能打算只得到一个单一的值或者固定数量的值。Python 提供了内置的 next 函数，能够让生成器(事实上，在 Python 2 中任何带有\_\_next\_\_方法的对象都称为 next)请求它的下一个值。

之前的 fibonacci 函数输出一个无穷的斐波那契数列。不用迭代整个函数，而是可以一次请求一个值。

首先，只是通过调用 fibonacci 函数并且保存返回值来创建自己的生成器。因为函数包含的是 yield 语句而不是 return 语句，所以 Python 解释器知道只返回这个 generator 对象。

```
>>> gen = fibonacci()  
>>> gen  
<generator object fibonacci at 0x101555dc8>
```

此时，值得注意的是 fibonacci 函数中的代码实际上没有运行。解释器唯一完成的就是识别生成器的出现并返回一个 generator 对象，该对象在每运行一次代码时就请求一个值。

可以使用内置的 next 函数请求第一个值，如下所示：

```
>>> next(gen)  
1
```

现在(只是现在)，fibonacci 函数中的实际代码已经运行(为了解释尽可能清晰，在循环的结尾加了一个显式的 continue 语句)。

```
def fibonacci():  
    numbers = []  
    while True:  
        if len(numbers) < 2: # True; numbers == []  
            numbers.append(1)  
        else:  
            numbers.append(sum(numbers))  
            numbers.pop(0)  
        yield numbers[-1]  
        continue
```

输入函数，它首先开始 while 循环的第一次迭代。因为此刻 numbers 列表是空的，值 1 被追加到列表中。最后运行 yield numbers[-1] 语句。此刻，生成器已经得到了一个输出值，因此执行暂停，并且输出值 1。执行在此处结束，continue 语句还没有执行。

现在，再次调用 next(gen)，如下所示：

```
>>> next(gen)
```

1

在暂停的地方继续执行，这意味着首先要执行 continue 语句。

```
def fibonacci():
    numbers = []
    while True:
        if len(numbers) < 2:
            numbers.append(1)
        else:
            numbers.append(sum(numbers))
            numbers.pop(0)
        yield numbers[-1]
        continue
```

这将返回到 while 循环的顶部。numbers 列表仅仅有一个成员(就是[1]),因此 len(numbers) 仍然小于 2，并且再次在 if 语句处选择再次执行路径。numbers 列表现在是[1,1]，然后输出列表的最后一个元素，停止执行。

```
def fibonacci():
    numbers = []
    while True:
        if len(numbers) < 2: # True; numbers == [1]
            numbers.append(1)
        else:
            numbers.append(sum(numbers))
            numbers.pop(0)
        yield numbers[-1]
        continue
```

现在，再次调用 next(gen)，如下所示：

```
>>> next(gen)
2
```

在上次执行结束的地方继续执行，这意味着接下来就是执行 continue 语句。

```
def fibonacci():
    numbers = []
    while True:
        if len(numbers) < 2:
            numbers.append(1)
        else:
            numbers.append(sum(numbers))
            numbers.pop(0)
        yield numbers[-1]
        continue
```

continue 语句将解释器发送回 while 循环的顶部。但现在当再次遇到 if 语句的时候选择 else 的路线，因为 number 列表现在包含两个元素([1,1])。随后，这两个元素的和被追加到

# www.itbaizhan.cn

列表的末尾，并且第一个元素被移除。再一次运行 `yield` 语句，并且它输出列表中的最后一个元素 2。

```
def fibonacci():
    numbers = []
    while True:
        if len(numbers) < 2: # False; numbers == [1, 1]
            numbers.append(1)
        else:
            numbers.append(sum(numbers))
            numbers.pop(0)
        yield numbers[-1]
        continue
```

如果再次调用 `next(gen)`，解释器将会选择相同的路径(因为 `numbers` 列表的长度仍然是 2)。当然，现在 `numbers` 列表已经从[1,1]变成了[1,2]，因此结果也不一样。3 被追加到列表的后面，1 在一开始就被移除，然后输出 3。

```
>>> next(gen)
3
```

如果继续请求更多的值，将会看到这一重复的模式。虽然运行的是同样的代码，但只会针对已更新的 `numbers` 列表运行，因此输出数值会延续斐波那契数列。

```
>>> next(gen)
5
>>> next(gen)
8
>>> next(gen)
13
>>> next(gen)
21
```

注意，有些事情并没有发生。例如，没有把庞大的斐波那契数列存储在内存中。只是存储了最新的两个数字，因为通过这两个数字可以计算出该数列的下一个数字。生成器废弃了过期的数据。这关系到生成器是否会无限期地继续下去，因为一旦保留了不必再保留的前一个值，最终这个列表将填满剩余内存。

与之相似，当有明确要求时，生成器仅仅计算数列中指定要求的值。此刻在代码执行中，生成器不必确定 `next` 将要输出的值(如果请求的话)是 34，因为生成器可能不会被请求。

### 3.2.2 StopIteration 异常

当使用带有生成器的其他函数时，可以有多条潜在的退出路径。例如，下面的普通函数有多条退出路径，这是使用多个 `return` 语句实现的：

```
def my_function(foo, add_extra_things=True):
    foo += '\nadded things'
```

```
if not add_extra_things:  
    return foo  
foo += '\n added extra things'  
return foo
```

这个函数一般在代码块的结尾返回。但是如果关键参数 `add_extra_thing` 有值且值为 `False`，则会转而执行函数第三行前面的 `return` 语句，并且函数执行在此将被截断。

这样做存在充足的理由，并且生成器一定有一种机制可以实现类似的目的。

## 1. Python 2

在一定程度上，正确的方法取决于所使用的 Python 的版本。在 Python 2 中，生成器不允许有 `return` 语句。如果打算编写一个包含 `yield` 和 `return` 语句的函数，会得到语法错误，如下所示：

```
>>> def my_generator():  
...     yield 1  
...     return  
...  
File "<stdin>", line 3  
SyntaxError: 'return' with argument inside generator
```

相反，Pyhton 提供了一个内置的异常 `StopIteration`，用来实现相同的目的。迭代生成器并且抛出 `StopIteration` 时，标志着生成器迭代完成并且已退出。在这种情况下捕获异常，并且没有回溯。另一方面，如果使用 `next` 函数，则抛出 `StopIteration` 异常。

考虑下面这个简单的生成器：

```
>>> def my_generator():  
...     yield 1  
...     yield 2  
...     raise StopIteration  
...     yield 3
```

如果迭代过这个生成器，会得到值 1 和 2，然后这个生成器将会干净地退出。`yield 3` 语句永远不会执行(与在 `return` 语句后面的代码相似)。

```
>>> [i for i in my_generator()]  
[1, 2]
```

如果在生成器上手动运行 `next`，前两个 `next` 调用将输出值，而第三个(以及任何后续的)调用将抛出一个 `StopIteration` 异常，如下所示：

```
>>> gen = my_generator()  
>>> next(gen)  
1  
>>> next(gen)  
2  
>>> next(gen)  
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in my_generator
StopIteration
```

### 2. Python 3

在 Python 3 中，情况类似，但是有一个额外的语法选项。Python 3 中消除了 `yield` 和 `return` 不能在一个函数中共存的限制。在这种情况下，有效地使用 `return` 实际上等同于 `raise StopIteration` 的功能。

值得注意的是，如果在 `return` 语句中返回一个值，它也不会成为最终输出的值。相反，这个值会被作为异常信息发送。考虑下面的语句：

```
return 42
```

该语句等同于下面的语句：

```
raise StopIteration(42)
```

非常重要的是，它与下面的语法不同：

```
yield 42
return
```

在想要兼容 Python 2 和 Python 3 的代码中，显式地使用 `raise StopIteration` 方式可能是更可取的。在仅运行于 Python 3 的代码中，两种方式的区别可能不是那么重要。

## 3.3 生成器之间的交互

到目前为止，之前已经研究过的生成器都是单向沟通的。它们都是将值输出到调用代码，不给生成器发送任何东西。

但是，生成器的协议也提供了一个额外的 `send` 方法，该方法允许生成器的反向沟通。因为 `yield` 语句实际上就是一个表达式，所以这是可以的。除了得到它的返回值以外，如果使用 `send` 方法而不是 `next` 重启生成器，那么提供给 `send` 方法的值实际上能被赋给 `yield` 表达式的结果。

考虑下面这个生成器，它会按顺序返回完全平方数。这是一个很普通的生成器。

```
def squares():
    cursor = 1
    while True:
        yield cursor ** 2
        cursor += 1
```

但是，你可能希望让生成器向前或向后移动到某一个值。对生成器代码做一点小小的修改即可实现这个功能，如下所示：

```
def squares(cursor=1):
```

# 尚学堂·百战程序员

第 I 部分 函数

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
while True:  
    response = yield cursor ** 2  
    if response:  
        cursor = int(response)  
    else:  
        cursor += 1
```

现在，将输出表达式的结果赋给 `response` 变量(如果有且仅有一个结果——你不会希望给该变量赋值 `NONE`)。

这使你能在 `squares` 生成器内跳跃输入，如下所示：

```
>>> sq = squares()  
>>> next(sq)  
1  
>>> next(sq)  
4  
>>> sq.send(7)  
49  
>>> next(sq)  
64
```

这里发生了什么？首先，解释器进入了生成器，并且被要求输出两个值(1 和 4)。但在下一次，会将值 7 发送给生成器。`Squares` 生成器内的代码逻辑就像是一个值被发送回来，然后将该值赋给 `cursor` 变量。因此，`cursor` 不再递增到 3 而是被设置为 7。

然后，生成器像以前一样继续执行。解释器返回到这个 `while` 循环的顶部。因为现在 `cursor` 是 7，所以将被输出的值为  $49(7^2)$ 。编写这个生成器，以便它仅从这里继续，因此当再次调用 `next` 时，`cursor` 像以前一样递增，变成了 8，之后输出的 `next` 值是  $64(8^2)$ 。

对如何(是否)发送这些被处理的值完全由生成器确定。本章前面介绍的生成器只是忽略了它们。相比之下，生成器使用发送给 `cursor` 的值作为一次性的值，然后返回到它原来的位置，如下所示：

```
def squares(cursor=1):  
    response = None  
    while True:  
        if response:  
            response = yield response ** 2  
            continue  
        response = yield cursor ** 2  
        cursor += 1
```

这个 `squares` 生成器的版本恰好实现输入指定值，然后在返回输入值之前值的位置继续输出：

```
>>> sq = squares()  
>>> next(sq)  
1  
>>> next(sq)
```

```
4
>>> sq.send(7)
49
>>> next(sq)
9
```

这里的不同之处完全在于生成器的行为。对于如何表现 send 没有什么魔法。send 的目的是提供一个与生成器双向交互的机制。确定是否(如何)处理发送给生成器的值是生成器的责任。

## 3.4 迭代对象与迭代器

在 Python 中，生成器是一种迭代器。Python 中的迭代器是包含`_next_`方法的任何对象(因此，能响应`next`函数)。

这是不同于迭代对象的，迭代对象是任何定义了`_iter_`方法的对象。可迭代对象的`_iter_`方法负责返回一个迭代器。

举例说明这种微妙的区别，考虑在 Python 3 中的`range`函数(在 Python 2 中称为`xrange`)。实际上，人们普遍认为`range`对象就是生成器。但它们却不是，如下所示：

```
>>> r = range(0, 5)
>>> r
range(0, 5)
>>> next(r)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'range' object is not an iterator
```

这是让很多人混淆的地方，因为在 Python 中，首先要学习的一般是像`for i in range(0, 5)`这样的习惯用法。该用法能工作是因为`range`函数返回了一个迭代对象。

但是`range`对象的`_iter_`方法返回的实际迭代器是一个生成器，并且它如期响应`next`方法。

```
>>> r = range(0, 5)
>>> iterator = iter(r)
>>> iterator
<range_iterator object at 0x10055ecc0>
>>> next(iterator)
0
>>> next(iterator)
1
```

另外，正如所料，在生成器完成输出值以后调用`next`将会抛出`StopIteration`异常。

```
>>> next(iterator)
2
>>> next(iterator)
```

```
3
>>> next(iterator)
4
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

当理解生成器的时候，记住：生成器可以是迭代器，但不一定是迭代对象。相似的，并非所有的迭代对象都是迭代器。

**注意：**与之相似，实际上并非所有的迭代器都是 generator 类的实例。本例中的迭代器是 range\_iterator 的实例，它实现了相似的模式。但是，作为一种实现细节，它缺少 send 方法。

## 3.5 标准库中的生成器

Python 的标准库中包含几个生成器，有些可能已经用过，有些甚至都没有意识到它们也是生成器。

### 3.5.1 range

在之前关于迭代对象和迭代器的讨论中，学习了 range 函数，它返回一个可迭代的 range 对象。

**注意：**如前所述，这个函数在 Python 2 中称为 xrange。

这个 range 对象的迭代器是个生成器。它返回序列值，这些值从 rang 对象的底层值开始一直到它的顶端值。默认情况下，它的序列就是让每一个当前值加 1 并作为下一个值输出。但是 range 函数有一个可选的第三方参数 step，能让你指定不同的增量(包括负值)。

### 3.5.2 dict.items 及其家族

在 Python 中，内置的字典类包括 3 个允许迭代所有字典的方法，并且这 3 个方法都是迭代器是生成器的迭代对象：keys、values 和 items。

**注意：**在 Python 2 中，这 3 个方法被称为 iterkeys、itervalues 和 iteritems。

这些方法的目的是允许迭代键、值或包含一个字典中键与值的二元组(条目)，如下所示：

```
>>> dictionary = {'foo': 'bar', 'baz': 'bacon'}
>>> iterator = iter(dictionary.items())
>>> next(iterator)
('foo', 'bar')
>>> next(iterator)
('baz', 'bacon')
```

在此使用生成器的一项价值是防止需要以另一种格式创建一个额外的字典副本(或部分字典)。dict.items 并不需要将整个字典重新格式化为包含二元组的列表。被请求时，它仅仅一次返回一个二元组。

如果在迭代期间试图修改字典，可能会看到副作用，如下所示：

```
>>> dictionary = {'foo': 'bar', 'baz': 'bacon'}
>>> iterator = iter(dictionary.items())
>>> next(iterator)
('foo', 'bar')
>>> dictionary['spam'] = 'eggs'
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

因为 items 迭代器是一个仅从引用的字典中读取数据的生成器，如果运行时字典表发生了变化，它将不知道自己应该做什么。面对这种模糊性，它会拒绝可能的猜测，并且抛出一个 RuntimeError 错误。

### 3.5.3 zip

Python 包含了一个名为 zip 的内置函数，该函数有多种可迭代对象并且一起迭代所有的对象，输出每个迭代对象(在元组中)的第一个元素，接着输出第二个元素，然后输出第三个元素，依此类推，直到到达最短的迭代对象的最后一个元素。下面就是一个示例：

```
>>> z = zip(['a', 'b', 'c', 'd'], ['x', 'y', 'z'])
>>> next(z)
('a', 'x')
>>> next(z)
('b', 'y')
>>> next(z)
('c', 'z')
>>> next(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

使用 zip 的原因与使用 dict.items 及其家庭的原因相似。它的目的就是在不同的结构中输出其迭代对象的返回成员，一次输出一个集合。如果将所有数据都复制到内存中并不是必需的，那么将缓解对内存的需求。

## 3.5.4 map

内置函数 `map` 是 `zip` 函数的表亲，该函数将一个能接受 N 个参数和 N 个迭代对象的函数作为参数，并且计算每个迭代对象的序列成员的函数结果，当它到达最短的迭代对象的最后一个元素时停止。

与 `zip` 相似，生成器在这里被用作迭代器，这是因为它不适合提前计算所有值。毕竟，这些数据可能需要也可能不需要。当且仅当每个值被请求时，才会计算值。

```
>>> m = map(lambda x, y: max([x, y]), [4, 1, 7], [3, 4, 5])
>>> next(m)
4
>>> next(m)
4
>>> next(m)
7
>>> next(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

和以前一样，处理小的迭代对象时，这依然是一个无关紧要的操作。但是，如果指定一个更大的数据结构，利用生成器可以节省大量的时间和避免内存消耗，因为不需要一次性转换和计算整个结构。

## 3.5.5 文件对象

在 Python 中使用生成器的最常见情况就是打开文件对象。尽管在 Python 中有多种与打开文件交互的方式，并且对于小文件最常见的就是调用 `read` 读取整个文件到内存中，但这个文件对象支持生成器模式，该模式从硬盘中一次读取一行文件内容。当操作较大文件时，这一点很重要。因为将整个文件读取到内存中并不总是合理的。

由于历史原因，文件对象有一个特殊方法，称作 `readline`，该方法一次读取一行数据，但也可以实现生成器协议并对文件调用 `next` 来实现相同的功能。

考虑下面的简单文件：

```
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5
```

在 Python shell 中，可以使用内置函数 `open` 读取文件。结果对象除了其他身份外还是生成器。

```
>>> f = open('lines.txt')
>>> next(f)
```

```
'line 1\n'  
>>> next(f)  
'line 2\n'
```

注意生成器一次读取一行并且输出整行，包括尾部的换行符(\n)。

如果打算在文件读取完毕后调用 next，StopIteration 将被如期抛出。

```
>>> next(f)  
'line 5\n'  
>>> next(f)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration
```

值得注意的是，此处的 `_next_` 和 `readline` 不完全是彼此的别名。一旦文件读取到结尾，`_next_` 函数就会抛出 StopIteration 异常，就像其他任何生成器一样，而 `readline` 实际上捕获了这个异常并且返回一个空字符串：

```
>>> next(f)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration  
>>> f.readline()  
''
```

## 3.6 何时编写生成器

本质上讲，使用生成器有两个主要理由。这两个理由来自于相同的基本概念：只有当需要值时才会确定这个值，而不是提前准备好。

这里的基本原则就是：通过自己的代码提前去做一系列工作或存储一堆数据是没有好处的。通常并不需要大的数据块。即使需要所有数据，但是如果不需要一次性处理所有数据，仍然可以仅存储需要的数据。

从基本原则中扩展出来的两个用例都需要分块访问数据，还需要分块计算数据。

### 3.6.1 分块访问数据

编写生成器的首要(也是最普遍的)原因是需要涵盖必须分块访问数据的情况，但是这种情况下没必要存储整个副本。

这正是之前介绍的文件对象生成器与 `dict.items`(及其家族)方法中本质上所发生的事情。处理小文件时，将整个文件读到内存中是非常合理的，并且可以对内存字符串做任何需要的处理。

另一方面，如果文件很大呢？如果需要重建一个大型的字典呢？有时，将操作数据复制到内存中并不是可行的操作。这时，就需要分块访问数据。

当用生成器方法迭代大型文件时，文件有多大并不重要。一次读取和输出一行。当用 dict.items 迭代字典时，不必关心源字典有多大。迭代器将一次迭代一块数据，然后仅输出包含两个元素的元组。

编写的生成器应用了相同的原则。生成器在任何打算迭代大量数据的情况下都是有效的工具，并且没有必要一次性地在内存中存储或复制整个数据。

### 3.6.2 分块计算数据

编写生成器的第二个常见理由是仅在需要它时计算数据。考虑一下本章前面讨论的 range 函数和 fibonacci 函数。必须循环遍历 0 到最大值之间每个数字的程序不必存储包含范围内每个数字的列表。对于每次只对当前数值加 1 直到达到最大值而言，内存是足够的。

与之相似，fibonacci 函数不需要计算每个斐波那契数字(这是一个不可能完成的任务，因为有无穷的数字在这个数列中——很快会越来越多)。生成器仅仅需要确定下一个单一的斐波那契数字并且输出它。

因为有时序列中的每个元素的计算可能相当昂贵，所以这可能是很重要的理由，而且计算整个系列的数据是没有价值且没有必要的。

#### 序列可以是无穷的

一方面，之前关于 fibonacci 函数的简要讨论中实际上一些序列是无穷的。在这种情况下，用列表表示整个序列是不可能的，但是生成器却可以。

这是因为生成器并不需要了解必须生成的每个值。它只需要生成下一个值。斐波那契数列永远持续下去这一点并不重要。只要生成器存储最近的数列中的两个值，它就能合理地计算下一个值。

这样做是没有问题的。在这种情况下调用生成器代码的主要任务就是处理生成器表示无穷序列的结果，然后在合适的时候跳出这个序列。

### 3.7 何时使用生成器单例模式

关于生成器的一个重要事实(经常被忽略)是很多生成器是单例模式的。最为常见的情况是，对象既是迭代对象又是迭代器。因为迭代对象只返回 self，所以在一个对象上重复调用 \_\_iter\_\_ 将会返回同一个对象。这在本质上意味着对象仅支持一个活动的迭代器。

简单的生成器函数并不是单例模式的。多次调用函数将返回不同的生成器，如下所示：

```
>>> gen1 = fibonacci()
>>> next(gen1), next(gen1), next(gen1), next(gen1), next(gen1)
(1, 1, 2, 3, 5)
>>> gen2 = fibonacci()
>>> next(gen2)
1
>>> next(gen1)
```

下面的可迭代类可以实现相似的目的，并通过`_init_`方法返回自己：

```
class Fibonacci(object):
    def __init__(self):
        self.numbers = []

    def __iter__(self):
        return self

    def __next__(self):
        if len(self.numbers) < 2:
            self.numbers.append(1)
        else:
            self.numbers.append(sum(self.numbers))
            self.numbers.pop(0)
        return self.numbers[-1]

    def send(self, value):
        pass

    # For Python 2 compatibility
    next = __next__
```

这是一个 Fibonacci 类，它实现了生成器的协议。但注意它也是一个迭代对象，并且将自己作为参数响应`iter`，这就意味着每个 Fibonacci 对象只有一个迭代器：它自己。

```
>>> f = Fibonacci()
>>> i1 = iter(f)
>>> next(i1), next(i1), next(i1), next(i1), next(i1)
(1, 1, 2, 3, 5)
>>> i2 = iter(f)
>>> next(i2)
8
```

这样做本身并没有什么错误。但值得注意的是，有些生成器可以用单例模式实现，然而其他生成器则并非如此。一定要理解迭代对象和迭代器之间的关系，并且弄明白一个迭代对象是否允许有多个迭代器。对于该问题的答案是：有些迭代对象可以有多个迭代器，而有些迭代对象则不可以。

### 3.8 生成器内部的生成器

一般来说，一个函数调用其他函数是可行的。这是开发人员构造可重用代码的一种关键方式。同样，一个生成器调用其他生成器也是可行的。Python 3.3 中引入了新的`yield from`语句，旨在为生成器提供一种调用其他生成器的直接方式。

考虑下面两个普通且有限的数据生成器：

```
def gen1():
    yield 'foo'
    yield 'bar'

def gen2():
    yield 'spam'
    yield 'eggs'
```

在 Python 3.3 之前，在封装生成器中，将这些子生成器组合成一个生成器的常用方法是显式地迭代它们，如下所示：

```
def full_gen():
    for word in gen1():
        yield word
    for word in gen2():
        yield word
```

也可以使用 `itertools.chain` 方法来实现这一点：

```
def full_gen():
    for word in itertools.chain(gen1(), gen2()):
        yield word
```

为此，Python 3.3 中提供了一个更干净的语法 `yield from` syntax，并且在另一个函数内调用函数看起来更贴切。

```
def full_gen():
    yield from gen1()
    yield from gen2()
```

这个语法的使用被称为生成器委托。但实际上，`full_gen` 的前两个实现是不等价的。这是因为前一个实现放弃了用 `send` 发送给生成器的值。

在另一方面，`yield from` 语法保留了这个值，因为生成器仅仅是委托给另一个生成器。这意味着任何被发送给封装生成器的值都将被发送给当前的委托生成器，这样开发人员就无须再处理这种情况。

## 3.9 小结

在 Python 中，生成器在执行计算或者迭代大量数据时是很有价值的工具，虽然它仅存储和计算实际需要的数据。这意味着可以节省大量的内存和性能方面的成本。

当需要处理大量数据或计算工作并且不需要提前处理所有工作时，可以考虑使用生成器。也可以把生成器当成一种表示无穷或分支序列的方式。

在第 4 章中，将从魔术方法的介绍开始，学习 Python 中的类。

## 第Ⅱ部分

# 类

---

- 第4章 魔术方法
- 第5章 元类
- 第6章 类工厂
- 第7章 抽象基类

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 4

## 章

# 魔术方法

可以将 Python 类定义为一个方法的长列表，在一定的情况下使用类的实例时，就可以调用这些方法。例如，类可以通过定义 `_eq_` 方法来定义该类的实例与另一个实例是否相等。如果类中存在 `_eq_` 方法的定义，那么类在使用 “`==`” 操作符进行相等性测试时，会自动调用该方法。

所谓的“魔术方法”被设计用于重载 Python 的操作符或内置方法。魔术方法可以通过“`__`”语法定义，从而避免程序员在没有意向使用重载时碰巧定义了同名方法。魔术方法使内置类(包括诸如整型和字符串等基本类型)提供的约定与自定义类提供的约定保持一致。如果希望在 Python 中进行相等性测试，可以总是使用 “`==`” 完成该操作，而无须考虑比对对象是两个整数还是两个为某个特定应用程序写的类实例，甚至是两个不相关类的实例。

## 4.1 魔术方法语法

在 Python 中，魔术方法遵循统一的模式——将下划线放到方法名称的两端。例如，当一个类的实例初始化时，将会执行 `_init_` 方法(而不是 `init`)。

采用这种约定在一定程度上起到了未雨绸缪的作用。如果不以下划线作为方法名的开始与结束，可以按照自己的喜好命名方法，而无须考虑方法名之后被 Python 赋予特殊的(而且非故意)意义。

当口头(例如在大会上做演讲)提到该方法时，很多人选择使用杜撰的术语“dunder”描述魔术方法。因此，`_init_` 最终的发音为 dunder-init。

每个魔术方法都有特定的目的；当特定语法出现时，它作为执行的钩子(译者注：所谓

# 尚学堂·百战程序员

第二部分 类

www.itbaizhan.cn

钩子就是在特定事件发生时，能够为响应事件而调用的代码或函数，回调函数就算是钩子的一种类型)。例如，`_init_`会在创建类的实例时执行。请看下面的简单类：

```
class MyClass(object):
    def __init__(self):
        print('The __init__ method is running.')
```

当然，该类没有实现任何功能，仅在初始化时进行标准输出打印。但已经可以足够让人理解`_init_`在何时被触发执行。

```
>>> mc = MyClass()
The __init__ method is running.
>>>
```

这里需要意识到的重点是并没有直接调用`_init_`方法，仅仅是 Python 的解释器知道在对象初始化时调用该方法。

所有的魔术方法都以这种方式生效，都需要特定的函数名称与方法签名(某些时候方法签名是一个变量)，然后该方法就会在特定情况下被调用。

前文提到的`_eq_`方法有两个强制参数：第一个是`self`参数和第二个作为比对对象的位置参数。

```
class MyClass(object):
    def __eq__(self, other):
        # All instances of MyClass are equivalent to one another, and they
        # are not equivalent to instances of other classes.
        return type(self) == type(other)
```

注意`_eq_`方法接受第二个参数，这是由于在 Python 中使用“`==`”进行相等性检查时，会执行`_eq_`方法，此时等号另一边的对象会赋值给该方法的第二个参数。

在本例中，`_eq_`仅仅是根据两个参数是否都为`MyClass`类的实例来检测是否相等，因此，将会得到如下结果：

```
>>> MyClass() == MyClass()
True
>>> MyClass() == 42
False
```

`MyClass`的两个不同实例相等，这是由于`isinstance(other, type(self))`结果为`True`。另一方面，42 是`int`类型的数据，而不是`MyClass`的实例，因此`_eq_`(并且因此是`==`操作符返回`False`)。

## 4.2 可用的魔术方法

Python 解释器能够识别大量用于不同目的的魔术方法，从对比检查和排序，到为实现多种语言特性的钩子。本书已经在第 2 章和第 3 章展示过其中一部分。

#### 4.2.1 创建与销毁

该类方法在类的实例创建或销毁时执行。

##### 1. \_\_init\_\_ 方法

在创建实例后会立即执行对象的 \_\_init\_\_ 方法。该方法必须接受一个位置参数(self)，然后可以接受任意数量的必要或可选位置参数，以及任意数量的关键字参数。

该方法的签名非常灵活，因为传递到类初始化调用的参数会被传递给 \_\_init\_\_ 方法。

考虑下面包含 \_\_init\_\_ 方法的类，该方法接受可选的关键字参数：

```
import random

class Dice(object):
    """A class representing a dice with an arbitrary number
    of sides.
    """
    def __init__(self, sides=6):
        self._sides = sides

    def roll(self):
        return random.randint(1, self._sides)
```

为了初始化一个标准的六面骰子，调用类时无须任何参数： die=Dice()。该代码创建了 Dice 实例(后面会对它进行更多解释)，并调用了新实例的 \_\_init\_\_ 方法，除了 self 参数外没有传递任何参数。由于并未提供 sides 参数，因此使用该参数的默认值 6。

只需直接将 sides 参数在调用 Dice 的构造函数时传递给它，该函数会把参数传递给 \_\_init\_\_ 函数，而无须再创建一个 D20 类。

```
>>> die = Dice(sides=20)
>>> die._sides
20
>>> die.roll()
20
>>> die.roll()
18
```

值得注意的是， \_\_init\_\_ 方法并没有创建新对象(该操作由 \_\_new\_\_ 完成)。该方法旨在为创建后的对象提供初始化数据。

实际上这意味着 \_\_init\_\_ 方法并不返回(也不应该返回)任何值。在 Python 中所有的 \_\_init\_\_ 都不返回值，如果用 Return 返回值则会导致 TypeError 错误。

\_\_init\_\_ 方法或许是在自定义类中用的最多的魔术方法。大多数类在初始化时都需要一些额外变量从而以某种方式自定义其实现，而 \_\_init\_\_ 方法是实现这类逻辑最合适的方法。

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

## 2. \_\_new\_\_方法

\_\_new\_\_方法实际上在\_\_init\_\_方法之前执行，用于创建类的实例。然而\_\_init\_\_方法负责在实例创建后对其进行自定义，\_\_new\_\_方法才是实际上创建并返回实例的方法。

\_\_new\_\_方法永远是静态的。同样它也无需显式装饰。第一个也是最重要的参数是创建实例所需要的类(按照惯例，命名为cls)。

在大多数情况下，\_\_new\_\_方法的其他参数会被完整复制到\_\_init\_\_方法中。参数在调用类构造函数时首先会被传递给\_\_new\_\_方法(这是由于其先被调用)，然后再传递给\_\_init\_\_方法。

在实际应用中，大多数类无需定义\_\_new\_\_方法。该方法在Python中的内置实现已经足够。一个类需要定义该方法时，几乎都需要首先在实现本类逻辑之前引用父类的实现，如下所示：

```
class MyClass(object):
    def __new__(cls, [...]):
        instance = super(MyClass, cls).__new__(cls, [...])
        [do work on instance]
        return instance
```

通常，会希望\_\_new\_\_方法返回一个已经被初始化后类的实例。但是某些情况下，并不需要这么做。注意，只有在通过\_\_new\_\_方法返回当前类的实例时才会执行\_\_init\_\_方法。如果返回的不是当前类的实例，那么就不会调用\_\_init\_\_方法。

这样做主要是在某些情况下返回了其他类的实例，因此\_\_init\_\_也会被其他类中定义的\_\_new\_\_方法触发，而执行两个不同类的\_\_init\_\_方法很可能导致问题。

## 3. \_\_del\_\_方法

与\_\_new\_\_和\_\_init\_\_方法在对象创建时被调用，而\_\_del\_\_方法在对象被销毁时被调用。

在Python中，对于开发者来说很少会直接销毁对象(如果需要，应该使用del关键字销毁)。Python的内存管理机制能够很好地胜任这项工作，因此由垃圾回收器完成对象的销毁工作被普遍接受。

也就是说，不管对象以何种方式销毁都会触发\_\_del\_\_方法执行，无论是直接删除，或是由垃圾回收器进行内存回收。可以通过下面被销毁时打印出信息的实践来验证这一机制。

```
class Xon(object):
    def __del__(self):
        print('AUUUUUUGGGGGHHH!')
```

如果创建了Xon对象但没有将其赋值给变量，那么它会被垃圾回收器标记为可回收，当其他语句执行时该对象会被垃圾回收器迅速回收。

```
>>> Xon()
<__main__.Xon object at 0x1022b8890>
>>> 'foo'
AUUUUUUGGGGGHHH!
```

```
'foo'  
>>>
```

这里发生了什么？首先，`Xon` 创建对象(但并没赋值给变量，因此 Python 解释器没有理由保留该对象)。接下来，解释器接收到一个不可变的字符串，并为其分配内存(然后立刻释放内存，这是因为该字符串也没赋值给变量，但在这无关紧要)。

在使用的特定版本的解释器中(CPython 3.4.0 和 CPython 2.7.6)，内存操作导致垃圾回收器遍历表。找到 `Xon` 对象并删除它，这将触发该对象的`_del_`方法，该方法会在对象被传递到垃圾回收器时执行 `del` 方法内的输出语句并删除该对象(如下代码所示)。

如果直接删除 `Xon` 对象，也能看到类似(但更快)的行为，如下所示：

```
>>> x = Xon()  
>>> del x  
AUUUUUUGGGGGHH!
```

在两种情况中，适用原则是一致的。无论是在代码中直接删除对象还是由垃圾回收器自动触发，都一样会调用`_del_`方法。

值得一提的是，`_del_`方法通常无法引发任何有意义的异常。这是由于该方法通常在后台由垃圾回收器触发，因此并没有一种好方法可以触发异常事件。因此在`_del_`方法中引发的任何异常都仅仅是在标准输出窗口打印一些错误信息，并且通常在该方法中引发异常并不合适。

## 4.2.2 类型转换

在 Python 中存在多个用于将复杂对象转换为简单对象或常用数据类型的魔术方法。例如，`str`、`int` 和 `bool` 等类型在 Python 中很常用，因此对于复杂对象来说知道自身使用简单类型的等价表示将会十分有用。

### 1. `_str_`、`_unicode_` 与 `_bytes_` 方法

目前为止，最常用的类型转换魔术方法是`_str_`。该方法接受一个位置参数(`self`)，并在对象被传递给 `str` 的构造函数时被调用，然后返回一个字符串。

```
>>> class MyObject(object):  
...     def __str__(self):  
...         return 'My Awesome Object!'  
...  
>>> str(MyObject())  
'My Awesome Object!'
```

字符串类型的使用十分普遍，因此为类定义`_str_`方法就变得很有用。

对此情况还需要注意一点，在 Python 2 中，字符串使用 ASCII 字符串，而在 Python 3 中，字符串使用 Unicode 字符串。这会导致很多问题，因此在本书中将使用一整章阐述该问题(第 8 章)。

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

这里只需要指出一点，在 Python 2 中存在 Unicode 编码的字符串，而 Python 3 引入了 bytes(有时也叫 bytestrings)类型，该类型基本可以类比 Python 2 中的 ASCII 编码字符串。

这些字符串家族拥有自己的魔术方法。Python 2 荣誉出品了 `_unicode_` 方法，该方法在对象传递给 `unicode` 的构造函数时被调用。类似的，Python 3 荣誉出品了 `_bytes_` 方法，该方法在对象传递给 `bytes` 的构造函数时被调用。上述两种情况中方法都会返回合适的类型。

`_str_` 方法还在其他特定场景被调用(本质上，底层调用 `_str_` 方法)。例如，当通过 `str` 的参数遇到格式化字符串 `%s` 时，如下所示：

```
>>> 'This is %s' % MyObject()
'This is My Awesome Object!'
```

在上述情况下，格式化方法就比较智能。例如，如果在 Python 2 中格式化 `unicode` 对象时遇到 `%s`，Python 会首先尝试使用 `_unicode_` 方法，看下面在 Python 2.7 中运行的方法：

```
>>> class Which(object):
...     def __str__(self):
...         return 'string'
...     def __unicode__(self):
...         return u'unicode'
...
>>> u'The %s was used.' % Which()
u'The unicode conversion was performed.'
>>> 'The %s was used.' % Which()
'The string conversion was performed.'
```

## 2. `__bool__` 方法

另一个常见需求是对象需要界定 `True` 或 `False`，无论是通过表达式转换为布尔类型，还是在某些情况下需要该对象布尔类型的等价形式(比如在 `if` 语句中使用到该对象)。

该类型的操作在 Python 3 中由 `__bool__` 魔术方法处理，而在 Python 2 中该方法被命名为 `__nonzero__`。无论是哪个版本，该方法都接受一个位置参数(`self`)并返回 `True` 或 `False`。

通常无需显式定义一个 `__bool__` 方法。如果未定义 `__bool__` 方法，但定义了 `len__` 方法(稍后将进一步介绍)，就将使用后者，从而会导致重复。

## 3. `__int__`、`__float__` 与 `__complex__` 方法

某些情况下，将复杂对象转为基本类型的数字十分有价值。如果一个对象定义了一个返回 INT 类型的 `__int__` 方法，那么该对象被传递给 `int` 的构造函数时，`int` 方法会被调用。

类似地，如果对象定义了 `__float__` 与 `__complex__` 方法，那么这些方法会在各自传递给 `float` 或 `complex` 的构造函数时被调用。

**注意：**Python 2 拥有单独的 Long 类型，因此它具有 `__long__` 方法。该方法的原理与所期望的完全一致。

### 4.2.3 比较

对象在进行相等性测试(==或!=)或不等性测试(例如，使用<、<=、>与>=)时进行比较。这些操作符与 Python 中的魔术方法一一对应。

#### 1. 二元相等性

下述方法支持使用==与!=进行相等性测试。

##### 1) \_\_eq\_\_ 方法

前面已介绍过，\_\_eq\_\_方法在两个对象使用==操作符进行比较时被调用。该方法必须接受两个位置参数(按照惯例，分别为self和other)，这是需要比较的两个对象。

在大多数情况下，首先检测左边对象的\_\_eq\_\_方法是否存在。如果左边对象定义了\_\_eq\_\_方法，那么该方法就会被使用(并且不再返回 NotImplemented)。否则，则使用右边对象中定义的\_\_eq\_\_方法(参数位置对调)。

考虑下面的类，在比较相等性时打印了一些无意义的代码(并且返回 False，除非是同一个对象)：

```
class MyClass(object):
    def __eq__(self, other):
        print('The following are being tested for equivalence:\n'
              '%r\n%r' % (self, other))
        return self is other
```

可以基于对象所在操作符的位置来查看顺序。

```
>>> c1 = MyClass()
>>> c2 = MyClass()
>>> c1 == c2
The following are being tested for equivalence:
<__main__.MyClass object at 0x1066de590>
<__main__.MyClass object at 0x1066de390>
False
>>> c2 == c1
The following are being tested for equivalence:
<__main__.MyClass object at 0x1066de390>
<__main__.MyClass object at 0x1066de590>
False
>>> c1 == c1
The following are being tested for equivalence:
<__main__.MyClass object at 0x1066de590>
<__main__.MyClass object at 0x1066de590>
True
```

注意，对象输出到标准输出窗口的顺序被交换。这是由于这两个对象被传递到\_\_eq\_\_方法的顺序被交换而导致。这还意味着在相等性检测时并没有对参数可交换的内在需求。除非有十分充足的理由，否则需要确保相等性检测始终可交换。

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

可以通过比较 MyClass 对象与其他类型的对象来观察该行为的另一方面。考虑下面只有一个`_eq_`方法的类，该方法的作用仅仅是返回 False：

```
class Unequal(object):  
    def __eq__(self, other):  
        return False
```

当对这些类的实例进行相等性测试时，基于它们调用的顺序可以看到不同的特征。当 MyClass 的实例在左边时，会调用它的`_eq_`方法。当 Unequal 的实例在左边时，则会调用该类的方法。

```
>>> MyClass() == Unequal()  
The following are being tested for equivalence:  
<__main__.MyClass object at 0x1066de5d0>  
<__main__.Unequal object at 0x1066de450>  
False  
>>> Unequal() == MyClass()  
False
```

对于传递给`_eq_`方法参数顺序的规则有一个例外：直接子类。如果被比较的两个对象中一个对象是另一个对象的直接子类，这会重载之前提到的规则，此时将使用子类的`_eq_`方法。

```
class MySubclass(MyClass):  
    def __eq__(self, other):  
        print('MySubclass\' __eq__ method is testing:\n' +  
              '%r\n%r' % (self, other))  
        return False
```

现在，无论提供给操作符的参数的顺序如何，都会调用相同的方法并传入相同顺序的参数。

```
>>> MyClass() == MySubclass()  
MySubclass' __eq__ method is testing:  
<__main__.MySubclass object at 0x1066de690>  
<__main__.MyClass object at 0x1066de450>  
False  
>>> MySubclass() == MyClass()  
MySubclass' __eq__ method is testing:  
<__main__.MySubclass object at 0x1066de5d0>  
<__main__.MyClass object at 0x1066de450>  
False
```

## 2) `_ne_`方法

`_ne_`方法与`_eq_`方法的功能相反，但原理一致，仅仅是该方法在使用!=操作符时使用。

通常，无须定义`_ne_`方法，只需要针对`_eq_`方法的返回值取反即可。如果没有定义`_ne_`方法，那么 Python 会调用`_eq_`方法，并对结果取反。

只有在不希望上述默认特征发生时才需要显式定义`_ne_`方法。

## 2. 相对比较

以下这些方法也处理比较操作，但使用比较操作符来测试相对值(比如`>`)。

### 1) `_lt_`、`_le_`、`_gt_` 和 `_ge_` 方法

`_lt_`、`_le_`、`_gt_` 以及 `_ge_` 方法分别与`<`、`<=`、`>`、`>=`操作符相匹配。与等值方法类似的是，上述方法都接受两个参数(按照惯例，为 `self` 和 `other`)，并在相对比较时根据运算符结果返回 `True` 或 `False`。

通常，无须全部定义上面 4 个方法，Python 解释器会认为`_lt_` 是`_ge_` 取反，`_gt_` 是`_le_` 取反。同理，Python 解释器会认为`_le_` 是由`_lt_` 与`_eq_` 分离得来，`_ge_` 是由`_gt_` 和`_eq_` 分离得来。

这意味着，实际上通常只需要定义`_eq_` 和`_lt_`(或`_gt_`)，那么所有 6 个比较操作符就能按照预期正常生效。

另一个定义这些方法的重要(但很容易忽视的)方面是这些方法内置了用于排序对象的 `sorted` 函数。因此，如果有一个列表中的所有对象都定义了这些方法，将对象传递到 `sorted` 就会基于比较方法的结果自动返回一个升序排序后的列表。

### 2) `_cmp_` 方法

`_cmp_` 方法是为对象定义相对比较的旧有(不推荐)方式。只有在之前描述的方法未被定义时，才检查该方法是否定义。

该方法接受两个位置参数(按照惯例，为 `self` 和 `other`)，如果 `self` 比 `other` 小，那么返回一个负整数；如果 `self` 比 `other` 大，则返回一个正整数；如果 `self` 和 `other` 相等，该方法返回 0。

`_cmp_` 在 Python 2 中已淘汰，在 Python 3 中不可用。

## 3. 操作符重载

这些方法提供了一种重载标准 Python 操作符的机制。

### 1) 二元操作符

一系列魔术方法可用于重载 Python 中的多种二元操作符，如`+`和`-`等。实际上对于每一个操作符，Python 都提供了 3 种魔术方法，每种方法都接受两个位置参数(按照惯例，为 `self` 和 `other`)。

第一类方法是普通方法(vanilla method)，表达式`x+y`与`x._add_(y)`匹配，这类方法仅仅是返回结果。

第二类方法是取反方法(reverse method)。只有在第一个操作对象不提供传统方法并且操作对象类型不同(或返回 `NotImplemented`)时才调用取反(操作符两边对象顺序交换)。这两类方法的拼写机制相同，只是取反方法在方法名称的开头加上了`r`。因此对于表达式`x+y`，如果 `x` 没有定义`_add_`方法，则调用`y._radd_(x)`。

第三类也是最后一类方法，是即席方法(in-place method)。在操作符即席修改第一个变

# 尚学堂·百战程序员

第Ⅱ部分 类

www.itbaizhan.cn

量时被调用(比如`+=`、`-=`等)。即席方法与第一类方法的拼写机制相同，只是这类方法仅仅是在正常方法开头加了`i`。因此表达式`x+=y`将会调用`x._iadd_(y)`。

通常，即席方法仅仅即席修改`self`并返回它。但这并不是严格的需求。值得注意的是，仅在直接方法没有清晰匹配的情况下，才需要定义一个即席方法。如果未定义即席方法，那么就会调用直接方法将值赋给操作符左边的对象并返回。

表 4-1 操作符重载的魔术方法

操作符	方法	取反	即席
<code>+</code>	<code>_add_</code>	<code>_radd_</code>	<code>_iadd_</code>
<code>-</code>	<code>_sub_</code>	<code>_rsub_</code>	<code>_isub_</code>
<code>*</code>	<code>_mul_</code>	<code>_rmul_</code>	<code>_imul_</code>
<code>/</code>	<code>_truediv_</code>	<code>_rtruediv_</code>	<code>_itruediv_</code>
<code>//</code>	<code>_floordiv_</code>	<code>_rfloordiv_</code>	<code>_ifloordiv_</code>
<code>%</code>	<code>_mod_</code>	<code>_rmod_</code>	<code>_imod_</code>
<code>**</code>	<code>_pow_</code>	<code>_rpow_</code>	<code>_ipow_</code>
<code>&amp;</code>	<code>_and_</code>	<code>_rand_</code>	<code>_iand_</code>
<code> </code>	<code>_or_</code>	<code>_ror_</code>	<code>_ior_</code>
<code>^</code>	<code>_xor_</code>	<code>_rxor_</code>	<code>_ixor_</code>
<code>&lt;&lt;</code>	<code>_lshift_</code>	<code>_rlshift_</code>	<code>_ilshift_</code>
<code>&gt;&gt;</code>	<code>_rshift_</code>	<code>_rrshift_</code>	<code>_irshift_</code>

这些方法可以重载 Python 中的所有二元操作符。在合理的情况下，自定义类可以(并且应该)重载二元操作符。

## 2) 除法

二元操作符中的除法(`/`)操作符，需要多花一点时间讨论。首先，需要一点背景知识。最开始，在 Python 中的两个`int`型数据进行除法运算，返回值的类型为`int`，而不是`float`。基本原理是两个数进行除法运算然后向下取整。因此`5/2`将会返回`2`，`-5/2`将会返回`-3`，如果希望得到`float`数据类型的结果，则至少需要其中一个值的类型为`float`。因此`5.0/2`则返回`2.5`。

由于许多开发人员觉得这违反直觉，因此 Python 3 改变了该特征。在 Python 中对两个`Int`型数据做除法返回`float`型的数据，即使结果是整数也是如此。因此`5/2`是`2.5`，`4/2`是`2.0`(不是`2`)，这是 Python 3 语言引入的向后不兼容的变更。

由于 Python3 引入了向后不兼容的变更，因此 Python 2 后续的版本使用了可以让开发

人员“可选”新特征的机制：特殊模块`_future_`用于导入未来特征。在 Python 2.6 和 2.7 中，开发人员可以通过加入`from _future_ import division`来“可选”Python 3 的特征。

这里的讨论很重要，因为涉及具体使用哪一个魔术方法。表 5-1 中的`_truediv_`方法（与同类方法）是 Python 3 的方法。Python 2 最初只提供`_div_`方法且对于/操作符调用该方法，除非从`_future_`中导入了/操作符，在这种情况下就是 Python 3 中调用`_truediv_`的机制。

在大多数情况下，在 Python 2 中执行的代码最终如何处理除法并不可知。这意味着需要同时定义`_truediv_`与`_div_`方法。在绝大多数情况下，这两个方法可以彼此匹配，这是完全可行的，如下所示：

```
class MyClass(object):
    def __truediv__(self, other):
        [...]
        div = __truediv__
```

或许使用`_truediv_`方法且将`_div_`作为别名才是“正确的”选择。宽泛的原则是最终在 Python 3 下执行的代码应该以 Python 3 为目标去编写，同时兼容 Python 2，而不是相反。

### 3) 一元操作符

Python 还提供了 3 个一元操作符：`+`、`-`与`~`。注意其中有两个操作符既是一元操作符也是二元操作符。这并不会引起问题，解释器能够根据表达式确定操作符被用于一元还是二元。

一元操作符方法只接受一个位置参数(`self`)，执行操作并返回结果。这 3 个方法的名称分别为`_pos_`（与`+`匹配）、`_neg_`（与`-`匹配）和`_invert_`（与`~`匹配）。

一元操作符更加直接。比如表达式`~x`，就是调用`x._invert_()`。考虑下面的仿 string 类，该类可以以逆序的方式返回字符串。

```
class ReversibleString(object):
    def __init__(self, s):
        self.s = s

    def __invert__(self):
        return self.s[::-1]

    def __str__(self):
        return self.s
```

在 Pyton 解释器中，可以看到以下结果：

```
>>> rs = ReversibleString('The quick brown fox jumped over the lazy dogs.')
>>> ~rs
'.sgod yzal eht revo depmuj xof nworb kciuq ehT'
```

那么，这里发生了什么？创建`ReversibleString`对象并赋值为`rs`。第二个语句，`~rs`，仅仅是一个一元表达式。结果并未赋值给一个变量，这意味着结果直接被丢弃。`rs` 变量并未

# 尚学堂.百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

即席修改。但解释器还是会展示出结果，也就是一个表示字符串的 str 对象，但显示顺序为逆序。

注意返回值是一个 str，而不是 ReversibleString。这里并不需要相同类型的对象作为操作对象，`__invert__`方法也是如此。

并不知道为什么没有返回 ReversibleString 类型的值，但通常情况下应该返回相同数据类型的对象。

```
class ReversibleString(object):
    def __init__(self, s):
        self.s = s

    def __invert__(self):
        return type(self)(self.s[::-1])

    def __repr__(self):
        return 'ReversibleString: %s' % self.s

    def __str__(self):
        return self.s
```

ReversibleString 的迭代通过`__invert__`返回了一个新的 ReversibleString 实例。这里添加了一个用于演示目的的自定义 repr，因为让解释器在输出时提供内存地址没有用处。

注意：你或许注意到这里使用了 `type(self)`，而不是直接调用 `ReversibleString()`。这确保了如果 ReversibleString 是子类，那么该子类可以正确使用。

现在，Python 的解释器显示了略微不同的输出结果：

```
>>> rs = ReversibleString('The quick brown fox jumped over the lazy dogs.')
>>> ~rs
ReversibleString: .sgod yzal eht revo depmuj xof nworb kciuq ehT
```

这里返回的对象是 ReversibleString，而不是 str 对象。这意味着结果可以被多次逆序。

```
>>> ~~rs
ReversibleString: The quick brown fox jumped over the lazy dogs.
```

这很直接。调用了 rs 对象的`__invert__`方法，然后对表达式的结果再次调用了`__invert__`方法。也就是，前面的调用结果等价于 `rs.__invert__().__invert__()`。

## 4. 重载常见方法

Python 包含了很多内置方法(最常见的例子是 `len` 方法)，该方法被广泛使用并将对象作为操作符。因此，Python 提供了对象传递到这些方法时被调用的一些魔术方法。

### 1) `__len__`方法

以这种方式重载的最常见方法非 `len` 莫属，该方法是以 Python 的方式确定一个条目的

“长度”。一个字符串的长度是一个字符串总字符的数量，一个列表的长度是列表所包含的项的数目。

可以通过定义 `_len_` 方法描述对象的长度，该方法接受位置参数(`self`)并返回一个整型值。

考虑下面这个表示一段时间的类：

```
class Timespan(object):
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def __len__(self):
        return (self.hours * 3600) + (self.minutes * 60) + self.seconds
```

该类本质上是接受小时、分钟和秒数作为参数，并计算对应的秒数作为对象的长度。

```
>>> ts = Timespan(hours=2, minutes=30, seconds=1)
>>> len(ts)
9001
```

值得注意的是 `_len_` 方法，如果定义了该方法，通常还被用于确定对象被类型转换为 `bool` 或用于 IF 语句时，其值为 `True` 还是 `False`，除非该对象还同时定义了 `_bool_` 方法(或者在 Python2 中，定义了 `_nonzero_` 方法)。

这实际上和对于一段时间的预期相同，因此通常无须再额外定义 `_bool_` 方法。

```
>>> bool(Timespan(hours=1, minutes=0, seconds=0))
True
>>> bool(Timespan(hours=0, minutes=0, seconds=0))
False
```

在 Python 3.4 中，添加了一个额外方法：`_length_hint_`，用于估计一个对象的长度，所谓估计就是允许返回值比对象实际长度多一些或少一些，这样可以降低性能开销。该方法接受一个位置参数(`self`)并且必须返回一个大于 0 的整数。

## 2) `_repr_` 方法

Python 中一个最重要的(也是经常被忽视的)方法是 `repr`。任何对象都可以定义 `_repr_` 方法，该方法接受一个位置参数(`self`)。

为什么 `repr` 重要？对象的 `repr` 方法用于确定该对象在 Python 交互式终端中的显示方式。

在 Python 中，将对象生成为`<__main__.Object at 0x102cdf950>`这种形式往往没有用处。在大多数情况下，一个对象的类以及其内存地址并不是希望获得的值。

定义 `_repr_` 方法可以让对象成为更有用的代表值。考虑下面定义了 `_repr_` 方法的 `Timespan` 类：

```
class Timespan(object):
    def __init__(self, hours=0, minutes=0, seconds=0):
```

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

```
self.hours = hours
self.minutes = minutes
self.seconds = seconds

def __repr__(self):
    return 'Timespan(hours=%d, minutes=%d, seconds=%d)' % \
        (self.hours, self.minutes, self.seconds)
```

现在，在终端上使用 Timespan 对象时会发生什么？

```
>>> Timespan()
Timespan(hours=0, minutes=0, seconds=0)
>>> Timespan(hours=2, minutes=30)
Timespan(hours=2, minutes=30, seconds=0)
```

这比显示内存地址有用多了！

注意，除了显示所有 Timespan 的关键属性外，`repr` 打印了一个用于初始化 Timespan 的有效表达式。在终端使用对象时，这非常有用。它能够直观地显示出正在使用的是一个对象，具体而言是 Timespan 对象。仅仅打印出时间信息使你无法确认所使用的对象是 str 或 timedelta 对象还是自定义的 Timespan 对象。如果将其复制粘贴到 Python 解析器，则可以对其代码进行解析。那很好啊。

在此要注意一个更重要的区别：`repr` 和 `str` 的目的不同。如何描述其区别则取决于读取内容的不同。但简单来说，一个对象的 `repr` 供程序员(也有可能是机器)阅读，而对象的 `str` 的用途更加广泛。你并不希望 Timespan 对象的 `str` 方法的返回结果类似于初始化类的代码。更多情况下，该方法供人阅读。

对于对象的 `repr` 方法，返回用于重构对象的有效 Python 表达式非常有用。很多 Python 的内置对象就是采用的这种方式。空列表的 `repr` 方法的值为 `[]`，该表达式用于创建一个空列表。

当这种实现不可能或不现实时，最佳实践是让返回值看上去明显就是一个对象，并对其关键属性进行描述。比如说，Timestamp 对象的另一种 `repr` 实现方式返回值为：`<Timestamp: X hours, Y minutes, Z seconds>`。Python 解释器无法解析该值(不像之前的 `repr` 实现)，但该对象是什么就变得很清晰，也不会有人错误地认为这个值可以被 Python 解析。

## 3) `_hash_` 方法

另一个被经常忽视的内置函数是 `hash` 函数。`hash` 函数的作用是通过数字化表达式唯一标识对象。

当一个对象传递给散列函数时，调用其 `_hash_` 方法(如果定义了 `hash`)。`_hash_` 方法接受一个位置参数(`self`)，并返回一个整型值，该整型值可以为负数。

对象类提供了 `_hash_` 函数，通常返回该对象的 `id`。对象的 `id` 值是与实现方式具体相关的，在 CPython 中，该值是其内存地址。

然而，如果一个对象定义了 `_eq_` 方法，则 `_hash_` 方法会隐式地被赋值为 `None`。这样做是由于通常哈希值的目的很模糊。取决于对象的使用方式，每一个对象拥有哈希值且

保持唯一是最佳实践，并且相等的对象其哈希值也应该相等。

因此，如果一个类能够理解相等且可哈希化时，其必须显式定义一个`_hash_`方法。

在 Python 的生态系统中哈希被用于多处。其中两个最常见的应用是用于字典的键值与`set`对象。仅有可哈希化的对象可以作为字典的键值。同样，仅有可哈希的对象可以在 Python 的`set`对象中存在。在上述两种情况下，哈希值用于确定一个对象是否是`set`对象的成员以及将某个对象与字典键值比对从而进行键查找。

#### 4) `_format_`方法

另一个常用的 Python 内置函数是`format`函数，该函数可以根据 Python 的格式化规范来格式化不同种类的对象。

任何对象都能提供`_format_`方法，该方法在对象被传递到`format`时调用。该方法接受两个位置参数，第一个为`self`，第二个为格式化规范的字符串。

在 Python 3 中，`str.format`方法倾向使用`_format_`方法替换%操作符来处理字符串内的占位符。如果被传递给`str.format`的对象带有`_format_`方法作为参数，则将调用该方法。

```
>>> from datetime import datetime
>>>
>>>
>>> class MyDate(datetime):
...     def __format__(self, spec_str):
...         if not spec_str:
...             spec_str = '%Y-%m-%d %H:%M:%S'
...         return self.strftime(spec_str)
...
>>>
>>> md = MyDate(2012, 4, 21, 11)
>>>
>>> '{0}'.format(md)
'2012-04-21 11:00:00'
```

由于字符串仅仅使用了占位符`{0}`，且未提供任何额外的格式化信息，如果没有指定任何格式规范，则使用默认规范。但要注意如果提供了格式化规范会发生什么：

```
>>> '{0:%Y-%m-%d}'.format(md)
'2012-04-21'
```

当使用`format`方法时，只有上述情况下会调用`_format_`。如果没有使用`%-`，则不会调用该方法。

#### 5) `_instancecheck_`与`_subclasscheck_`

虽然在 Python 中大多数类型检查使用所谓的鸭子类型(duck typing)(如果`obj.looking()`像一个鸭子的外观，并且`obj.quack()`像鸭子的叫声，则`obj`很可能就是一个鸭子)来完成，但还可以使用内置的`isinstance`方法检查一个对象是否是某个类的实例。类似的，可以使用`issubclass`检查一个类是否继承于另一个类。

极少需要自定义该特征。如果对象是所提供的类或其任意子类(这也是你所希望的)的实

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

例，`isinstance`方法返回 `true`。同样，如果提供的两个参数是同一个类，`issubclass`(尽管名称表示的是子类)返回 `True`(这也是你希望的结果)。

尽管这样，偶尔也需要允许类伪装其身份。Python 2.6 引入了`_instancecheck_`与`_subclasscheck_`方法使其成为可能。这两种方法都接受两个参数，第一个参数是`self`，第二个参数是用于比较的类(与`isinstance`方法的第一个参数一样)。这允许类决定哪个对象可能会伪装成其实例或子类。

## 6) `_abs_` 与 `_round_` 方法

Python 提供了内置的`abs` 与 `round` 函数，分别用于返回绝对值与取整后的值。

虽然很少需要自定义类的上述特征，但可以分别通过定义`_abs_` 与 `_round_` 方法完成自定义。这两个方法都接受一个位置参数(`self`)，并返回一个数字类型的值。

## 5. 集合

很多对象都是其他不同种类对象的集合。最复杂的类功能上是来自属性的集合(以某种有意义的方式排序)与对象中定义的方法集合。

Python 通过几种方式来理解一个对象与另一个对象是否为“成员关系”。例如，对于列表和字典，可以通过表达式 `needle in haystack`(`needle` 是被查找的对象，而 `haystack` 是一个集合)检查一个对象是否是一个集合的成员。

字典由键组成，并且可以通过 `haystack[key]` 基于键值进行查找。同样，大多数对象拥有属性，这些属性在初始化时设置或通过其他方法设置，可以使用点操作符访问(`haystack.attr_name`)。

Python 提供了与所有这些对象交互的魔术方法。

### 1) `_contains_` 方法

`_contains_` 方法在对表达式(如 `needle in haystack`)求值时被调用。该方法接受两个位置参数(`self` 与 `needle`)，如果 `needle` 在集合中，则返回 `True`，否则返回 `False`。

该方法并不能严格确保某一对象在一个集合中，虽然该方法经常用于这种情况下。考虑下面这个表示一段时间区间的类。

```
class DateRange(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __contains__(self, needle):
        return self.start <= needle <= self.end
```

在该示例中，`_contains_` 方法确定了某个日期是否在给定时间区间内。

```
>>> dr = DateRange(date(2015, 1, 1), date(2015, 12, 31))
>>> date(2015, 4, 21) in dr
True
>>> date(2012, 4, 21) in dr
False
```

## 2) \_\_getitem\_\_、\_\_setitem\_\_ 和 \_\_delitem\_\_ 方法

`__getitem__` 方法及其同类方法被用于对集合(如字典)、索引或部分集合(如列表)进行键查找。在上述两种情况中，基本的表达式都是 `haystack[key]`。

`__getitem__` 方法接受两个参数：`self` 与 `key`。该方法在集合能找到元素时返回对应值，否则引发对应的异常。引发何种异常取决于具体情况，但通常都是 `IndexError`、`KeyError` 或 `TypeError`。

`__setitem__` 方法也用于同样情况下，但它用于设置集合中元素的值，而不是用于查找。该方法接受 3 个位置参数：`self`、`key` 与 `value`。

并非所有支持元素查找的对象也同时需要支持元素修改。换句话说，如果需要，可以在定义 `__getitem__` 方法的情况下不定义 `__setitem__` 方法。

最后，`__delitem__` 方法通常在使用 `del` 关键字(例如，`del haystack[key]`)删除键时被调用。

## 3) \_\_getattr\_\_ 与 \_\_setattr\_\_ 方法

Python 类用作集合的另一种主要方式是作为属性与对象的集合。当 `date` 对象包含 `year`、`month` 和 `day` 时，这些都是属性(在本例中被设置为整型)。

无论是通过点(如 `obj.attr_name`)还是使用 `getattr` 方法(比如 `getattr(obj, 'attr_name')`)，`__getattr__` 方法都会在试图获取一个对象的属性时被调用，

然而，与其他魔术方法不同，`__getattr__` 仅在“用常规方式无法找到属性时才被调用”。换句话说，Python 解释器首先进行标准的属性查找，如果发现属性则返回。如果没有匹配属性(换句话说，引发 `AttributeError` 错误)，只有在这种情况下才会调用 `__getattr__` 方法。

在另一方面，其工作机制类似之前讨论过的 `__getitem__()`。该方法接受两个位置参数(`self` 与 `key`)并返回合适的值，或是引发 `AttributeError` 异常。

类似的，`__setattr__` 的用法与 `__getattr__` 相同。其在赋值给一个对象时被调用，无论是通过点操作符还是使用 `setattr` 方法。与 `__getattr__` 不同的是，该方法一直会被调用(否则该方法没有存在的意义)，因此，如果希望使用传统实现时，应该调用基类方法。

## 4) \_\_getattribute\_\_ 方法

`__getattr__` 方法只有在无法找到属性时才被调用，因为这是在常规情况下希望的行为(否则会很容易陷入无限循环的大坑)。然而，`__getattribute__` 与 `__getattr__` 不同，会被无条件调用。

在这里，逻辑顺序是首先调用 `__getattribute__`，在正常情况下负责执行传统的属性查找。如果类定义了 `__getattribute__`，其变为负责调用基类的实现。如果(并且只有在这种情况下)`__getattribute__` 引发 `AttributeError` 异常，调用 `__getattr__` 方法。

## 4.3 其他魔术方法

除了之前描述的魔术方法外，还有其他一些魔术方法。特别是 Python 使用 `__iter__` 与 `__next__` 方法实现了一个迭代协议。由于在第 3 章已经进行过详细讨论，因此这里不再赘述。

类似的，Python 还实现了所谓上下文管理器的丰富语言特性，该特性利用了 `_enter_` 与 `_exit_` 魔术方法。由于在第 2 章已进行详细讨论，因此在此就不再赘述。

## 4.4 小结

由 Python 语言提供的对于类的魔术方法为不同自定义类之间提供了一致的数据模型。这除了能够以可预见的方式为不同类型的对象交互提供纽带之外，还极大地提升了语言的可读性。

并不需要在自定义类中实现上述所有或部分的魔术方法。当编写类时，考虑你所需要的功能。然而，如果功能与已经定义的方法匹配，那么实现魔术方法的方式更加可取，而不是自定义一个方法名称。

在第 5 章中，将学习有关元类的知识。

# 第 5 章 元类

Python 中的类也是一种对象。

这是一个关键概念。在 Python 中，几乎所有的一切都是对象，包括函数和类。这意味着函数与类都可以作为参数提供、以类实例的成员形式存在，且可以完成其他对象所能完成的工作。

类是一个对象意味着什么？第 4 章已经讨论过对象实例化的原理。对象在实例化过程中会调用 `__new__` 与 `__init__` 方法创建新对象，该过程对于类而言也不例外。作为对象的类本身也是另一种类的实例，用于创建类。

负责生成其他类的类就是元类(Metaclass)。“Meta-”是一个 Greek 前缀，意味着“post-”或“after”。例如，亚里士多德的部分作品称为“物理学(The Physics)”，而后的称为“形而上学(Metaphysics)”，这仅仅意味着该部分内容是在物理学之后的成果。然而，赋予该前缀的意义也随着时代而不断演化为对自身某种程度的引用。如果在一个会议中对另一个会议进行规划，该会议就可被称为元会议(meta-meeting)。

本章内容涵盖了元类。首先深入研究 Python 对象模型背后的设计理念，以及元类、类和对象之间的关系，然后探索使用元类的一些特定方式。

## 5.1 类与对象

类与类的实例之间的关系有两重意思，且非常一目了然。首先，类定义了其实例的属性和行为。其次，类可以作为创建实例的工厂。

基于这一点，只需要额外理解的是要意识到该种关系可以继承。当实例化所编写的类时，类作为实例属性和行为的定义，并完成实例的生成。定义类时，仅仅是使用一个特殊

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

的替代语法代替不同类的实例化，即实例化 type 类。

## 5.1.1 直接使用 type

通过直接使用 Type 而非使用 Python 的关键字 Class 创建类，可以很好地说明这个概念。从语法上讲，这种方式十分丑陋，但有助于理解其背后的机制。

因此，考虑下面一组简单的类：

```
class Animal(object):
    """A class representing an arbitrary animal."""

    def __init__(self, name):
        self.name = name

    def eat(self):
        pass

    def go_to_vet(self):
        pass

class Cat(Animal):
    def meow(self):
        pass

    def purr(self):
        pass
```

显然，Animal 类代表一个动物，且定义了一些动物能够做的事情，比如吃东西和被带到兽医处。Cat 子类额外定义了猫叫与猫喘鸣，其他动物并不包含这些函数(方法内留空，读者可以自己想象)。

这里所发生的是 Python 解释器执行到代码段的开始部分时，即 class Animal(object) 这行，实例执行的是调用 type 的构造函数。前所述，type 是 Python 中的内置类，该类是其他类对象的默认类。它是创建其他类的默认类——或者说是默认元类。

然而，可以直接通过 type 创建类。type 构造函数接受 3 个位置参数：name、bases 与 attrs。其中 name 参数(一个字符串)仅仅是类的名称。bases 参数是该类的基类的元组。Python 支持多重继承，这也是称为元类的理由。如果只继承单个类，只需要发送只包含一个对象的元组。attrs 参数是类中所有属性的字典。

### 1. 创建类

下面代码基本等价于之前的 class Animal 代码块：

```
def init(self, name):
    self.name = name
```

```
def eat(self):
    pass

def go_to_vet(self):
    pass

Animal = type('Animal', (object,), {
    '__doc__': 'A class representing an arbitrary animal.',
    '__init__': init,
    'eat': eat,
    'go_to_vet': go_to_vet,
})
```

显然，这并不是实例化新类的最佳方式。另外要注意，上面的代码只是基本等价于之前的代码块。也有一些区别，最值得注意的区别是该代码块中的 `init`、`eat`、`go_to_vet` 方法并没有附加到位于命名空间下的类。这一点值得注意，但对于理解本例的目的来说并不重要。

下面关注对于 `type` 的调用。第一个参数只是一个字符串'Animal'。这里有一些重复。发送该字符串，将其作为类名称，但也将 `type` 调用的结果赋给变量 `Animal`。`Class` 关键字可以为你完成上面的操作。由于这里直接调用 `type`，因此必须手动将结果赋给变量，就像在为其他类的新实例所做的一样。

第二个参数是只包含一个成员的元组: `{object,}`。这表示 `Animal` 类继承自 `object`，与之前的类一样。需要在元组尾部加逗号，以 Python 解释器将其作为元组。括号在 Python 中有其他用处，因此对于只包含一个元素的元组需要在结尾处加括号。

第三个参数是一个字典，定义了类的属性，等价于 `class` 代码块内缩进的代码部分。首先定义了映射到原始类中函数的函数，然后将这些函数传入 `attrs` 字典。字典的键用于确定类中属性的名称。这里值得注意的是文档字符串。Python 解释器自动将类中的文档字符串赋给 `_doc_` 属性。由于是直接实例化 `type`，因此必须手动赋值。

## 2. 创建子类

可以用类似的方式创建 `Cat` 类，如下所示：

```
def meow(self):
    return None

def purr(self):
    return None

Cat = type('Cat', (Animal,), {
    'meow': meow,
    'purr': purr,
})
```

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

该段代码与之前基本一样。最大的区别是现在继承的是 Animal 对象而不是 object 对象。这里传递的是 Animal 类自身。另外，注意 type 的第二个参数仍然是只包含一个元素的元组。无须传递(Animal, object)。实际上 object 作为 Animal 的基类已经融入 Animal 类。只有在多重继承的情况下才需要传递包含多个元素的元组。

## 5.1.2 type 链

考虑下面 Cat 类的实例：

```
louisoix = Cat(name='Louisoix')
```

注意，这行代码包含 3 个部分：louisoix 是一个对象，也是 Cat 的实例；Cat 类也是一个对象(这是由于类也是对象)，同时也是 type 的实例；type 是(继承)链的顶端。

也可以用另一种方式观察上述代码。将一个对象传递给 type，会返回该对象的类，如下所示：

```
>>> type(5)
<type 'int'>
```

那么，观察下述链：

```
>>> type(louisoix)
<class '__main__.Cat'>

>>> type(Cat)
<class 'type'>
>>> type(type)
<class 'type'>
```

在此，type 类是基类，它位于链的顶端，因此 type(type)返回其自身。

注意：在 Python 2 的终端中，输出结果显示为<type 'type'>，而不是<class 'type'>。这没问题，仍然是同一个类型，仅仅是在终端窗口的输出结果上有所不同。

## 5.1.3 type 的角色

type 是 Python 中的主要元类。默认情况下，使用 class 关键字创建的普通类都使用 type 作为其元类。

通俗来讲，可以为类(Cat)和其实例(louisoix)引用 type 作为其元类。

除此之外，type 也是其他元类的基类。这可以通过 object 作为所有类的基类进行类比。正如 object 是类继承链中的最高级，type 也是元类层级的最高级。

## 5.2 编写元类

从语法角度来讲，编写元类非常简单。只需要声明一个继承自 type 的类(使用 class 关键字)。对象模型的精妙之处就在此。类只是对象，元类也只是类。元类的行为继承自 type。因此，任何 type 的子类都可以作为元类。

在开始介绍示例之前，注意不要声明或使用任何不是直接继承自 type 的元类。这会破坏 Python 的多重继承机制。Python 的继承模型要求任何类都只能有一个元类。只有(并且只能在这种情况下)一个元类是另一个元类的直接子类(在这种情况下实际上最终使用的是子类)时，才能够接受一个类继承自使用不同子类的两个类。如果尝试实现一个不继承 type 类的元类，同时继承任何使用该元类的类与其他使用 type 作为元类的类(几乎所有类都使用 type 作为元类)，都会破坏多重继承。不会希望这么做。

### 5.2.1 \_\_new\_\_方法

自定义元类必须定义的最重要方法是\_\_new\_\_方法。该方法实际上用于处理类的创建，且必须返回一个新类。

\_\_new\_\_方法是一个类方法(并不需要显式定义)。在自定义元类中发送到\_\_new\_\_方法的参数必须与发送到 type 的\_\_new\_\_方法的参数保持一致，必须有 4 个位置参数。

第一个参数是元类自身，伪装成一个参数的方式与绑定方法类似。按照惯例，将该参数命名为 cls。

除此之外，\_\_new\_\_方法接受如下 3 个位置参数：

- 首先是一个 string 类型的类名称(name)。
- 其次是类继承基类的元组(bases)。
- 最后是一个类应该包含属性的字典(attrs)。

元类中\_\_new\_\_的大多数自定义实现都应该确保其调用基类实现，并在此代码之后完成自定义代码。

### 5.2.2 \_\_new\_\_与\_\_init\_\_方法

在此回忆一下\_\_new\_\_与\_\_init\_\_的区别。在类或元类中，\_\_new\_\_方法负责创建和返回对象。

与之相反，\_\_init\_\_方法负责在对象创建后自定义对象，且不返回任何值。

在普通类中，通常都不会自定义\_\_new\_\_方法。而定义自定义\_\_init\_\_方法却非常普遍。这是由于由 object 提供的\_\_new\_\_实现基本足够，但也是有必要的。重载\_\_new\_\_方法(即使直接继承 object)需要调用基类方法并小心返回结果(新实例)。相反，重载\_\_init\_\_方法非常容易且相对没有风险。对象的\_\_init\_\_实现是空操作，且该方法不返回任何值。

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

编写自定义元类时，行为则会发生变化。自定义元类通常会重载 `__new__` 方法，但通常不会实现 `__init__` 方法。重载 `__init__` 方法时时，要记住必须总是调用基类的实现。实际上，`type` 的 `__new__` 实现会提供要用到的对象并返回新的实例。

## 5.2.3 元类示例

在深入自定义行为的元类之前，先考虑下面这个仅仅展示所有前面提到功能的自定义元类：

```
class Meta(type):
    """A custom metaclass that adds no actual functionality."""

    def __init__(cls, name, bases, attrs):
        return super(Meta, cls).__new__(cls, name, bases, attrs)
```

本讨论还未探索在类创建的过程中如何使用 `class` 关键字给元类赋值(稍后将详细介绍)。但可以通过直接调用 `Meta` 元类来创建使用 `Meta` 元类的类，这类似于之前直接调用 `type` 的例子。

```
>>> C = Meta('C', (object,), {})
```

该代码创建一个类 `C`，该类是 `Meta` 的实例而不是 `type` 的实例。观察下面的输出：

```
>>> type(C)
<class '__main__.Meta'>
```

这与看到的正常类不同，如下所示：

```
>>> class N(object):
...     pass
...
>>> type(N)
<class 'type'>
```

## 5.2.4 元类继承

值得注意的是，元类可继承。因此，`C` 的子类也是 `Meta` 的实例，而不是 `type` 的直接实例，如下面的代码和图 5-1 所示。

```
>>> class D(C):
...     pass
...
>>> type(D)
<class '__main__.Meta'>
```

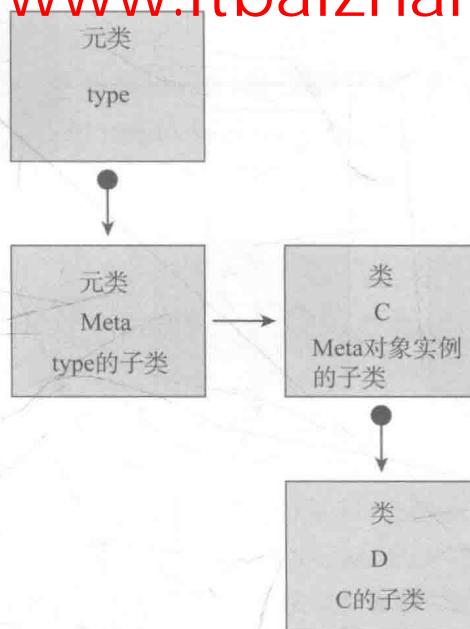


图 5-1 元类继承

在本示例中，D 是 Meta 的实例，这并不是由于 D 显式声明了元类，或是调用 Meta 创建了它，而是由于它的基类是 Meta 的实例，因此 D 也是 Meta 的实例。

这里需要注意的重点是，在大多数情况下类只有一个元类。即使在多继承的情况下，也是如此。如果一个类的多个子类有不同的元类，Python 的解释器会通过检查元类的起源来解决该冲突，如果其元类关系为直接继承，则最终使用子类。

考虑下面既是 C(Meta 的实例)又是 N(type 的实例)的子类。

```
>>> class Z(C, N):
...     pass
...
>>> type(Z)
<class '__main__.Meta'>
```

图 5-2 展示了在代码中所发生的事情。

代码中发生了什么？Python 解释器被告知创建类 Z，Z 同时是 C 和 N 的子类。这等同于 type('Z', (C, N), {})。

首先，Python 解释器检查 C，并发现 C 是 Meta 的实例。然后检查 N，并发现 N 是 type 的实例。这是潜在的冲突。这两个基类却不同的元类。然而，Python 解释器同时发现 Meta 是 type 的直接子类，因此得知其可以安全地使用 Meta，并这样做了。

如果两个元类中的一个并不是另一个的直接子类会发生什么？现在存在一个冲突，而 Python 解释器并不知道该如何解决，因此会拒绝尝试执行，如下所示：

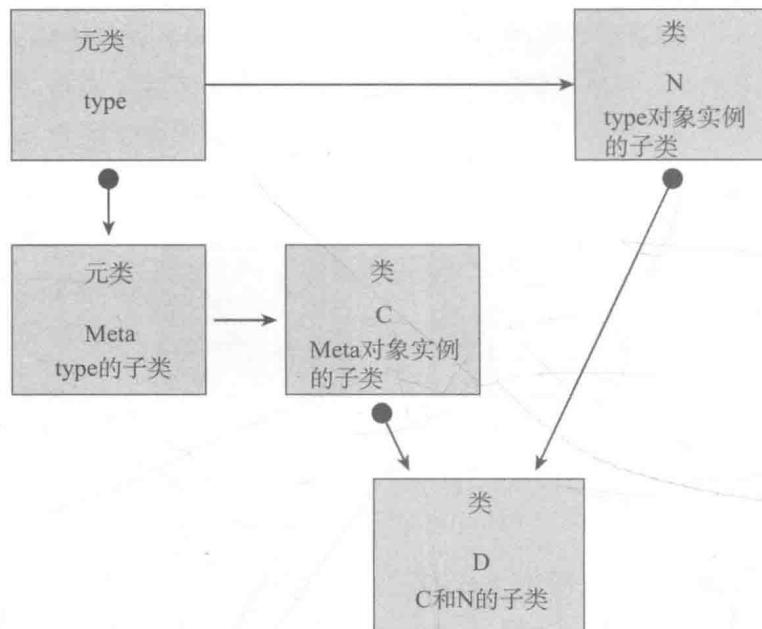


图 5-2 子类的元类继承

```
>>> class OtherMeta(type):
...     def __new__(cls, name, bases, attrs):
...         return super(OtherMeta, cls).__new__(cls, name, bases, attrs)
...
>>> OtherC = OtherMeta('OtherC', (object,), {})
>>>
>>> class Invalid(C, OtherC):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __new__
TypeError: Error when calling the metaclass bases
    metaclass conflict: the metaclass of a derived class must be a (non-
        strict) subclass of the metaclasses of all its bases
```

出现这种结果是由于 Python 对于每个类只能有一个元类，并且在不明确的情况下拒绝尝试使用其中的元类。

## 5.3 使用元类

在深入研究更复杂的元类之前，先探索一下如何使用元类。虽然允许直接实例化元类（正如之前 type 和 Meta 示例中展示的），但并不是最佳方式。

Python 中的 class 提供了在不使用 type 作为元类的情况下，声明元类的机制。然而，根据使用 Python 版本的不同，定义所使用元类的语法也不同。

### 5.3.1 Python 3

在 Python 3 中，元类与基类(如果存在)在一起声明。语法类似于在函数声明或调用中的关键字参数，而关键字参数为 metaclass。

之前，通过直接调用 Meta 创建类 C。下面是 Python 3 中推荐的方式：

```
class C(metaclass=Meta):  
    pass
```

此处的 class 关键字调用与直接调用 Meta 创建类完成的任务相同，但还是推荐使用这种语法风格。

这里要注意的是，并没有显式指定 object 作为基类。在本书使用的大多数示例中，都显式指定 object 作为基类。这是因为本书希望示例既能在 Python 2 中运行，也可以在 Python 3 中运行。在 Python 2 中，这样做很重要，因为继承于 object 的子类才是使类成为“新式类”(很久以前在 Python 2.2 中被引入，用于改变 Python 的方法解析顺序以及其他函数的运作机制)的关键。直接继承 object 的方式用于确保向后兼容，这迫使开发人员“可选”地使用新式类。

在 Python 3 中，该版本是向后兼容的，所有的类都是新式类，无须显式直接继承自 object，因此在此不必这样做。也就是说，之前的代码等同于下述代码：

```
class C(object, metaclass=Meta):  
    pass
```

这种风格能够让你更直观地区分基类之间的不同，这里使用的语法风格类似于函数声明中的位置参数部分，而不是以关键字参数语法的形式来声明元类。这些参数必须以给定的顺序声明，metaclass 作为最后一个参数，就像函数参数那样。

在 Python 3 中，当直接继承 object 时，可以使用任意语法风格(显式地包含或省略)。

### 5.3.2 Python 2

对于元类声明，Python 2 有完全不同的语法。Python 3 并不支持 Python 2 的语法，Python 2 也不支持 Python 3 的语法(跳到 5.3.3 小节，查看如何在不同版本中正确地声明元类)。

声明元类的 Python 2 语法实际上是将 \_\_metaclass\_\_ 属性赋给类。考虑之前调用 Meta 创建类 C 的代码，在 Python 2 中等价于如下代码：

```
class C(object):  
    __metaclass__ = Meta
```

在本例中，元类在类中赋值。这没有问题。Python 解释器在调用类关键字时搜索元类赋值代码并用 meta 而不是 type 来创建新类。

### 5.3.3 需要跨版本执行的代码怎么办

由于 Python 3 中对 Python 语言做了引入了向后不兼容的部分，因此 Python 开发人员需

要考虑将同样代码运行在不同版本而得到相同结果的策略。

其中一种最流行的方式是引入名为 six 的工具，该工具由 Benjamin Peterson 编写，并由 PyPI 提供。

six 工具为声明元类提供了两种方法：通过创建一个替身类并将其用作直接基类，或者使用装饰器添加元类。

第一种方法(也就是替身类方法)看上去如下所示：

```
import six

class C(six.with_metaclass(Meta)):
    pass
```

这里发生了什么？six.with\_metaclass 创建了一个直接继承 object 的虚类，并使用 Meta 作为其元类，该类并不完成任何工作。通过将该类用作 C 的基类，并基于元类与类继承交互的方式(之前讨论过)，无论在任何 Python 版本下，C 现在都是 Meta 的实例。

根据使用元类的不同，有时上述解决方案可能不起作用。因为 six.with\_metaclass 实际上是实例化一个类，某些元类可能会完成一些任务，而完成的任务有可能会与抽象基类不兼容。

six 提供了另一种方式，将元类赋给一个类，即使用装饰器：@six.add\_metaclass。语法如下：

```
import six

@six.add_metaclass(Meta)
class C(object):
    pass
```

此时在 Python 2 或 Python 3 中的实现结果变为相同。类 C 使用 class 关键字与 Meta 元类(而不是使用 type 元类)创建。装饰器在不实例化抽象类的情况下完成了这项工作。

### 5.3.4 跨版本兼容性在何时重要

因为在 Python 2 与 Python 3 中语法不兼容，所以在此探讨何时使用“纯粹”的语言方式就变得非常重要，这也正是介绍 six 工具的时机。

在不深入探讨理论的前提下，简略的指导原则是，如果正在使用 Python 2，那么未来可能会将代码迁移到 Python 3 中，并希望编写跨版本兼容的代码。这使得使用 six 工具完成很多任务变得合适(元类只是其中之一)，因此将 six 工具引入代码库就非常明智。而与之相反，如果已经使用 Python 3 环境，很少需要再将代码迁移回 Python 2，那么直接编写 Python 3 的代码应该没有问题。

## 5.4 何时使用元类

学习元类时，一个比较棘手的问题是，需要理解何时是最适合使用元类。实际上，大多数代码使用传统的类与对象的结构都没有问题，并不真的需要使用元类。

同时，使用元类额外增加了一层复杂性。实际上阅读代码的次数要远大于写代码的次数，因此，通常使用简单的办法实现目标并解决问题是最佳方式。

也就是说，当适合使用元类时，元类能够让代码变得更加容易理解。能够发现元类可以使代码更加简单而不是更加复杂是一项有价值的技能。

### 5.4.1 说明性类声明

使用自定义元类最常见的原因是在类声明与类结构之间创建描述，尤其是在创建供其他开发人员使用的 API 时。

#### 1. 现有的示例

首先，考虑一个来自 wild 的示例。很多 Python 开发人员已经对 Django 模型非常熟悉，Django 模型是一个流行的 Web 框架。该模型通常对应于关系数据库中分散的数据库表。

Django 模型的声明十分简单。下面的示例模型表示一本书：

```
from django.db import models

class Book(models.Model):
    author = models.CharField(max_length=100)
    title = models.CharField(max_length=250)
    isbn = models.CharField(max_length=20)
    publication_date = models.DateField()
    pages = models.PositiveIntegerField()
```

根据对 Python 中普通类的了解，你认为这里会发生什么？很明显，`models.CharField`、`models.DateField` 及其他类似属性是对象的实例。所以，创建 `Book` 实例时，如果需要访问这些属性，则需要获取这些实例。

如果熟悉 Django，就知道事实并非如此。如果希望获得 `Book` 实例的 `author` 属性，将会得到字符串。对于 `title` 与 `isbn` 属性也是如此。`publication_date` 属性则为 `datetime.date` 对象，`pages` 为 `int` 对象。如果未将属性提供给模型，则返回结果为 `None`。

这是如何发生的？用于查看时区分该类如何声明(用于生成它所提供的代码)与如何构建的底层原理是什么？声明该类时，它的属性是复杂的域对象。但查看该类的实例时，同一个属性被赋给特定书的值。

对于上面问题的回答是：该 Django 模型使用随 Django 一同发行的元类 `ModelBase`。使用 Django 时，它被隐藏起来，因为 `django.db.models.Model` 使用了 `ModelBase` 元类。因此，子类可以直接获得它。

ModelBase 完成了很多工作(Django 是一个成熟的框架，它的 ORM 已经经历了多个版本的迭代)。但它所完成的主要工作是在 Django 的模型类的声明方式与它们对象的组织方式之间进行转换。对于 Django 来说，有一个非常简单直接的模型声明方式。一个模型代表一个表；模型中的属性对应于表中的列。

Django 生态系统中的实例表示表中的行。当访问实例中的一个域时，真正想要的其实是该行的值。因此，特定的 Book 实例可能是 The Hobbit，因此在本例中你会希望 book.title 是'The Hobbit'。

本质上，在此使用元类是可取的，因为它使声明 Book 类与访问 Book 实例的数据变得很整洁，即使这些属性不匹配，也可使用凭直觉获取的 API。

## 2. 示例说明

深入 ModelBase 实现的每一个细节已经超出了本书的讨论范围，但该特定概念的实现实际上非常简单。

首先，创建模型类时，回忆一下该类的属性被传递给字典中的元类的 `__new__` 方法，该字典通常被命名为 attrs。在本示例模型中，该字典会包含 author、title 等作为键。这些键对应的值是 Field 对象(所有这些类都是 django.db.models.Field 的子类)。

ModelBase 元类有一个用于迭代 attrs 字典以寻找 Field 子类的 `__new__` 方法。该方法发现的任何 attrs 字典的域都会弹出到另一个位置——另一个名为 fields 的字典(实际上在该类的另一个对象 `_meta` 中)，除了了解实际 field 类的位置之外，其他实现细节并不是特别重要，隐藏内部 Django 代码，仅在需要时提取这些代码。而只想编写 Django 模块的普通人并不需要查看这些代码。

之后，创建实例时，对应于域的属性被实例化并设置为 `None`(除非为该行提供了默认值或特定值)这种情况下优先使用该值。现在，访问该实例的属性时，返回的是该行的值而不是 Field 子类。与之类似，值可以以一种直观的方式编写，而不再需要 Field 参与。

本质上，元类所完成的工作是接受类声明，重组类属性的结构，然后使用新结构创建类。

## 3. 为什么该示例充分利用了元类

设计 API 时该模式非常有效。一个优秀 API 的主要目标是尽可能的简单，并且尽量少包含用例代码。这意味着不仅声明类应该简单直接，而且使用该类也应该简单直接。

在 Django 模型的示例中，这两个目标有些许冲突。而 ModelBase 元类解决了该冲突。

使用元类是填补这一缺口的优秀方式。本质上它们通过将类声明置于前台，然后在 `__new__` 中将该类的声明转换为实际类的结构。

### 5.4.2 类验证

元类的另一个重要作用是用于类验证。如果一个类必须遵循特定接口，元类将会是强制使其符合规范的有效方式。通常，更倾向于使用合理的默认值解决该问题。但有时这并

不现实。

例如，一个类需要设置两个属性中的一个，但不需要两个属性都设置。如果其中一个属性设置为 `unset`(与设置为 `None` 相反)，就很难通过合理的默认值实现。

可以使用元类实现这个理念。下面的简单元类需要类包含 `foo` 属性或 `bar` 属性：

```
class FooOrBar(type):
    def __new__(cls, name, bases, attrs):
        if 'foo' in attrs and 'bar' in attrs:
            raise TypeError('Class %s cannot contain both `foo` and '
                            '`bar` attributes.' % name)
        if 'foo' not in attrs and 'bar' not in attrs:
            raise TypeError('Class %s must provide either a `foo` '
                            'attribute or a `bar` attribute.' % name)
        return super(FooOrBar, cls).__new__(cls, name, bases, attrs)
```

下面的 Python 3 类使用该元类并遵循该接口：

```
>>> class Valid(metaclass=FooOrBar):
...     foo = 42
...
>>>
```

一切正常工作。如果尝试对这两个属性都设置值或都不设置值，会发生什么情况呢？

```
>>> class Invalid(metaclass=FooOrBar):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __new__
TypeError: Class Invalid must provide either a `foo` attribute or a `bar` attribute.
>>>
>>> class Invalid(metaclass=FooOrBar):
...     foo = 42
...     bar = 42
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __new__
TypeError: Class Invalid cannot contain both `foo` and `bar` attributes.
```

这个特定的实现存在一个问题。它的子类并不会继承该功能。原因是元类直接检查 `attrs` 属性，但这只包含所声明的类的属性集。它并不知道任何继承自基类的属性。

```
>>> class Valid(metaclass=FooOrBar):
...     foo = 42
...
>>> class AlsoValid(Valid):
...     pass
```

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __new__
TypeError: Class AlsoValid must provide either a `foo` attribute or a `bar` attribute.
```

这是个问题。毕竟，`AlsoValid`类也有效。它包含一个`foo`属性。因此必须用另一种通过检查基类属性实现`FooOrBar`元类的方式。

```
class FooOrBar(type):
    def __new__(cls, name, bases, attrs):
        answer = super(FooOrBar, cls).__new__(cls, name, bases, attrs)
        if hasattr(answer, 'foo') and hasattr(answer, 'bar'):
            raise TypeError('Class %s cannot contain both `foo` and '
                            '`bar` attributes.' % name)
        if not hasattr(answer, 'foo') and not hasattr(answer, 'bar'):
            raise TypeError('Class %s must provide either a `foo` '
                            'attribute or a `bar` attribute.' % name)
        return answer
```

这里的区别是什么？这次是在实例化类返回之前检查它的属性，而不是查看`attrs`字典。

新类在`__new__`方法的第一行调用`type`的构造函数从基类获得所有属性。因此，对`hasattr`的调用正常工作，而不必管属性是在当前类中声明还是继承自基类。

在不使用元类的情况下可以实现这一点吗？完全可以。可以通过编写一个简单方法接受该类作为参数并完成同样的检查。实际上，这是一种用于装饰器的优秀方式，但必须手动将类发送给验证方法。而使用元类，就只需要在创建类时处理。有时，显式检查更可取；但其他时候却不然。这仅仅取决于用例。

## 5.4.3 非继承属性

还可以将元类用作一种工具，使类中特定的属性不会自动继承。想要这样做的最常见场景是与其他元类行为结合。例如，假设一个元类为它的类提供功能，但一些类被创建为抽象类，因此并不希望所谓的功能在这种情况下执行。

实现这一点，显而易见的一种方式是允许类设置`abstract`属性，并且仅在`abstract`未设置或设置为`False`时才执行元类的特殊功能。

```
class Meta(type):
    def __new__(cls, name, bases, attrs):
        # Sanity check: If this is an abstract class, then we do not
        # want the metaclass functionality here.
        if attrs.get('abstract', False):
            return super(Meta, cls).__new__(cls, name, bases, attrs)

        # Perform actual metaclass functionality.
        [...]
```

但这种方法存在一个问题。与其他属性类似，子类将继承 `abstract` 属性。这意味着任何子类都需要显式声明自己并不是抽象类，这显得很怪异。

```
class AbstractClass(metaclass=Meta):
    abstract = True

class RegularClass(AbstractClass):
    abstract = False
```

直觉上，希望在所有抽象类中声明 `abstract` 属性，但对于该属性不应该继承。事实证明这很容易实现，因为并不需要像元类那样读取 `attrs` 字典，元类可以修改该属性，并在不需要时将该属性丢弃。

在本例中，可以通过从 `attrs` 字典中弹出 `abstract` 值来实现这一点，如下所示：

```
class Meta(type):
    def __new__(cls, name, bases, attrs):
        # Sanity check: If this is an abstract class, then we do not
        # want the metaclass functionality here.
        if attrs.pop('abstract', False):
            return super(Meta, cls).__new__(cls, name, bases, attrs)

        # Perform actual metaclass functionality.
        [...]
```

这里的区别不明显，但很重要。`abstract` 属性在实际创建的类中被完全移除。在本例中，`AbstractClass` 不会获得元类的功能，而且实际上 `abstract` 属性已经不存在。最重要的是，这意味着子类不再继承该属性，这也与希望的行为完全一致。

## 5.5 显式选择的问题

之前提供的两个例子都是元类可以解决但不使用元类也能解决的问题。实际上，元类的大多数主要用例都可以不用元类解决。

例如，类装饰器可以很容易地处理类遵循特定接口的需求。但装饰所有类是一件繁琐的工作，装饰器可以确保设置了 `foo` 或设置了 `bar`，但很难确保同时设置了这两个值。

这就提出了一个重要问题。使用元类实现上述需求的价值是什么？元类可以提供而类装饰器无法提供的价值是什么？

对于这类问题的回答大部分取决于最终类的使用方式。使用元类与使用类装饰器的关键区别是使用装饰器要求将类装饰器显式应用到每一个子类。如果程序员在实现子类的过程中忘记应用装饰器，就不会进行检查。

与之相反，元类是自动的，并且对声明和使用类的程序员透明。很少(如果存在)有 API 要求程序员直接使用元类，但其中很多 API 要求程序员继承由 API 包提供的基类。通过将元类赋给该基类，所有的子类也都会收到该元类。这使最终程序员即使不了解元类的功能，

也能应用该功能。

简言之，Zen of Python 中的第一行阐述“显式比隐式更好”。但与该文档中的大多数事情类似，该格言适用大多数场景，直到遇到不适合的场景。例如，提到外来信息或样板代码时，显式更好。与之类似，有时显式只是意味更容易维护，但并不总是合适。

## 5.6 meta-coding

对于元类的操作越来越多时，元类的优势就开始凸显。将每个 Django 模型使用一个显式的装饰器进行标记并不合理且难以维护。

与之类似，考虑“元编码(meta-coding)”的情形。在该上下文中，术语 meta-coding 指代用于查看应用程序中其他代码的代码。例如，考虑应该记录其自身代码的代码。

元类使一个类的实例记录所有对它自身的方法调用很容易实现。下面的元类会导致其类记录它的函数调用(除了将结果打印到标准输出 sys.stdout，而不是实际记录日志之外)：

```
class Logged(type):
    """A metaclass that causes classes that it creates to log
    their function calls.

    """
    def __new__(cls, name, bases, attrs):
        for key, value in attrs.items():
            if callable(value):
                attrs[key] = cls.log_call(value)
        return super(Logged, cls).__new__(cls, name, bases, attrs)

    @staticmethod
    def log_call(fxn):
        """Given a function, wrap it with some logging code and
        return the wrapped function.
        """

        def inner(*args, **kwargs):
            print('The function %s was called with arguments %r and '
                  'keyword arguments %r.' % (fxn.__name__, args, kwargs))
            try:
                response = fxn(*args, **kwargs)
                print('The function call to %s was successful.' %
                      fxn.__name__)
                return response
            except Exception as exc:
                print('The function call to %s raised an exception: %r' %
                      (fxn.__name__, exc))
                raise
        return inner
```

首先查看这里发生了什么。Logged 被声明为 type 的子类，这意味着 Logged 是一个元类。Logged 类有一个\_\_new\_\_方法，该方法完成的工作是迭代 attrs 字典中的所有属性，检

查它们是否为可调用的(使用 Python 的内置函数 `callable`)，如果是，对其进行包装。

包装函数自身非常简单，尤其是在你已经对装饰器的概念很熟悉的情况下。它声明了一个执行一些逻辑(在本例中，调用 `print`)的局部函数，调用作为参数传递给 `log_call` 的函数。如果想了解更多关于该模式的信息，参阅第 1 章器，其中大量使用了该模式。

类使用该元类时会发生什么？考虑下面使用 `Logged` 作为其元类的 Python 3 类：

```
class MyClass(metaclass=Logged):  
    def foo(self):  
        pass  
  
    def bar(self):  
        raise TypeError('oh noes!')
```

创建 `MyClass` 的实例时，会发现调用其方法变为如下所示：

```
>>> obj = MyClass()  
>>> obj.foo()  
The function foo was called with arguments (<__main__.MyClass object at  
0x1022a37f0>,) and keyword arguments {}.  
The function call to foo was successful.
```

如果尝试调用 `obj.bar()`，会收到异常。

```
>>> obj.bar()  
The function bar was called with arguments (<__main__.MyClass object at  
0x1022a37f0>,) and keyword arguments {}.  
The function call to bar raised an exception: TypeError('oh noes!',)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 19, in inner  
  File "<stdin>", line 5, in bar  
TypeError: oh noes!
```

机智的读者可能已经注意到，实例化 `MyClass` 时，为什么没有调用 `__init__` 的日志？毕竟，`__init__` 是调用可调用的。看上去仅仅是对 `foo` 与 `bar` 的调用才会记录日志。

但是，回忆一下，元类遍历 `attrs` 字典中的所有属性，并且并没有在 `MyClass` 类中显式定义 `__init__`。`__init__` 方法继承自 `object`，这正是你希望的行为。否则，子类会导致 `log_call` 装饰器对同一个可调用函数重复应用，从而导致重复的 `print` 语句。

但是，通过显式定义 `__init__`，可以观察到打印信息的行为。

```
>>> class MyClass(metaclass=Logged):  
...     def __init__(self):  
...         pass  
...  
>>>  
>>> obj = MyClass()  
The function __init__ was called with arguments (<__main__.MyClass object
```

```
at 0x1022a3550>,) and keyword arguments {}.  
The function call to __init__ was successful.
```

同时还要注意，即使并没有在 Python shell 中显式调用 `__init__`，仍然会记录该方法，这是由于在创建新实例时 Python 解释器在底层调用了 `__init__` 方法。

然而，这很重要，该行为只有在创建类时出现。如果一个方法是在类创建之后被添加的（通常在任何情况下都不应该出现），该方法就不会被包装。

```
>>> MyClass.foo = lambda self: 42  
>>> obj.foo()  
42
```

在本例中，对 `foo` 的调用不会输出日志信息，这是由于 `MyClass` 已经被创建，因此元类已经完成了它的工作。因此，只得到一个普通函数调用而不是包装后的函数调用。

## 5.7 小结

在 Python 中，元类是非常强大的工具。类是一等对象这一事实使得在类声明的外部可以对类进行操作。使用元类就可以实现这一点。

Python 语言中元类的出现克服了很多其他面向对象语言的限制——面向对象的语言在编码时要静态声明类。

最终结果是 Python 的对象模型是世界上最好的模型。它结合了传统类结构语言的简单性与遵循其他模型语言的强大功能，如 JavaScript 与 LUA 中的原型继承。

一个常见的错觉是元类难以理解。但 Python 对象模型的部分强大之处在于其简单性与一致性。实际上，元类是一种非常直观的实现，它为 Python 语言添加了强大的功能。

第 6 章将介绍创建类的另一种方式——即席创建类。

# 6

第 章

## 类 工 厂

正如第 5 章“元类”中所述，Python 类也是对象。类在 Python 中为一等公民的事实使得采用其他强大的设计模式成为可能。类工厂就是这类模式中的一种。本质上，类工厂就是一个在运行时创建类的函数。该概念允许创建类时根据情况决定其属性，比如说，根据用户输入创建属性。

本章涵盖类工厂的内容，首先回顾即席生成类，然后展示如何在函数中实现这一点。之后，还阐述了几种常见的应用类工厂的场景。

### 6.1 类型回顾

回忆一下第 5 章中的讨论，正如 Python 中的其他对象一样，类也是由一个类实例化。例如，假如创建一个名为 Animal 的类，如下所示：

```
class Animal(object):
    """A class representing an arbitrary animal."""

    def __init__(self, name):
        self.name = name

    def eat(self):
        pass

    def go_to_vet(self):
        pass
```

Animal 类在其构造函数被调用时负责创建 Animal 对象。与 Animal 类创建对象的方式相同的是，Animal 本身也是一个对象，它的类是 type——Python 中一个用于创建其他类的

内置类。

`type` 是基本元类，自定义元类(如第 5 章所述)继承 `type` 类。

也可以直接调用 `type` 类代替 `class` 关键字创建一个类。`type` 接受 3 个位置参数：`name`、`bases` 与 `attrs`，分别对应于类名称、类的基类(以元组的形式)以及类的属性(字典类型)。

## 6.2 理解类工厂函数

顾名思义，类工厂函数是一个用于创建并返回类的函数。

考虑之前的 `Animal` 示例。可以使用 `type` 而不是 `class` 关键字创建一个与该类等价的类，如下所示：

```
def init(self, name):
    self.name = name

def eat(self):
    pass

def go_to_vet(self):
    pass

Animal = type('Animal', (object,), {
    '__doc__': 'A class representing an arbitrary animal.',
    '__init__': init,
    'eat': eat,
    'go_to_vet': go_to_vet,
})
```

有几个原因导致这种方式并不理想。其中一个原因是这种写法会将函数置于和 `Animal` 同一层命名空间下。通常，不使用 `Class` 关键字而直接使用 `type` 并不是理想的办法，除非真需要这么做。

然而，有些时候还真需要这么做。在这类情况下，可以通过将代码放入一个函数中从而减少混乱，并将函数分发使用。这就是一个类工厂。考虑下面使用函数创建示例 `Animal` 类的情况：

```
def create_animal_class():
    """Return an Animal class, built by invoking the type
    constructor.
    """
    def init(self, name):
        self.name = name

    def eat(self):
        pass
```

```
def go_to_vet(self):
    pass

    return type('Animal', (object,), {
        '__doc__': 'A class representing an arbitrary animal.',
        '__init__': init,
        'eat': eat,
        'go_to_vet': go_to_vet,
    })
```

这里发生了什么变化？之前混入到命名空间(以及 Animal 类本身的创建)中的 init、eat 与 go\_to\_vet 函数全部移到了 create\_animal\_class 函数中。

现在，就可以通过调用上述函数获得一个自定义创建的 Animal 类，如下所示。

```
Animal = create_animal_class()
```

这里需要注意的重点是，如果多次调用 create\_animal\_class 函数会返回不同的类。也就是说，尽管所返回的类有相同的名称与属性，其实它们并不是同一个类。这些类之间的相似性是基于每次运行函数时都会赋值相同的字典键与相似的函数。

换句话说，所返回的类之间的相似性并不确定。函数不能接受一个或多个参数并返回不同类并没有具体的原因。实际上，这也是类工厂函数的整个目的。

考虑下面多次调用 create\_animal\_class 函数所返回的不同类：

```
>>> Animall = create_animal_class()
>>> Animal2 = create_animal_class()
>>> Animall
<class '__main__.Animal'>
>>> Animal2
<class '__main__.Animal'>
>>> Animall == Animal2
False
```

类似的，考虑下面的实例：

```
>>> animal1 = Animall('louisoix')
>>> animal2 = Animal2('louisoix')
>>> isinstance(animal1, Animall)
True
>>> isinstance(animal1, Animal2)
False
```

尽管这两个类内部都是调用 Animal，但这并不是同一个类。它们是函数两次执行所返回的不同结果。

本例通过调用 type 创建 Animal 类，但这并不是必须的。使用 class 关键字创建类更加简单。即使在函数中使用 class 关键字并在函数结尾部分返回类也同样生效：

```
def create_animal_class():
    """Return an Animal class, built using the class keyword
```

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
and returned afterwards.  
"""  
class Animal(object):  
    """A class representing an arbitrary animal."""  
    def __init__(self, name):  
        self.name = name  
  
    def eat(self):  
        pass  
  
    def go_to_vet(self):  
        pass  
  
    return Animal
```

大多数情况下，使用 `class` 关键字而不是直接调用 `type` 创建类更加可行。然而，并非在所有情况下都可以这么做。

## 6.3 决定何时应该编写类工厂

编写类工厂函数的主要原因是在需要基于运行时的信息(如用户输入)创建类时。而 `class` 关键字假定你已经在编码时知道需要赋值给类的属性(虽然这并不是必要的)。

如果在编码时并不知道需要赋值给类的属性，类工厂函数将会是一个方便的替代办法。

### 6.3.1 运行时属性

考虑下面创建类的函数，但这次，该类的属性可以基于传递给函数的参数而变化：

```
def get_credential_class(use_proxy=False, tfa=False):  
    """Return a class representing a credential for the given service,  
    with an attribute representing the expected keys.  
    """  
  
    # If a proxy, such as Facebook Connect, is being used, we just  
    # need the service name and the e-mail address.  
    if use_proxy:  
        keys = ['service_name', 'email_address']  
    else:  
        # For the purposes of this example, all other services use  
        # username and password.  
        keys = ['username', 'password']  
  
        # If two-factor auth is in play, then we need an authenticator  
        # token also.  
        if tfa:  
            keys.append('tfa_token')  
  
    # Return a class with a proper __init__ method which expects
```

```
# all expected keys.
class Credential(object):
    expected_keys = set(keys)

    def __init__(self, **kwargs):
        # Sanity check: Do our keys match?

        if self.expected_keys != set(kwargs.keys()):
            raise ValueError('Keys do not match.')

        # Write the keys to the credential object.
        for k, v in kwargs.items():
            setattr(self, k, v)

    return Credential
```

`get_credential_class` 函数请求获得所发生的登录类型的信息——是传统登录方式(使用用户名与密码)还是使用 OpenID 服务登录。如果是传统登录方式, 或许还需要双因素认证, 也就是额外需要验证令牌。

该函数返回一个类(而不是一个实例), 用于表示合适的凭据类型。例如, 如果将 `use_proxy` 变量设置为 `True`, 则返回的类会包含设置为 `['service_name', 'email_address']` 的 `expected_keys` 属性, 代表通过代理身份验证所要的密钥。而向该函数传入的参数 `use_proxy` 为 `false` 时, 将会返回带有不同 `expected_keys` 属性的类。

然后, 类的 `__init__` 方法检查从 `expected_keys` 属性中获得的关键字参数。如果不匹配, 则构造函数引发异常。如果匹配, 将值写入实例。

可以使用 `class` 关键字而不是调用 `type` 在函数中创建该类。这是由于 `class` 代码块在 `def` 代码块中, 该类是在函数作用域内创建的。

### 1. 理解应该这样做的原因

你或许会问在本例中类工厂是否有价值。毕竟, 仅有三种可能性。硬编码这些类而不是即席动态创建类。从本例中可以很容易推断出硬编码类也并不合理。

毕竟, 大量网站并不会使用多种验证方式。例如, 一些网站使用用户名, 另一些网站使用 e-mail 地址。对于开发服务来说, 更倾向于使用同一个 API 键与一个或多个密码令牌。

并没有一个网站需要(至少是依赖)通过编程方式来决定使用哪个凭据, 但假如是一个为大量不同的第三方网站提供凭据的服务就不同了。这类网站更倾向将所需的键与值类型存储在数据库中。

现在, 突然间你就拥有了一个能够根据数据库查询结果生成属性的类。这很重要, 因为数据库查询在运行时而不是在编码时发生。现在突然就有了一个类, 这个类可以拥有无限种 `expected_keys` 属性, 而完全靠手动编码实现这点就变得不再现实。

将这类数据存入数据库同时也意味着, 随着数据改变, 代码却无须变更。一个网站或许需要修改或增加它支持的凭据类型, 现在只需要在数据库中添加或删除行即可, 而

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

Credential 类却无须修改即可继续使用。

## 2. 属性字典

仅仅是某些属性只有在执行时可知并不是使用类工厂的必要条件。常常，可以即席将属性写入类，或是类仅仅存储一个包含任意属性集合的字典。

如果该方案可行，那么该方案就更加简单、更加直接。

```
class MyClass(object):
    attrs = {}
```

属性字典的最常见的缺点在于：当写一个子类继承现有类，而现有类无法直接控制时，需要现有的功能可以对修改过的属性进行操作。稍后将看到子类的示例。

## 3. 扩充 Credential 类

考虑一个只有一个表的 credential 数据库，该表只有两列：一列是服务名称(比如 Apple 或 Amazon)，一列是凭据键(比如 username)。

很明显，该模拟数据库还是太简单，无法覆盖所有用例。在本例中，并不支持登录的一些备用模式(比如 OpenID)。同时，本例并不会以特定顺序(比如用户名在密码之前)展示凭据。这些都没有问题，对于证明理论已经足够。

现在，考虑一个从该数据库(以 CSV 平面文件方式存储)读取数据并返回适当类的类工厂。

```
import csv


def get_credential_class(service):
    """Return a class representing a credential for the given service,
    with an attribute representing the expected keys.
    """
    # Open our "database".
    keys = []
    with open('creds.csv', 'r') as csvfile:
        for row in csv.reader(csvfile):
            # If this row does not correspond to the service we
            # are actually asking for (e.g., if it is a row for
            # Apple and we are asking for an Amazon credential class),
            # skip it.
            if row[0].lower() != service.lower():
                continue

            # Add the key to the list of expected keys.
            keys.append(row[1])

    # Return a class with a proper __init__ method which expects
    # all expected keys.
    class Credential(object):
```

```
expected_keys = keys

def __init__(self, **kwargs):
    # Sanity check: Do our keys match?
    if set(self.expected_keys) != set([i for i in kwargs.keys()]):
        raise ValueError('Keys do not match.')

    # Write the keys to the credential object.
    for k, v in kwargs.items():
        setattr(self, k, v)

return Credential
```

现在，`get_credential_class` 函数的输入部分被完全替换。参数不再是凭据类型，而是用来指定谁使用该凭据。

例如，一个示例 CSV “数据库” 看上去如下所示：

```
Amazon,username
Amazon,password
Apple,email_address
Apple,password
GitHub,username
GitHub,password
GitHub,auth_token
```

`get_credential_class` 接受的值是一个字符串，与 CSV 文件的第一列对应。因此，调用 `get_credential_class('GitHub')` 将会返回一个包含 `username`、`password` 与 `auth_token` 属性的类。CSV 文件中与 Apple 和 Amazon 对应的行将会被跳过。

#### 4. 表单示例

可以从流行的 Web 框架 Django 的表单 API 中看到该概念的应用。该框架包含一个抽象类，`django.forms.Form`，用于创建 HTML 表单。

Django 表单有一个接受在表单中声明属性与表单域和表单数据分隔符的自定义元类。如果知道是哪一个表单域，那么在 API 中创建一个凭据表单就会非常容易。

```
from django import forms

class CredentialForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

另一方面，如果不知道是哪一个域(正如之前的示例中所示)，这将会是一个非常复杂的任务。因此类工厂就成为完美的解决方案。

```
import csv
```

# 尚学堂.百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
from django import forms

def get_credential_form_class(service):
    """Return a class representing a credential for the given service,
    with attributes representing the expected keys.
    """
    # Open our "database".
    keys = []
    with open('creds.csv', 'r') as csvfile:
        for row in csv.reader(csvfile):
            # If this row does not correspond to the service we
            # are actually asking for (e.g. if it is a row for
            # Apple and we are asking for an Amazon credential class),
            # skip it.
            if row[0].lower() != service.lower():
                continue

            # Add the key to the list of expected keys.
            keys.append(row[1])

    # Put together the appropriate credential fields.
    attrs = {}
    for key in keys:
        field_kw = {}
        if 'password' in key:
            field_kw['widget'] = forms.PasswordInput
        attrs[key] = forms.CharField(**field_kw)

    # Return a form class with the appropriate credential fields.
    metaclass = type(forms.Form)
    return metaclass('CredentialForm', (forms.Form,), attrs)
```

在本例中，将自定义 Credential 类替换为 Django form 的子类。不再需要设置 expected\_keys 属性。而是为每一个预期键设置一个属性。之前的代码将这些放入一个字典，并创建一个新的 form 子类并返回。

值得一提的是，Django 的 Form 类使用的是一个继承 type 的自定义元类。因此，调用其构造函数而不是直接使用 type 非常重要。在最后两行中获得 forms.Form 的元类，并直接使用其构造函数。

本例中值得注意的另一件事是，是否真的有必要使用元类构造函数而不是使用 class 关键字创建类。这里无法通过 class 关键字创建类是由于，即使在一个函数内，你也无法创建类并且将属性写入类，并且在类创建后，元类的行为将不再会被应用到该类的属性(在第 5 章中已详细解释了该行为)。

## 6.3.2 避免类属性一致性问题

另一个编写类工厂函数的原因是处理类与实例之间属性不同的问题。

## 1. 类属性与实例属性

下面两段代码没有产生等价的类或实例：

```
#####
### CLASS ATTRIBUTE #####
#####

class C(object):
    foo = 'bar'

#####

#####
### INSTANCE ATTRIBUTE #####
#####

class I(object):
    def __init__(self):
        self.foo = 'bar'
```

对于这两个类，首先也是最显著的区别是能够从什么位置访问 foo 属性。C.foo 属性返回字符串，而 I.foo 属性引发 AttributeError 异常，这并不令人感到惊讶。

```
>>> C.foo
'bar'
>>> I.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'I' has no attribute 'foo'
```

毕竟，foo 作为 C 的一个属性被实例化，但并不作为 I 的属性被实例化。由于直接访问 I 而不是 I 的实例，因此\_\_init\_\_函数还没有被运行。即使 I 的实例被创建后，也只有实例包含 foo 属性，而类并不包含该属性。

```
>>> i = I()
>>> i.foo
'bar'
>>> I.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'I' has no attribute 'foo'
```

然而，这里的 C 和 I 之间的区别较小，该区别包括如果修改后的 foo 属性与它的实例不同时会发生什么。

考虑下面两个已实例化的 C 实例：

```
>>> c1 = C()
>>> c2 = C()
```

现在修改其中一个实例的 foo 属性，如下所示：

# 尚学堂·百战程序员

第 II 部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
>>> c1.foo = 'baz'
```

可以看到，现在 c2 实例依然使用类的属性，而 c1 有自己的属性。

```
>>> c1.foo  
'baz'  
>>> c2.foo  
'bar'
```

在此进行的查找并不十分相同。c1 写了一个实例属性，名称为 foo，值为'baz'。而 c2 并没有这种实例属性。但是，由于类 C 有该属性，因此查找使用类属性。

考虑如果修改类属性时会发生的情况，如下所示：

```
>>> C.foo = 'bacon'  
>>> c1.foo  
'baz'  
>>> c2.foo  
'bacon'
```

在这里，c1.foo 不受影响，这是由于 c1 有名称为 foo 的实例属性。而 c2.foo 的值却发生了改变，因为实例中并没有这种属性。因此，当一个类的属性改变时，请观察实例的改变。

在 Python 的内部数据模型中可以通过这两个实例的 `_dict_` 属性来观察这一点。

```
>>> c1.__dict__  
{'foo': 'baz'}  
>>> c2.__dict__  
{}
```

在正常情况下，特殊的 `_dict_` 属性存储对象的所有属性(和值)。但也有例外。类 A 可能会自定义一个 `_getattr_` 或 `_getattribute_` 方法(正如在第 4 章“魔术方法”中所讨论的)，或定义一个特殊属性 `_slots_`，该属性也会引入替代属性行为(该属性很少使用，只有在特定场景下内存使用量很重要时才被使用，本书不包含相关内容)。注意 c1 在 `_dict_` 中存在 foo 键，而在 c2 中不存在。

## 2. 类方法的限制

当类中定义了类方法时情况就变得真正有趣。记住类方法是那些并不需要类的实例就可以执行的方法，但需要类本身。它们通常使用`@classmethod` 装饰器装饰一个方法来完成声明，并且方法的第一个参数按照传统会被命名为 `cls` 而不是 `self`。

考虑下面的类 C，该类中包含能够访问并返回 foo 的类方法：

```
class C(object):  
    foo = 'bar'  
  
    @classmethod  
    def classfoo(cls):  
        return cls.foo
```

在 classfoo 方法的上下文中，foo 属性在类中而不是实例中被显式访问。使用新的类定义重新运行该示例，并考虑如下代码：

```
>>> c1.foo  
'baz'  
>>> c1.classfoo()  
'bacon'  
>>> c2.classfoo()  
'bacon'
```

实际上，无法从类方法中访问实例属性。这也是类方法的要点，毕竟，它们并不需要一个实例。

### 3. 使用类工厂尝试

需要类工厂的一个最大原因是当你继承一个现有类并且所依赖的类属性必须调整时。

本质上，在你无法控制的代码中，如果一个已存在的类设置某个必须自定义的类属性，类工厂是生成带有重载属性的恰当子类的一种恰当方式。

考虑这样一种情况，当一个类中包含了必须在运行时(或是在静态代码中子类的选择过多时)被重载的属性。在这种情况下，类工厂将会是一个非常有效的方案。下面继续使用有启发性的类 C 的示例：

```
def create_C_subclass(new_foo):  
    class SubC(C):  
        foo = new_foo  
    return SubC
```

这里的重点是并不需要在类创建之前，也就是函数运行时，知道 foo 的值。与其他大多数类工厂的使用并无二致，是关于在运行时获得属性值。

以执行 C 子类的 classfoo 类方法创建类的方式将会返回你期望的结果。

```
>>> S = create_C_subclass('spam')  
>>> S.classfoo()  
'spam'  
>>> E = create_C_subclass('eggs')  
>>> E.classfoo()  
'eggs'
```

值得注意的是，在很多情况下，创建一个仅仅在\_init\_方法中接受该值的子类就简单多了。然而，也有一些情况使用这种方法并不可行。例如，父类依赖于类方法，此时将一个新值赋给实例并不会导致类方法接收到新值，这时该子类创建的模型将会是一个有价值的解决方案。

#### 6.3.3 关于单例模式问题的解答

顾名思义，让类工厂函数难以使用的一点是类工厂返回的是类，而不是类的实例。

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

这意味着如果你需要一个实例，则必须调用类工厂函数返回的结果才可以。例如，实例化一个使用 `create_C_subclass` 生成的子类，正确代码应该是 `create_C_subclass('eggs')()`。

这种做法并没有错，但可能并不是你所需要的结果。有些时候，通过类工厂创建的类功能上类似单例模式。单例模式是一种只允许一个实例的类模式。

在函数中生成类的情况下，有可能函数的目的就是作为一个类构造函数。最终开发人员必须不断考虑如何再次实例化所生成的类。

如果不需要面对在其他的地方重用类或类工厂可以处理类重用的情况，就不需要处理这种情况，让类工厂返回其创建类的实例而不是类本身完全是合理且有用的。

继续使用 C 的示例，考虑下面的工厂：

```
def CPrime(new_foo='bar'):
    # If `foo` is set to 'bar', then we do not need a
    # custom subclass at all.
    if new_foo == 'bar':
        return C()

    # Create a custom subclass and return an instance.
    class SubC(C):
        foo = new_foo
    return SubC()
```

现在，调用 `CPrime` 将会返回合适的 C 子类的实例，该类带有按需修改后的 `foo` 属性。

这种方式存在的一个问题，很多(很可能是绝大多数)类需要将参数发送给 `_init_` 方法，此时该函数就无法处理这种情况。考虑一个凭据表单的示例，该示例包含一个返回实例的方法。

```
import csv

from django import forms
def get_credential_form(service, *args, **kwargs):
    """Return a form instance representing a credential for the
    given service.

    """
    # Open our "database".
    keys = []
    with open('creds.csv', 'r') as csvfile:
        for row in csvfile.reader(csvfile):
            # If this row does not correspond to the service we
            # are actually asking for (e.g. if it is a row for
            # Apple and we are asking for an Amazon credential class),
            # skip it.
            if row[0].lower() != service.lower():
                continue

            # Add the key to the list of expected keys.
            keys.append(row[1])

    # Put together the appropriate credential fields.
```

```
 attrs = {}
for key in keys:
    field_kw = {}
    if 'password' in key:
        field_kw['widget'] = forms.PasswordInput
    attrs[key] = forms.CharField(**field_kw)

# Return a form class with the appropriate credential fields.
metaclass = type(forms.Form)
cls = metaclass('CredentialForm', (forms.Form,), attrs)
return cls(*args, **kwargs)
```

这段代码与之前的类工厂相比并没有太多变化。只有如下两处变更：

- 首先，将`*args`与`**kwargs`添加到函数签名中。
- 其次，最后一行代码返回所创建的类的实例，并将`*args`与`**kwargs`传递给实例。

现在你有了一个功能完整的类工厂，该类工厂返回它所创建的`form`类的实例。这就提出了最后一点，对于最终开发人员来说很容易将该函数与一个类混淆，除非最终开发人员查看内部工作机制。因此，或许应该将该函数的信息写到开发规范的命名约定中。

```
def CredentialForm(service, *args, **kwargs):
    [...]
```

在 Python 中，函数通常以小写字母命名，并用下划线作为单词分隔符。然而，该函数被实际使用它的开发人员用作类构造函数，因此，通过修改命名约定，表示`CredentialForm`是一个类名称。

另外，这种命名方式也与用于实例的类的名称相匹配，由于第一个参数与元类的构造函数相匹配，因此`CredentialForm`是类的内部名称。

## 6.4 小结

当需要在运行时而不是编码时确认类属性时，类工厂的强大之处就显而易见。Python 语言能够精确处理这种情况，这是由于类是第一类对象，与其他对象创建的方式类似。

另一方面，类包含一些未知属性增加了不确定性。因此方法的编写必须允许属性缺失，而在正常情况下，都会假定属性存在。

在运行时声明类的功能非常强大，但是以牺牲简单性为代价。这并没有什么问题，当你遇到类工厂是最好选择的情况时，这种代码机制也会很明显，并且没有其他解决问题的直接方式。直接而言，如果类工厂是解决某个问题的好方法，那么一定也是最简单的方法。

该规则适用于通用编程规则，但在这里尤其有用。

在第 7 章“抽象基类”中，将讨论 Python 字符串与字节字符串，以及如何尽可能轻松地使用字符串数据。

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



第 7 章

## 抽象基类

如何知道正在使用的对象是否符合一个给定规范？在 Python 中回答该问题的常见答案被称作 *duck typing* 模式。如果它看起来像一只鸭子并且叫起来像一只鸭子，那么它大概就是一只鸭子。

在处理编程和对象时，问题通常可以转化为一个对象是否实现了给定方法，或包含一个特定的属性。如果这个对象有一个 `quack` 方法，你就有恰当的证据证明它是一只鸭子。此外，如果你只需要一个 `quack` 方法，实际上它是否是一只鸭子就没那么重要了。

这通常是一个非常有用的构造，并且它能轻而易举地在 Python 这种松散类型系统中实现。它强调构成问题而不是身份问题，强调 `hasattr` 函数而不是 `isinstance` 函数。

但是有时，身份很重要。例如，假定你正在使用要求输入遵循特殊身份的类库。或者，有时检查不同种类的属性和方法显得过于繁琐时。

Python 2.6 和 Python 3 中引入了抽象基类的概念。抽象基类是一个分配身份的机制。它们是回答“从本质上讲，这是一只鸭子吗？”这个问题的一种方式。抽象基类也提供了一个标明抽象方法的机制，就是要求其他实现提供关键性功能，这些功能是在基类实现中不主动提供的功能。

本章探究抽象基类，它们为什么存在？以及如何使用它们？

### 7.1 使用抽象基类

抽象基类的基本目的就是提供有点形式化的方法，来测试一个对象是否符合特定规范。

如何确定你正在处理的对象是列表？这十分简单——只需要调用 `isinstance` 将变量与列表类进行比较，然后查看函数是返回 `True` 还是 `False`。

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
>>> isinstance([], list)
True
>>> isinstance(object(), list)
False
```

另一方面，你编写的代码真的需要一个列表吗？考虑这种情况，你只是去读取像列表一样的对象，但是绝不会修改该对象。在这种情况下，可以接受一个元组来代替列表。

这个 `isinstance` 方法的确提供了一个针对多个基类测试的机制，如下所示：

```
>>> isinstance([], (list, tuple))
True
>>> isinstance((), (list, tuple))
True
>>> isinstance(object(), (list, tuple))
False
```

但是，这也不是你真正想要的，毕竟，一个自定义序列类也完全可以被接受，假如它使用 `__getitem__` 方法接受升序的整数和切片对象(例如，Django 中的 `QuerySet` 方法)。因此，只是针对能够显式地被识别出来的类使用 `isinstance` 可能会返回错误的 `false`，从而使允许使用的对象不被允许使用。

当然，也可以测试 `__getitem__` 方法是否存在。

```
>>> hasattr([], '__getitem__')
True
>>> hasattr(object(), '__getitem__')
False
```

此外，这不是一个完整的解决方案。与 `isinstance` 检查不同，它不产生 `false` 结果。相反，它会产生 `true` 结果，因为不仅仅只有类似列表的对象实现了 `__getitem__` 方法。

```
>>> hasattr({}, '__getitem__')
True
```

从本质上讲，仅仅对某个属性或者方法是否存在进行测试有时不足以确定该对象是否符合你所寻找的参数。

抽象基类提供了声明一个类是另一类的派生类的机制(无论它是否是另一个类的派生类)。该机制并没有影响实际的对象继承关系或是改变方法解析顺序。其目的是声明性的，它提供了一种断言对象符合协议的方式。

此外，抽象基类提供了一种要求子类实现指定协议的方式。如果一个抽象基类要求实现指定方法，并且子类没有实现这个方法，然后当试图创建子类时解释器会抛出一个异常。

## 7.2 声明虚拟子类

Python 2.6、2.7 和 Python 3 的所有版本都提供了一个名为 `abc` (表示抽象基类)的模块，

该模块提供了一些使用抽象基类的工具。

abc 模块提供的第一个内容是名为 ABCMeta 的元类。任何抽象基类，无论它们的目的是什么，必须使用 ABCMeta 元类。

所有抽象基类可以任意地声明它是任意具体类的父类(不是派生类)，包括在标准库的具体类(甚至那些使用 C 语言实现的类)。ABCMeta 的实例通过使用 register 方法提供了对声明的实现(记住，这些使用 ABCMeta 作为它们元类的类都是类本身)。

考虑一个注册自身作为 dict 的父类的抽象基类(注意下面的代码使用的是 Python 3 的元类语法)。

```
>>> import abc  
>>> class AbstractDict(metaclass=abc.ABCMeta):  
...     def foo(self):  
...         return None  
...  
>>> AbstractDict.register(dict)  
<class 'dict'>
```

这并没有对 dict 类本身进行任何修改。在此没有发生显著的变化，至关重要的是，dict 的方法解析没有发生改变。你并不会突然发现 dict 拥有了 foo 方法。

```
>>> {}.foo()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'dict' object has no attribute 'foo'
```

这样做就使得 dict 对象也被标识为 AbstractDict 的实例，并且现在 dict 自身也被标识为一个 AbstractDict 的子类。

```
>>> isinstance({}, AbstractDict)  
True  
>>> issubclass(dict, AbstractDict)  
True
```

请注意，反过来执行却不是这样的结果。AbstractDict 不是 dict 的子类。

```
>>> issubclass(AbstractDict, dict)  
False
```

## 7.2.1 声明虚拟子类的原因

为了理解声明虚拟子类的原因，请回忆一下本章开始时打算读取类似列表对象的实例。实例需要像 list 或 tuple 一样可被遍历，并且还需要一个 \_\_getitem\_\_ 方法来接收整型参数。另一方面没有必要限制只接受 list 或 tuple。

为此，抽象基类提供了一种非常好的可扩展机制。之前的示例表明，可以使用 isinstance 来检查一个类的元组。

# 尚学堂·百战程序员

第II部分 类

www.itbaizhan.cn

```
>>> isinstance([], (list, tuple))  
True
```

但是，这并不是真的可扩展。如果在你的实现中检查 list 或 tuple，并且使用你的类库的人打算发送一些其他的类似列表的对象，而对象并不是 list 或 tuple 的子类，此时就遇到了难以实现扩展的问题。

抽象基类提供了解决这个问题的方案。首先，定义一个抽象基类并且对它注册 list 和 tuple，如下所示：

```
>>> import abc  
>>> class MySequence(metaclass=abc.ABCMeta):  
...     pass  
...  
>>> MySequence.register(list)  
<class 'list'>  
>>> MySequence.register(tuple)  
<class 'tuple'>
```

现在修改 `isinstance`，检测 `MySequence` 而不再检测(`list, tuple`)。当针对 `list` 或 `tuple` 进行检测时仍将返回 `True`，而针对其他对象的检测将会返回 `False`。

```
>>> isinstance([], MySequence)  
True  
>>> isinstance((), MySequence)  
True  
>>> isinstance(object(), MySequence)  
False
```

到目前为止，与之前的情况一样。但有一个关键的区别。考虑这样一种情况，一个开发人员正使用类库并期望一个 `MySequence` 对象，并且因此，也期望一个 `list` 或 `tuple`。

当(`list, tuple`)在类库中是硬编码时，开发人员什么也做不了。但是，`MySequence` 是一个在类库中定义的抽象基类。这意味着开发人员能导入它。

一旦开发人员导入该类库，这个完全类似列表的自定义类就能简单地被注册在 `MySequence` 中：

```
>>> class CustomListLikeClass(object):  
...     pass  
...  
>>> MySequence.register(CustomListLikeClass)  
<class '__main__.CustomListLikeClass'>  
>>> issubclass(CustomListLikeClass, MySequence)  
True
```

开发人员可以将 `CustomListLikeClass` 的实例传递到期望 `MySequence` 的类库。现在，当类库执行 `isinstance` 检测时，检查会通过，因此准许接受该对象。

## 7.2.2 使用 register 作为装饰器

自 Python 3.3 以来，使用 ABCMeta 元类的类所提供的 register 方法也可以用作装饰器。

如果你创建一个应该被注册为 ABCMeta 子类的新类，那么一般情况下可以像下面一样注册它(使用在前面示例中定义的 MySequence 抽象基类)：

```
>>> class CustomListLikeClass(object):
...     pass
...
>>> MySequence.register(CustomListLikeClass)
<class '__main__.CustomListLikeClass'>
```

但是，注意，register 方法会返回传递给它的类。这样的工作机制使得 register 也能被用作装饰器。register 接受一个可调用函数，同时返回一个可调用函数(这个示例中，是完全相同的可调用函数)。

下面的代码作用相同：

```
>>> @MySequence.register
... class CustomListLikeClass(object):
...     pass
...
>>>
```

你能确认这和以前你做的 issubclass 检查一样。

```
>>> issubclass(CustomListLikeClass, MySequence)
True
```

值得注意的是，在 Python 3.3 中加入了装饰器行为。在 Python 2 中和 Python 3.2 以及以前的版本中，抽象基类中的 register 方法返回的结果是 None，而不是被传递的类。

这意味着在之前的版本中，register 不能被用作装饰器。如果你正在编写跨版本兼容 Python 2 和 Python 3 的代码，或者正在编写可以在 Python 3 之前的版本上运行的代码，都应该避免将 register 用作装饰器。

## 7.2.3 \_\_subclasshook\_\_

对于大多数目的，使用一个使用 ABCMeta 元类的类，然后使用由 ABCMeta 提供的 register 方法就完全足以获得你所需要的结果。但是，为所有希望的子类手动注册这种情况并不合理。

用 ABCMeta 元类创建的类可以选择性地定义一个特殊的魔术方法，称为 `__subclasshook__`。

`__subclasshook__` 方法必须被定义为一个类方法(使用 `@classmethod` 装饰器定义)并且接收一个额外的位置参数，该参数是被测试的类。它可以返回三个值：True、False 或 NotImplemented。

返回值是 True 和 False 的情况是很清晰的。如果被测试类被认为是子类，那么

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

subclasshook 方法的返回结果是 True；而如果被认为不是子类，那么返回值是 False。

考虑传统的“鸭子类型”范例。在“鸭子类型”范例中最根本的问题是，一个对象是否有某个方法或者属性(比如是否它叫起来像只鸭子)，而不是对象是否是这个或那个类的子类。抽象基类可以用subclasshook 实现这个概念，如下所示：

```
import abc

class AbstractDuck(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, other):
        quack = getattr(other, 'quack', None)
        return callable(quack)
```

该抽象基类声明，任何带有 quack 方法(但是该方法不是一个不可调用的 quack 属性)的类都被认为是它的子类，而任何其他类都不是它的子类。

```
>>> class Duck(object):
...     def quack(self):
...         pass
...
>>>
>>> class NotDuck(object):
...     quack = 'foo'
...
>>> issubclass(Duck, AbstractDuck)
True
>>> issubclass(NotDuck, AbstractDuck)
False
```

这里需要注意的一件重要事情是，当subclasshook 方法被定义时，它优先于 register 方法。

```
>>> AbstractDuck.register(NotDuck)
<class '__main__.NotDuck'>
>>> issubclass(NotDuck, AbstractDuck)
False
```

在此需要用到 Not Implemented。如果subclasshook 方法返回 Not Implemented，然后(并且只有然后)传统检测的路径就会查看已注册的类是否已被选中。

考虑下面这个修改过的 AbstractDuck 类：

```
import abc

class AbstractDuck(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, other):
        quack = getattr(other, 'quack', None)
```

```
if callable(quack):  
    return True  
return NotImplemented
```

这里仅有的变化就是 `quack` 方法好像已经不存在，`__subclasshook__` 方法返回值是 `NotImplemented` 而不是 `False`。现在，注册表已经被检查，并且之前注册的类将作为子类返回。

```
>>> issubclass(NotDuck, AbstractDuck)  
False  
>>> AbstractDuck.register(NotDuck)  
<class '__main__.NotDuck'>  
>>> issubclass(NotDuck, AbstractDuck)  
True
```

本质上讲，第一个例子说的是，“如果它叫起来像只鸭子，那么它就是 `AbstractDuck`，”然后第二个例子是说，“如果它叫起来像只鸭子或者直接说它是 `AbstractDuck`，那么它就是 `AbstractDuck`。”

当然，注意如果这样做，就必须能够处理接收的任何东西。如果依赖于调用 `quack` 方法，那么使得该方法可选择对你没有任何好处。

那么，这样做的价值何在？这样做可以仅简单地针对你所需的方法做 `hasattr` 或 `callable` 检查。

在一种相对简单的情况下，使用抽象基类或许实际上是一个障碍。例如，使用一个替身来检查单一方法是否存在只会增加不必要的复杂度。

对于复杂的情况，使用抽象基类就有一些价值了。首先，在区分上有价值。抽象基类为整个测试定义了一个存在的统一位置。使用抽象基类子类的任何代码仅仅是使用 `issubclass` 或 `isinstance` 方法。随着需求的变化，仅在一个地方存放一致性检查的代码。

另外，`NotImplemented` 作为 `__subclasshook__` 的可用返回值增加了一些功能。它提供了一种确保绝对匹配或不匹配给定协议的机制，这也是对于自定义类作者显式可选的方式。

## 7.3 声明协议

抽象基类的另一个主要价值在于它有声明协议的功能。在之前的示例中，已经介绍了抽象基类是如何使一个类能够声明它自身可以通过类型检查测试的。

但是，抽象基类也可以定义子类必须提供的内容。这类似于在其他诸如 Java 等面向对象的语言中接口的概念。

### 7.3.1 其他现有的方法

即使不使用抽象基类也能解决这种基本问题。因为抽象基类是一个相对较新的语言特性，下面这几种方法十分常见。

# 尚学堂·百战程序员

第II部分 类

[www.itbaizhan.cn](http://www.itbaizhan.cn)

## 1. 使用 NotImplementedError

考虑一个构建为带有特定功能的类，但是这个类省去了一个关键的方法，以便于这个方法可以被子类实现。

```
from datetime import datetime

class Task(object):
    """An abstract class representing a task that must run, and
    which should track individual runs and results.
    """
    def __init__(self):
        self.runs = []

    def run(self):
        start = datetime.now()
        result = self._run()
        end = datetime.now()
        self.runs.append({
            'start': start,
            'end': end,
            'result': result,
        })
        return result

    def _run(self):
        raise NotImplementedError('Task subclasses must define '
                               'a _run method.'')
```

这个类的目的是运行某种类型的任务并追踪何时执行这些任务。很容易直观地理解类如何也能提供日志记录或者类似功能。

但是，基类 Task 不能提供任务主体。这需要由子类完成。相反，Task 类提供一个 shell 方法 \_run，该方法仅抛出带有有用错误信息的 NotImplementedError。任何未能重写 \_run 的子类大都会引发这个错误，如果打算调用 Task 类本身的 run 方法，也会得到这个错误。

```
>>> t = Task()
>>> t.run()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in run
File "<stdin>", line 20, in _run
NotImplementedError: Task subclasses must define a _run method.
```

## 2. 使用元类

NotImplementedError 不是声明协议唯一的方式。另一种声明协议的常用方式是使用元类。

```
from datetime import datetime, timezone
```

```
class TaskMeta(type):  
    """A metaclass that ensures the presence of a _run method  
    on any non-abstract classes it creates.  
    """  
  
    def __new__(cls, name, bases, attrs):  
        # If this is an abstract class, do not check for a _run method.  
        if attrs.pop('abstract', False):  
            return super(TaskMeta, cls).__new__(cls, name, bases, attrs)  
  
        # Create the resulting class.  
        new_class = super(TaskMeta, cls).__new__(cls, name, bases, attrs)  
  
        # Verify that a _run method is present and raise  
        # TypeError otherwise.  
        if not hasattr(new_class, '_run') or not callable(new_class._run):  
            raise TypeError('Task subclasses must define a _run method.')  
  
        # Return the new class object.  
        return new_class  
  
  
class Task(metaclass=TaskMeta):  
    """An abstract class representing a task that must run, and  
    which should track individual runs and results.  
    """  
  
    abstract = True  
  
    def __init__(self):  
        self.runs = []  
  
    def run(self):  
        start = datetime.now(tz=tzzone.utc)  
        result = self._run()  
        end = datetime.now(tz=tzzone.utc)  
        self.runs.append({  
            'start': start,  
            'end': end,  
            'result': result,  
        })  
        return result
```

这个示例类似于前面的示例，指示有一些细微的区别。第一个区别就是 Task 类本身，虽然它仍然被实例化，但是不再声明 `_run` 方法，因此面向公用的 `run` 方法会抛出 `AttributeError` 错误。

```
>>> t = Task()  
>>> t.run()  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 12, in run
AttributeError: 'Task' object has no attribute '_run'
```

但是，更为重要的区别在于子类。因为当子类被创建时元类会运行`__new__`方法，解析器将不再允许创建没有`_run`方法的子类。

```
>>> class TaskSubclass(Task):
...     pass
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 16, in __new__
NotImplementedError: Task subclasses must define a _run method.
```

### 7.3.2 抽象基类的价值

以上两种方法都是有价值的，但是说它们的缺点是有点“即席(ad hoc)”也并无不妥。

抽象基类提供了一种呈现相同模式的更正式的方法。它们提供了一个使用某个抽象基类声明协议的机制，并且子类一定要提供一个符合该协议的实现。

`abc` 模块提供了一个名为`@abstractmethod` 的装饰器，它指定了一个必须被所有子类重写的特定方法。该方法体可以是空的(`pass`)，或者可能包含一个子类方法可能选择使用`super` 调用的实现。

考虑一个使用`@abstractmethod` 装饰器替代自定义元类的 `Task` 类。

```
import abc
from datetime import datetime, timezone

class Task(metaclass=abc.ABCMeta):
    """An abstract class representing a task that must run, and
    which should track individual runs and results.
    """
    def __init__(self):
        self.runs = []

    def run(self):
        start = datetime.now(tz=timezone.utc)
        result = self._run()
        end = datetime.now(tz=timezone.utc)
        self.runs.append({
            'start': start,
            'end': end,
            'result': result,
        })
        return result

    @abc.abstractmethod
    def _run(self):
```

pass

同样，前两个例子几乎完全相同，但略有不同。首先，注意 Task 类本身不能被实例化。

```
>>> t = Task()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Task with abstract methods _run
```

这与 NotImplementedError 方法不同，NotImplementedError 方法允许基类 Task 被实例化。

对于不能正确重写 \_run 方法的子类，在错误的情况下它与之前的两个方法的差别也是细微的。在第一个示例中，使用 NotImplementedError，最终在 \_run 方法被调用的地方引发 NotImplementedError 错误。在第二个示例中，使用自定义的 TaskMeta 元类，当这个抽象类被创建时引发 TypeError 错误。

当使用抽象基类时，解析器乐于去创建一个不实现基类中所有(或者任何一个)抽象方法的子类。

```
>>> class Subtask(Task):
...     pass
...
>>>
```

但是，解析器不愿做的就是实例化这个类。实际上，这样做给出的错误信息与 Task 类给出的完全一样，逻辑上是与你期望的完全一致。

```
>>> st = Subtask()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Subtask with abstract methods _run
```

但是，一旦你定义一个重写抽象方法的子类，那么它就能正常工作，并且也能够实例化子类。

```
>>> class OtherSubtask(Task):
...     def _run(self):
...         return 2 + 2
...
>>>
>>> ost = OtherSubtask()
>>> ost.run()
4
```

而且，如果检查 runs 属性，会发现关于 run 的信息已经被保存了，如下所示：

```
>>> ost.runs
[{'result': 4, 'end': datetime.datetime(...), 'start': datetime.datetime(...)}]
```

实际上，基于以下几个原因，这是解决这个问题的一个非常有用的方法。首先(并且可

能是最重要的), 这个方法是正式而不是临时的。抽象基类被明确地提出作为满足这种特定需求的解决方案, 依据这一概念, 理想情况下应该有且仅有一种实现这种需求的“正确”方法。

其次, @abstractmethod 装饰器非常简单, 并且可以在打算编写一个模板代码时避免出现大量潜在的错误。举个示例, 如果在 TaskMeta 元类中意外地仅检查 attrs 字典中存在 \_run, 而不允许子类中存在 \_run 会如何呢? 很容易犯这样的错误, 并且会导致 Task 子类不能成为自身的子类, 除非每次都手动重写 \_run 方法。使用@abstractmethod 装饰器, 你将在不用过多考虑的情况下获得正确的行为。

最后, 这个方法使得引入中间实现变得很容易。考虑一个有 10 个抽象方法而不是 1 个抽象方法的抽象基类。理所当然会有一个完整的子类树, 在树中链上的高级别的子类实现一些常用方法, 但是将在抽象状态中的方法留给其他子类去实现。公正地讲, 也能够使用自定义元类方法实现该功能(在 TaskMeta 示例中, 通过在每一个中间类中声明 abstract=True)。但是, 当使用@abstractmethod 装饰器时, 基本上能够得到你直观上希望的行为。

当然, 如果需要这类功能, 那么这是一个不使用抽象基类的充足理由, 也就是所需支持的 Python 版本不包含 abc 模块。由于 abc 模块在 Python 2.6 中被引入, 并且许多 Python 包不再支持比 2.6 还旧的版本, 因此这种情况变得非常少见。

### 7.3.3 抽象属性

也可以将属性声明为抽象属性(也就是使用@property 装饰器的方法)。但是, 实现此目的的正确方法有点取决于所支持的 Python 版本。

在 Python 2.6 到 3.2 中(包含跨版本兼容的代码), 正确的方法是使用@abstractproperty 装饰器, 这个装饰器由 abc 模块提供。

```
import abc

class AbstractClass(metaclass=abc.ABCMeta):
    @abc.abstractproperty
    def foo(self):
        pass
```

在 Python 3.3 中, 这个方法已经过时, 因为@abstractmethod 方法已经被更新到能与 @property 装饰器共同协作。因此, 让一个特殊的装饰器提供两种功能现在变得多余。如此, 下面的示例与前面的示例完全相同, 但是仅仅在 Python 3.3 及以上版本中可以实现:

```
import abc

class AbstractClass(metaclass=abc.ABCMeta):
    @property
```

www.itbaizhan.cn

```
@abc.abstractmethod  
def foo(self):  
    pass
```

试图实例化 AbstractClass 的子类且不重写 foo 方法将会导致引发一个错误。

```
>>> class InvalidChild(AbstractClass):  
...     pass  
...  
>>> ic = InvalidChild()  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: Can't instantiate abstract class InvalidChild with abstract methods foo
```

但是，重写抽象方法的子类能够被实例化。

```
>>> class ValidChild(AbstractClass):  
...     @property  
...     def foo(self):  
...         return 'bar'  
...  
>>>  
>>> vc = ValidChild()  
>>> vc.foo  
'bar'
```

### 7.3.4 抽象类或静态方法

与属性一样，你可能希望将@abstractmethod 装饰器与一个类方法或静态方法组合使用(也就是说，用@classmethod 或@staticmethod 装饰器装饰的方法)。

这是一个小技巧。在 Python 2.6 到 3.1 版本中根本没有提供一种方式来实现这一点。Python 3.2 中的确提供了一种实现方式，也就是使用@abstractclassmethod 或者@abstractstaticmethod 装饰器来实现。这些功能与之前的抽象属性的示例很相似。

在 Python 3.3 以后，通过修改@abstractmethod 从而兼容@classmethod 和@staticmethod 装饰器改变了这一点，并且废弃了 Python 3.2 版本中的方法。

在本例中，由于在 Python 3 中编写的大多数代码通常只兼容 Python 3.3 和以上版本，你更倾向的是分别使用两个装饰器。但是，如果需要兼容 Python 3.2，并且不需要兼容之前的 Python 版本(包括所有 Python 2 的版本)，那么这些装饰器就可用。

考虑下面这个使用 Python 3.3 语法的抽象类：

```
class AbstractClass(metaclass=abc.ABCMeta):  
    @classmethod  
    @abc.abstractmethod  
    def foo(cls):  
        return 42
```

继承该类的子类在未重写该方法的情况下照样能够正常工作，但是子类不能被实例化。

```
>>> class InvalidChild(AbstractClass):
...     pass
...
>>> ic = InvalidChild()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class InvalidChild with abstract methods foo
```

尽管如此，实际上抽象方法本身能够被直接调用，且不会引发任何错误。

```
>>> InvalidChild.foo()
42
```

一旦抽象方法在子类中被重写，那么就能够实例化该子类。

```
>>> class ValidChild(AbstractClass):
...     @classmethod
...     def foo(cls):
...         return 'bar'
...
>>> ValidChild.foo()
'bar'
>>> vc = ValidChild()
>>> vc.foo()
'bar'
```

## 7.4 内置抽象基类

除了通过 `abc` 模块建立自己的抽象基类之外，Python 3 标准类库也为语言提供了少量的内置抽象基类，尤其是选择使用一些特殊类用于实现常用模式(例如序列、可变序列、迭代器等)。最常用的抽象基类是用于集合的抽象基类，它们存在于 `collections.abc` 模块中。

绝大多数内置的抽象基类都提供了抽象和非抽象的方法，并且通常是继承 Python 内置类的替代方法。例如，继承于 `MutableSequence` 相对于继承 `list` 或 `str` 可能是更好的选择。

抽象基类能被划分为两种基本类别：一种需要和检查单一方法(比如 `Iterable` 和 `Callable`)，另一种作为普通内置 Python 类型的替身。

### 7.4.1 只包含一个方法的抽象基类

Python 提供了 5 个抽象基类，每个基类包含一个抽象方法，并且抽象基类的 `__subclasscheck__` 方法仅仅检查该方法是否存在。这些抽象基类如下所示：

- `Callable` (`__call__`)
- `Container` (`__contains__`)
- `Hashable` (`__hash__`)
- `Iterable` (`__iter__`)

- `Sized(_len_)`

任何包含相应方法的类都会自动地被当作相关抽象基类的子类。

```
>>> from collections.abc import Sized  
>>>  
>>> class Foo(object):  
...     def __len__(self):  
...         return 42  
...  
>>> issubclass(Foo, Sized)  
True
```

与之相似，类可以直接作为抽象基类的子类，并期望重写相关方法。

```
>>> class Bar(Sized):  
...     pass  
...  
>>> b = Bar()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: Can't instantiate abstract class Bar with abstract methods __len__
```

除了这 5 个抽象基类，还有另外一个抽象基类 `Iterator`。`Iterator` 有点特殊，它继承自 `Iterable`，提供 `_iter_` 的实现(就是返回自身并可以被重写)，并且添加抽象方法 `_next_`。

## 7.4.2 可供集合使用的抽象基类

在 Python 3 中，另一个主要类型的内置抽象基类是那些用于识别与 Python 主要集合类(`list`、`dict` 和 `set`)功能类似的子类。

有 6 个这样的类，它们每两个一组(包含一个不可变类和一个可变类)被划分为三个类别。

第一类是 `Sequence` 和 `MutableSequence`。这两个抽象基类在 Python 中分别起到类似于元组或者列表的作用。`Sequence` 抽象基类只需要 `_getitem_` 和 `_len_` 方法。但是，它也提供许多用于列表和元组的其他常用方法的实现，例如 `_contains_` 和 `_iter_` 方法等。这里的主要目的在于可以继承 `Sequence` 并仅仅定义你需要的内容，并且 Python 提供序列的其他常用功能。当然，`list`、`tuple` 和 `set` 也都被当作 `Sequence` 的子类。

`MutableSequence` 也类似，但是它引入了即席修改序列的概念。因此，它添加 `_setitem_`、`_delitem_` 和 `insert` 方法作为抽象方法，并且为 `append`、`pop` 提供功能。原则仍然相同，你必须仅定义所需要的内容作为可变序列，并且 Python 为其他情况提供了一个类似列表的方法。`list` 和 `set` 已经被认为是 `MutableSequence` 的子类。

其他两个类别是 `Mapping` 和 `Set`，`Mapping` 和 `Set` 与 `MutableMapping` 和 `MutableSet` 一起出现。`Mapping` 用于类似字典(dictionary-like)的对象(与 `dict` 相似，并且 `dict` 被认为是它的子类)，而 `Set` 用于无序的集合(与 `set` 类似，且 `set` 被当作一个子类)。在这两种情况下，它们指定一些关键方法(使用与 `dict` 和 `set` 对应的名称)作为抽象方法，并为除关键方法之外的其他部分提供实现。

## 使用内置抽象基类

这些抽象基类的主要目的是提供一个测试常用集合类型的方法。而不是仅仅测试对象是否是一个 list，而是检测是否为 MutableSequence(或者如果你不需要修改它，可以只检测是否为 Sequence 对象)。不检测是否为 dict 对象，而是检测是否为 MutableMapping 对象。

这使得代码更加灵活。如果使用你类库的人不需要为单独目的创建一个类似列表的对象或一个类似字典的对象，他仍然可以将对象传递给你的代码，不需要做额外的工作就可以使用这些对象。这使得你的代码可以对对象进行测试，确保得到的对象是所期望的对象，并且有了允许传入兼容对象的灵活性，从而使得传入对象可以不是你所期望的具体对象。

### 7.4.3 额外的抽象基类

标准类库中还有其他一些抽象基类在此没有详细介绍。尤其是，`numbers` 模块包含用于实现不同类型数字的抽象基类。

## 7.5 小结

抽象基类最重要的作用是提供了一个正式和动态的方式来回答这个问题，“你正在获取的对象是你认为的类型吗？”它可以克服一些仅仅检测特定属性是否存在或仅仅检测是否为特定类这两种方法的缺点。这很有价值。

但值得记住的是，尽管看上去即席方法优于抽象基类，但是抽象基类依然更像是一个君子协定。Python 解析器将捕获一些明显的违规行为(例如，在子类中未能实现抽象方法)。但是，确保子类做正确的事情是实现者的责任。有很多事情抽象基类并没有进行检测。例如，它并没有检测方法签名或返回类型。

从本章的介绍中可知，仅仅由于一个类实现抽象基类并不能保证它实现的是正确的或是你所期望的方式。这并不是什么新内容。就像一个类包含某个特定方法并不意味着该方法就能做正确的事一样。检查一个对象是否有 `quack` 方法很容易，但是确定 `quack` 方法是否真的使对象叫起来像一只鸭子就困难得多。

这没有问题。使用类似 Python 的动态语言编写软件的一部分是接受这种君子协议的存在。当用一个正式的、合理的方式去声明和确定对象是否符合一个类型或者协议时这非常有价值。而抽象基类就提供了这些价值。

第 8 章将介绍 Unicode 和 ASCII 字符串，以及在 Python 语法中如何高效地处理这两种字符串。

## 第Ⅲ部分

# 数 据

---

- 第8章 字符串与 Unicode
- 第9章 正则表达式

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 第 8 章

## 字符串与 Unicode

在编写 Python 应用程序中最让人头疼的事情之一就是对字符串数据的处理，尤其是当字符串包含常见拉丁字符之外的字符时。

为表示字符串数据，其中最早期的一个标准就是众所周知的 ASCII，它是美国标准信息交换码。ASCII 定义了一个用于代表常见字符的字典，这些字符包括从“A”到“Z”(包括大写和小写)、数据“0”到“9”，以及一些常见符号(比如句号、问号等)。

然而，ASCII 依赖这样一个假设：每一个字符与一个字节匹配，由于存在太多字符，因此导致了问题。最终，现在使用所谓的 Unicode 生成文本。

在 Python 中，有两种不同种类的字符串数据：文本字符串与字节字符串。两种字符串之间可以互相转换。理解正在处理的是哪种字符串数据非常重要，从而可以对字符串的使用保持一致。

在本章中，你将会学到文本字符串与字节字符串的区别，以及这两类字符串在 Python 2 与 Python 3 中实现方式的区别。你还将学到在 Python 程序中与字符串数据打交道时如何处理常见问题。

### 8.1 文本字符串与字节字符串

数据一直以字节方式存储。诸如 ASCII 与 Unicode 的字符集负责使用字节数据生成对应的文本。

ASCII 生成文本的方式非常简单直接。通过定义一个匹配表，在该表中每一个字符对应 7 位。一个常见的 ASCII 码的超集是 latin-1(在后面会详细讨论)，但使用 8 位维护这个系统。通常，使用十进制或十六进制表示字符。因此，每当 ASCII 码遇到由数字 65(或十

六进制 0x41)表示的字节时，就会知道该数据对应字符 A。

实际上，Python 本身定义了两个函数，用于在单整型字节与对应的字符之间转换，这两个函数分别为 `ord` 与 `chr`。“`ord`”是“ordinal”的缩写。`ord` 函数接受一个字符并返回在 ASCII 表中对应的整数，如下所示：

```
>>> ord('A')  
65
```

`chr` 方法则相反。接受一个整型并返回在 ASCII 表中对应的字符，如下所示：

```
>>> chr(65)  
'A'  
>>> chr(0x41)  
'A'
```

ASCII 的根本问题是假设字符与字节 1:1 对应。这是一个非常严重的限制，因为 256 个字符完全无法包含不同语言的字符。Unicode 通过使用 4 字节来表示每个字符解决了该问题。

## Python 中的字符串数据

实际上，Python 语言有两种不同的字符串：一个用于存储文本，另一个存储原始字节。文本字符串内部使用 Unicode 存储，而字节字符串存储原始字节并显示 ASCII(例如，当发送到 `print` 时)。

Python 2 与 Python 3 为文本字符串与字节字符串使用不同(但重叠)的名称进一步导致混淆。Python 3 的术语更有意义，因此你应该学习 Python 3 中的命名并在使用时与 Python 2 中定义的进行转换。

### 1. Python 3 字符串

在 Python 3 中，文本字符串类型(使用 Unicode 数据存储)被命名为 `str`，字节字符串类型被命名为 `bytes`。正常情况下，实例化一个字符串会得到一个 `str` 实例，如下所示：

```
>>> text_str = 'The quick brown fox jumped over the lazy dogs.'  
>>> type(text_str)  
<class 'str'>
```

如果你希望获得一个 `bytes` 实例，则需要在文本之前加上 `b` 字符。

```
>>> byte_str = b'The quick brown fox jumped over the lazy dogs.'  
>>> type(byte_str)  
class 'bytes'
```

可以在一个 `str` 与一个 `bytes` 之间进行转换。`str` 类包含一个 `encode` 方法，用于使用特定编码将其转换为一个 `bytes`。在大多数情况下，在编码数据时你会希望使用 UTF-8 作为编码。`encode` 方法接受一个必要参数，该参数是一个代表特定编码的字符串。

```
>>> text_str.encode('utf-8')
b'The quick brown fox jumped over the lazy dogs.'
```

与此类似，`bytes` 类包含了一个 `decode` 方法，也接受一个编码作为单个必要参数，并返回一个 `str`。不过，解码是一个更有意思的问题。不应该武断地说总是应该将数据以 UTF-8 解码，因为来自其他源的数据并不一定使用 UTF-8 编码。你必须根据数据被编码时采用的编码进行解码。你将会在本章后面学到更多关于这方面的知识。

Python 3 永远不会尝试隐式地在一个 `str` 与一个 `bytes` 之间进行转换。其方法是需要你显式地使用 `str.encode` 与 `bytes.decode` 方法在文本字符串与字节字符串之间进行转换(现实是需要你指定编码)。对于大多数应用程序来说，这是更可取的行为，因为这有助于避免处理常见英文文本时程序正常工作，但在遇到意料外的字符时出错。

这也意味着仅包含 ASCII 字符的文本字符串并不等于仅包含 ASCII 字符的字节字符串。

```
>>> 'foo' == b'foo'
False
>>>
>>> d = {'foo': 'bar'}
>>> d[b'foo']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: b'foo'
```

对于文本字符串与字节字符串共同执行几乎任何操作都会引发 `TypeError` 异常，如下所示：

```
>>> 'foo' + b'bar'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
```

只有%操作符例外，该操作符在 Python 中用于字符串格式化。尝试将一个文本字符串插入到一个字节字符串将会如期望那样引发 `TypeError` 异常。

```
>>> b'foo %s' % 'bar'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'str'
```

另一方面，尝试将一个字节字符串插入到一个文本字符串却可以生效，但并不按照直觉返回期望的结果。

```
>>> 'foo %s' % b'bar'
"foo b'bar'"
```

这里发生的是操作符接受 `b'bar'` 的值，该值是一个 `bytes` 对象的实例。首先寻找该值的 `_str_` 方法，而 `bytes` 对象实际上包含该方法。它返回文本字符串 "`b'bar'`"，该字符串的前缀为 `b'` 后缀为 '`'`。这个值与 `_repr_` 方法返回的值相同。

## 2. Python 2 字符串

Python 2 字符串的工作机制类似，但有一些微妙却重要的区别。

第一个区别是类名称。Python 3 中的 str 类在 Python 2 中名称为 unicode。如果就这一点并没有问题。但是，Python 3 的 bytes 类在 Python 2 中名称为 str。这意味着在 Python 3 中 str 是一个文本字符串，而在 Python 2 中 str 是一个字节字符串。如果使用 Python 2，理解这个区别非常重要。

若不使用前缀实例化字符串，则返回一个 str 实例(记住，这是一个字节字符串！)

```
>>> byte_str = 'The quick brown fox jumped over the lazy dogs.'  
>>> type(byte_str)  
<type 'str'>
```

如果你希望在 Python 2 中得到一个文本字符串，则在字符串之前加上 u 字符，如下所示：

```
>>> text_str = u'The quick brown fox jumped over the lazy dogs.'  
>>> type(text_str)  
<type 'unicode'>
```

与 Python 3 不同，Python 2 会在文本字符串与字节字符串之间尝试进行隐式转换。该工作机制是，如果解释器遇到一个不同种类字符串的混合操作，解释器首先会将字节字符串转换为文本字符串，然后对文本字符串执行操作。

这种工作机制使得对一个字节字符串和一个文本字符串的操作最终会返回一个文本字符串：

```
>>> 'foo' + u'bar'  
u'foobar'
```

解释器使用默认编码进行这个隐式解码。在 Python 2 中，默认编码几乎总是 ASCII。Python 定义了一个方法，sys.setdefaultencoding，用于为文本字符串与字节字符串之间进行隐式转换提供默认编码。

```
>>> import sys  
>>> sys.setdefaultencoding()  
'ascii'
```

这意味着之前 Python 3 中的很多示例在 Python 2 中会展示出不同的行为。

```
>>> 'foo' == u'foo'  
True  
>>>  
>>> d = {u'foo': u'bar'}  
>>> d['foo']  
u'bar'
```

### str.encode 和 unicode.decode

Python 2 的字符串处理行为中有点奇怪的方面是文本字符串有一个 decode 方法，而字

节字符串有一个 encode 方法。

你永远不会希望用到这些方法。

从理论上讲，这两个方法的目的是确保不用过于担心输入变量类型是什么。调用 encode 可以将任意类型的字符串转换为字节字符串，或使用 decode 将任意类型字符串转换为文本字符串。

但在实际使用中，这会让人迷惑并导致灾难，这是由于如果方法接受了“错误”的输入字符串(也就是，一个已经是所希望的输出类型的类型)，该方法会尝试进行两次转换，其中一次隐式转换使用 ASCII。

考虑下面的 Python 2 示例：

```
>>> text_str = u'\u03b1 is for alpha.'  
>>>  
>>> text_str.encode('utf-8')  
'\xce\xb1 is for alpha.'  
>>>  
>>> text_str.encode('utf-8').encode('utf-8')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
UnicodeDecodeError: 'ascii' codec can't decode byte 0xce in position 0:  
    ordinal not in range(128)
```

请求将某个对象使用 UTF-8 进行编码但回报错信息，报错信息显示说文本无法被解码为 ASCII 码，这十分奇怪。但这就是 Python 2 中为了对一个字节字符串执行 encode(该方法用于文本字符串)所尝试进行的隐式转换。

对于解释器来说，最后一行代码等同于下面的代码：

```
text_str.encode('utf-8').decode('ascii').encode('utf-8')
```

这永远不是你希望的。

不重复执行编码解码方法看上去很容易，但你遇到类似这种错误的形式往往并不是像示例中那样盲目地执行两次 encode，而是在执行 encode 或 decode 之前没有首先检查所使用的数据类型。在 Python 2 中，文本字符串与字节字符串频繁混合使用，因此很容易出现你得到的结果并非是你所希望的结果的情况。

### unicode\_literals

如果使用的是 Python 2.6 或更新的版本，只要你愿意，可以使其中的部分行为与 Python 3 中的相同。Python 定义了一个名称为 `_future_` 的特殊模块，可以使用该模块主动选择未来的行为。

在本例中，导入 `unicode_literals` 会导致字符串文本遵循 Python 3 的约定，虽然仍然使用的是 Python 2 的类名称。

```
>>> from __future__ import unicode_literals  
>>> text_str = 'The quick brown fox jumped over the lazy dogs.'  
>>> type(text_str)
```

```
<type 'unicode'>
>>> bytes_str = b'The quick brown fox jumped over the lazy dogs.'
>>> type(bytes_str)
<type 'str'>
```

一旦调用了 `from __future__ import unicode_literals`, Python 2.6 及更高版本中没有前缀的字符串就变为文本字符串(unicode), 而使用前缀 `b` 创建字节字符串(python 2 的 str)。

导入该模块并不会将 Python 2 中的其他字符串处理机制转换为遵循 Python 3 的行为。解释器仍然会在字节字符串与文本字符串之间进行隐式转换, 并且 ASCII 依然是默认编码。

尽管如此, 大多数代码中指定的字符串希望使用的是文本字符串而不是字节字符串。因此, 如果你所编写的代码不需要支持 Python 2.6 及更低版本, 使用 Python 3 的行为则更加明智。

### 3. six

虽然过渡到更加清晰的 Python 3 命名方式非常重要, 但 Python 2 与 Python 3 为文本字符串与字节字符串提供不同类名称这一事实是导致混淆的根源。

为了协助处理这种混淆, 流行的 Python 类库 `six` 是一个以编写能够正确运行在 Python 2 与 Python 3 下的模块为中心的类库, 该类库为这些类型提供了别名, 因此在必须运行在这两个平台的代码中进行引用时要保持一致。文本字符串类(在 Python 3 中是 `str`, 在 Python 2 中是 `unicode`)的别名是 `six.text_type`, 而字节字符串(在 Python 3 中是 `bytes`, 在 Python 2 中是 `str`)类的别名是 `six.binary_type`。

## 8.2 包含非 ASCII 字符的字符串

大多数 Python 程序以及几乎所有需要处理用户输入(无论是直接输入, 或者来自文件、数据库等)的程序都必须能够处理任意字符, 包括不在 ASCII 表中的字符。在文本字符串与字节字符串中转换 ASCII 字符非常直观(在 UTF-8 编码中, 实际上是空操作)。但涉及非 ASCII 字符时就会变得复杂, 尤其是在使用时未留意使用的是文本字符串还是字节字符串的情况下。

### 8.2.1 观察区别

考虑一个包含非 ASCII 字符的文本字符串, 如下面代码中的文本字符串 “Hello, World”, 使用 Google 将其翻译成希腊文(注意代码是 Python 3 代码):

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> type(text_str)
<class 'str'>
```

对于这个文本字符串, 首先要注意的是其完全无法使用 ASCII 编码编码为 bytes 实例。

```
>>> text_str.encode('ascii')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-3:
ordinal not in range(128)
```

这是由于 ASCII 中不包含希腊字符,因此 ASCII 编码无法将其转换为原始的字节数据。这没有问题,然而,这是 utf-8 编码的用武之地,如下所示:

```
>>> text_str.encode('utf-8')
b'\xce\x93\xce\xb5\xce\xb9\xce\xb1\xcf\x83\xce\xb1\xcf\x82,
\xcf\x84\xce\xbf\xce\xbd\xce\xba\xcf\x8c\xcf\x83\xce\xbc\xce\xbf.'
```

在此要注意的是,首先,这是你遇到的第一个其文本字符串与字节字符串看上去完全不同的字符串。文本字符串的 `repr` 看上去像是人类可读的希腊文,而字节字符串的 `repr` 看上去是机器可读的。

另外,注意字符串的长度也不同。

```
>>> byte_str = text_str.encode('utf-8')
>>> len(text_str)
20
>>> len(byte_str)
35
```

为什么会这样?请记住 Unicode 的存在是为了解决:ASCII 假定字节与字符 1:1 的对应关系,这会实际限制可用的字符个数。

Unicode 通过破除该限制使得允许更多字符。UTF-8 字符是变长的。一个 Unicode 字符最小可以是 1 个字节(对于 ASCII 表中的字符),最大可以是 4 个字节。

在希腊文本的示例中,大多数字符是两个字节,这也是为什么字节字符串的长度几乎是文本字符串长度的两倍。然而,对于空格、句号和逗号(在字节字符串中很明显)都是 ASCII 字符,并且每个字符只占用一个字节。

## 8.2.2 Unicode 是 ASCII 的超集

为什么在打印到屏幕时只包含 ASCII 字符的文本字符串与字节字符串看上去非常类似,而包含 Unicode 的字符串却看上去完全不同呢?

按照约定,打印 ASCII 范围内的字节与 ASCII 字符保持一致。另外,Unicode 的这种结构使得它是 ASCII 的超集。这意味着在拉丁字母表中的字符以及常见标点符号在 Unicode 字符串与字节字符串中以同样的方式表示。

这包含另外一个重要意义,即任何有效的 ASCII 文本同时也是有效的 Unicode 文本。

## 8.3 其他编码

Unicode 并不是在原始字节数据与可读文本表示之间进行转换的唯一可用编码。还有

# 尚学堂·百战程序员

第III部分 数 据

[www.itbaizhan.cn](http://www.itbaizhan.cn)

很多其他编码，其中一些编码被广泛使用。

一个广为人知的常见编码是 ISO-8859 标准，通俗叫法是 latin-1(为清楚起见，本章的余下部分使用“Latin-1”而不是 ISO-8859 指代该编码)。

与 Unicode 类似，该编码是 ASCII 的超集，为很多除英语之外的语言中发现的字形添加了支持。然而，顾名思义，其被设计为只支持字母依赖于拉丁字形的语言，因此并不适用于使用其他字母的语言(比如希腊文、中文、日文、俄文、韩文等)。

实际上不可能使用 latin-1 编码生成之前的希腊文字符串，正如下面 Python 3 示例中所证明的：

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> text_str.encode('latin-1')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'latin-1' codec can't encode characters in position 0-3: ordinal not in range(256)
```

## 编码互不兼容

虽然很多编码被构建为 ASCII 的超集，但它们之间通常并不相互兼容，认识到这一点非常重要。除去 ASCII 码，在 latin-1 与 utf-8 编码之间几乎没有重叠部分。

考虑使用这两种编码对字节字符串进行编码的区别。

```
>>> text_str = 'El zorro marrón rápidamente saltó por encima de los perros vagos.'
>>> text_str.encode('utf-8')
b'El zorro marr\xc3\xb3n r\xc3\xalpido salt\xc3\xb3 por encima de los perros vagos.'
>>> text_str.encode('latin-1')
b'El zorro marr\xf3n r\xelpido salt\xf3 por encima de los perros vagos.'
```

因此，用一种编码所编码的字符串无法使用另一种编码进行解码。如果你尝试将一个字节字符串使用 latin-1 编码而使用 utf-8 解码，Unicode 编码会意识到遇到无效字符序列并解码失败。

```
>>> text_str.encode('latin-1').decode('utf-8')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf3 in position 13:
invalid continuation byte
```

更糟的是，如果你尝试将一个字节字符串使用 utf-8 编码但使用 latin-1 解码，该编码(包含更多字符)会成功返回一个文本字符串，但该字符串是乱码。

```
>>> text_str.encode('utf-8').decode('latin-1')
'El zorro marrÃ³n rÃ;pido saltÃ³ por encima de los perros vagos.'
```

无法基于字节字符串的内容推断使用的是哪种编码。然而，很多常见的文档格式与数据传输协议提供了声明使用的是哪一种编码的机制。另一方面，文档也有可能错误地指定其字符编码。

## 8.4 读取文件

文件总是存储字节。因此，为了使用从文件中读取的文本数据，必须首先将其解码为一个文本字符串。

### 8.4.1 Python 3

在 Python 3 中，文件正常情况下会自动为你解码。考虑下面包含 Unicode 文本的文件，使用 UTF-8 解码：

```
Hello, world.  
Γεια σας, τον κόσμο.
```

在 Python 3 中打开并读取该文件会得到一个文本字符串(而不是字节字符串)。

```
>>> with open('unicode.txt', 'r') as f:  
...     text_str = f.read()  
...  
>>> type(text_str)  
<class 'str'>
```

该代码示例做了几个需要重点理解的关键假设。

所做的最大的假设是如何解码文件。文本文件并没有声明它们如何编码。解释器无法得知其读取的是 UTF-8 文本还是 Latin-1 文本或是其他编码的文本。

Python 3 决定使用哪种编码取决于运行它的系统。这可以通过函数 `locale.getpreferredencoding()` 获知。在 Mac OS X 以及大多数 Linux 系统下，首选编码是 UTF-8。

```
>>> import locale  
>>> locale.getpreferredencoding()  
'UTF-8'
```

然而，大多数 Windows 系统使用名为 Windows-1252 或 CP-1252 的另一种编码来编码文本文件，在 Windows 上的 Python 3 中运行相同代码则反映这一点。

```
>>> import locale  
>>> locale.getpreferredencoding()  
'cp1252'
```

需要明确注意的重点是，`locale.getpreferredencoding()` 提供的首选编码基于底层系统的运行方式。这是基于反射的，而不是基于规则。在任意系统中保存(使用系统默认工具)的包含特殊字符的文本文件，都可以在 Python 3 中打开并正确解码。

然而，文件并不总是在同一种操作系统中被保存和打开，这也是假设带来的问题。

## 1. 指定编码

Python 3 通过为 `open` 方法提供一个可选的 `encoding` 关键字，允许你显式声明文件的编码。该参数以字符串的形式接受一个编码，与 `encode` 和 `decode` 类似。

由于示例 Unicode 文件的规则是使用 UTF-8 编码，因此可以显式地通知解释器使用该编码来解码文件。

```
>>> with open('unicode.txt', 'r', encoding='utf-8') as f:  
...     text_str = f.read()  
...  
>>> type(text_str)  
<class 'str'>
```

由于文件通过 UTF-8 编码，并且使用 UTF-8 解码，因此文本字符串包含预期的数据。

```
>>> text_str  
'Hello, world.\nΓεια σας, τον κόσμο.\n'
```

## 2. 读取字节

另一个所做的隐式假设是(逻辑上处于使用哪一种编码解码文件之前)文件是否需要被解码。

你或许希望以字节字符串而不是文本字符串的形式读取文件。这么做有两个常见原因。最常见的原因是所接收到的是一个非文本数据(例如，所读取的是一个图像)。然而，另一个潜在原因是无法确定所读取文本文件的编码。

为了读取字节字符串而不是文本字符串，为发送给 `open` 方法的第二个字符串参数添加字符 `b`。例如，考虑读取同一个文件，但以字节字符串形式读取 Unicode，如下所示：

```
>>> with open('unicode.txt', 'rb') as f:  
...     byte_str = f.read()  
...  
>>> type(byte_str)  
<class 'bytes'>
```

查看 `byte_str` 变量显示在第二行文本字符串中的原始字节：

```
>>> byte_str  
b'Hello, world.\n\xce\x93\xce\xb5\xce\xb9\xce\xb1 \xcf\x83\xce\xb1\xcf\x82,  
 \xcf\x84\xce\xbf\xce\xbd \xce\xba\xcf\x8c\xcf\x83\xce\xbc\xce\xbf.\n'
```

该变量被解码的形式与来自其他源提供的字节字符串解码的形式并无不同。

```
>>> byte_str.decode('utf-8')  
'Hello, world.\nΓεια σας, τον κόσμο.\n'
```

在处理编码不确定的文件时，这是一种有效的策略。数据可以以字节的形式从文件中被安全地读取，然后程序可以通过编程确定该如何对其进行解码。

## 8.4.2 Python 2

在 Python 2 中，无论以何种方式打开文件，read 方法总是返回一个字节字符串。

```
>>> with open('unicode.txt', 'r') as f:  
...     byte_str = f.read()  
  
...  
>>> type(byte_str)  
<type 'str'>
```

注意，b 修饰符并没有被应用在 open 方法的第二个参数中，但仍然返回一个 str 实例(在 Python 2 中是一个字节字符串)。

可以使用 decode 获得一个文本字符串，就像对来自任意源的字节字符串那样。

```
>>> byte_str  
'Hello, world.\n\xce\x93\xce\xb5\xce\xb9\xce\xb1 \xcf\x83\xce\xb1\xcf\x82,  
 \xcf\x84\xce\xbf\xce\xbd \xce\xba\xcf\x8c\xcf\x83\xce\xbc\xce\xbf.\n'  
>>>  
>>> byte_str.decode('utf-8')  
u'Hello, world.\n\u0393\u03b5\u03b9\u03b1 \u03c3\u03b1\u03c2,  
 \u03c4\u03bf\u03bd \u03ba\u03cc\u03c3\u03bc\u03bf.\n'
```

由于 Python 2 总是提供字节字符串，因此 open 函数并没有提供 encoding 关键字参数，如果尝试提供一个参数，则会引发 TypeError 异常。

如果编写希望运行在 Python 2 上的代码，最好且最安全的方式是以二进制模式(使用 b)打开文件，如果希望获取文本数据，请自行解码。

## 8.4.3 读取其他源

可以从很多不同的源中读取文本数据，而不仅仅是从文件读取。现代程序接受直接的用户输入，通过协议(比如 HTTP)接受输入，从数据库中读取，并使用诸如扩展标记语言(XML)或 JSON 等序列化格式传输数据。

Python 提供了很多库和工具读取来自不同元的不同类型数据。例如，Python 2.6 及之后版本中的 json 模块可用于序列化与反序列化 JSON 数据。例如，pyyaml 库读取 YAML 文件，psycopg2 库从 PostgreSQL 数据库中读写数据。

这些库中的大多数(但并不是所有)返回文本字符串。然而，熟悉所使用的库并且知道获取的是文本字符串还是字节字符串则是你的责任。另外，有些库在不同版本的 Python 下的行为或许不同，在 Python 2 中返回字节字符串，而在 Python 3 中返回文本字符串。确保使用合适的类型非常重要。

## 8.4.4 指定 Python 文件编码

很多文档格式提供声明所使用的是哪一种编码的方式。例如，XML 文件的开头部分可能如下所示：

# 尚学堂·百战程序员

第III部分 数 据

www.itbaizhan.cn

```
<?xml version="1.0" encoding="UTF-8"?>
```

这是开始一个 XML 文件的常见方式。注意 `encoding` 属性。该属性声明该文本数据使用 UTF-8 进行编码。由于 XML 文件声明这是它所使用的编码，因此读取 XML 的程序将使用 UTF-8 由字节到文本来解码其发现的所有文本。

有些时候，在 Python 源文件中声明一个编码是必须的。例如，假设一个 Python 源文件包含一个其中含有 Unicode 字符的字符串文字。在 Python 2 中，解释器假设 Python 源文件使用 ASCII 编码，这会导致出错。

考虑下面存储为 `unicode.py` 的模块：

```
text_str = u'Γεια σας, τον κόσμο.'  
print(text_str)
```

在 Python 3.3 或更高版本(由于 Python 3.0-3.2 缺乏 `u` 前缀)中运行该模块不会遇到任何问题。

```
$ python3.4 unicode.py  
Γεια σας, τον κόσμο.
```

然而，在 Python 2 中运行同一个模块会在第一行报语法错误，这是由于 Python 2 解释器希望获得的是 ASCII 编码。

```
$ python2.7 unicode.py  
File "unicode.py", line 1  
SyntaxError: Non-ASCII character '\xe9' in file unicode.py on line 1, but  
no encoding declared; see http://www.python.org/peps/pep-0263.html  
for details
```

正如错误信息所提示，Python 模块实际上可以声明一个编码，类似于在 XML 文件可以声明编码那样。默认情况下，Python 2 期望文件使用 ASCII 进行编码，Python 3 期望文件使用 UTF-8 进行编码。

为了重写这部分，Python 允许你在模块开头包含一个注释，并以特定的形式进行格式化。解释器会读取该注释并将其作为编码声明进行使用。

为 Python 文件指定编码的格式如下：

```
# -*- coding: utf-8 -*-
```

在此可以使用任何传递给 `encode` 与 `decode` 的编码。因此，诸如 `ascii`、`latin-1`、`cf-1252` 等值都可以接受(当然，假设该文件是以这些编码进行编码)。

考虑包含 `coding` 声明的同一个模块：

```
# -*- coding: utf-8 -*-  
text_str = u'Γεια σας, τον κόσμο.'  
print(text_str)
```

如果在 Python 2 中运行这个已修改的文件，将很顺利而不会引发语法错误。

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
$ python2.7 unicode.py
Γεια σας, τον κόσμο.
```

注意，如果选择手动为 Python 模块指定一种编码，那么确保所指定的编码正确则是你的责任。正如其他文档格式那样，Python 模块也有可能声明的是一种编码而使用的是另一种编码。

如果你碰巧指定了错误的编码，那么显示出来的字符串就像一堆垃圾。考虑同样的文件如果声明使用 latin-1 编码会发生什么(实际上是使用 utf-8 字符)。

```
# -*- coding: latin-1 -*-
text_str = u'Γεια σας, τον κόσμο.'
print(text_str)
```

无论在 Python 2 中还是在 Python 3.3 以上版本中运行这段代码都产生同样的结果，完全是一堆乱码。

```
$ python3.4 unicode.py
Í»í»í»í» í»í»í»í» í»í»í»í» í»í»í»í».
```

由于 latin-1 编码可以接受几乎任意字节流，因此并不会实际识别出这并不是 latin-1 编码文本，而是会返回不正确的数据。某些编码(比如 utf-8)更加严格，该编码会报错而不显示乱码。报错的情况更可取，但这两种都不是你希望的结果。因此正确声明编码非常重要。

注意，这还取决于终端显示字符的能力。如果所使用的终端不支持 Unicode，很可能引发异常。

## 8.5 严格编码

除了支持所有 Unicode 范围的字符之外，utf-8 作为编码的一个关键优势在于它是一种“严格”的编码。这意味着它不仅仅是接受任意字节流并解码，通常，它还可以检测无效的非 Unicode 字节流并报错。

当处理编码未知的字节流(因为无法根据推断完全确定编码)时这是一种有效的模式。例如，假如你认为一个字节流可能是 utf-8 也可能是 latin-1 时，可以尝试两者，如下所示：

```
try:
    text_str = byte_str.decode('utf-8')
except UnicodeDecodeError:
    text_str = byte_str.decode('latin-1')
```

当然，这并不是灵丹妙药。例如，如果你获得一个编码为完全不同的字节字符串时会发生什么？由于 Latin-1 是一个悲观编码，因此它会将其不正确地解码。

### 8.5.1 不触发错误

有时在使用严格编码(比如 utf-8 或 ascii)解码或编码字符串时，你并不希望当编码遇到

# 尚学堂·百战程序员

第III部分 数 据

[www.itbaizhan.cn](http://www.itbaizhan.cn)

它无法正确处理的文本时，触发一个严格错误。

encode 与 decode 方法提供了一种在遇到一组其无法处理的字符时，能够以不同行为响应的机制。这两个方法都有第二个可选参数 errors，参数类型为字符串。默认值为 strict，它会引发诸如 UnicodeDecodeError 类的异常。另外两个常见的错误处理程序为 ignore 与 replace。

ignore 错误处理程序仅仅是跳过所有编码无法解码的字节。考虑如果你尝试用 ASCII 解码希腊字符时会发生什么，如下所示：

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> byte_str = text_str.encode('utf-8')
>>> byte_str.decode('ascii', 'ignore')
'', ''
```

ASCII 码并不知道如何处理任何希腊字符，但它知道如何处理空格与句号。因此，保留这些符号，但剥离所有外国字符。

replace 错误处理程序也是如此，但与跳过无法识别的字符不同，它将这些字符替换为一个占位字符。具体的占位字符基于情况(是编码还是解码以及所使用的编码类型)而稍有区别，但通常是一个问号(?)或是一个特殊的 Unicode 问号的菱形字符(◊)。

如果尝试使用 ascii 编码来解码希腊文本并使用 replace 错误处理程序，结果将如下所示：

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> byte_str = text_str.encode('utf-8')
>>> byte_str.decode('ascii', 'replace')
'?????? ????? ???? ?????? ?????????????!'
```

如果尝试使用 ascii 编码将希腊文本编码为一个字节字符串并使用 replace 错误处理程序会得到如下结果：

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> text_str.encode('ascii', 'replace')
b'????? ???, ??? ?????.'
```

你或许会注意到，当使用 replace 错误处理程序时，替换字符的数量或许与实际文本字符串中的字符并不是 1:1 的对应关系。当使用 ascii 编码来解码字节字符串时，编码无法获知每个字符对应多少字节，所以最终会比在文本字符串中的实际字符显示更多的问号。

## 8.5.2 注册错误处理程序

如果内置的错误处理程序无法满足需求，还可以注册额外的错误处理程序。codecs 模块(也就是定义默认错误处理程序的地方)公开了一个用于注册额外错误处理程序的函数，名称为 register\_error。该函数接受两个参数：错误处理程序的名称与实际进行错误处理的函数。

该函数接收将被引发的异常，并负责再次引发它，引发另一个异常，或是返回一个合适的字符串值，用于替换结果字符串中的内容。

异常实例包含 `start` 属性与 `end` 属性，它们分别与编码无法编码或解码的子字符串的开始与结束位置相对应。异常实例还包含一个 `reason` 属性，该属性对应于人类可读的解释，用于解释为什么无法编码或解码字符，另外，还包含一个 `object` 属性，该属性对应于原始字符串。

如果返回一个替换值，错误函数必须返回一个包含两个元素的元组。第一个元素是替换字符或字符串，第二个元素是在原始字符串中编码或解码应该继续进行的位置。通常情况下，这与异常实例中的 `end` 属性相对应。如果你这样做，请注意返回的 `start` 位置。这非常容易导致死循环。

下面的示例仅使用不同的替代字符替换字符。

```
def replace_with_underscore(err):
    length = err.end - err.start
    return ('_' * length, err.end)
codecs.register_error('replace_with_underscore', replace_with_underscore)
```

该错误处理程序替换未知的字符，但使用的是下划线而不是问号。如果使用 `ascii` 解码一个包含 Unicode 希腊文本的字节字符串以及使用该错误处理程序，则结果如下：

```
>>> text_str = 'Γεια σας, τον κόσμο.'
>>> byte_str = text_str.encode('utf-8')
>>> byte_str.decode('ascii', 'replace_with_underscore')
'_____ , _____.'
```

## 8.6 小结

处理字符串数据可以让人异常受挫。可能会很容易遇到这种情况，你创建一个程序一直能够正常工作，直到遇到与期望编码不同的文本数据。

当可能时，应尽量在程序中处理文本字符串。在接收到字节字符串后第一时间对其进行解码是一个好主意。同样，当输出数据时，应努力尽可能晚地将文本字符串编码为字节字符串。

有时解码很困难。你或许不知道一个字节字符串的编码方式，或者你被告知一种编码，但该编码是错误的。这很有挑战性，并且没有一个简单方案。

记住，Python 解释器在这里是你的朋友。如果处理不确定的数据，并且你不知道编码，你或许可以交互式地使用不同的编码来解码示例数据，直到你找到一种编码使得解码后的数据看起来合理。当然，该手动方法假设你为其编码的数据与你所使用的示例数据总是类似。

需要记住的重点是，当处理字符串数据时，要确保你总是知道所处理的字符串类型。最糟糕和令人沮丧的问题是，你认为是一个文本字符串但却收到一个字节字符串，或是相反。请确保这些字符串数据的直观性。

第9章介绍正则表达式，它是一种在字符串中搜索匹配给定模式数据的机制。

# 第 9 章

## 正则表达式

正则表达式是一个以简单直观的方式通过寻找模式(而不是寻找文本字符串)匹配文本的工具。例如，可以使用 Python `in` 关键字检查一个具体的文本字符串是否在另一个字符串中，如下所示。

```
>>> haystack = 'My phone number is 213-867-5309.'  
>>> '213-867-5309' in haystack  
True
```

然而有时，你并没有想要匹配的具体字符串。例如，假如你希望知道在一个字符串中是否存在任何有效的电话号码该怎么办？更进一步，假如你希望知道在一个字符串中是否存在任何有效的电话号码同时也想知道电话号码是什么，该怎么办呢？

这也是正则表达式的用武之地。它的目的是指定一个文本模式从而识别该模式是否在一个更大的文本字符串中。正则表达式可以识别匹配模式的文本是否存在，并且还能将一个模式分解为一个或多个子模式，并展示每个子模式匹配的文本。

本章探索正则表达式(为了简短，使用 `regexes`)。首先，你将会学习如何在 Python 中通过使用 `re` 模块执行正则表达式。然后探索多种正则表达式，从简单的正则表达式开始，由浅入深。最后，学习有关正则表达式替代品的内容。

### 9.1 使用正则表达式的原因

使用正则表达式有两个常见的原因。

第一个原因是数据挖掘——也就是说，当你希望在一大堆文本中找到一小堆文本时(与给定模式匹配)。区别出与给定类型的信息(比如，`e-mail` 地址、URL、电话号码等)相匹配

的文本是非常常见的需求。

作为人类，我们总是基于模式识别这类信息是否存在。一个电视广告显示以.com 或.org 结尾的字符，我们根据直觉知道这是一个 Web 地址。

第二个原因是验证。可以使用正则表达式确认获得的数据是你所期望的。认定“外部”数据不可信通常是明智的，尤其是来自用户的数据。正则表达式可以帮助确定这些未经测试的数据是否有效。

这里的推论是正则表达式是用于将数据转为一致格式的有价值的工具。例如，电话号码能够以多种有效方式编写，如果你请求用户输入电话号码，会倾向于接受所有格式。然而，你真正想要的是存储实际电话号码的数字，然后能够以一致的格式显示。除了能够用于验证之外，正则表达式对于这类数据转换也同样有用。

## 9.2 Python 中的正则表达式

Python 标准库为使用正则表达式提供了 re 模块。

re 模块提供的主要函数是 search。它的目的是接受一个正则表达式(针)和一个字符串(草堆)，并返回发现的第一个匹配。如果完全没有找到匹配，re.search 返回 None。

尽量通过最简单的正则表达式来考虑 re.search 的实际使用，也就是搜索一个简单的字母数字字符串。

```
>>> import re  
>>> re.search(r'fox', 'The quick brown fox jumped...')  
<_sre.SRE_Match object; span=(16, 19), match='fox'>
```

在此，正则表达式解析器的工作十分简单。它在字符串中寻找单词 fox，并返回匹配的对象。

注意：re 模块还提供了一个名称为 match 的函数，该函数看上去类似于 search。但它们之间有一个重要的区别：match 仅仅搜索从字符串开始的第一个匹配。当你实际想要的是在一个字符串的任何位置找到匹配时，很容易错误地使用 re.match。如果你需要这么做，最好总是使用 re.search 并且使用^锚(在本章后面讨论)。

### 9.2.1 原始字符串

善于观察的读者或许已经注意到所指定的正则表达式略微不同：r'fox'。在字符串出现之前的字符 r 代表“raw”(并不是代表“regex”)。

原始字符串与正常字符串的区别是原始字符串不会将\字符解释成一个转义字符。这意味着，无法转义引号来避免字符串结束。

但是，原始字符串对于正则表达式尤其有用，这是因为正则表达式引擎自身需要\字符

对其自身进行转义。因此，对于正则表达式使用原始字符串很常见且有用。实际上，使用原始字符串表示正则表达式如此普遍，以至于一些语法高亮引擎实际上会为原始字符串提供正则表达式的语法高亮。

### 9.2.2 match 对象

match 对象有多种用于获取关于匹配信息的方法。`group` 方法无可争议是最重要的。它返回一个包含匹配文本的字符串，如下所示：

```
>>> match = re.search(r'fox', 'The quick brown fox jumped...')  
>>> match.group()  
'fox'
```

你或许会好奇该方法的名称为什么是 `group`。这是由于正则表达式可以拆分为多个只调出匹配子集的子组。你将会很快学到更多关于这方面的知识。

`match` 对象还有几个其他方法。`start` 方法提供了在原始字符串中匹配开始的索引，`end` 方法提供了在原始字符串中匹配结束的索引。

`groups` 与 `groupdict` 方法用于调出正则表达式的一部分。你会在后面关于正则表达式与反向引用的部分学到这些方法。

最后，`re` 属性包含用于匹配的正则表达式，`string` 属性包含被用作被搜索对象的字符串，`pos` 属性被设置为在字符串中搜索开始的位置。

### 9.2.3 找到多个匹配

`re.search` 的一个限制是它仅仅返回最近一个以 `match` 对象形式的匹配。如果在一个字符串内存在多个匹配，`re.search` 只会返回第一个匹配。通常这也正是你期望的结果。但是，有时需要当多个匹配存在时进行多个匹配。

`re` 模块为此提供了两个函数：`findall` 与 `finditer`。这两个方法都返回所有非重叠匹配，包括空匹配。`re.findall` 方法返回一个列表，`re.finditer` 返回一个生成器。

然而，这里有一个关键区别。这些方法并没有实际返回一个 `match` 对象。而是，它们仅仅根据正则表达式的内容以字符串或元组的形式返回匹配自身。

考虑一个 `findall` 的示例：

```
>>> import re  
>>> re.findall(r'o', 'The quick brown fox jumped...')  
['o', 'o']
```

在本例中，它返回带有两个字符 o 的列表，这是由于字符 o 在字符串中出现两次。

## 9.3 基本正则表达式

最简单的正则表达式是那些仅包含简单的字母数字字符的表达式——不包含任何其他字符。这很容易被忽视。很多正则表达式使用直接文本匹配。

字符串 Python 是一个有效的正则表达式。它仅匹配单词。默认正则表达式区分大小写，因此它不会匹配 python 或 PYTHON。

```
>>> re.search(r'Python', 'python')
>>> re.search(r'Python', 'PYTHON')
```

然而，它会匹配在更大文本段中的单词。它将会匹配在 Python 3 或 This is Python code 或其他类似文本段中的单词，如下所示：

```
>>> re.search(r'Python', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
>>> re.search(r'Python', 'This is Python code.')
<_sre.SRE_Match object; span=(8, 14), match='Python'>
```

当然，使用正则表达式仅仅匹配纯文本正则表达式基本上没有价值。毕竟，使用 in 操作符检测一个字符串是否在另一个字符串中出现会非常简单，并且使用 str.index 完成找到子字符串在大字符串中的位置已经绰绰有余。

正则表达式强大之处在于能够指定用于匹配的文本模式。

### 9.3.1 字符组

字符组允许指定一个字符应该与一组可能出现的字符之一，而不仅仅是一个单独字符匹配。你可以使用方括号并在方括号内列出所有可能的字符从而表示一个字符组。

例如，考虑一个应该与 Python 或 python 匹配的正则表达式：[Pp]ython。

这里发生了什么？在正则表达式中第一个标记实际上是一个带有两个选项(P 和 p)的字符组。每个字符都可以匹配，但不匹配其他字符。剩下的 5 个字符仅仅是文字字符。

下面的正则表达式匹配什么？

```
>>> re.search(r'[Pp]ython', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
>>> re.search(r'[Pp]ython', 'python 3')
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

正则表达式在字符串 Python 3 中匹配单词 Python 以及在字符串 Python 3 中匹配单词 python。可是它并没使得整个单词不区分大小写。它并不会与所有字符都是大写的单词匹配，例如：

```
>>> re.search(r'[Pp]ython', 'PYTHON 3')
>>>
```

另一种对于这类字符组的使用是对于有多种拼写的单词。正则表达式 gr[ae]y 会匹配 gray 或 grey，允许你快速识别并提取任意一种拼写。

```
>>> re.search(r'gr[ae]y', 'gray')
<_sre.SRE_Match object; span=(0, 4), match='gray'>
```

还值得注意的是，类似这种字符组匹配且仅仅匹配一个字符。

```
>>> re.search(r'gr[ae]y', 'graey')
>>>
```

这里，正则表达式引擎成功匹配文字 g，然后是文字 r。接下来，引擎被给予字符组[ae]，并与 a 匹配。现在，字符组已经匹配完成，然后引擎向后移动位置。在正则表达式中下一个字符是一个 y，但是字符串中下一个字符为 e。这并不匹配，因此正则表达式解析器向后移动，重新开始开头的 g。但它到达字符串末尾无法找到匹配时，返回 None。

## 1. 区间

一些常见的字符组非常大。例如，考虑匹配任意数字。每一次都提供[0123456789]很不明智。而提供所有包括大小写在内的字母就更不明智了。

为了适应这点，正则表达式引擎在字符组中使用连字符(-)代表区间。匹配任意数字的字符组可以写为[0-9]。对于一个字符组使用多个区间也是可能的，只需要一个接一个地提供范围。[a-z]字符组只匹配小写字符，[A-Z]字符组只匹配大写字符。它们可以组合为——[A-Za-z]，从而可以同时匹配大写或小写。

```
>>> re.search(r'[a-zA-Z]', 'x')
<_sre.SRE_Match object; span=(0, 1), match='x'>
>>> re.search(r'[a-zA-Z]', 'B')
<_sre.SRE_Match object; span=(0, 1), match='B'>
```

当然，你或许也想匹配连字符。这惊人地常见。匹配(例如)字母数字字符、连字符、下划线的原因有很多。假如你需要这么做时会发生什么？

可以转义连字符：[A-Za-z0-9\-\\_]。这会告诉正则表达式引擎你希望的是一个字面上的连字符。但是，转义通常使得连字符更加难以阅读。也可以将连字符作为字符组的第一或最后一个字符，正如在[A-Za-z0-9\-\\_]中。在本例中，引擎将会将字符解译为字面的连字符。

## 2. 取反

迄今为止的字符组都由可能出现的字符定义。但是，你或许希望根据可能不会出现的字符定义一个字符组。

可以通过在字符组开头使用^字符，从而可以反转一个字符组(意味着它会匹配任何指定字符之外的其他所有字符)。

```
>>> re.search(r'^[a-z]', '4')
<_sre.SRE_Match object; span=(0, 1), match='4'>
>>> re.search(r'^[a-z]', '#')
<_sre.SRE_Match object; span=(0, 1), match='#'>
```

# 尚学堂·百战程序员

第III部分 数 据

www.itbaizhan.cn

```
>>> re.search(r'[^a-z]', 'X')
<sre.SRE_Match object; span=(0, 1), match='X'>
>>> re.search(r'[^a-z]', 'd')
>>>
```

在该场景中，正则表达式解析器寻找所有不在 a 到 z 范围内的字符。因此，它会匹配数字、大写字符、符号，但不会匹配小写字符。

这里尤其需要注意的是正则表达式在这里寻找什么。它寻找的字符与出现在字符组中的所有字符不匹配。它不是寻找(不会匹配)字符的缺失。

考虑正则表达式 `n[^e]`。这意味着字符 n 接下来的字符可以是除 e 之外的所有字符。

```
>>> re.search(r'n[^e]', 'final')
<sre.SRE_Match object; span=(2, 4), match='na'>
```

在该例中，它与单词 final 匹配，匹配部分是 na。字符 a 是匹配的一部分，这是因为它是非 e 的单个字符。

如你所料，该正则表达式在 n 后面跟随 e 时会匹配失败。

```
>>> re.search(r'n[^e]', 'jasmine')
>>>
```

这里，正则表达式引擎只到了字符串中的 n 位置，但无法匹配下一个字符，因为它是一个 e，并且因此没有匹配。

但是，正则表达式也无法匹配处于字符串末尾的 n。

```
>>> re.search(r'n[^e]', 'Python')
>>>
```

该正则表达式发现单词 Python 中的 n。然而，这就是它所获得的全部，字符串中没有剩下能够匹配`[^e]`的字符，并且因此，匹配失败。

### 3. 快捷方式

几种普通字符组还在正则表达式引擎中有几个预定义的快捷方式。如果你希望定义“单词”，你的直觉或许是使用`[A-Za-z]`。但是，很多单词都使用该区间以外的字符。

正则表达式引擎提供了一个快捷方式，`\w`，与“任意单词字符”匹配。“任意单词字符”是如何定义的，根据环境的不同有所区别。在 Python 3 中，它基本上与几乎任何语言的任意单词匹配。在 Python 2 中，它仅仅与英语单词字符匹配。无论是哪个版本，它都会匹配数字、下划线与连字符。

`\d` 快捷方式匹配数字字符。在 Python 3 中，它还与其他语言的数字字符匹配。在 Python 2 中，它只匹配`[0-9]`。

`\s` 快捷方式匹配空白字符，比如空格、`tab`、换行等。空白字符所包含的列表在 Python 3 中要比 Python 2 中长很多。

最后，`\b` 快捷方式匹配一个长度为 0 的子串。但是，它仅仅在一个单词开始或结尾处

匹配。这被称为词边界字符快捷方式。

```
>>> re.search(r'\bcorn\b', 'corn')
<_sre.SRE_Match object; span=(0, 4), match='corn'>
>>> re.search(r'\bcorn\b', 'corner')
>>>
```

正则表达式引擎在这里匹配 corn，由于 corn 单独是一个单词，但无法匹配单词 corner，这是由于结尾的\b 不匹配(因为下一个字符是 e，是一个单词字符)。

值得一提的是这些快捷方式无论在字符组内还是字符组外同样生效。例如，正则表达式\w 将会匹配任意单词字符。

```
>>> re.search(r'\w', 'Python 3')
<_sre.SRE_Match object; span=(0, 1), match='P'>
```

由于 re.search 仅仅返回第一个匹配，它与字符 P 匹配然后完成。考虑使用相同的正则表达式和字符串时，re.findall 的结果是什么。

```
>>> re.findall(r'\w', 'Python 3')
['P', 'y', 't', 'h', 'o', 'n', '3']
```

注意正则表达式与字符串中除了空格之外的所有字符匹配。在 Python 正则表达式引擎中\w 快捷方式包括数字。

\w、\d 与\s 快捷方式还包括取反快捷方式：\W、\D 与\S。这些快捷方式匹配快捷方式中包含字符之外的所有字符。再次注意这仍然需要至少出现一个字符。它们无法与空字符串匹配。

还有一个对于\b 的取反快捷方式，但机制有所不同。\\b 匹配在单词开始或结束位置长度为 0 的子字符串，而\\B 匹配不在单词开始或结束位置长度为 0 的子字符串。这本质上反转了之前示例中 corn 与 corner 示例的结果。

```
>>> re.search(r'corn\\B', 'corner')
<_sre.SRE_Match object; span=(0, 4), match='corn'>
>>> re.search(r'corn\\B', 'corn')
>>>
```

#### 4. 字符串的开始与结束

有两个特殊字符意味着字符串的开始与结束。

^字符指定字符串的开始，如下所示：

```
>>> re.search(r'^Python', 'This code is in Python.')
>>> re.search(r'^Python', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
```

注意第一个命令无法产生一个匹配。这是由于字符串并不是以单词 Python 开始，而^字符需要正则表达式与字符串开始部分匹配。

类似，\$字符指定一个字符串的结束，如下所示：

# 尚学堂·百战程序员

第III部分 数 据

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
>>> re.search(r'fox$', 'The quick brown fox jumped over the lazy dogs.')
>>> re.search(r'fox$', 'The quick brown fox')
<sre.SRE_Match object; span=(16, 19), match='fox'>
```

再次注意第一个命令无法产生一个匹配。这是由于单词 fox 并不是出现在字符串末尾，而这是\$字符要求的。

## 5. 任意字符

.字符是最后一个快捷方式字符。它代表任何单个字符。但是，它仅仅只能出现在方括号字符组以外。

考虑下面使用.字符的简单正则表达式：

```
>>> re.search(r'p.th.n', 'python 3')
<sre.SRE_Match object; span=(0, 6), match='python'>
>>> re.search(r'p..hon', 'python 3')
<sre.SRE_Match object; span=(0, 6), match='python'>
```

上述两个示例中，句号代替一个单独字符。在第一个示例中，正则表达式引擎在正则表达式中找到.字符。在字符串中，它看到字符 y，并且匹配然后继续到下一个字符(t 对比 t)。

在第二个示例中，发生同样的事情。每一个句点符号匹配一个并且仅仅匹配一个字符。这两个句点分别匹配 y 和 t，然后正则表达式引擎继续到下一个字符(这次，为 h 对比 h)。

注意，只有一个.无法匹配的字符，也就是换行(\n)。让.字符与换行符匹配是可能的，但这在本章后面进行讨论。

### 9.3.2 可选字符

目前为止，所有你看到的正则表达式都是在正则表达式中的字符与被搜索的字符串中的字符保持 1:1 的关系。

然而有时，一个字符或许是可选的。再次考虑有多种拼写方式单词的示例，但这次，利用单词是否包含某个字符区分拼写，比如“color”与“colour”，或者“honor”与“honour”。

可以在一个正则表达式中使用? 符号指定一个字符、字符组或其他基本单元可选，这意味着正则表达式引擎将会期望该字符出现零次或一次。

例如，可以使用正则表达式 honou?r 将英国拼写方式的“honour”与单词“honor”匹配。

```
>>> import re
>>> re.search(r'honou?r', 'He served with honor and distinction.')
<sre.SRE_Match object; span=(15, 20), match='honor'>
>>> re.search(r'honou?r', 'He served with honour and distinction.')
<sre.SRE_Match object; span=(15, 21), match='honour'>
```

在这两个示例中，正则表达式包含 4 个文字字符，hono。这些能够将 hono 与 honor 和 honour 匹配。正则表达式遇到的下一个字符是可选的 u。在第一个示例中，没有 u，但这没有问题，因为正则表达式将其标记为可选。在第二个示例中，u 存在，这也没问题。在这两个示例中，正则表达式寻找文本字符 r，并且找到该字符，因此完成匹配。

### 9.3.3 重复

目前为止，你仅仅学习了关于仅出现一次并且只有一次的字符(或字符组)，或者可选字符(出现 0 次或一次)。然而，有时你需要同样的字符或字符组重复出现。

你或许期望一个字符组连续重复出现一定数量的次数，比如在一个电话号码中。美国电话号码由国家号 1(通常忽略)，一个 3 个数字的区号，然后是 7 个数字的电话号码，电话号码的前三位与后四位通过一个连字符、句号或类似符号分开。

可以使用 {N} 指定一个标记必须重复给定次数，N 字符与标记应该重复的次数对应。

下面使用一个正则表达式识别一个 7 位数字的本地电话号码(暂时忽略国家号码与区域号码): [\d]{3}-[\d]{4}。

```
>>> re.search(r'[\d]{3}-[\d]{4}', '867-5309 / Jenny')
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

在本例中，正则表达式引擎开始寻找三个连续数字。然后发现他们(867)，移动到连字符。由于该连字符并不在一个字符组里，它并没有任何特殊含义，仅仅是与字面连字符匹配。然后正则表达式找到最后 4 个连续数字(5309)并返回匹配。

#### 1. 重复区间

有时，你或许并不知道标记应该重复的确切次数。电话号码可能包含的数字个数是静态的，但很多数字数据并不是按这种方式标准化。

例如，考虑信用卡安全码。在美国发行的信用卡在背面包含一个特殊的安全码，通常被称为“CVV 码”。大多数信用卡品牌使用 3 位安全码，你可以使用 [\d]{3} 匹配。然而，美国运通卡使用 4 位安全码([\d]{4})。

假使你希望能够匹配上述两种情况会怎么样？重复区间在这里就能够派上用场。这里的语法是 {M,N}，M 是下界而 N 是上界。

值得注意的是该边界是包含性的。如果你希望匹配 3 个数字或 4 个数字，正确的语法是 [\d]{3,4}。你或许被诱导(基于使用 Python 切片的经验)相信不包含上界(并且你应该使用的是 {3,5})。然而，正则表达式不是以这种方式工作。

```
>>> re.search(r'[\d]{3,4}', '0421')
<_sre.SRE_Match object; span=(0, 4), match='0421'>
>>> re.search(r'[\d]{3,4}', '615')
<_sre.SRE_Match object; span=(0, 3), match='615'>
```

在这两种情况中，正则表达式引擎找到一系列与所期望匹配的数字，并且返回一个匹配。

当给予匹配 3 个字符或 4 个字符的选择时，正则表达式如何决定是否是一个有效的匹配？答案是，在大多数情况下，正则表达式是“贪心”的，意味着尽可能多地匹配字符。在这个简单示例中，这意味着如果有 4 个数字，将会匹配 4 个数字。

偶尔，这种情况并不可取。通过在重复操作符之后放置一个? 字符，它会导致重复被

认为是“惰性”的，意味着引擎将会尽可能少地返回有效匹配。

```
>>> re.search(r'[\d]{3,4}?', '0421')
<sre.SRE_Match object; span=(0, 3), match='042'>
```

? 字符用于另一个目的的重复对于解析器并不会导致任何混淆，因为该字符出现在重复语法之后，而不是一个用于匹配的标记。

注意在这种情况下的? 字符并不是用于使得重复段可选。它仅仅是，如果给予选择匹配 3 个数字还是 4 个数字的机会，它仅仅选择匹配 3 个。

注意：注意用于使得标记可选的? 字符本质上正是{0,1}的别名。

## 2. 开闭区间

你还或许遇到一种情况是对于标记能够重复的次数并没有一个上界。例如，考虑一个传统的街道地址。通常以一个数字开头(暂时忽略例外情况，并断言总是以数字开头)，但数字可以是任意长度。8 位数字的街道号码并不是技术上无效的。

在上述情况下，你可以忽略上界，但保持, 字符从而指定上界为 $\infty$ 。例如，{1,}用于指定出现 1 次或多次，且没有上界。

```
>>> re.search(r'[\d]{1,}', '1600 Pennsylvania Ave.')
<sre.SRE_Match object; span=(0, 4), match='1600'>
```

如果你不指定一个下界，该语法同样生效，在这种情况下，下界假定为 0。

## 3. 速写

你可以使用两个速写字符指定常见的重复情况。可以使用+字符代替{1,}(一个或多个)。类似的，你可以使用\*代替{0, }(0 个或多个)。

因此，之前的示例可以使用+重写为如下所示：

```
>>> re.search(r'[\d]+', '1600 Pennsylvania Ave.')
<sre.SRE_Match object; span=(0, 4), match='1600'>
```

使用+与\*通常使得一个正则表达式更容易阅读，当能够使用这两种方式时，它们是更可取的方式。

## 9.4 分组

正则表达式提供了一个机制将表达式分成组。当使用分组时，除了获得整个匹配，你能够在匹配中选择每一个单独组。你可以在一个正则表达式中使用圆括号指定分组。

下面是一个简单的本地电话号码的示例。然而，这次，每组数字都是一个分组。

# www.itbaizhan.cn

```
>>> match = re.search(r'(\d{3})-(\d{4})', '867-5309', re.DOTALL)
>>> match
<sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

如前所述，可以对 match 对象使用 group 方法返回完整匹配。

```
>>> match.group()
'867-5309'
```

re 模块的 match 对象提供了一个方法，groups，用于返回一个对应每一个单个分组的元组。

```
>>> match.groups()
('867', '5309')
```

通过将正则表达式像这样分解为子组，你可以不仅仅快速获得匹配，而且获得匹配中的具体数据部分。

只获取单独分组也是可能的，通过给 group 方法传递一个对应你希望获得分组的参数（注意这里分组编号以 1 开始）。

```
>>> match.group(2)
'5309'
```

通过使用分组，你可以以多种不同形式接受一个电话号码而仅仅只取出重要的数据，也就是一个电话号码的实际数字。

```
>>> re.search(
...     r'(\+?1)?[ .-]?(\?(\d{3}))?[ .-]?(\d{3})[ .-]?(\d{4})',
...     '(213) 867-5309')
<sre.SRE_Match object; span=(0, 14), match='(213) 867-5309'>

>>> re.search(
...     r'(\+?1)?[ .-]?(\?(\d{3}))?[ .-]?(\d{3})[ .-]?(\d{4})',
...     '213-867-5309')
<sre.SRE_Match object; span=(0, 12), match='213-867-5309'>

>>> re.search(
...     r'(\+?1)?[ .-]?(\?(\d{3}))?[ .-]?(\d{3})[ .-]?(\d{4})',
...     '213.867.5309')
<sre.SRE_Match object; span=(0, 12), match='213.867.5309'>

>>> re.search(
...     r'(\+?1)?[ .-]?(\?(\d{3}))?[ .-]?(\d{3})[ .-]?(\d{4})',
...     '2138675309')
<sre.SRE_Match object; span=(0, 10), match='2138675309'>

>>> re.search(
...     r'(\+?1)?[ .-]?(\?(\d{3}))?[ .-]?(\d{3})[ .-]?(\d{4})', '+1'
...     '(213) 867-5309')
<sre.SRE_Match object; span=(0, 17), match='+1 (213) 867-5309'>
```

# 尚学堂·百战程序员

第III部分 数 据

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
>>> re.search(  
...     r'(\+?1)?[ .-]?\\(?((\\d){3})\\)?[ .-]?((\\d){3})[ .-]?((\\d){4})', '1  
...     (213) 867-5309')  
<_sre.SRE_Match object; span=(0, 16), match='1 (213) 867-5309'>  
  
>>> re.search(  
...     r'(\+?1)?[ .-]?\\(?((\\d){3})\\)?[ .-]?((\\d){3})[ .-]?((\\d){4})',  
...     '1-213-867-5309')  
<_sre.SRE_Match object; span=(0, 14), match='1-213-867-5309'>
```

相比于你已经遇到的正则表达式，该正则表达式略微复杂一点。考虑它自身的每个独特部分将会更容易解析。

第一节是`(+?1)?[ .-]?`。它用于寻找几乎你遇到的任何形式(+1 或 1，或可能为连字符)的美国国家号码。

第二节是`\(?((\d){3})\\)?[ .-]?`，用于抓取区号，接着是一个可选的连字符或空白。区号可能会以圆括号的形式提供(在美国电话号码中非常普遍)。

正则表达式的剩下部分是最后电话号码的 7 位数字，与之前你已经见过的一样。

无论电话号码如何格式化，正则表达式都能与之匹配。并且虽然完整匹配仍然基于提供的原始数据格式化，但分组保持一致。

```
>>> match = re.search(  
...     r'(\+?1)?[ .-]?\\(?((\\d){3})\\)?[ .-]?((\\d){3})[ .-]?((\\d){4})',  
...     '213-867-5309')  
>>> match.groups()  
(None, '213', '867', '5309')  
  
>>> match = re.search(  
...     r'(\+?1)?[ .-]?\\(?((\\d){3})\\)?[ .-]?((\\d){3})[ .-]?((\\d){4})',  
...     '+1 213-867-5309')  
>>> match.groups()  
('+1', '213', '867', '5309')
```

分组之间唯一的区别是基于对于国家码来说提供的是什么。如果忽略，那么它就不会被捕获，并且在它的位置提供 None。第二个到第四个分组都一致地包含 3 段(国内)电话号码。

## 9.4.1 零分组

示例一直使用 group 方法返回完整匹配，而不是一个单独分组。实际上，从一开始调用 group 方法获得完整匹配看上去是一个非常奇怪的命名法。

为什么它以这种方式生效？group 的目标实际上是从匹配中返回一个单独分组。它接受一个可选参数，也就是所需返回分组的号码。如果忽略参数(正如之前示例一直做的那样)，默认值为 0。

# www.itbaizhan.cn

在正则表达式中，分组的计数是基于它们在正则表达式中的位置，从 1 开始。

零分组是特殊的，对应完整的匹配。这就是为什么分组是以 1 开始。通过不提供参数调用 group，你请求的就是零分组，因此，获得完整匹配。

## 9.4.2 命名分组

除了有按位置编号的分组之外，Python 正则表达式引擎还提供一个命名分组的机制。该功能实际上最开始由 Python 正则表达式实现引入，虽然其他语言现在已经跟进。

一个命名分组的语法是在开始的“(”符号之后立刻添加?P<group\_name>。可以通过重写为(?P<first\_three>[\d]{3})-(?P<last\_four>[\d]{4})指定本地电话号码正则表达式使用命名分组。

```
>>> match = re.search(r'(?P<first_three>[\d]{3})-(?P<last_four>[\d]{4})',
...                      '867-5309')
>>> match
<sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

首先，注意命名分组还仍然是位置分组。仍然可以(如果选择这么做)以下面方式查找分组：

```
>>> match.groups()
('867', '5309')
>>> match.group(1)
'867'
```

使用命名分组开辟了查找一个分组的另外两种方式。第一种，分组名称可以作为一个字符串传递给 group 方法。

```
>>> match.group('first_three')
'867'
```

此外，match 对象提供了一个 groupdict 方法。该方法在大多数方面与 groups 方法类似，除了它返回的是一个字典而不是元组，字典键对应分组的名称。

```
>>> match.groupdict()
{'first_three': '867', 'last_four': '5309'}
```

值得注意的是 groupdict，与 groups 类似，并不会返回完整匹配；它仅仅返回子组。同时，如果混合命名分组与非命名分组，非命名分组不会出现在由 groupdict 返回的字典中。

```
>>> match = re.search(r'(?P<first_three>[\d]{3})-([\d]{4})', '867-5309')
>>> match.groups()
('867', '5309')
>>> match.groupdict()
{'first_three': '867'}
```

在本例中，只有第一组(命名为 first\_three)是命名分组，第二组只是一个编码分组。因此，当调用 groups 时，这两个分组都在元组中返回。但是，当调用 groupdict 时，只有 first\_three

分组被包含在结果中。

从维护角度来说，命名分组十分有价值。你或许会在后面代码中引用一个分组。如果你首选使用命名分组，向正则表达式中添加一个新分组从而使得在后面代码中不需要更新分组号，这是由于已经存在的名称保持不变。

### 9.4.3 引用已经存在的分组

正则表达式还提供了一种引用一个之前匹配分组的机制。有些时候，你或许会寻找同样一个子匹配，该匹配会接下来再次出现。

例如，如果你尝试解析一段 XML 代码，你或许想要悲观地寻找任意有效的开始标记，比如<([\w\_-]+)>。然而，你想要确保同样的结束标记存在。

重复使用该模式两次并不足以解决该问题。一方面，它会正确匹配你希望的模式。

```
>>> re.search(r'<([\w_-]+)>stuff</([\w_-]+)>', '<foo>stuff</foo>')
<sre.SRE_Match object; span=(0, 16), match='<foo>stuff</foo>'>
```

另一方面，它还会匹配实际上不应该匹配的模式。

```
>>> match = re.search(r'<([\w_-]+)>stuff</([\w_-]+)>', '<foo>stuff</bar>')
>>> match
<sre.SRE_Match object; span=(0, 16), match='<foo>stuff</bar>'>
>>> match.group(1)
'foo'
>>> match.group(2)
'bar'
```

这里，正则表达式引擎正确发现<foo>是一个开始 XML 标记，匹配它，然后将文本 foo 赋值给子组。然后匹配文字字符 stuff，然后去匹配 XML 结束标记。

此时，你直觉上会希望匹配失败，因为 XML 结束标记是</bar>，与开始标记的<foo> 不相同。

但是，正则表达式并不会这么做。它仅仅被通知匹配</and>这两个包装字符以及这两个字符之间的单词字符。由于 bar 满足需求，引擎会匹配它，将其赋值给第二个子组，并返回一个匹配。

此时你真正希望的是正则表达式引擎需要与第一个分组使用同样一个子匹配。这应该使得字符串<foo>stuff</foo>匹配，但字符串<foo>stuff</bar>不匹配。

正则表达式引擎提供一种完成这种工作的方式——使用回溯引用。回溯引用引用在一个正则表达式中前面匹配的分组，并导致正则表达式解析器期望同一个匹配文本再次出现。

可以使用\N 回溯引用编号分组。因此，\1 将会匹配第一个分组，\2 匹配第二个分组等。该语法能够最多匹配前 99 个分组。

考虑下面使用回溯引用的 XML 正则表达式：

```
>>> match = re.search(r'<([\w_-]+)>stuff</\1>', '<foo>stuff</foo>')
>>> match
```

# www.itbaizhan.cn

```
<_sre.SRE_Match object; span=(0, 16), match='<foo>stuff</foo>'>
>>> match.groups()
('foo',)
```

注意，在此目前只有一个子组。在之前的示例中，有两个子组，它们都包含文本 foo。然而，在本例中，回溯引用取代了第二个分组。

然而，一个更重要的区别是该正则表达式所不能匹配的标记。

```
>>> re.search(r'<([\w_-]+)>stuff</\1>', '<foo>stuff</bar>')
>>>
```

在本例中，正则表达式引擎成功匹配 XML 结束标签。但是，由于 bar 与 foo 并不是同样的文本，匹配失败。

**警告：**不应该实际使用自定义正则表达式来解析 XML。而是使用 lxml 或类似工具。对于解析 HTML，使用类似 BeautifulSoup 的包。本示例的目的仅仅是解释这类回溯引用的工作机制。

## 9.5 先行断言

之前，你学到关于取反字符组，这使得能够匹配在字符组之外的任意字符。正如之前提到，该方法使得由取反字符组匹配的字符成为匹配的一部分，并且它将完全无法匹配不存在的任意字符。

然而，有一种机制能够基于之后内容是否存在接受或拒绝一个匹配，而不需要接下来的内容作为匹配的一部分。它被称为先行断言。

之前一个取反字符组的示例是 n[^e]——一个 n 接下来是一个不是 e 的字符。这会匹配在 final 中的 na，但无法与 jasmine 中的字符匹配，也无法与 Python 中的字符匹配。

一个使用取反先行断言的正则表达式采用语法 n(?![^e])。

```
>>> re.search(r'n(?![^e])', 'final')
<_sre.SRE_Match object; span=(2, 3), match='n'>
>>> re.search(r'n(?![^e])', 'jasmine')
>>> re.search(r'n(?![^e])', 'Python')
<_sre.SRE_Match object; span=(5, 6), match='n'>
```

这些结果与当使用一个取反字符组时有所不同。在第一个示例中，使用单词 final，正则表达式再次匹配，但匹配本身不同。取反字符组使得字符 a 作为匹配的一部分，而取反先行断言并不这样做，返回的匹配仅仅是字符 n。

第二个结果最相似，jasmine 中的 n 与正则表达式中的字符 n 匹配。然而，由于 n 接下来是一个 e，它不符合条件，匹配失败。

最终结果区别最大，由于该匹配实际上成功，而并没有使用一个取反字符组。正则表

# 尚学堂·百战程序员

第三部分 数 据

[www.itbaizhan.cn](http://www.itbaizhan.cn)

达式引擎匹配在 Python 中的 n。然后到达字符串的结尾。由于 n 后面并没有跟着一个 e，匹配成功并返回。

值得注意的是虽然这看上去像分组语法，在本例中，分组却无法解决问题。

```
>>> match = re.search(r'n(?!e)', 'final')
>>> match
<_sre.SRE_Match object; span=(2, 3), match='n'>
>>> match.groups()
()
```

正则表达式引擎还支持一种不同类型的先行断言，称为正向先行断言。这需要匹配之后紧接着是指定字符或问号之后的字符，但仍然不会使得这些字符作为匹配的一部分。

正向先行断言的语法仅仅需要将字符!替换为=。考虑该正则表达式：

```
>>> re.search(r'n(?=e)', 'jasmine')
<_sre.SRE_Match object; span=(5, 6), match='n'>
```

在本例中，正则表达式引擎匹配单词 jasmine 中的 n。在这么做之后，它验证接下来的字符是 e，正如正则表达式需要的那样。因为这样，匹配完成并返回。与之前相同，先行断言并没有创建分组。

没有 e，匹配失败，如下所示。

```
>>> re.search(r'n(?=e)', 'jasmin')
>>>
```

在本例中，正则表达式引擎再次匹配 n，但由于 n 之后并没有跟着一个 e，匹配失败。

## 9.6 标记

有时，你需要轻微改变正则表达式引擎的行为。正则表达式引擎在大多数包括 Python 在内的语言中，提供了少量用于修改整个表达式行为的标记。

在使用 re.search 或类似函数时，Python 引擎提供了几种可以发送到一个正则表达式的标记。在 re.search 的例子中，它接受第三个参数作为标记。

### 9.6.1 不区分大小写

最简单直接的标记是 re.IGNORECASE，它会导致正则表达式变为不区分大小写。

```
>>> re.search(r'python', 'PYTHON IS AWESOME', re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 6), match='PYTHON'>
```

当使用 re.IGNORECASE 时，仍然会返回字符串中发现的匹配，而不是正则表达式中发现的匹配。

re.IGNORECASE 也是 re.I 的别名。

## 9.6.2 ASCII 与 Unicode

在 Python 2 与 Python 3 中，一些字符快捷方式的工作机制存在区别。例如，在 Python 3 中\w 匹配几乎所有语言的单词，而不是仅仅匹配拉丁字母。

re 模块提供了能够让 Python 2 遵循 Python 3 行为的标记，以及让 Python 3 遵循 Python 2 行为的标记。

re.UNICODE(别名为 re.U)标记强制正则表达式引擎遵循 Python 3 的行为。在 Python 2 与 Python 3 中都定义了该标记，因此被设计运行在这两个平台中的代码可以安全使用该标记。注意，如果在 Python 3 中你尝试对字节字符串使用 re.U，解析器会引发一个错误。

re.ASCII(别名为 re.A)标记强制正则表达式遵循 Python 2 的行为。与 re.UNICODE 不同，re.ASCII 标记在 Python 2 中不可用。如果运行在 Python 2 与 Python 3 的代码需要使用 re.ASCII，则选择使用合适的字符组，或是在应用标记之前进行版本检测。

## 9.6.3 点匹配换行符

re.DOTALL 标记(别名为 re.S，该别名与 Perl 或其他地方使用的术语一致)导致.字符除了匹配其他字符之外，还匹配换行符。

```
>>> re.search(r'.+', 'foo\nbar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search(r'.+', 'foo\nbar', re.DOTALL)
<_sre.SRE_Match object; span=(0, 7), match='foo\nbar'>
```

在第一个命令中，正则表达式引擎必须匹配一个或多个任意字符。它匹配 foo，然后遇到换行符并停止，因为.字符正常情况下不会匹配换行符。

然而，在第二个命令中，当传递 re.DOTALL 时，.字符能够匹配的字符就包括了换行符。因此，正则表达式引擎(保持贪心)继续执行直到到达字符串结尾，整个字符串作为匹配结果返回。

## 9.6.4 多行模式

re.MULTILINE 标记(别名为 re.M)导致仅能够匹配字符串开始与结束的^与\$字符可以匹配字符串内任意行的开始与结束。

```
>>> re.search(r'^bar', 'foo\nbar')
>>> re.search(r'^bar', 'foo\nbar', re.MULTILINE)
<_sre.SRE_Match object; span=(4, 7), match='bar'>
```

在第一个命令中，^字符能够匹配字符串的开始部分。因此，单词 bar 并不匹配，因为它并不是字符串的开始部分。

但是，在第二个命令中，使用了 re.MULTILINE 标记。因此，^字符仅仅只需要一行的开始。由于换行符在 bar 之前，因此能够匹配 bar 并返回。

## 9.6.5 详细模式

re.VERBOSE 标记(别名为 re.X)允许复杂的正则表达式以更容易阅读的方式表示。

该标记做两件事。首先，它导致所有的空白(除了在字符组中)被忽略，包括换行符。其次，它将#字符(同样，除非在字符组内)当作注释字符。

这使得正则表达式能够有简单的注释，当正则表达式越来越复杂时非常有价值。下面的两个命令等价：

```
>>> re.search(r'(?P<first_three>[\d]{3})-(?P<last_four>[\d]{4})', '867-5309')
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
>>> re.search(r"""(?P<first_three>[\d]{3})      # The first three digits
...           -                                # A literal hyphen
...           (?P<last_four>[\d]{4})      # The last four digits
...           """", '867-5309', re.VERBOSE)
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

## 9.6.6 调试模式

re.DEBUG 标记(没有别名)在编译正则表达式时将一些调试信息输出到 sys.stderr。

```
>>> re.search(r'(?P<first_three>[\d]{3})-(?P<last_four>[\d]{4})',
...             '867-5309', re.DEBUG)
subpattern 1
max_repeat 3 3
in
    category category_digit
literal 45
subpattern 2
max_repeat 4 4
in
    category category_digit
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

## 9.6.7 使用多个标记

偶尔，你或许需要同时使用多个标记。为了完成这点，使用|(位或)操作符。例如，如果需要 re.DOTALL 与 re.MULTILINE 标记，正确的语法是 re.DOTALL | re.MULTILINE 或 re.S | re.M。

## 9.6.8 内联标记

通过在正则表达式开始部分加上一个特殊语法，在一个正则表达式内使用标记也是可能的。使用简易形式的标记，看上去如下所示：

```
>>> re.search('(?i)FOO', 'foo').group()
```

'foo'

注意开头的(?i)。这等同于使用 re.IGNORECASE 标记。然而，该语法不如显式发送标记。同时，标记的完整形式将不生效。(?ignorecase)无效并且会引发异常。

## 9.7 替换

正则表达式引擎并不仅仅局限于识别一个模式是否在字符串中存在。它还能够执行字符串替换，基于在原始字符串中的分组返回一个新字符串。

在 Python 中的替换方法是 re.sub。它接受 3 个参数：正则表达式、用于替换的字符串、被搜索的原始字符串。只有实际匹配被替换，因此如果并没有匹配，re.sub 最终不执行任何操作。

re.sub 允许从被替换的字符串中的正则表达式模式使用同样的回溯引用。考虑一个从电话号码中剥离无关格式数据的任务。

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'\2\3\4',
...     '213-867-5309')
'2138675309'
```

因为该正则表达式匹配几乎所有的电话号码，并仅将电话号码的实际数字分组，无论原始号码如何被格式化，都能得到相同的数据。

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'\2\3\4',
...     '213.867.5309')
'2138675309'
```

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'\2\3\4',
...     '2138675309')
'2138675309'
```

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'(213) 867-5309')
'2138675309'
```

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'\2\3\4',
...     '(213) 867-5309')
'1 (213) 867-5309'
```

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?( [\d]{3})[ .-]?( [\d]{4})',
...     r'\2\3\4',
...     '+1 213-867-5309')
```

# 尚学堂·百战程序员

第III部分 数 据

www.itbaizhan.cn

'2138675309'

被替换的字符串不仅限于使用对于字符串的回溯引用；其他字符按字面被解释。因此，`re.sub` 也可以被用于格式化。例如，假如你希望以一致的方式显示而不是存储一个电话号码时会发生什么？`re.sub` 能够处理这一点，如下所示：

```
>>> re.sub(r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
...         r'(\2) \3-\4',
...         '+1 213-867-5309')
'(213) 867-5309'
```

除了替换字符串增加了圆括号、空格与连字符之外，这里的一切都与之前的示例一致。因此，结果也一样。

## 9.8 已编译的正则表达式

Python 正则表达式实现的一个最终功能为：已编译的正则表达式。`re` 模块包含了一个函数：`compile`，它返回一个已编译的正则表达式对象，该对象之后可以被复用。

`re` 模块缓存它即席编译的正则表达式，因此在大多数情况下，使用 `compile` 并没有很大的性能优势。但是，用于传递正则表达式对象却十分有用。

`re.compile` 函数返回一个正则表达式对象，该对象内正则表达式以数据的形式存在。这些对象有它们自己的 `search` 和 `sub` 方法，这些方法省略了第一个参数(也就是正则表达式自身)。

```
>>> regex = re.compile(
...     r'(\+?1)?[ .-]?\((?([\d]{3})\)\)?[ .-]?([\d]{3})[ .-]?([\d]{4})'
... )
>>> regex.search('213-867-5309')
<_sre.SRE_Match object; span=(0, 12), match='213-867-5309'>
>>> regex.sub(r'(\2) \3-\4', '+1 213.867.5309')
'(213) 867-5309'
```

同时，使用 `re.compile` 有另一个好处。正则表达式对象的 `search` 方法允许使用在 `re.search` 中不可用的两个额外参数。这两个参数分别是被搜索字符串的开始与结束位置，它们可用来让你减少对部分字符串的考虑。

```
>>> regex = re.compile('[\d]+')
>>> regex.search('1 mile is equal to 5280 feet.')
<_sre.SRE_Match object; span=(0, 1), match='1'>
>>> regex.search('1 mile is equal to 5280 feet.', pos=2)
<_sre.SRE_Match object; span=(19, 23), match='5280'>
```

发送的这两个值在返回的匹配对象的 `pos` 与 `endpos` 属性中可用。

## 9.9 小结

正则表达式是用于查找、解析与验证数据非常有效的工具。对于从未使用过正则表达式的人来说，它看上去很吓人，但如果按步骤学习，就能够掌握。

此外，掌握正则表达式将使得你能够在不需要模式匹配算法的情况下执行更加复杂的解析与格式化任务。

但是，在非必要时请慎用正则表达式。有时，使用几行直接进行字符串比对的代码更加直接。就像任何工具那样，只有它们是合适的解决方案时，才使用正则表达式，而不是有更简单方法存在时不选择简单方法。

类似的，请记住正则表达式通常不适用于解析非常复杂的结构。如果你正在解析一个复杂的文档格式，可能更应该去寻找一个帮助你处理这种格式的类库。

第 10 章将介绍如何在 Python 中测试应用程序。

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



## 第IV部分

### 其他高级主题

---

- 第 10 章 Python 2 与 Python 3
- 第 11 章 单元测试
- 第 12 章 CLI 工具
- 第 13 章 asyncio 模块
- 第 14 章 代码风格

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 第 10 章

## Python 2 与 Python 3

在本书其他章节(尤其是第 5 章“元类”与第 8 章“字符串与 Unicode”)中, 你已经学习了 Python 2 与 Python 3 处理某些事情所存在的区别。

实际上, Python 3 是对 Python 编程语言的重大升级。遍观 Python 历史, Python 非常强调兼容性, 避免变更可能导致大量现有代码无法使用。这并不意味着该语言从不淘汰任何语法, 当然, 同样也需要重点关注向后兼容性。

Python 3.0 是不遵循该规则的特例。正如任何复杂语言或系统的开发人员那样, Python 的开发人员做了一些后来被认为是错误的决定。因此, Python 3.0 可以被认为是以牺牲向后兼容性修复这些错误的一种努力。

由于现有的 Python 程序已经非常普遍, 因此官方在一定时间内同时支持 Python 2 与 Python 3 ——从而给出一定时间使得社区从旧版本迁移到新版本。Python 2.6 与 Python 3.0 基本上同时发布, Python 2.7 大概是在一年半之后发布的(大概在 Python 3.1 发布之后一整年)。

当前, Python 2.7 甚至是 Python 2.6 依然被普遍使用。因此, 理解 Python 2 与 Python 3 的区别以及掌握这两个版本之间的切换就非常重要。

本章讨论 Python 2 与 Python 3 的区别, 并讨论在双生态系统之间的切换策略。

### 10.1 跨版本兼容性策略

Python 3 引入一系列向后不兼容的变更(以及很多向后兼容变更, 但本章并不会过多关注这部分内容)。大多数向后不兼容变更是聚焦于移除二义性, 确保只有一种清晰的方法可用来解决问题、要么是更新语言解决怪异的地方, 或是使得 Python 行为更加现代。

由于 Python 3 并不打算成为一个向后兼容的发布版本, 因此不要预期 Python 2 代码不

经任何修改就可以在 Python 3 上执行。实际上，现有的很多 Python 2 模块无法在 Python 3 上执行，或是执行会产生不同结果，甚至有些代码会产生语法错误。

也就是说，可以使用多种策略编写代码，从而使得代码可以在不同生态系统上执行。

## 10.1.1 \_\_future\_\_ 模块

在某些情况下，有用的 Python 3 行为可以“向后导入”到 Python 2.6 和 Python 2.7 中。你可以使用 `__future__` 模块(已经在 Python 语言中存在一些时间了)实现这点。

`__future__` 模块提供了向 Python 语言逐渐引入一项功能的机制，使得该特性最开始可以可选引入，然后最终变成语言的默认行为。

例如，`yield` 以及之后的 `with` 都是使用该模块导入，最终这两个特性都作为关键字加入到 Python 中。由于将一个新关键字添加到语言将会使得现有使用上述关键字作为变量名称的代码遭到破坏，这些关键字需要被逐渐引入。对于一个 Python 版本，可以通过使用诸如下面的语句可选地加入新关键字：

```
from __future__ import with_statement
```

至于 `with`，该语句在 Python 2.5 中可用。通过上面的代码使得 `with` 和 `as` 成为关键字。如果执行的代码将上述单词作为标识符，会收到一个警告。然后，在 Python 2.6 中，`with` 与 `as` 变成一个常规关键字。然而，即使在这个时候，importing `with_statement` from `__future__` 依然有效(只是不再可选)。这使得使用 `with` 的代码可以在 Python 2.5 以及之后的版本中运行。

同样的原则适用于在 Python 3 中引入的很多功能。可以在 Python 2.6 与 Python 2.7 中可选地引入某些或所有的 Python 3 功能，这使得编写在这两个版本下运行的代码更加容易管理。

如在本章展示 Python 2 与 Python 3 中不同的特定行为一样，你也可以学到这些区别并在 Python 2 中使用该方法选择使用 Python 3 中的特性。

## 10.1.2 2to3

当 Python 3 第一次发布时，处理 Python 2 与 Python 3 中共享源码的推荐机制是使用一个名称为 `2to3` 的工具。

`2to3` 是与当前 Python 版本一起发行的命令行工具。该工具的目标是尝试检测为 Python 2 编写的模块，并为转换为 Python 3 模块提供修补建议，甚至是自动转换模块。类似的工具 `3to2`，也可用(在 PyPI 上)于完成相反的工作。

考虑下面针对 `foo.py` 的转换，这是一个非常简单的单行 Python 2 模块：

```
$ cat foo.py
print 'foo'
```

该模块在 Python 2 中有效但在 Python 3 中无效，这是由于在 Python 3 中 `print` 是一个函数，而不是一个语句(本章后面会详细解释)，在 Python 2 中生效的代码在 Python 3 中报错。

www.itbaizhan.cn

```
$ python2.7 foo.py
foo
$ python3.4 foo.py
  File "foo.py", line 1
    print 'foo'
               ^
SyntaxError: invalid syntax
```

这是一个很直观(向后不兼容)的变更，遍历代码库手动修改该段代码将会是一项艰巨的任务，这就是 2to3 可以处理的问题。通过对该文件执行 2to3，可以获得关于 2to3 认为必须做什么的一些信息。

```
$ 2to3 foo.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored foo.py
--- foo.py (original)
+++ foo.py (refactored)
@@ -1 +1 @@
-print 'foo'
+print('foo')
RefactoringTool: Files that need to be modified:
RefactoringTool: foo.py
```

默认情况下，2to3 实际上不会做任何工作。它只是告诉你必须完成的工作并提供修补建议，它已经发现在第一行的 print 语句并建议变更为函数，但它并不会实际去修改文件并变更代码(在下一小节讨论)。

```
$ cat foo.py
print 'foo'
```

### 写入变更

然而，2to3 可以写入它能够确定的变更。实现这一点最简单的办法是添加-w 标记，该标记会即席重写文件(再次注意这将会即席重写文件，因此你应该理解你在做什么)。

```
$ 2to3 -w foo.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored foo.py
--- foo.py      (original)
+++ foo.py      (refactored)
@@ -1 +1 @@
-print 'foo'
+print('foo')
RefactoringTool: Files that were modified:
```

RefactoringTool: foo.py

果然, foo.py 文件已经在磁盘中被修改, 现在在 Python 3 中运行它就不会再引发报错。

```
$ cat foo.py
print('foo')
$ python3.4 foo.py
Foo
```

### 10.1.3 限制

遗憾的是, 2to3 工具并不能处理所有可能的情况, 因此仅仅是对一个模块执行 2to3 并不能确保任何在 Python 2 中有效的模块会神奇地变为一个有效的 Python 3 模块。

2to3 底层的工作机制是包含很多 fixer, fixer 是在特定 Python 2 代码与其等价 Python 3 代码之间转换层的一个术语。例如, 有一个名称为 print 的 fixer 负责处理由 print 语句到 print 函数的转换。甚至可以启用或禁用特定的 fixer(分别通过--fix 和--nofix)。

另一个 2to3 更基本的限制是使用 2to3 需要分别为 Python 2 与 Python 3 维护两个代码库。官方的建议是只有在编写 Python 2 代码并将其转换为 Python 3 代码部署时才使用 2to3。在实际应用中, 这很让人受挫, 且在大多数大项目中不可行。

有一种更好的办法。

### 10.1.4 six

six 是一个由 Benjamin Peterson 编写的 Python 模块, 用于在 Python 2 与 Python 3 之间提供唯一来源兼容性。在 2to3 中, 为 Python 2 编写代码, 然后程序执行并生成对应的 Python 3 代码。然而, six 遵循另一种逻辑。使用 six, 用 Python 3 语法编写一个单独模块可以正确执行在 Python 2.6 和 Python 2.7 下。

相比于 2to3, 该方法有几种优势, 但最重要的区别是只需要维护一份代码。同样的代码可以在两种环境中执行。除此之外, six 以单独模块的形式分发, 这使得在需要时无须依靠依赖项管理器就可以简单包含在代码内。

six 从根本上完成的工作是为 Python 2 与 Python 3 之间变更的元素提供唯一接口。例如, 在第 8 章中学到 Python 2 的 unicode 类与 Python 3 的 str 类是同一个类。six 模块提供了 six.text\_type, 用于在这两个环境中匹配正确的类。

例如, 下面两行代码在 Python 3 中相同:

```
>>> str('foo')
>>> six.text_type('foo')
```

另外, 下面两行代码在 Python 2 中相同:

```
>>> unicode('foo')
>>> six.text_type('foo')
```

six 的关键限制是通常情况下你的程序只有在无须支持 Python 2.6 之前版本时，six 才是一个可行方案。虽然 six 本身可以在更早的 Python 版本下执行，但在 Python 2.5 以及更老版本中无法通过 `_future_` 引入一些 Python 3 功能意味着很难确保行为的一致性。也就是说，如果你能确保所使用的功能在老版本的 Python 中生效，six 也通常会生效。

好消息是，如果你正在阅读本书，很难想象你还真正需要支持 2008 年发布的 Python 2.6 与之前的版本，该版本现在几乎被完全舍弃。所有现代 Linux 分发至少是 Python 2.6 以上，并且这还是很多年前的老黄历。如果使用的是 Windows，你很可能需要自己安装 Python，因此不太可能需要老版本的 Python。

six 现在已经是大多数人针对编写需要执行在 Python 2 与 Python 3 环境下代码的推荐机制。随着本章探讨在 Python 2 与 Python 3 之间的区别，你将会学到通过一个唯一接口使得在两个环境中获得相同结果的 six 语法。如果你正在编写需要在 Python 2 与 Python 3 中执行的代码，这或许是你希望使用的方式。

## 10.2 Python 3 中的变更

Python 2 与 Python 3 之间存在很多变更。其中一些变更非常巨大，而其他一些变更仅仅只包含类似重命名模块一样简单。

### 10.2.1 字符串与 Unicode

或许 Python 3 最彻底的变更是字符串类型为 Unicode 而不再是 ASCII，并且在程序中接收的大多数字符串通常都是 Unicode。

该变更非常巨大，以至于本书在第 8 章中使用大量篇幅详细介绍 Python 处理文本数据的主题。下面是一个快速复习。

在 Python 2 中，字符串默认是字节字符串，而在 Python 3 中是 Unicode 字符串。Python 3 行为可以通过 `from __future__ import unicode_literals` 导入到 Python 2 中，如果你正在编写需要在两种环境下运行的代码，绝对应该使用这种方式。

同时，字节字符串与文本字符串的名称不同。在 Python 2 中，`str` 类用于表示字节字符串，`unicode` 类用于表示文本字符串。在 Python 3 中，则变为 `bytes` 与 `str`。这意味着名称为 `str` 的类名称在两个版本中都存在，但并不是一回事。`six` 模块给它们赋予的别名分别为 `six.binary_type` 与 `six.text_type`。

注意：更多信息，请参阅第 8 章。

### 10.2.2 Print 函数

如之前的示例那样，Python 改变了 `print` 的工作机制。在 Python 2 中，`print` 是一个特

殊语句，如下所示：

```
print 'The quick brown fox jumped over the lazy dogs.'
```

默认情况下，print 会写到 sys.stdout 并在字符串结尾附加\n。然而，print 可以通过使用特殊语法>>打印到其他地方。

```
import sys
print >> sys.stderr, 'The quick brown fox jumped over the lazy dogs.'
```

在 Python 3 中，print 变得更加正常。首先且最重要的是，它现在是一个函数，这意味着可以使用括号像调用函数那样调用 print。

```
print('The quick brown fox jumped over the lazy dogs.')
```

仍然可以输出到非 sys.stdout 的地方。Python 3 print 函数接受一个名为 file 的关键字参数(默认值为 sys.stdout)，用于处理这种情况：

```
import sys
print('The quick brown fox jumped over the lazy dogs.', file=sys.stderr)
```

此外，新的 print 函数更加灵活，这是由于可以通过使用 end 关键字参数改变默认在字符串结尾附加\n 的行为。

这仍然会输出到 sys.stdout 中，但不会在输出之前在字符串的末尾附加\n。

Python 3 print 函数在 Python 2.6 和 Python 2.7 的 \_\_future\_\_ 模块中可用。

```
from __future__ import print_function
```

**注意：**如果使用更早版本的 Python，six 通过 six.print\_ 提供了同样的功能(注意结尾下划线用于不和 Python 关键字混淆)。参数与 print 函数的参数完全一致。作为提醒，通常情况下不要尝试在 Python 2.5 以及更低版本中尝试用一份代码同时兼容 Python 2 与 Python 3 两个平台。

### 10.2.3 除法

在 Python 2 中，除法(/)操作符作用于两个整型数，会返回一个 int。这一直是 Python 2 中导致混淆的一个根源，因为大多数人本能会预期两个整型相除会在适当时返回一个浮点数。考虑下面的 Python 2 代码：

```
>>> 4 / 2
2
>>> 5 / 2
2
```

5 除以 2 将返回 2 违反直觉。这种结果是由于这是一个整型除法。解释器做除法是，

获得正确结果 2.5，然后向下取整获得一个整型从而保持类型一致性。然而，这通常不是你在动态语言中真正希望的结果。

避免这一点的方法是使除数或被除数为 float 类型。

```
>>> 5.0 / 2  
2.5
```

Python 3 通过将整型除法总是返回 float 修复了该行为，这也是在动态语言中你通常希望的结果。

```
>>> 4 / 2  
2.0  
>>> 5 / 2  
2.5
```

如果你希望从一个除法操作中获得整型结果，可使用“向下取整除法”操作符，//，该操作符无论提供任何类型的参数，总是返回一个整型。

```
>>> 4 // 2  
2  
>>> 5 // 2  
2
```

Python 3 的行为更加可取，但向后不兼容。如果编写需要在两个环境下执行的代码，`_future_` 模块将再次成为你的朋友。可以通过下面的代码选择在 Python 2.6 和 Python 2.7 中使用 Python 3 的行为：

```
from __future__ import division
```

这也是单一代码跨平台的推荐机制。

#### 10.2.4 绝对与相对导入

在 Python 模块中引用包的主要方式是通过导入。然而，当发起 `import foo` 时实际会发生什么呢？这要视情况而定。

在 Python 2 中，解释器尝试(在标准库之后)的第一件事是相对导入。这意味着它会在导入模块的同一个目录中寻找名称为 `foo.py`(或者 `foo/-init-.py`)的模块。如果找到该模块，则完成；它会返回模块并使 `foo` 命名空间下的属性可用。

如果解释器找不到这样一个文件(目前为止是最常见的情况)，它会开始在 `sys.path` 中的所有目录下寻找一个匹配的模块。在正常情况下，这会包含所有已安装的 Python 包。这种类型的导入被称为绝对导入。

该行为可能导致问题。例如，仅仅是在一个目录中添加重复命名的模块会导致该目录中的其他模块失效，因为突然开始执行相对导入而不是绝对导入。

Python 3 仅仅通过尽可能地移除相对导入改变该行为。所有的导入都是绝对导入。如

果想要一个相对导入(某些情况下更好), 则必须使用特殊语法显式要求, 也就是使用英文句号作为开头。

```
import .foo
```

这会告诉解释器导入一个名称为 foo 的模块, 该模块与当前模块位于同一目录下。在这种情况下, 只会尝试相对导入(不会从标准库中导入, 也不会从 sys.path 中的目录下导入)。解释器还提供了一个..语法, 用于定位目录树中的上一层。

这里的 Python 3 行为更加安全并且是一种更加明显的方式, 但这会破坏向后兼容性。如果正在维护一个在 Python 2 和 Python 3 下执行的程序或分发版本, 则可以通过 \_\_future\_\_ 模块选择 Python 3 的行为, 如下所示:

```
from __future__ import absolute_import
```

这会导致你的模块使用 Python 3 的导入行为。除非显式使用相对导入语法, 否则只会从标准库或已安装的模块位置进行导入。

## 10.2.5 “老式风格”类的移除

Python 2.2 引入了在当时被称为新式类的部分。本质上, 这些是修复 Python 类特定继承问题(尤其是, 在多重继承下方法解析的顺序被破坏)的一种尝试, 统一数据模型, 并引入一些新功能(比如 super)。

为了与老版本 Python 的保持向后兼容性, 解释器会自主选择。在 Python 2 中的类默认是老式风格。

```
>>> class Foo:  
...     pass  
...  
>>> type(Foo)  
<type 'classobj'>
```

可以通过显式继承任何一个新式风格类来创建一个新式风格的类, 最值得注意的是 object, 它是新式风格类层级树的最高层。

```
>>> class Foo(object):  
...     pass  
...  
>>> type(Foo)  
<type 'type'>
```

在 Python 3 中, 旧式风格类被完全移除。少数在 Python 2 标准库中存在的旧式风格类全部被转换为新式风格。仍然允许显式继承 object 类, 但不再必须。

你或许会注意到本书示例(包括那些明显说明是 Python 3 的代码)全部显式继承 object。如果编写的代码只在 Python 3 下执行, 就不需要这么做。然而, 如果编写的代码既需要在 Python 2 下也需要在 Python 3 下执行, 你仅仅需要继续像在 Python 2 中那样显式继承 object。

类。这在 Python 3 中仍然有效，意味着无论在哪种环境下，这些类都是新式类。

如果检测一个变量是否为类，可以通过 `six.class_types`。在 Python 2 中，`six.class_type` 是一个包含 `type` 与 `classobj` 的元组，而在 Python 3 中该元组仅包含 `type`。

## 10.2.6 元类语法

Python 3 还变更了给类赋值一个自定义元类的语法。在 Python 2 中，给一个类赋值一个自定义元类使用 `_metaclass_` 属性。

```
class Foo(object):
    __metaclass__ = FooMeta
```

在 Python 3 中，元类成为类声明自身的一部分。

```
class Foo(object, metaclass=FooMeta):
    pass
```

这两种语法不兼容。无法在 Python 2 的类声明中使用 `metaclass` 作为关键字，而在 Python 3 中 `_metaclass_` 属性不生效。

`six` 库为该问题提供了一个解决方案。它提供了两种机制(`six.with_metaclass` 与 `six.add_metaclass`)，用于在创建类时给类赋一个元类。

### 1. `six.with_metaclass`

`six.with_metaclass` 函数接受需要的元类以及所有子类作为参数，并返回一个新类继承的存根类。语法上，使用方式如下所示：

```
class Foo(six.with_metaclass(FooMeta, object)):
    pass
```

`six` 底层的机制是创建一个继承自 `object` 的空类，并使用 `FooMeta` 作为元类。它返回该类并作为 `Foo` 的唯一继承类。这使得 `Foo` 有了 `FooMeta` 元类(无论是在 Python 2 还是 Python 3 下)以及适当的父类，但底层添加了一个没用的额外基类(存根类)。

可以通过查看新类的方法解析顺序来实际观察该机制。

```
>>> import six
>>>
>>> class FooMeta(type):
...     pass
...
>>> class Foo(six.with_metaclass(FooMeta, object)):
...     pass
...
>>> Foo.__mro__
(<class '__main__.Foo'>, <class 'six.NewBase'>, <type 'object'>)
```

注意在方法解析顺序中的中间类：`six.NewBase`。这就是 `six` 创建的存根类。它继承于

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

object(在调用 six.with\_metaclass 时通知)。如果你查看它,你将会看到这就是 FooMeta 元类被赋值的地方。

```
>>> NewBase = Foo.__mro__[1]
>>> NewBase
<class 'six.NewBase'>
>>> type(NewBase)
<class '__main__.FooMeta'>
```

确实,查看 Foo 类也显示了元类为 FooMeta,这是由于它继承自 NewBase,该类也是一个 FooMeta 类。

```
>>> type(Foo)
<class '__main__.FooMeta'>
```

## 2. six.add\_metaclass

six 模块还提供了 add\_metaclass,该方法实现同一个目标,但方式略微不同。第一个区别是 API.add\_metaclass 作为类装饰器使用。

```
@six.add_metaclass(FooMeta)
class Foo(object):
    pass
```

这里的结果本质上相同。可以通过检查 Foo 的类型看到它是 FooMeta。

```
>>> type(Foo)
<class '__main__.FooMeta'>
```

然而,底层实现的方式却不同。with\_metaclass 通过创建一个存根类并将其置于类继承树中实现这一点,而 add\_metaclass 避免了该方式。当使用该方法时并不会在方法解析顺序中存在存根类。

```
>>> Foo.__mro__
(<class '__main__.Foo'>, <type 'object'>)
```

add\_metaclass 的底层实现方式是类最终被构建两次。第一次,创建一个“正常”类,然后装饰器接收该类并将其替换为带有合适元类的类,然后返回该类。效率会有略微降低,但最终得到的是略微纯净的结果。

## 10.2.7 异常语法

正如对 print 所做的修改,Python 3 为了移除不寻常(并且有点武断)的语法改变了异常语法。

在 Python 2 中,引发一个异常的语法最开始看上去如下所示:

```
raise ValueError, 'Invalid value.'
```

如果执行该语句,在 Python 2 中会发生什么呢?解释器会创建一个新 ValueError 对象

并将字符串作为它的唯一参数。当对象创建完毕后，解释器引发异常。

换言之，真正发生的事情是仅仅调用创建一个类(在本例中为 `ValueError`)实例的代码。因此，它应该看上去如下所示，并且在 Python 3 中，就是如此。使用逗号的不同寻常的语法被移除，只剩下直接对象的实例化。

```
raise ValueError('Invalid value.')
```

由于异常只是对象(继承 `Exception` 的子类也是如此)，并且因为在 Python 2 中引发异常对象有效，因此这里展示的 Python 3 语法无须修改就可以在 Python 2 中生效。

只需要始终使用该语法，即使是对于只运行在 Python 2 下的代码也是如此。这意味着无须再担心该区别。

## 1. 处理异常

除了改变引发异常的语法外，Python 3 中还引入了处理异常的语法。在 Python 2 中，`except` 语句看起来如下所示(同样，注意逗号)：

```
try:  
    raise ValueError('Invalid value.')  
except ValueError, ex:  
    print('%s' % ex)
```

Python 3 改变了该语法，使其更加清晰。Python 2 中的逗号被替换为 `as` 关键字(该关键字在 Python 2.5 中被引入，但用于其他目的)。

```
try:  
    raise ValueError('Invalid value.')  
except ValueError as ex:  
    print('%s' % ex)
```

这里展示的 Python 3 语法同样在 Python 2.6 与 Python 2.7 中有效。如果所编写的代码只需要在 Python 2.6 以及更新版本下执行，就应该使用 `as` 关键字代替旧语法。

## 2. 异常链

Python 3 还对异常处理添加了一个重要的新功能，也就是异常链。本质上，某些时刻会出现这种情况，在解释器处理一个异常时(在 `except` 子句中)，引发了另一个异常。在 Python 2 中，所有关于原始异常的信息会丢失。

在 Python 3 中，不再如此。当触发第二个异常时，将会把原始异常赋给 `_context_` 属性。

此外，Python 3 提供了显式指定另一个异常作为一个异常的“子句”的机制，使用新语法：`raise...from`。

```
raise DatabaseError('Could not write') from IOError('Could not open file.')
```

该代码会创建 `DatabaseError` 异常和 `IOError` 异常。后面的异常会被赋值为前面异常的子句。工作原理是 Python 3 中的异常现在有了一个 `_cause_` 属性，通常情况下值为 `None`，当该语法被调用时该值会被设置为合适的异常。`_cause_` 属性被认为优先于 `_context_` 属性。

使用它的合适时机是什么？最常见情况是类似于实现多个后端用于数据存储、任务执行或其他类似任务的框架，但希望暴露一个通用错误类，这样使用框架的程序员只需要处理一类错误。在 Python 2 中，这样一个模型需要在底层丢失异常数据，但在 Python 3 中，这些信息可以保留。

遗憾的是，Python 2 完全不支持这类异常链，并且 Python 2 中 `raise...from` 不是有效语法。然而，six 库提供了 `six.raise_from`。它接受两个参数(这两个异常)，在 Python 3 中附加到异常上下文中，而在 Python 2 中仅仅是忽略第二个参数。如果所编写的代码希望在两个环境中执行并希望利用 Python 3 中的异常链，就应该使用 `six.raise_from`。

## 10.2.8 字典方法

Python 2 中的 dict 类包含 3 个在 Python 3 中被改变的方法：`keys`、`values` 与 `items`。在 Python 2 中，这几种方法返回一个包含适当内容的 list 对象。

```
>>> d = {'foo': 'bar'}
>>> d.keys()
['foo']
```

这对于小字典完全没有问题，但对于大字典(尤其是 `values` 和 `items`)则会导致问题，这是由于生成大量数据的内存副本。

在大多数情况下，副本并不是你所需要的。你仅仅是希望遍历所请求的数据。对此任务来说生成器(见第 3 章“生成器”)是更好的方案。实际上，Python 2 提供了这类生成器：`iterkeys`、`itervalues` 和 `iteritems`。

```
>>> d = {'foo': 'bar'}
>>> gen = d.iterkeys()
>>> gen
<dictionary-keyiterator object at 0x10732d7e0>
>>> next(gen)
'foo'
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

此外，Python 2 还为这几个生成器提供了视图，名称分别为 `viewkeys`、`viewvalues` 和 `viewitems`。这些视图对象仅仅引用原始字典。结果是如果原始字典改变，视图会随之改变。

在 Python 3 中，只保留视图，返回 list 与仅返回一个生成器的方法被移除(视图也作为生成器)。然而，在 Python 3 中，视图方法现在使用原始方法名称：`keys`、`values` 与 `items`。

如果希望编写运行在 Python 2 与 Python 3 两个平台下的代码，six 模块提供了 `six.viewkeys`、`six.viewvalues` 与 `six.viewitems`，它们会与 Python 2 和 Python 3 中合适的方法相匹配。

### 10.2.9 函数方法

Python 2 与 Python 3 都提供了查看函数属性的方式，比如它们的名称、函数内的代码以及函数接受的参数。推荐的方式是使用 `inspect` 模块，但与函数对象直接交互的代码十分常见，因此，`six` 模块为其提供了一个接口。

Python 中函数的大量属性在 Python 2.6(不是 Python 3.0)中被重命名。在此之前，这些属性被认为是私有 API，因此 Python 开发人员认为在 Python 2.6 中允许重命名这些属性的决定能够被社区接受。

Python 2.5 中的属性名称为 `func_closure`、`func_code`、`func_defaults` 与 `func_globals`。在 Python 2.6 中，这些属性被重命名，移除了 `func_` 前缀并替换为双下划线(例如，`__closure__`)。

考虑下面这个简单函数的 `__defaults__` 元组：

```
>>> def foo(x=5):
...     return x + 3
...
>>> foo.__defaults__
(5,)
```

这段代码告知第一个可选参数有一个默认值 5。由于这里只有一个可选参数，因此在元组中只有一个元素。

`six` 模块提供了无论在哪个 Python 版本中都能返回正确属性的别名。它们是 `six.get_function_closure`、`six.get_function_code`、`six.get_function_defaults` 与 `six.get_function_globals`。这些函数都接受函数作为参数，如下所示：

```
>>> import six
>>> six.get_function_defaults(foo)
(5,)
```

### 10.2.10 迭代器

Python 3 略微改变了迭代的结构。在 Python 2 中，迭代期望一个无参的 `next` 方法。在 Python 3 中，它变为 `_next_`。

如果需要一个在 Python 2 与 Python 3 中都能够正确运行的迭代器，正确的方案是使用一个不完成任何工作但仅调用 `_next_` 的 `next` 方法，如下所示：

```
class CompatableIterator(object):
    def next(self):
        return self._next_()
```

任何继承 `CompatableIterator` 的子类都将收到一个 `next` 方法，该方法只调用 `_next_`，这使得迭代器在 Python 2 与 Python 3 中都可以正常工作。

然而，`six` 模块实际上提供的是类 `six.Iterator`。实际上，它比之前在 Python 2 中提供实

现的示例更加好用，仅仅是在 Python 3 中给对象分配一个别名。

因此，如果需要建立一个必须在 Python 2 与 Python 3 中都能运行的迭代器，只需要使其继承于 six.Iterator 并定义一个`_next_`方法而不是`next`方法。

## 10.3 标准库重定位

除了提供一些新功能以及在语法中有一些变更，Python 3 还对标准库中的一些模块做了改动。

通常来说，如果要维护的代码需要同时在 Python 2 与 Python 3 中执行，six 提供了获取正确模块的唯一接口。它们位于`six.moves`。

### 10.3.1 合并“高效”模块

对于 Python 2 中的`pickle`与`StringIO`模块，在 Python 2 标准库中它们都存在功能相同的副本。第一个是 Python 实现，第二个是用 C 编写的更高效实现。

Python 3 将两个模块合并为一个单独的模块，因此开发人员无须考虑使用的模块是 C 实现还是使用一个特定库的 Python 实现(当使用库时这类细节通常不重要)。

#### 1. io

Python 2 中的`StringIO`与`cStringIO`合并为一个单独模块`io`。

为了便于在 Python 2 与 Python 3 中执行一个单独模块，six 提供了`six.moves.cStringIO`，用于给类(不是模块)一个别名。因此，`six.moves.cStringIO`等价于 Python 2 中的`cStringIO.StringIO`以及 Python 3 中的`io.StringIO`。

例如，下面两种导入在 Python 3 中等价：

```
>>> from io import StringIO  
>>> from six.moves.cStringIO import StringIO
```

#### 2. pickle

`pickle`模块也类似于`io`模块。Python 2 提供了`pickle`与`cPickle`模块，通常，第二个模块会高效许多。Python 3 将这两个模块合并后的模块命名`pickle`。导入`cPickle`不再有效。

同样，six 为该模块提供了别名从而匹配正确的模块，而不用关心所运行的 Python 版本。但在本例中，`six.moves.cPickle`指向模块而不是类别名。这使得你可以从`pickle`模块中导入特定方法。

下面两行代码在 Python 3 中等价：

```
>>> import pickle  
>>> from six.moves import cPickle as pickle
```

### 10.3.2 URL 模块

Python 2 有 3 个与 URL 打交道的模块：urllib、urllib2 与 urlparse。每个模块所包含的内容一直令人混淆。

在实践中，通常需要一起使用它们，因此 Python 3 完全重组这些模块，将其合并到一个单独模块：urllib。大多数来自 urlparse 模块(主要关注读取 URL 并将其分解为单独部分)的方法现在位于 urllib.parse 中。

此外，很多关于解析的方法(比如 quote 与 unquote)从 urllib 移到了 urllib.parse 中。

Python 3 中重组后的 urllib 模块包含 4 个子模块：error、parse、request 与 response。six 模块为同样 4 个子模块提供了 six.moves.urllib，它将合适的方法按照它们在 Python 3 中被组织的方式集合到一起。

如果所编写的代码既需要在 Python 2 中运行也需要在 Python 3 中运行，并且你还使用了任何在 urllib 或其 Python 2 中等价的模块，就应该使用 six.moves.urllib。

### 10.3.3 重命名

Python 3 还重命名了某些模块，以及一些内置函数。表 10-1 列出了一些常见的被重命名或移动的函数，以及对应的 six.moves 函数别名。

表 10-1 常见的被重命名或移动的函数

PYTHON 3	PYTHON 2	six.moves
Configparser	ConfigParser	Configparser
filter	itertools.ifilter	filter
input	raw_input	input
map	itertools imap	map
range	xrange	range
functools.reduce	reduce	reduce
socketserver	SocketServer	socketserver
zip	itertools izip	zip

### 10.3.4 其他包重组

此外，在 Python 2 和 Python 3 之间还有很多包被重组，但少量不常用的包在 six 中没有别名。这些包包括 xml 与 tkinter 等。

如果使用这些包编写单一来源实现的代码，请查看包文档，以获取有关被移动项的信息。

如果遇到一个已被移动的模块或属性，则可以使用 six.add\_move 函数告知 six.moves 相关信息。如果一个模块被移动，将 six.MovedModule 对象发送到 add\_move。

six.MovedModule 构造函数接受 3 个参数：移动的名称（以及当从 six.moves 导入时被引用的方式）、旧模块名称与新模块名称，后两者的类型为字符串。

例如，下面的代码使得 six.moves.ttk 在 Python 2 中是 ttk 的别名，而在 Python 3 中是 tkinter.ttk 的别名：

```
>>> import six
>>> six.add_move(MovedModule('ttk', 'ttk', 'tkinter.ttk'))
```

如果一个模块内的属性被移动，则将 six.MovedAttribute 对象发送到 six.add\_move。MovedAttribute 构造函数接受两个额外参数，分别为属性的新旧名称，这些名称都为字符串类型。

## 10.4 版本检测

偶尔，你会遇到在 Python 2 中与 Python 3 中工作机制不同的情况，并且没有一种简单的接口能够让你的代码在两个版本中功能一致。

在这种情况下，six 模块提供了两个常量，six.PY2 与 six.PY3。它们根据当前运行的 Python 版本被设置为 True 或 False。

## 10.5 小结

Python 3 相对 Python 2 迈进了一大步。Python 3 使得语言更加简洁高效。而另一方面，由于向后兼容性的问题，Python 社区接受 Python 3 的步伐也很缓慢。

如果你正在编写 Python 2 代码，可以考虑使代码无须修改就能在 Python 3 中运行的这种编写方式。当最终需要转换到 Python 3 时，这会非常有益。

此外，这是强调自动化测试重要性的一个好地方。所编写的代码在不同条件下运行，你必须尽可能地确保在每个环境下代码都能以相同的方式工作。为此，可以使用一个健壮的单元测试套件，它能够在所有支持的环境下自动执行。对于从 Python 2 转换到 Python 3 来说，有一个功能测试套件或许是前置步骤，同时有一个可管理且能够在两个环境中执行的单一来源代码库也是必须的。

第 11 章将阐述有关测试的更多细节，包括如何在多环境中进行测试。

# 第 11 章

## 单元测试

当考虑到测试你所编写的代码时，能够想到的第一件事很可能是直接运行你的程序。如果程序能够运行，至少你可以知道程序没有任何语法错误(所提供的每个模块都被导入)。

类似地，如果你提供合适的输入，且没有获得回溯(traceback)，就知道针对这些输入程序可以成功执行。并且，如果输出结果与所期望的结果相匹配，那么这就是一个程序能够正常工作的额外归纳证据。

但是，这种方式有一系列关键限制。第一个限制是对于较大的程序，不可能测试所有场景。虽然对所有潜在场景进行尽可能完整的测试非常重要，但避免该限制几乎不可能。

第二个限制是(也是本章主要介绍的内容)是时间。对于大多数应用程序来说，对于程序所做的所有改动都手动测试所有场景并不现实，这是因为遍历这些场景非常耗时。

但可以通过自动化测试稍微减少该限制。自动化测试套件可以在你不在或忙别的事情时执行，这样就可以更早和更加频繁地测试你的工作，从而可以节省大量的时间。

本章介绍了有关测试的部分内容。尤其是，重点介绍了使用由 Python 标准库提供的内置工具进行的单元测试(比如 unittest 与 mock)，以及一些用于测试的常用包。

### 11.1 测试的连续性

那么，到底什么是单元测试？此外，与功能性测试或完整性测试或其他类型的测试有什么区别？为了解答该问题，本章讨论两种不同的测试场景。

#### 11.1.1 副本生态系统

首先，考虑一个非常完整的测试环境。如果编写一个主要在服务器上运行的应用程序，

这或许需要一个包含相关数据副本的“中间”服务器，其中潜在导致代码出问题的行为可以安全地被执行。对于一个脚本或桌面应用程序，原则是相同的。这需要程序的运行环境包含所需的修改或涉及的所有元素。

在这种场景中，程序所有需要的事物都需要模拟实际生产环境。如果连接到特定类型的数据库，该数据库也需要在测试环境中出现(只是在不同的位置)。如果从一个 Web 服务中获取数据，你仍然需要在测试环境中发起同样的请求。

本质上，在副本生态系统中，任何程序依靠的外部依赖都必须以同样的方式出现和设置。

这种类型的测试环境不仅被设计用于测试特定代码，还会测试实际生产环境中的完整生态环境。在应用程序的不同组件之间来回传递的任何数据都需要以完全相同的方式传输。

针对诸如此类副本生态系统的自动化测试通常称为系统测试。该术语表示针对完整复制出的生态系统进行的测试。这类测试被设计用于不仅测试指定代码，还检测导致外部环境破坏的变更。

## 11.1.2 隔离的环境

另一个不同类型的测试是测试一段指定的代码段，可以在隔离环境中完成这项测试。

在副本生态系统中，外部需求与依赖(比如数据库、外部服务或类似部分)全部被复制下来。另一方面，在隔离环境下执行的测试通常阻断测试代码与外部依赖，主要关注实际代码所做的工作。

这类阻断通过规定外部服务或依赖接收给定输入并返回给定输出来完成。这类测试并不显式地测试程序与其他服务的交互性。而是测试应用程序使用从其他服务接收的数据所完成的工作。

例如，考虑根据一个人的结婚日期确定其年龄的函数。它首先从外部数据库获得关于此人的信息(生日与结婚纪念日)，并计算两个日期的交汇点从而确定这个人的当前年龄。

该函数看上去或许如下所示：

```
def calculate_age_at_wedding(person_id):
    """Calculate the age of a person at his or her wedding, given the
    ID of the person in the database.
    """
    # Get the person from the database, and pull out the birthday
    # and anniversary datetime.date objects.
    person = get_person_from_db(person_id)
    anniversary = person['anniversary']
    birthday = person['birthday']

    # Calculate the age of the person on his or her wedding day.
    age = anniversary.year - birthday.year

    # If the birthday occurs later in the year than the anniversary, then
    # subtract one from the age.
```

```
if birthday.replace(year=anniversary.year) > anniversary:  
    age -= 1  
  
# Done; return the age.  
return age
```

当然，如果尝试实际运行该函数，将会失败。该函数依赖于另一个没有在本例中定义的函数 `get_person_from_db`。通过阅读注释与代码可以直观地理解它是从数据库获得特定类型的记录并返回一个类似字典的对象。

当测试一个类似这样的函数时，副本生态系统会重新构建数据库，使用特定 ID 请求某个人的记录，测试函数并返回期望的年龄。与之相反，在一个隔离环境中的测试完全避免了与数据库打交道。一个隔离的环境会声明你已经获得特定记录，并针对该记录测试函数的余下部分。

这类测试将代码与其他内容(甚至是其程序自身的其他代码部分)隔离的测试被称为单元测试。

### 11.1.3 优点与缺点

这两类基本测试都各有优点与缺点，并且大多数应用程序都必须将这两种测试作为一个健壮测试框架的一部分。

#### 1. 速度

在隔离环境中执行的单元测试最大的优点之一是速度。针对完整的生态环境副本的测试通常有非常长的安装与卸载时间。此外，用于在多个组件之间传输数据的 I/O 需求通常是导致应用程序运行最慢的原因之一。

与之相反，在一个隔离环境中执行的测试通常非常快。相比从数据库获取与 ID 对应的行并通过管道传输所花费的时间，之前示例中用于执行确定一个人年龄的算法所花费的时间要少得多(几个量级)。

有一套执行速度非常快的隔离测试很有价值，这是因为通过快速测试，你能够非常频繁地执行这些测试并得到反馈。

#### 2. 交互性

隔离测试之所以快，主要原因是这些测试是隔离的。隔离测试假定应用程序所需的多个服务之间的交互性不存在问题。

然而，这些交互性也需要测试。这也是为什么你还需要在副本生态系统中进行测试。这使你可以确保这些服务继续按你所期望的方式交互。

## 11.2 测试代码

本章的重点是单元测试。因此，如何编写执行之前示例中 `calculate_age_at_wedding` 函数

数的测试？你的目标不是实际与数据库交互并返回一个人的记录，因此你必须测试该函数并提供相关信息。

## 11.2.1 代码布局

在很多情况下，目前最好且最直观地测试这样一个函数的方式是按照更加容易测试的方式重构代码。

在 `calculate_age_at_wedding` 函数的示例中，你根本不需要从数据库获取任何记录。根据你的应用程序，让该函数接受完整的记录而不是 `person_id` 变量或许没有问题(甚至更加可行)。换言之，直到在调用数据库之前，接力棒都不会交给该函数，并且该函数唯一做的工作就是执行算法。

以这种方式重构代码还可以使得函数能够接受的数据类型更多。任何使用恰当键值的字典类对象都能被接受。

下面经过裁剪的函数仅计算年龄，并期望接收一个完整的个人记录(从何处获得该记录并不重要)。

```
def calculate_age_at_wedding(person):
    """Calculate the age of a person at his or her wedding, given the
    record of the person as a dictionary-like object.
    """
    # Pull out the birthday and anniversary datetime.date objects.
    anniversary = person['anniversary']
    birthday = person['birthday']
    # Calculate the age of the person on his or her wedding day.
    age = anniversary.year - birthday.year

    # If the birthday occurs later in the year than the anniversary, then
    # subtract one from the age.
    if birthday.replace(year=anniversary.year) > anniversary:
        age -= 1

    # Done; return the age.
    return age
```

在很多方面，该函数几乎与之前版本完全相同。唯一的变更是对于 `get_person_from_db` 的调用已被移除(以及对应的注释与文档字符串已更新，以匹配函数的最新功能)。

## 11.2.2 测试函数

当提到测试函数时，现在问题就变得非常简单。仅仅传入一个字典并确保获得正确的结果。

```
>>> from datetime import date
>>>
>>> person = {'anniversary': date(2012, 4, 21),
...             'birthday': date(1986, 6, 15)}
```

```
>>> age = calculate_age_at_wedding(person)
>>> age
25
```

当然，这里存在两个限制。首先，该测试仍然需要手动在交互式终端中执行。而单元测试套件的价值在于它能够自动执行。

第二个(甚至更加重要)的限制是该测试针对仅有的一个输出只有一个输入。假设你在第二天将该函数弄乱并将其替换为下面的形式：

```
def calculate_age_at_wedding(*args, **kwargs):
    return 25
```

该测试仍然会通过，即使该函数已经被完全破坏。

确实，该测试并没有覆盖这个函数其中的一些部分。毕竟，函数中还有一个 if 代码块，该代码块根据生日是否在结婚纪念日之前或之后来选择是否执行分支。至少，你希望确保测试能够将这两个代码路径都执行到。

下面的测试函数实现这一点：

```
from datetime import date

def test_calculate_age_at_wedding():
    """Establish that the `calculate_age_at_wedding` function seems to
    calculate a person's age at his wedding correctly, given a
    dictionary-like object representing a person.
    """
    # Assert that if the anniversary falls before the birthday in a
    # calendar year, that the calculation is done properly.
    person = {'anniversary': date(2012, 4, 21),
              'birthday': date(1986, 6, 15)}
    age = calculate_age_at_wedding(person)
    assert age == 25, 'Expected age 25, got %d.' % age

    # Assert that if the anniversary falls after the birthday in a calendar
    # year, that the calculation is done properly.
    person = {'anniversary': date(1969, 8, 11),
              'birthday': date(1945, 2, 15)}
    age = calculate_age_at_wedding(person)
    assert age == 24, 'Expected age 24, got %d.' % age
```

现在你有了一个能够被自动流程执行的函数。Python 包含一个测试执行程序，稍后将介绍它。另外，该测试覆盖了函数的两个不同排列。

当然，它并没有覆盖所有可能的输入(不可能实现这一点)，但提供了一个略微完整的检查。

但需要切记测试并不是一个彻底的检查。它仅仅测试你提供的输入与输出。例如，该测试并没有涉及如果发送到 calculate\_age\_at\_wedding 函数的参数不是字典会怎样，或是发送带有错误键值的字典，或是使用的是 datetime 对象而不是 date 对象，或发送的结婚周年纪念日小于生日，或大量其他排列等情况下会发生什么。这并不会导致问题，你只需要重

点理解测试的局限是什么即可。

### 11.2.3 assert 语句

测试函数使用的 assert 语句是什么？考虑单元测试的基本定义。单元测试是一个或一组断言。在本例中，你断言如果发送一个包含指定日期且格式正确的字典，可以得到一个整型结果。

在 Python 中，assert 是一个关键字，并且 assert 语句几乎只用于测试(虽然并不需要在测试代码中出现它)。assert 语句期望发送给它的表达式的值为 True。如果是 True，则 assert 语句不做任何操作；否则，会引发 AssertionError 异常。你可以选择在引发 AssertionError 时提供一个自定义错误信息，如前面的示例中所示。

在编写测试时，你希望在测试失败时使用 AssertionError 作为被引发的异常，可以通过直接引发该异常或是(通常)使用 assert 语句断言测试的通过条件，这是由于当编译测试失败时，所有的单元测试框架都会捕获错误并正确处理。

## 11.3 单元测试框架

由于你的测试是一个函数，因此接下来的步骤是设置执行该测试的流程(以及其他任何你所编写的用于测试应用程序其他部分的函数)。

有一些单元测试框架，比如 py.test 和 nose，能够以第三方包的形式获得。不过，Python 标准库也随之附带了健壮的单元测试框架，可以从标准库的 unittest 模块中获得。

考虑前面示例中的测试函数，但该函数通过 unittest 模块执行。

```
import unittest
from datetime import date

class Tests(unittest.TestCase):
    def test_calculate_age_at_wedding(self):
        """Establish that the `calculate_age_at_wedding` function seems
        to calculate a person's age at his wedding correctly, given
        a dictionary-like object representing a person.
        """
        # Assert that if the anniversary falls before the birthday
        # in a calendar year, that the calculation is done properly.
        person = {'anniversary': date(2012, 4, 21),
                  'birthday': date(1986, 6, 15)}
        age = calculate_age_at_wedding(person)
        self.assertEqual(age, 25)

        # Assert that if the anniversary falls after the birthday
        # in a calendar year, that the calculation is done properly.
        person = {'anniversary': date(1969, 8, 11),
                  'birthday': date(1945, 2, 15)}
        age = calculate_age_at_wedding(person)
```

```
self.assertEqual(age, 24)
```

在很多方面，该代码与前面的代码并无不同。但有一些关键区别。第一个区别是你现在拥有一个继承于 `unittest.TestCase` 的类。`unittest` 模块期望使用 `unittest.TestCase` 子类能够找到测试组。每一个测试都必须是其名称以 `test` 开头的函数。按照推论，由于测试本身现在是一个类的方法而不是未绑定函数，因此它有一个参数 `self`。

另一个关键区别是 `assert` 语句被替换为对 `self.assertEqual` 的调用。`unittest.TestCase` 类为 `assert` 提供了大量的包装器，用于标准化错误消息以及提供一些其他样板。

### 11.3.1 执行单元测试

现在是时候在 `unittest` 框架中实际执行这个测试了。为此，将函数与测试类保存到同一个模块下，比如 `wedding.py`。

Python 解释器提供了一个标记，`-m`，用于接收一个来自标准库或 `sys.path` 的模块，并把该模块作为脚本执行。`unittest` 模块支持以这种方式执行它，并接受被测试的 Python 模块（如果将模块命名为 `wedding.py`，那么这里提供的就是 `wedding`）。

```
$ python -m unittest wedding
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
OK
```

这里发生了什么？`wedding` 模块被载入，`unittest` 模块发现了一个 `unittest.TestCase` 子类。它实例化该类并执行以单词 `test` 开头的所有方法，`test_calculate_age_at_wedding` 方法就是以 `test` 开头。

对于成功执行的测试，`unittest` 输出会打印一个句号字符(.)；若测试执行失败，则打印字母(F)；若发生错误，则输出字母(E)；另外还有一些其他情况，比如遇到希望跳过的测试会打印字母(S)。由于这里只有一个测试并且已成功执行，因此只能看到一个.符号，之后就是输出结论。

#### 1. 失败

你可以通过改变测试条件故意让测试失败，从而观察当测试失败时会发生什么。

为了阐述这一点，在 `Tests` 类中添加下面的方法：

```
def test_failure_case(self):
    """Assert a wrong age, and fail."""
    person = {'anniversary': date(2012, 4, 21),
              'birthday': date(1986, 6, 15)}
    age = calculate_age_at_wedding(person)
    self.assertEqual(age, 99)
```

以下是一个类似的测试，只是它断言年龄为 99，这会导致错误。现在执行该测试，观

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

察会发生什么。

```
$ python -m unittest wedding
.F
=====
FAIL: test_failure_case (wedding.Tests)
Assert a wrong age, and fail.

-----
Traceback (most recent call last):
  File "wedding.py", line 50, in test_failure_case
    self.assertEqual(age, 99)
AssertionError: 25 != 99

-----
Ran 2 tests in 0.000s

FAILED (failures=1)
```

现在你有了两个测试。第一个测试是之前已通过测试的主测试，第二个测试由于是假年龄，测试失败。

如果直接执行函数，当引发 `AssertionError` 异常时只能得到一个标准的回溯。然而，`unittest` 模块实际上捕获到该错误并跟踪错误，并在测试执行的最后友好地输出结果。

此时这或许看上去像是无关紧要的区别，但如果你有上百个测试，区别就会很大。当遇到第一个未捕获的错误时，Python 模块将会终止，因此测试将会在第一个失败处停止。而当使用 `unittest` 时，测试会继续执行，并且在最后获得所有失败信息。

`unittest` 输出结果还包含测试函数、`docstring` 的开始部分以及完整的回溯，因此可以很容易找到失败的测试并调查原因。

## 2. 错误

一个错误与一个测试失败之间只有很小的区别。引发 `AssertionError` 异常的测试是测试失败，而其他类型的异常会被认为是错误。

考虑如果被测试的 `person` 变量是一个空字典会是什么。在 `wedding` 模块的 `Tests` 类中加入下面的函数：

```
def test_error_case(self):
    """Attempt to send an empty dict to the function."""
    person = {}
    age = calculate_age_at_wedding(person)
    self.assertEqual(age, 25)
```

现在如果执行测试会发生什么？

```
$ python -m unittest wedding
.EF
=====
ERROR: test_error_case (wedding.Tests)
Attempt to send an empty dict to the function.
```

```
Traceback (most recent call last):
  File "wedding.py", line 55, in test_error_case
    age = calculate_age_at_wedding(person)
  File "wedding.py", line 10, in calculate_age_at_wedding
    anniversary = person['anniversary']
KeyError: 'anniversary'
```

```
=====
FAIL: test_failure_case (wedding.Tests)
Assert a wrong age, and fail.
```

```
Traceback (most recent call last):
  File "wedding.py", line 50, in test_failure_case
    self.assertEqual(age, 99)
AssertionError: 25 != 99
```

```
Ran 3 tests in 0.000s
```

```
FAILED (failures=1, errors=1)
```

现在有了 3 个测试。之前的两个测试分别通过和失败，第三个测试报错。该错误会引发 `KeyError` 错误，而不是 `AssertionError` 错误，这是因为 `calculate_age_at_wedding` 函数需要字典中的 `anniversary` 键(该键并不在字典中)。

对于大多数实用目的来说，你或许不会过多关心一个失败与一个错误的区别。它们都是测试失败，只是方式略微不同。

### 3. 跳过测试

也可以将一个测试标记为在特定情况下跳过测试。例如，一个应用程序被设计为既能在 Python 2 下执行也能在 Python 3 下执行，但特定测试只有在环境是这两个中的一个时才有意义。可以将测试声明为只有在特定条件下执行，而不是让测试失败。

`unittest` 模块提供了装饰器接受一个表达式的 `skipIf` 与 `skipUnless` 装饰器。`skipIf` 装饰器会导致如果表达式求值结果为 `True` 时跳过测试，`skipUnless` 装饰器在表达式求值结果为 `False` 时跳过测试，这两个装饰器都接受第二个必需参数，该参数是一个用于描述跳过测试原因的字符串。

为了实际查看跳过的测试，在 `Tests` 类中加入下面的函数(为了保证输出结果大小合理，将会移除失败和错误测试的结果)。

```
@unittest.skipIf(True, 'This test was skipped.')
def test_skipped_case(self):
    """Skip this test."""
    Pass
```

这个使用 `unittest.skipIf.True` 装饰的函数是一个有效的 Python 表达式，并且求值结果明显为 `True`。现在来看当执行测试时会发生什么：

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
$ python -m unittest wedding
.s
-----
Ran 2 tests in 0.000s
OK (skipped=1)
```

被跳过的测试对应的输出结果是 `s`, 而不是表示测试通过的传统句号符号。使用小写字符而不是大写字符(如 `F` 和 `E` 那样)表示这并不是一个错误条件, 的确, 整个测试被认为成功了。

## 11.3.2 载入测试

目前为止, 你已经对一个单独模块运行了所有测试, 并且测试与被测试的代码都位于同一模块中。这对于一个测试用示例来说没有问题, 但对于大型应用程序就不可行了。

`unittest` 模块理解这一点, 并提供了从一个完整项目树中以编程的方式获取测试的扩展机制。默认类, 也就是适合大多数需求的类是 `unittest.TestLoader`。

如果你只是使用默认测试载入类(这也是大多数情况下你所希望使用的), 则可以使用关键字 `discover` 而不是被测试的模块名称来触发它。

```
$ python -m unittest discover
-----
Ran 0 tests in 0.000s
OK
```

测试去哪了? 测试发现遵循特定的规则, 从而确定从何处寻找测试。默认情况下, 它期望所有包含测试的文件命名遵循 `test*.py` 模式。

只测试符合以 `test` 开头的文件名也是你真正希望做的。测试发现的价值在于, 你可以将测试代码与其他代码相分离。因此, 如果将来自 `wedding.py` 文件的通过测试代码本身移动到一个匹配该模式(例如, `test_wedding.py`)的新文件中, 则测试发现系统会找到该文件(注意, 必须显式导入 `calculate_age_at_wedding` 函数, 因为它与被测试函数已经不在同一模块下)。

果然, 现在测试发现找到了测试:

```
$ python -m unittest discover
-----
Ran 1 test in 0.000s
OK
```

## 11.4 模拟

为了使 `calculate_age_at_wedding` 函数能够更加容易地进行单元测试，回忆一下为什么必须移除该函数的一部分。其理念是重构代码，将数据库调用的代码放到其他地方，使得该函数更容易被测试。

通常，以这种方式重构代码使得代码更容易测试是解决该问题的理想方式，但有时这并不现实或并不明智。若不再为了自动化测试通过重构代码隐式拆分特定功能，那么该如何显式拆分被测试的代码片段？

答案就是通过模拟。模拟是在测试中声明特定函数调用给出一个特定输出的过程，而函数调用本身会被禁止。此外，你可以以特定方式来断言你所期望的模拟调用。

从 Python 3.3 开始，`unittest` 模块发行了 `unittest.mock`，包含了用于模拟的工具。如果使用的是 Python 3.2 以及更早版本，你可以使用 `mock` 包，该包可以从 [www.pypi.python.org](http://www.pypi.python.org) 下载。

不同版本之间的 API 相同，但如何导入这些 API 就明显有变化。如果使用的是 Python 3.3，则使用 `from unittest import mock`；如果使用的是安装的包，则使用 `import mock`。

### 11.4.1 模拟函数调用

再次考虑 `calculate_age_at_wedding` 函数，该函数包含一个用于从未指定数据库获取记录的函数调用(如果你一直跟随练习，应该创建一个新文件)。

```
def calculate_age_at_wedding(person_id):
    """Calculate the age of a person at his or her wedding, given the
    ID of the person in the database.
    """
    # Get the person from the database, and pull out the birthday
    # and anniversary datetime.date objects.
    person = get_person_from_db(person_id)
    anniversary = person['anniversary']
    birthday = person['birthday']

    # Calculate the age of the person on his or her wedding day.
    age = anniversary.year - birthday.year

    # If the birthday occurs later in the year than the anniversary, then
    # subtract one from the age.
    if birthday.replace(year=anniversary.year) > anniversary:
        age -= 1

    # Done; return the age.
    return age
```

之前，测试该函数大部分通过改变函数本身来实现。为了可测试性你重构了代码。但你还希望能够测试无法修改或修改起来可能导致问题的代码。

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

首先，实际上 `get_person_from_db` 函数仍然不存在，因此你希望禁止对该函数的调用。为此，添加一个引发异常的函数。

```
def get_person_from_db(person_id):
    raise RuntimeError('The real `get_person_from_db` function '
                       'was called.')
```

此时，如果尝试执行 `calculate_age_at_wedding` 函数，则会得到一个 `RuntimeError`。对本例来说这非常方便，因为它将会在模拟无法正常工作时使其非常明显。测试会明显失败。

接下来执行测试。如果你只是尝试执行来自之前代码的测试，它将会失败(引发 `RuntimeError`)。你需要一种解决 `get_person_from_db` 调用的办法，这就是 `mock` 的用武之地。

`mock` 模块本质上是一个打补丁的库。它临时将给定命名空间的一个变量替换为一个名为 `MagicMock` 的特殊对象，然后在模拟范围结束后将变量还原为之前的值。`MagicMock` 对象本身非常自由。它基本上接受(并追踪)对其的任何调用，并返回任何你让它返回的值。

在本例中，你希望 `get_person_from_db` 方法在测试期间被替换为一个 `MagicMock` 对象。

```
import unittest
import sys

from datetime import date

# Import mock regardless of whether it is from the standard library
# or from the PyPI package.
try:
    from unittest import mock
except ImportError:
    import mock

class Tests(unittest.TestCase):
    def test_calculate_age_at_wedding(self):
        """Establish that the `calculate_age_at_wedding` function seems
        to calculate a person's age at his wedding correctly, given
        a person ID.

        """
        # Since we are mocking a name in the current module, rather than
        # an imported module (the common case), we need a reference to
        # this module to send to `mock.patch.object`.
        module = sys.modules[__name__]

        with mock.patch.object(module, 'get_person_from_db') as m:
            # Ensure that the get_person_from_db function returns
            # a valid dictionary.
            m.return_value = {'anniversary': date(2012, 4, 21),
                             'birthday': date(1986, 6, 15)}
            # Assert that that the calculation is done properly.
            age = calculate_age_at_wedding(person_id=42)
            self.assertEqual(age, 25)
```

这里新出现的重要内容是 `mock.patch.object`。该方法可以作为一个上下文管理器或一

个装饰器使用，并接收两个必要参数：被模拟的可调用对象所在的模块，以及字符串类型的被调用对象的名称。在本例中，由于函数以及测试都位于同一个文件(通常情况下，你并不会这么做)，因此必须获得一个对当前模块的引用，它总是 `sys.modules['__name__']`。

上下文管理器返回一个 `MagicMock` 对象，在前面的示例中为 `m`。然而，在可以调用被测试的函数之前，你必须指定所期望 `MagicMock` 完成的工作。在本例中，你希望它返回一个字典，该字典中包含某个人的有效记录。`MagicMock` 对象的 `return_value` 属性用于处理这一点。设置它意味着每次调用 `MagicMock`，它都会返回该值。如果没有设置 `return_value`，则返回另一个 `MagicMock` 对象。

如果对该模块执行测试，就会看到测试已通过(这里，新的模块被命名为 `mock_wedding.py`)。

```
$ python -m unittest mock_wedding
```

```
Ran 1 test in 0.000s
OK
```

#### 11.4.2 断言被模拟的调用

虽然该测试已通过，但从根本上而言，仍然在一个重要方面不完整。它模拟对 `get_person_from_db` 的函数调用，并根据输出结果测试该函数正确执行。

测试并没有真正验证是否调用了 `get_person_from_db` 函数。在某种程度上，这很多余。你知道调用实际发生了，否则就不会收到来自模拟对象的返回值。然而，有时调用模拟函数并不会返回值。

所幸的是，`MagicMock` 对象可以跟踪对它的调用。它存储关于被调用次数的信息并记录每一次调用而不是仅仅打印出返回值并完成。最终，`MagicMock` 提供了一些断言调用是否以特定方式发生的方法。

为此，最常用的方法或许就是 `MagicMock.assert_called_once_with`。该方法断言两件事：`MagicMock` 被调用且只被调用一次，并且使用了指定的参数签名。考虑一个增强后的测试函数，该函数能够确保使用期望的 person ID 来调用 `get_person_from_db` 方法：

```
class Tests(unittest.TestCase):
    def test_calculate_age_at_wedding(self):
        """Establish that the `calculate_age_at_wedding` function seems
        to calculate a person's age at his wedding correctly, given
        a person ID.
        """
        # Since we are mocking a name in the current module, rather than
        # an imported module (the common case), we need a reference to
        # this module to send to `mock.patch.object`.
        module = sys.modules['__name__']

        with mock.patch.object(module, 'get_person_from_db') as m:
```

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
# Ensure that the get_person_from_db function returns
# a valid dictionary.
m.return_value = {'anniversary': date(2012, 4, 21),
                  'birthday': date(1986, 6, 15)}

# Assert that that the calculation is done properly.
age = calculate_age_at_wedding(person_id=42)
self.assertEqual(age, 25)

# Assert that the `get_person_from_db` method was called
# the way we expect.
m.assert_called_once_with(42)
```

这里变化的部分是在代码的最后对 MagicMock 对象进行了检查，确保对它的调用如期进行。调用签名是一个单独的位置参数：42。这是在测试中使用的 person ID(在前面的几行代码中使用过该 ID)。它作为位置参数被发送，这是由于在原始函数中它是以这种方式作为参数提供的。

```
person = get_person_from_db(person_id)
```

注意，person\_id 作为一个单独的位置参数被提供，这也是记录 MagicMock 的方式。如果执行测试，将看到该测试仍然会通过：

```
$ python -m unittest mock_wedding
-----
Ran 1 test in 0.000s
OK
```

如果 MagicMock 的断言不正确会发生什么？测试会失败，并显示一个有用的失败信息，正如你可以通过修改 assert\_called\_once\_with 参数签名所看到的：

```
$ python -m unittest mock_wedding
F
=====
FAIL: test_calculate_age_at_wedding (wedding.Tests)
Establish that the `calculate_age_at_wedding` function seems

-----
Traceback (most recent call last):
  File "/Users/luke/Desktop/wiley/wedding.py", line 58, in
    test_calculate_age_at_wedding
    m.assert_called_once_with(84)
  File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/unittest
    /mock.py", line 771, in assert_called_once_with
    return self.assert_called_with(*args, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/unittest
    /mock.py", line 760, in assert_called_with
    raise AssertionError(_error_message()) from cause
```

```
AssertionError: Expected call: get_person_from_db(84)
Actual call: get_person_from_db(42)
```

```
Ran 1 test in 0.001s
```

在此会告知你 MagicMock 期望获得的调用，以及它实际接收的调用。如果没有发生调用或调用多于一次，就会获得类似的错误。

类似于 assert\_called\_once\_with 方法，assert\_called\_with 方法也有类似的功能，只是在 MagicMock 被调用超过一次时并不会测试失败，并且它只对最近的调用检查调用签名。

### 11.4.3 检查模拟

可以用几种其他方式检查 MagicMock 对象以确定发生了什么。你或许只是希望知道它是否被调用，或是它被调用的次数。你可能还希望断言调用顺序，或是查看调用签名的一部分。

#### 1. 调用次数与状态

最简单直接的两个问题是 MagicMock 是否被调用以及被调用的次数。

如果只是想知道 MagicMock 是否被调用过，可以检查 called 属性，在 MagicMock 第一次被调用时会将该属性设置为 True。

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.called
False
>>> m(foo='bar')
<MagicMock name='mock()' id='4315583152'>
>>> m.called
True
```

另一方面，你或许还想知道 MagicMock 被调用的确切次数。这也可以通过 call\_count 获知。

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.call_count
0
>>> m(foo='bar')
<MagicMock name='mock()' id='4315615752'>
>>> m.call_count
1
>>> m(spam='eggs')
<MagicMock name='mock()' id='4315615752'>
>>> m.call_count
2
```

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

MagicMock 类并没有断言是否被调用以及被调用次数的内置方法，但 unittest.TestCase 中的 assertEquals 与 assertTrue 方法完全可以胜任这项任务。

## 2. 多次调用

你或许还想在一个步骤内断言对 MagicMock 多次调用的组合。为此，MagicMock 对象提供了 assert\_has\_calls 方法。

为了使用 assert\_has\_calls，必须理解 mock 库中提供的 call 对象。每当发起一个对 MagicMock 对象的调用时，它都会在内部创建一个存储调用签名(并将其附加到对象内的 mock\_calls.列表)的 call 对象。如果签名匹配，则认为 call 对象相等。

```
>>> from unittest.mock import call  
>>> a = call(42)  
>>> b = call(42)  
>>> c = call('foo')  
>>> a is b  
False  
>>> a == b  
True  
>>> a == c  
False
```

这实际上就是 assert\_called\_once\_with 与类似方法的底层工作机制。它们创建一个新的 call 对象，然后确保与 mock\_calls 列表中的对象相等。

assert\_has\_calls 方法接受一个 call 对象的列表(或是其他类似对象，例如 tuple)。它还接受一个可选的关键字参数 any\_order，该参数值默认为 False。如果该值保持为 False，这意味着它期望调用顺序与列表中的顺序保持一致。如果该值设置为 True，只需要对 MagicMock 方法进行相同的调用即可，而不必关心对该方法的调用顺序。

下面所示是 assert\_has\_calls 的实际应用：

```
>>> from unittest.mock import MagicMock, call  
>>>  
>>> m = MagicMock()  
>>> m.call('a')  
<MagicMock name='mock.call()' id='4370551920'>  
>>> m.call('b')  
<MagicMock name='mock.call()' id='4370551920'>  
>>> m.call('c')  
<MagicMock name='mock.call()' id='4370551920'>  
>>> m.call('d')  
<MagicMock name='mock.call()' id='4370551920'>  
>>> m.assert_has_calls([call.call('b'), call.call('c')])
```

值得一提的是，虽然 assert\_has\_calls 期望调用的顺序保持一致，但它并不需要你发送调用的整个列表。从列表的任意一端调用都没有问题。

#### 11.4.4 检查调用

有时你或许并不想测试调用签名的完整性，或许只是想重点测试是否包含一个特定参数。这更难以完成。除了检查完整的调用签名外，并没有一个只供检查是否包含某个特定参数的方法。

然而，查看调用对象自身以及发送给它的参数也是可能的。工作机制是 call 类实际上是 tuple 的子类，并且调用对象是包含三个元素的元组，第二个和第三个参数是调用签名。

```
>>> from unittest.mock import call
>>> c = call('foo', 'bar', spam='eggs')
>>> c[1]
('foo', 'bar')
>>> c[2]
{'spam': 'eggs'}
```

通过直接查看 call 对象，可以获得一个位置参数的元组和一个关键字参数的字典。

该机制使你能够部分测试调用签名。例如，如果希望确保发送给调用的参数是字符串 bar，但并不关心其他参数时该怎么办？

```
>>> assert 'bar' in c[1]
>>> assert 'baz' in c[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert c[2]['spam'] == 'eggs'
```

一旦能够以元组的方式访问位置参数以及以字典的方式访问关键字参数，那么测试单个参数是否存在无非就是测试一个列表或字典中是否包含某个元素。

### 11.5 其他测试工具

在你为应用程序构建单元测试套件时，还有其他几种测试工作可以考虑。

#### 1. coverage

如何实际知道哪段代码被测试了呢？理想情况下，你希望在每次测试时执行尽可能多的测试，同时保持测试套件的执行速度。

如果想获知测试套件测试了多少代码，你会希望使用 coverage 应用程序，该程序可以从 [www.pypi.python.org](http://www.pypi.python.org) 下载。该程序最初由 Ned Batchelder 编写，是一个可以在测试执行中跟踪每个模块中所有代码行的工具，它提供一个报表列举哪些代码没有执行。当然，coverage 可以在 Python 2 中执行，也可以在 Python 3 中执行。

该应用程序通过安装 coverage 脚本，然后在调用包括单元测试脚本在内的任意 Python 脚本时使用 coverage run 取代 python 而生效。输出结果基本上相似。

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
$ coverage run -m unittest mock_wedding
```

```
Ran 1 test in 0.000s
```

```
OK
```

但如果查看字典，将会看到在过程中创建了.coverage 文件。该文件包含其中哪些代码被执行的信息。

可以通过使用 coverage report 命令查看这些信息。

```
$ coverage report
```

Name	Stmts	Miss	Cover
mock_wedding	22	1	95%

该报告展示了有多少语句已经执行以及有多少语句未执行。所以，你知道有一个语句被忽略，但并不知道是哪个语句。向命令行添加-m 会向输出结果添加哪一行代码被跳过。

```
$ coverage report -m
```

Name	Stmts	Miss	Cover	Missing
mock_wedding	22	1	95%	24

现在你知道第 24 行代码在测试时未执行(在 mock\_wedding.py 文件示例中，第 24 行对应于 RuntimeError 错误，在“真正的” get\_person\_from\_db 函数被调用时引发)。

coverage 应用程序还可以使用 coverage html 命令写入格式良好的 HTML 输出结果。这可以使未被执行的行使用红色高亮显示。此外，如果一个语句有多个分支(比如 if 语句)，则会使用黄色高亮标出只有一个路径被执行。

## 2. tox

很多 Python 应用程序都需要在包括 Python 2 和 Python 3 在内的多个 Python 版本上运行。如果所编写的应用程序在多版本环境下运行(甚至仅仅是多个小版本修订)，你会希望对所有环境执行测试。

尝试对所支持的所有版本手动执行测试很可能非常繁琐。如果需要完成这类工作，可以考虑使用 tox。tox 由 Holger Krekel 编写，是一个自动使用合适版本的 Python(由你已经安装的版本提供)创建虚拟环境(使用 virtualenv)并在这些环境内对其进行测试的工具。

## 3. 其他测试执行程序

本章主要关注由 Python 本身提供的测试执行程序，但还有很多其他替代方案。其中一些(如 nose 和 py.test)十分流行，它们添加了大量的功能以及用于扩展的钩子。

即使已经拥有了一个健壮的单元测试组件，这些库依然可以很容易被纳入现有程序，这是因为这两个工具都支持在盒子之外(译者注：也就是不会修改现有代码)使用单元测试。但是，这两个库还支持其他添加测试的方式。

这两个库都可以从 [www.pypi.python.org](http://www.pypi.python.org) 以及 Python 2.6 和以上版本执行。

[www.itbaizhan.cn](http://www.itbaizhan.cn)

## 11.6 小结

单元测试是一种确保代码随着时间改变而保持一致的强大方式。这是一种发现代码何时变更并做出适当调整的有效方式。

这是任何应用程序重要的一方面。拥有一个强壮的测试套件可以更容易地检测一些 bug 并察觉函数的行为何时发生变化，从而简化应用程序的维护。

第 12 章将介绍在命令行接口(CLI)上使用 Python 的 optparse 与 argparse 工具。

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 12 章

## CLI 工具

Python 应用程序以各种各样的形式出现，包括桌面应用程序、服务器应用程序、脚本、科学计算应用程序以及其他形式。

一些 Python 应用程序必须使用命令行界面(CLI)执行。它们可能要求输入数据，在脚本被调用时接受提供的参数。

本章介绍了 optparse 与 argparse，这是由 Python 标准库提供的用于编写由 CLI 执行的应用程序的工具。

### 12.1 optparse

optparse 是两个模块中较早由 Python 提供的，在 Python 2.7(也就是引入 argparse 的版本)后被名义上认定为已过时。然而，optparse 仍然被广泛使用，对于打算支持 Python 2.6 的版本来说是必须的，这使得它在 Python 生态系统中十分常见。

本质上，optparse 的存在用于提供一种清晰一致的方式来读取命令行，包括位置参数以及选项和开关。

#### 12.1.1 一个简单的参数

实际上，看完 optparse 的示例后理解起来非常简单。考虑下面这个用于接受来自 CLI 的选项的简单 Python 脚本：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
```

```
options, args = parser.parse_args()

print(' '.join(args).upper())
```

该脚本接受任意数量的参数，将它们转换为大写字符，并将其打印回 CLI。

```
$ python echo_upper.py
```

```
$ python echo_upper.py foo bar baz
FOO BAR BAZ
$ python echo_upper.py spam
SPAM
```

下面两小节将分解该示例。

## 1. `__name__ == '__main__'`

如果你没写过太多命令行脚本(或者在其他用例中遇到它)，那么可能会不太熟悉 `if __name__ == '__main__'`。在 Python 中，每个模块都有一个 `__name__` 属性，该属性总是被设置为当前正在执行的模块的名称。

值 `__main__` 比较特殊。当一个模块被直接调用时(比如在命令行执行)，`__name__` 属性会被设置为该值。

为什么要测试该值？Python 中几乎所有的.py 文件都可以作为模块导入，因此，同样可以导入本示例文件。在 CLI 脚本中，你或许不希望脚本直接被调用。CLI 脚本有时包含诸如调用 `sys.exit()` 从而结束整个程序的代码。该模块的选项和参数解析行为只有在直接调用时才有意义。因此，这种类型的代码应该放置于 `if __name__ == '__main__'` 测试之下。

注意，if 代码块并没有神奇之处，它仅仅是一个高层级的 if 语句。即使 if 测试失败，其他高层级的代码仍然会运行。此外要注意的是，将该测试放置在文件末尾是一种传统做法。

## 2. OptionParser

接下来，考虑创建一个 OptionParser 实例，随后调用它的 `parse_args` 方法。OptionParser 类是 optparse 模块中用于接受发送到 CLI 命令的参数和选项的主要类，并使命令行和参数生效。

该类的基本作用是告诉 OptionParser 实例你所期望的选项并知道如何处理这些选项。选项是一些以-或--开头的字符串，比如-v 或-verbose(稍后你将学习更多相关内容)。对于 `parse_args` 的调用会迭代所有解析器能够识别的选项，并将其置于 `parse_args` 返回的第一个变量中(在前面的示例中命名为 `options`)。任何剩下的参数都被认为是位置参数，它们置于第二个变量中(在前面示例中为 `args`)，类型为列表。

前面的示例没有使用 `options`，所有解析器接收的一切都被认为是位置参数。之后，该脚本接受该列表，将其连接为一个字符串，并转换为大写后打印出来。

值得注意的是，任何以连字符开头的参数都被认为是一个选项，如果你尝试发送一个解析器不能识别的选项，optparse 将会引发一个异常。此外，该异常会在 optparse 内部处理

# www.itbaizhan.cn

并调用 `sys.exit`, 所以你自己无法捕获这些错误。

```
$ python echo_upper.py --foo  
Usage: echo_upper.py [options]  
  
echo_upper.py: error: no such option: --foo
```

## 12.1.2 选项

位置参数通常并不是将信息输入到脚本的最直观的方法。当你的脚本目的很直接, 而且只有一两个位置参数时使用它们很合理。但是, 随着脚本变得更加可定制, 你通常会希望使用选项。

对于很多用例来说, 相比于位置参数, 选项有如下好处:

- 可以将它们设置为可选(且通常应该作为可选), 当未提供选项时可以使用合理的默认值。
- 选项也可以接受与一个键(选项的名称)相关的选项值, 这可以提升可读性。
- 多个选项可以按照任意顺序提供。

### 1. 选项类型

CLI 脚本可以接受两种常见的选项类型。

其中一种类型有时被称为一个标记或一个开关, 这是一种不需要或不接受带有值的选项。本质上, 在本例中, 选项是否出现决定了脚本的行为。

这种开关的两个常见示例是`--verbose` 与`--quiet`(通常分别以`-v` 和`-q` 的方式提供)。如果未指定该选项, 脚本正常执行; 但如果提供了这些参数, 则完成不同的工作(提供更多或更少的输出)。注意, 通常以`--quiet` 的方式使用该选项, 而不是以`--quiet=true` 或类似方式。该值根据开关是否出现隐式提供。

另一种类型的选项是期望给参数赋一个值, 而不仅仅把参数作为开关使用。大多数数据库客户端接受诸如`--host`、`--port` 等选项。它们如果作为开关并没有意义。你无法仅是提供`--host` 并期望数据库客户端推断实际的主机名称是什么。你必须提供所需连接的主机名称或 IP 地址。

### 2. 向 OptionParser 添加选项

一旦有了 `OptionParser` 实例, 就可以使用 `add_option` 方法向其添加选项。该步骤在 `OptionParser` 实例化之后但在链条的最后一步 `parse_args` 之前进行。

首先考虑添加一个简单开关, 它并没有实际期望一个参数。

```
import optparse  
  
if __name__ == '__main__':  
    parser = optparse.OptionParser()  
    parser.add_option('-q', '--quiet',
```

```
    action='store_true',
    dest='quiet',
    help='Suppress output.',
)
```

这会为`-q` 和`--quiet` 开关添加支持。注意，在 CLI 脚本中，同时有长格式与短格式选项的版本非常常见，因此 `optparse` 非常容易支持这一点。通过向 `add_option` 提供两个不同的字符串作为位置参数，`add_option` 方法理解它们应该被接受并且它们互为别名。

指定`--quiet` 标记的 `action` 关键字参数是一个标记，而并没有期望传入一个变量。如果不设置 `action` 关键字参数，该选项就被假设期待一个值(后面将介绍更多相关信息)。设置 `action` 为 `store_true` 或 `store_false` 意味着不期待任何值，如果提供了标记，值就分别为 `True` 或 `False`。

`dest` 关键字参数用于确定在 Python 中选项的名称。在 `options` 变量中特定选项的名称是 `quiet`。在很多情况下，你并不需要设置该参数。`OptionParser` 基于选项的名称推断一个合适的名称。但为了可读性与可维护性，总是显式设置该参数是一个好主意。

最后是 `help` 关键字参数，用于为该选项设置帮助文本。当用户使用`--help` 调用你的脚本时，这正是他看到的帮助信息。总是提供该参数是明智的。

值得注意的是，`optparse` 自动添加了一个`--help` 选项，并自动为其设置值。如果你仅使用示例选项并提供`--help` 选项调用一个脚本，你会获得有用的输出。

```
$ python cli_script.py --help
Usage: cli_script.py [options]

Options:
-h, --help  show this help message and exit
-q, --quiet Suppress output.
```

### 3. 带有值的选项

除开关之外，有时你需要的选项实际上期望带有与选项对应的值。这会增加一些复杂度。最大的原因是值在 Python 中是有类型的，而 CLI 并没有一个强壮的类型概念。本质上，所有的一切都是字符串。

首先，考虑一个接受字符串的选项，例如，可能被发送到数据库客户端的`--host` 标记。该选项或许是可选的。数据库客户端的最大用例是连接到位于同一台机器上的数据库，因此 `localhost` 是一个完全合理的默认值。

下面是一个仅将主机名称打印到标准输出的完整脚本：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-H', '--host',
                      default='localhost',
```

```
dest='host',
help='The host to connect to. Defaults to localhost.',
type=str,
)
options, args = parser.parse_args()
```

如果你不带参数调用该脚本，将会看到应用了默认值 localhost。

```
$ python optparse_host.py
The host is localhost.
```

下面通过添加一个--host 选项，重载了该默认值。

```
$ python optparse_host.py --host 0.0.0.0
The host is 0.0.0.0.
```

如果没有提供选项，optparse 将会报错。

```
$ python optparse_host.py --host
Usage: optparse_host.py [options]

optparse_host.py: error: --host option requires an argument
```

请关注对于 add\_option 的调用。很多方面与--quiet 标记不同。首先，你忽略了 action 关键字参数。该关键字参数的默认值(store)仅存储传入的值。如果希望使用非默认值，可以手动指定 action 参数的值。

其次，提供了一个 type 参数。大多数情况下，OptionParser 实例实际上会根据默认值的类型来推断类型(虽然，如果默认值为 None，这并不生效)，因此提供该参数通常是可选的。显式提供该参数通常会使代码在之后更容易阅读。type 的默认值也是 str。

最后，提供了一个 default 参数。大多数参数应该是可选的，这意味着它们应该有一个合理的默认值。在很多情况下，该默认值可能是 None。在主机名值的示例中，你选择 localhost 作为合理的默认值，因为让客户端与服务器在同一台机器上是一个常见的用例。

另外值得明确指出的是，从 options 变量读取值的方式并不是你所期望的方式——host 的值作为 options.host 读取。你或许期望 options 的值以一个字典的形式提供，在这种情况下 options['host'] 是正确的。但是，options 变量使用它自己的一个特殊类(称为 Values)提供，而每一个单独选项以属性的方式存在于该对象中。注意，如果你想要一个字典，options.\_\_dict\_\_ 将会为你提供对应的字典。

#### 4. 非字符串值

什么是非字符串的值？例如，继续使用客户端连接到某种数据库的示例，假如脚本应该接受一个端口号又如何？大多数数据库在默认端口上执行(PostgreSQL 使用 5432，MySQL 使用 3306 等)，但有时这种服务在其他端口上执行。

用于端口的选项看上去类似于用于主机的选项。

```
parser.add_option('-p', '--port',
```

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
    default=5432,
    dest='port',
    help='The port to connect to. Defaults to 5432.',
    type=int,
)
```

这里的关键区别是 `type` 现在被指定为 `int`。同样，`OptionParser` 将会根据默认值为整型 5432 推断出类型为 `int`。

在本例中，`OptionParser` 执行类型转换，当转换失败时引发相应的错误。考虑一个接受一个主机与端口的脚本，如下所示：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-H', '--host',
                      default='localhost',
                      dest='host',
                      help='The host to connect to. Defaults to localhost.',
                      type=str,
)
    parser.add_option('-p', '--port',
                      default=5432,
                      dest='port',
                      help='The port to connect to. Defaults to 5432.',
                      type=int,
)
    options, args = parser.parse_args()

    print('The host is %s, and the port is %d.' %
          (options.host, options.port))
```

同样，不带参数调用脚本会使这两个值都使用默认值。因为使用 `%d` 而不是 `%s` 格式化字符串，你就知道底层 `options.port` 是一个整型。

```
$ python optparse_host_and_port.py
The host is localhost, and the port is 5432.
```

如果尝试指定一个非整型的端口值，会得到一个错误。

```
$ python optparse_host_and_port.py --port=foo
Usage: optparse_host_and_port.py [options]

optparse_host_and_port.py: error: option --port: invalid integer value: 'foo'
$ echo $?
2
```

当然，如果指定一个有效的整型，它会重写默认值。

```
$ python3 optparse_host_and_port.py --port=8000  
The host is localhost, and the port is 8000.
```

### 5. 指定选项值

在命令行中如何指定选项值存在几种不同的方式。optparse 模块尝试支持所有方式。

#### 短格式语法

短格式语法是有一个连字符与一个单独字符的选项，比如-q、-H 或-p。如果选项接受一个值(如前面示例中的-H 与-p)，它必须写在紧挨着选项之后。在选项与值之间的空格是可选的(-Hlocalhost 与-H localhost 等价)，并且可以选择将值包裹在双引号内(-Hlocalhost 与-H "localhost" 等价)。但不能在短格式选项与值之间使用等号。

下面是 4 种使用短格式语法指定选项的有效方式：

```
$ python optparse_host_and_port.py -H localhost  
The host is localhost, and the port is 5432.  
$ python optparse_host_and_port.py -H "localhost"  
The host is localhost, and the port is 5432.  
$ python optparse_host_and_port.py -Hlocalhost  
The host is localhost, and the port is 5432.  
$ python optparse_host_and_port.py -H"localhost"  
The host is localhost, and the port is 5432.
```

在短格式语法中使用等号会导致该等号被认为是值的一部分，这并不是你希望的(注意在输出结果中的=)。对于非字符串选项，在解析器尝试将字符串转换为想要的类型并失败时，你通常会得到一个错误。

```
$ python optparse_host_and_port.py -H=localhost  
The host is =localhost, and the port is 5432.
```

并且，你可能会得到如下结果：

```
$ python optparse_host_and_port.py -H="localhost"  
The host is =localhost, and the port is 5432.
```

#### 长格式语法

对于长格式语法(也就是，--host 而不是-H)，所支持的排列略微不同。

现在在选项与选项值之间必须有分隔符(与-Hlocalhost 不同)。这给人直观的感觉。如果提供的是--hostlocalhost，解析器将永远不会知道选项结束与值开始的位置。分隔符可以是一个空格或一个等号(因此，--host=localhost 与--host localhost 等价)。

允许使用引号，但这是可选的(如果值中间包含空格，你当然希望使用引号)。

下面是 4 种使用长格式语法指定选项的有效方式：

```
$ python cli_script.py --host localhost  
The host is localhost, and the port is 5432.  
$ python cli_script.py --host "localhost"  
The host is localhost, and the port is 5432.  
$ python cli_script.py --host=localhost
```

```
The host is localhost, and the port is 5432.  
$ python cli_script.py --host="localhost"  
The host is localhost, and the port is 5432.
```

## 应该使用哪一种语法

在短格式语法与长格式语法之间的基本权衡是短格式语法在 CLI 中更容易输入，而长格式语法更加明显。

当编写 CLI 脚本时，应考虑既支持短格式语法，又支持长格式语法，尤其是对于将被频繁使用的语法来说更要如此(对于频繁使用的选项，仅提供一个长格式别名可能就已足够)。

当你调用 CLI 脚本时，如果所编写的代码需要提交到版本控制中，并在之后这些代码会被读取与维护，应考虑尽可能仅使用长格式语法。这使得 CLI 命令对于后续读取代码的人来说更容易靠直觉知道意思。

另一方面，对于你输入到提示符中的一次性命令，使用哪种语法可能并不重要。

## 6. 位置参数

也可以将位置参数发送给 optparse。实际上，任何没有附加到选项的参数都会被解析器认为是一个位置参数，并将其发送给由 parser.parse\_args() 返回的 args 变量。

```
import optparse  
  
if __name__ == '__main__':  
    parser = optparse.OptionParser()  
    options, args = parser.parse_args()  
  
    print('The sum of the numbers sent is: %d' %  
          sum([int(i) for i in args]))
```

任何发送到该脚本的参数都是 args 变量的一部分，该脚本尝试将其转换为整型并相加。

```
$ python optparse_sum.py 1 2 5  
The sum of the numbers sent is: 8
```

当然，如果你发送一个无法转换为整型的参数，将会得到一个异常。

```
$ python optparse_sum.py 1 2 foo  
Traceback (most recent call last):  
  File "optparse_sum.py", line 8, in <module>  
    print('The sum of the numbers sent is: %d' % sum([int(i) for i in args]))  
ValueError: invalid literal for int() with base 10: 'foo'
```

## 7. 计数器

除了简单的标记与直接值存储之外，还可以使用少量的其他选项类型。一种很少用但有时非常有用的类型是计数器标记。

大多数标记仅根据标记是否存在将一个布尔类型的值设置为 True 或 False。但是，一

一个相关的概念是允许多次指定一个标记从而加强效果。

考虑一个导致脚本更加冗长的-v 标记。一些程序允许重复指定-v 使脚本变得更加冗长。例如，一个名为 Ansible 的流行配置工具允许你最多指定-v 四次，从而提供极度冗长的输出。

可以通过给 add\_option 提供一个不同的 action 值实现这一点。考虑该脚本：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-v',
                      action='count',
                      default=0,
                      dest='verbosity',
                      help='Be more verbose. This flag may be repeated.')
)
options, args = parser.parse_args()

print('The verbosity level is %d, ah ah ah.' % options.verbosity)
```

注意，现在调用 add\_option 并指定 action='count'。这意味着每次发送标记时，该值都会递增 1。

通过调用下面的脚本，可以很容易看出计数器的作用。

```
$ python count_script.py
The verbosity level is 0, ah ah ah.
$ python count_script.py -v
The verbosity level is 1, ah ah ah.
$ python count_script.py -v -v
The verbosity level is 2, ah ah ah.
$ python count_script.py -vvvvvvvvvvvv
The verbosity level is 11, ah ah ah.
```

注意，在本例中你有两种指定短格式选项的有效方式：-v -v 与-vv 是等价的。这对于短格式选项也同样生效，提供这类参数不需要带有值。

另外，还要注意的是，指定 default 值为 0 很重要。如果你没有显式指定，OptionParser 使用默认值 None，通常这并不是你想要的(在本例中，当在脚本的最后一行尝试进行字符串插值时脚本会引发 TypeError 异常)。

最后注意，如果选择一个非 0 的默认值，标记函数就是一个递增值，而不是一个纯粹的计数器。因此，如果默认值为 1，并且你提供了两个-v 标记，值就会是 3(而不是 2)。

## 8. 列表值

有时，对于同一个选项你希望接受多个值，并将其以列表的形式提供给脚本。这与计数器选项基本类似，只是它每次接受一个值，而不是简单地让一个整型变量递增。

下面的脚本打印用户名，每次一个：

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-u', '--user',
                      action='append',
                      default=[],
                      dest='users',
                      help='The username to be printed. Provide this multiple times to '
                           'print the username for multiple users.')
    options, args = parser.parse_args()

    for user in options.users:
        print('Username: %s.' % user)
```

若不带-u 或--user 选项，执行该脚本不会生成任何输出。

```
$ python echo_usernames.py
$
```

但是，你可以为脚本提供一个或多个-u 或--user 选项，无论提供几个选项，OptionParser 都会使得它们以列表的形式可用：

```
$ python echo_usernames.py -u me
Username: me.
$ python echo_usernames.py -u me -u myself
Username: me.
Username: myself.
```

## 12.1.3 使用 optparse 的原因

虽然 optparse 已经过时多年，但该模块依然是用于解析选项最常用的模块。任何必须在 Python 2.6 或更早版本、Python 3.0 到 Python 3.2 之间版本上运行的代码，都必须使用 optparse，因为它的继任者 argparse，只在 Python 2.7 和 Python 3.3 中可用。

如果你正在使用 CLI 工具编写的代码必须运行在多个 Python 版本上，optparse 最可能是你多年内仍将继续使用的模块。类似的，由于你使用的工具很多依赖于 optparse，因此能够阅读使用 optparse 设计的模块代码就十分重要。

另一方面，注意 optparse 已经不会有后续版本的开发，因为它仍将过时。随着时间的推移，你所需要支持的 Python 版本的窗口向后迁移，你或许会决定将 optparse 中完成的工作迁移到 argparse 中。

## 12.2 argparse

Python 提供的第二个用于解析 CLI 参数与选项的库称为 argparse。argparse 模块被认为是 optparse 的继任者(optparse 已正式过时)。但是, argparse 模块仍然比较新。它在 Python 3.3 中被引入并移植到 Python 2.7 中。因此,任何需要在 Python 的更早版本中执行的代码仍然需要使用 optparse。

在很多方面, argparse 在概念上都类似于 optparse。基本原则是相同的。你创建一个解析器指定选项,以及一些所期望的类型与合理的默认值;然后解析器解析来自 CLI 的内容并对其进行相应的分组。

你在 argparse 模块中实例化且用于解析的类是 ArgumentParser, 虽然该类使用的一些语法与 optparse. OptionParser 所使用的不同,但其原则却十分相似。

### 12.2.1 本质

一个不支持任何实际参数或选项的基本 CLI 脚本看上去如下所示:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    args = parser.parse_args()

    print('The script ran successfully and did nothing.')
```

有一个需要注意的关键区别,除了重命名模块和类之外,还有 parse\_args 方法,不像与它等价的 optparse 那样返回一个二元组,而是返回一个同时包含位置参数与选项的对象。

另一个区别是位置参数被处理的方式。在 optparse 中,你并没有声明位置参数。第二个变量仅包含已经通知并由 optparse 处理之外的选项。与之相反, argparse 更加严格。对于每个位置参数,它都期望被通知,这使得帮助屏幕更有用,并且也能在接收的数据不符合预期的情况下引发一个错误。

因此,与最开始 optparse 示例不同,下面的代码在接收任何参数后引发一个错误,而不是将这些参数扔到剩下的篮子中。

```
$ python argparse_basic.py
The script ran successfully and did nothing.
$ python argparse_basic.py foo
usage: argparse_basic.py [-h]
cli_script.py: error: unrecognized arguments: foo
```

## 12.2.2 参数与选项

在 argparse 中，通过 ArgumentParser 对象的 add\_argument 方法添加位置参数与选项。对此的接口现在已经统一，这意味着 argparse 中的位置参数支持 str 之外的类型，并可以指定默认值。

### 1. 选项标记

第一类选项是标记，比如用于详细模式的 -v 或 --verbose，或者用于阻止大部分或所有输出结果的 -q 或 --quiet。这些选项并不期望有值。选项是否存在决定了在解析器中对应的布尔值。

指定标记的语法看上去如下所示：

```
parser.add_argument('-q', '--quiet',
                    action='store_true',
                    dest='quiet',
                    help='Suppress output.',
)
```

如果你已经熟悉了 optparse，这段代码对你来说会看上去很熟悉。除了方法名称外，并没有太多变更。

首先，注意 action 变量，该变量被设置为 store\_true，这就是为什么解析器并不期望有值的原因。大多数选项并不需要指定 action 的值(默认值为 store，用于存储它接收的值)。指定 store\_true 或 store\_false 是表示一个选项是标记且并不该接受任何值的最常见方式。

dest 关键字参数决定了在调用 parse\_args 返回的对象中查找被解析值的方式(在本例中，为 True 或 False)。这里使用的字符串是对象的属性名称(因此，可以使用 args.quiet 查找这个名称)。很多情况下，dest 关键字参数是可选的。ArgumentParser 根据选项自身的名称确定了初始名称。但为了可读性与可维护性，显式提供名称更加有用。

help 关键字参数决定当用户使用 -h 或 --help 参数调用脚本时所返回的内容。ArgumentParser 隐式提供附加到这些开关的一个帮助屏幕，因此应该总是在参数中指定一个 help。

### 2. 替代前缀

大多数 CLI 脚本使用连字符(-)作为选项前缀，这也是 ArgumentParser 默认期望的。但有些脚本可能使用不同的字符。例如，一段只打算在 Windows 环境执行的脚本可能倾向使用 / 字符，这会与很多 Windows 命令行程序保持一致。

可以通过为 ArgumentParser 构造函数提供 prefix\_chars 关键字参数来改变作为前缀使用的字符，如下所示：

```
import argparse
```

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(prefix_chars='/')
    parser.add_argument('/q', '//quiet',
                        action='store_true',
                        dest='quiet',
                        help='Suppress output.')
)
args = parser.parse_args()

print('Quiet mode is %r.' % args.quiet)
```

在本例中，将前缀字符变更为/。注意，这也意味着参数自身(传递给 add\_argument 的参数)必须进行相应的改变。

调用该脚本仍然简单直接。只需要使用/q 或//quiet(而不是-q 或--quiet)。

```
$ python argparse_quiet.py
Quiet mode is False.
$ python argparse_quiet.py /q
Quiet mode is True.
```

查看帮助屏幕，可以看到这一点：

```
$ python argparse_quiet.py /h
usage: argparse_quiet.py [/-h] [/q]

optional arguments:
  /h, //help  show this help message and exit
  /q          Suppress output.
```

注意，由于你将前缀字符变更为/，因此自动注册的 help 命令也随之发生了变更。

### 3. 带有值的选项

接受值的选项在本质上是类似的。考虑下面这个接受 host 值(比如数据库客户端)的脚本示例，将其转换成 argparse 版本：

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-H', '--host',
                        default='localhost',
                        dest='host',
                        help='The host to connect to. Defaults to localhost.',
                        type=str,
)
    args = parser.parse_args()

    print('The host is %s.' % args.host)
```

同样，如果你已经熟悉了 optparse，很容易就能注意到它们十分相似。关键字参数相同，并完成同样的任务。

这里需要重点关注的参数是 type，它控制最终期望值所属的 Python 类型。该值的常见类型是 int 或 float，还有少量其他类型也同样可用。

使用 argparse 对参数的解析与你使用 optparse 时略有不同。无论你使用的是短格式还是长格式语法，都可以使用一个空格或一个等于号来分隔选项和值。短格式语法(并且仅仅是短格式语法)也支持完全不分隔选项与值。短格式语法与长格式语法都允许使用引号包裹值。

因此，下面所有的命令等价：

```
$ python argparse_args.py -Hlocalhost  
The host is localhost.  
$ python argparse_args.py -H"localhost"  
The host is localhost.  
$ python argparse_args.py -H=localhost  
The host is localhost.  
$ python argparse_args.py -H="localhost"  
The host is localhost.  
$ python argparse_args.py -H localhost  
The host is localhost.  
$ python argparse_args.py -H "localhost"  
The host is localhost.  
$ python argparse_args.py --host=localhost  
The host is localhost.  
$ python argparse_args.py --host="localhost"  
The host is localhost.  
$ python argparse_args.py --host localhost  
The host is localhost.  
$ python argparse_args.py --host "localhost"  
The host is localhost.
```

## 选择

ArgumentParser 能够指定一个选项只能是枚举集合中的一个枚举值。

```
import argparse  
  
if __name__ == '__main__':  
    parser = argparse.ArgumentParser()  
    parser.add_argument('--cheese',  
                        choices=('american', 'cheddar', 'provolone', 'swiss'),  
                        default='swiss',  
                        dest='cheese',  
                        help='The kind of cheese to use',  
    )  
    args = parser.parse_args()
```

www.itbaizhan.cn

```
print('You have chosen %s cheese.' % args.cheese)
```

如果不带参数运行该脚本，正如所料，你将获得默认值。

```
$ python argparse_choices.py  
You have chosen swiss cheese.
```

还可以将默认值重写为 choices 元组中任意可用的选项。

```
$ python argparse_choices.py --cheese provolone  
You have chosen provolone cheese.
```

但是，如果你尝试提供一个在选项列表中不可用的值，将会获得一个错误。

```
$ python argparse_choices.py --cheese pepperjack  
usage: argparse_choices.py [-h] [--cheese {american,cheddar,provolone,swiss}]  
argparse_choices.py: error: argument --cheese: invalid choice: 'pepperjack'  
(choose from 'american', 'cheddar', 'provolone', 'swiss')
```

### 接受多个值

argparse 的一个额外功能是可以指定一个选项接受多个参数。可以设置一个选项接受非绑定数量的参数，或确切几个参数。可以使用 add\_argument 方法的 nargs 关键字参数实现这一点。

nargs 最简单的用法是指定一个选项接受特定数量的参数。考虑下面这个简单的脚本，该脚本接受一个期望两个而非一个参数的选项：

```
import argparse  
  
if __name__ == '__main__':  
    parser = argparse.ArgumentParser()  
    parser.add_argument('--madlib',  
        default=['fox', 'dogs'],  
        dest='madlib',  
        help='Two words to place in the madlib.',  
        nargs=2,  
    )  
    args = parser.parse_args()  
  
    print('The quick brown {0} jumped over the '  
        'lazy {1}.format(*args.madlib))
```

给 nargs 发送一个整数意味着选项期望参数的具体个数，并将以列表的形式返回这些参数(注意，如果指定 nargs 的值为 1，仍得到一个列表)。

如果忽略--madlib 参数，将获得在 add\_argument 调用中指定的默认列表。

```
$ python argparse_multiaargs.py  
The quick brown fox jumped over the lazy dogs.
```

提供两个参数会导致默认值被替换为所提供的参数。

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
$ python argparse_muliargs.py --madlib pirate ninjas
The quick brown pirate jumped over the lazy ninjas.
```

但是，如果提供的参数个数不是两个，那么该命令将失败。

```
$ python argparse_muliargs.py --madlib pirate
usage: argparse_muliargs.py [-h] [--madlib MADLIB MADLIB]
argparse_muliargs.py: error: argument --madlib: expected 2 arguments
$ python argparse_muliargs.py --madlib pirate ninjas cowboy
usage: argparse_muliargs.py [-h] [--madlib MADLIB MADLIB]
argparse_muliargs.py: error: unrecognized arguments: cowboy
```

在第一个示例中，`--madlib` 选项只能够使用一个参数，但由于它期望两个参数，因此命令失败。在第二个示例中，`--madlib` 参数成功使用了它所期望的两个参数，但仍然剩下一个位置参数。解析器不知道如何处理该参数，因此该命令同样失败。

或许希望允许任意数量的参数，这可以通过给 `nargs` 关键字参数提供`+`或`*`来实现。值`+`表明选项期望提供一个或更多值，而值`*`表明选项期望提供 0 个或更多值。

考虑下面这个简单的加法脚本：

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--addends',
                        dest='addends',
                        help='Integers to provide a sum of',
                        nargs='+',
                        required=True,
                        type=int,
    )
    args = parser.parse_args()

    print('%s = %d' % (
        ' + '.join([str(i) for i in args.addends]),
        sum(args.addends),
    ))
```

如果运行该脚本，就可以看到它提供了下面的等式：

```
$ python argparse_sum.py --addends 1 2 5
1 + 2 + 5 = 8
$ python argparse_sum.py --addends 1 2
1 + 2 = 3
```

注意，提供给 `nargs` 的值`+`实际上意味着一个或更多值，而不是两个或更多值。该脚本可以仅接收一个参数。

```
$ python argparse_sum.py --addends 1
1 = 1
```

## 4. 位置参数

使用 argparse(而不是使用 optparse)时, 必须显式声明位置参数。如果没有显式声明, 解析器在完成解析后, 并不期望剩下任何参数, 如果参数仍然存在, 则会引发错误。

位置参数的声明等价于选项的声明, 只是省略了开头的连字符。例如, 前面示例中的 --addends 选项的形式就比较糟糕。选项应该是可选的。

以位置参数的方式完成同样的工作非常容易。

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('addends',
        help='Integers to provide a sum of',
        nargs='+',
        type=int,
    )
    args = parser.parse_args()

    print('%s = %d' % (
        ' + '.join([str(i) for i in args.addends]),
        sum(args.addends),
    ))
```

代码几乎相同, 只有--addends 参数被替换为 addends, 没有了双连字符的前缀。这导致解析器期望一个位置参数。

为什么要为位置参数提供名称? (毕竟, optparse 并不需要位置参数名称), 答案在于所提供的名称在程序的--help 输出结果中会用到。

```
$ python cli_script.py --help
usage: cli_script.py [-h] addends [addends ...]

positional arguments:
  addends      Integers to provide a sum of

optional arguments:
  -h, --help  show this help message and exit
```

注意 help 顶部附近的 usage 一行中的单词 addends。它提供了关于该参数所期望值的更多信息。此外, 与由 optparse 提供的 help 不同, 位置参数作为帮助屏幕的一部分被记录。

可以用同样的方式调用该脚本, 这次不使用--addends 选项。

```
$ python cli_script.py 1 2 5
1 + 2 + 5 = 8
```

## 5. 读取文件

当编写 CLI 应用程序时一个常见的需求是读取文件。argparse 模块提供了一个可以被

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

发送给 `add_argument` 方法中 `type` 关键字参数的特殊类，也就是 `argparse.FileType`。

`argparse.FileType` 类期望参数被发送给 Python 的 `open` 函数，不包括文件名称（文件名称由用户在调用程序时提供）。如果你打开文件进行读取，这可能并没有什么。因为 `open` 默认就是只打开文件进行读取。但是，任何在 `add_argument` 函数开始的位置参数之后的传递给 `open` 函数的参数都可以赋值给 `FileType`，并且它们都会被传递给 `open` 方法。

考虑下面这个程序，它从非默认位置读取一个配置文件：

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--config-file',
                        default='/etc/cli_script',
                        dest='config',
                        help='The configuration file to use.',
                        type=argparse.FileType('r'))
)
args = parser.parse_args()

print(args.config.read())
```

默认情况下将会从 `/etc/cli_script` 读取，但可以使用 `-c` 或 `--config-file` 选项指定一个不同的文件。不需要强迫你自己打开文件并以文本的形式提供选项，仅需要提供一个 `open` 文件对象。

```
$ echo "This is my config file." > foo.txt
$ python cli_script.py --config-file foo.txt
This is my config file.
```

注意该文件期望退出。如果未退出，将获得一个错误。

```
$ python cli_script.py --config-file bar.txt
usage: cli_script.py [-h] [-c CONFIG]
cli_script.py: error: argument -c/--config-file: can't open 'bar.txt':
[Errno 2] No such file or directory: 'bar.txt'
```

## 12.2.3 使用 argparse 的理由

如果仅使用 Python 2.7 或 Python 3.3 以及更高版本，有很多理由支持使用 `argparse` 而不使用 `optparse`。`argparse` 模块基本上支持 `optparse` 的所有功能，并添加了一些额外功能，比如多参数、对文件的良好支持等。

此外，`argparse` 对于位置参数的处理与对选项的处理更加一致，结果是能获得更加健壮的处理与更有用的帮助输出。

`argparse` 唯一且主要的缺点是在老版本的 Python 中不可用。如果你仍然需要支持 Python 2.6 或 Python 3.2，那么仍需要坚持使用 `optparse`。

### 12.3 小结

optparse 与 argparse 模块为需要从命令行读取数据的 Python 程序提供了非常好的支持。

当前从 optparse 到 argparse 的转换很有挑战性，因为你或许会发现自己需要基于过时的模块编写代码，从而支持今天仍然广泛使用的 Python 版本。如果需要从事该领域的工作，你或许需要在一段时间内对这两个模块保持熟悉。

第 13 章将介绍 asyncio 模块，该模块是 Python 3.4 中的新模块，用于支持异步工作。

尚学堂.百战程序员  
[www.itbaizhan.cn](http://www.itbaizhan.cn)



# 第 13 章

## asyncio 模块

通常情况下，大多数 Python 应用程序都按顺序执行。也就是说，应用程序从定义的入口点执行到定义的出口点，每次执行都是从开始到结束的一个单独进程。

但对于很多异步语言却相反，这类语言包括 JavaScript 和 Go。比如说，JavaScript 高度依赖异步机制，所有后台发起的 Web 请求都使用一个单独的线程，并在数据加载后依赖回调来执行正确的函数。

一个语言是否应该使用同步或异步方式解决大多数问题并没有标准答案，但存在对于特定问题一种模块比另一种更加有效的情况。这也是为什么存在 `asyncio` 模块的原因。`asyncio` 模块使得在 Python 中出现需要异步操作解决的问题时，更加简单。

当前，`asyncio` 还是一个预览版模块。目前估计不会为了向后兼容做一些修改(由于一旦库被加入标准库，Python 会极力避免对其进行修改)，`asyncio` 很可能在 Python 的后续版本中进行大版本修订。

`asyncio` 模块在 Python 3.4 中引入，在 Python 2 中不可用。如果你使用的是 Python 3.3，则可以从 PyPI 中获取该模块；目前该库还不是标准库。因此，如果你希望使用由 `asyncio` 提供的功能，则会使你的程序只能执行在新版的 Python 中。另外，`asyncio` 模块在 Python 3.4 的生命周期中进行了大量开发，因此尽量使用最新的版本。

由于 Python 应用程序是顺序执行的，因此如果你对异步语言了解得不多，那么很多概念对你来说可能会很陌生。本章将会详细解释这些概念。

### 13.1 事件循环

大多数异步应用程序实现异步的基本机制是通过在后台执行的事件循环。当有代码需

要执行时，这些代码才会被注册到事件循环中。

将一个函数注册到事件循环会导致它变为一个任务。事件循环负责在获得任务后，马上执行它。另一种方式是事件循环有时在等待一定时间后，再执行任务。

虽然你可能并不熟悉编写使用事件循环的代码，但你使用的大多数程序却高度依赖该机制。几乎所有的服务器都是一个事件循环。举例来说，一个数据库服务器等待连接和查询进来，并在获得连接和查询后，尽快执行查询。如果两个不同的连接带有两个不同的查询，其会被分配优先级并都被执行。桌面应用程序也是事件驱动的，所显示的屏幕允许在多个位置进行输入并对输入进行响应。大多数视频游戏同样也是事件循环。游戏等待控制输入，并基于输入采取行动。

## 一个简单的事件循环

在大多数情况下，你并不需要自己创建一个事件循环对象。可以通过 `asyncio.get_event_loop()` 函数返回一个 `BaseEventLoop` 对象。实际上，你所获得的是一个子类，具体是哪个子类会根据平台的不同而不同。你并不需要过多考虑实现细节。所有平台上的 API 相同。但是，在某些平台上会有功能限制。

当第一次获得循环对象时，其并未执行。

```
>>> loop = asyncio.get_event_loop()
>>> loop.is_running()
False
```

### 1. 执行循环

下面的事件循环中还没有注册任何内容，但你还是可以执行它：

```
>>> loop.run_forever()
```

然而，有一个小问题。如果执行上述代码，你将失去对 Python 解释器的控制权，这是由于程序陷入了死循环。按 `Ctrl+C` 键来结束循环重新获得解释器的控制权(当然，这会使循环停止)。

遗憾的是，`asyncio` 并没有一个“触发并遗忘”的方法，可以在另外的线程中执行循环。对于大多数应用程序代码来说，这并不是一个非常大的障碍，因为你编写一个服务或者守护程序的目的或许是在前台执行，并等待其他进程发起命令。

然而，对于测试或实验来说，这就会带来相当大的挑战，大多数 `asyncio` 的方法实际上并不是线程安全的。对于本章的大多数示例，只需要通过不陷入死循环即可避免这一点。

### 2. 注册任务并执行循环

任务主要使用 `call_soon` 注册到循环，注册顺序是 FIFO(先进先出)队列。因此，本章的大多数示例会在末尾加入一个任务，用于停止循环，如下所示：

```
>>> import functools
>>> def hello_world():
```

```
...     print('Hello world!')
```

```
...     ...
```

```
>>> def stop_loop(loop):
```

```
...     print('Stopping loop.')
```

```
...     loop.stop()
```

```
...     ...
```

```
>>> loop.call_soon(hello_world)
```

```
Handle(<function hello_world at 0x1003c0b70>, ())
```

```
>>> loop.call_soon(functools.partial(stop_loop, loop))
```

```
Handle(functools.partial(<function stop_loop at 0x101ccf268>,
```

```
    <asyncio.unix_events._UnixSelectorEventLoop
```

```
    object at 0x1007399e8>), ())
```

```
>>> loop.run_forever()
```

```
Hello world!
```

```
Stopping loop.
```

```
>>>
```

在本例中，将 `hello_world` 函数注册到循环，然后再注册 `stop_loop` 函数。当循环开始时(使用 `loop.run_forever()`)，这两个任务都按照顺序执行。由于第二个任务用于停止循环，因此在该任务完成后，它会退出循环。

### 3. 延迟调用

也可以注册一个晚一点被调用的任务。可以通过 `call_later` 方法实现这一点，该方法接受延迟时间(单位是秒)和被调用的函数名称作为参数。

```
>>> loop.call_later(10, hello_world)
```

```
TimerHandle(60172.411042585, <function hello_world at 0x1003c0b70>, ())
```

```
>>> loop.call_later(20, functools.partial(stop_loop, loop))
```

```
TimerHandle(60194.829461844, functools.partial(
```

```
    <function stop_loop at 0x101ccf268>,
```

```
    <asyncio.unix_events._UnixSelectorEventLoop object at 0x1007399e8>), ())
```

```
>>> loop.run_forever()
```

注意，同一时间有可能会出现两个或者更多个延迟调用。如果发生了这种情况，那么先后顺序就无法确定。

### 4. 偏函数(Partial)

你或许已经注意到了上面示例中使用的 `functools.partial` 方法。大多数接受函数的 `asyncio` 方法仅仅接受函数对象(或其他被调用元素)，但这些函数在被调用时并没有带参数。`functools.partial` 方法就是为了解决该问题。`partial` 方法本身接受参数与关键字参数，这些参数在底层函数被调用时被传给底层函数。

例如，上面示例中的 `hello_world` 函数实际上完全没有必要。该函数与 `functools.partial(print, 'Hello world!')` 等价。因此上面的示例可以写成下面的形式：

```
>>> import functools
```

```
>>> def stop_loop(loop):
```

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
...     print('Stopping loop.')
...     loop.stop()
...
>>> loop.call_soon(functools.partial(print, 'Hello world! '))
Handle(functools.partial(<built-in function print>, 'Hello world!'), ())
>>> loop.call_soon(functools.partial(stop_loop, loop))
Handle(functools.partial(<function stop_loop at 0x101ccf268>,
    <asyncio.unix_events._UnixSelectorEventLoop object
     at 0x1007399e8>), ())
>>> loop.run_forever()
Hello world!
Stopping loop.
>>>
```

那为什么要使用 `partial` 方法呢？毕竟，通常可以将这类调用封装到不需要参数的函数中。答案就是调试时更有用。`Partial` 对象知道调用函数用的是哪个参数。`partial` 函数以数据的形式表示参数，`partial` 在被调用时使用这些数据执行合适的函数调用。与之相反，`hello_world` 函数仅仅是一个函数。函数内的方法调用是代码。并不存在查看 `hello_world` 函数并取出其底层数据的简单方法。

可以通过创建一个 `partial` 函数，然后查看其底层函数与参数找出函数与偏函数之间的区别：

```
>>> partial = functools.partial(stop_loop, loop)
>>> partial.func
<function stop_loop at 0x10223e488>
>>> partial.args
(<asyncio.unix_events._UnixSelectorEventLoop object at 0x102238b70>,)
```

## 5. 任务结束前执行循环

在任务结束之前一直执行循环也是可能的，如下所示：

```
>>> @asyncio.coroutine
... def trivial():
...     return 'Hello world!'
...
>>> loop.run_until_complete(trivial())
'Hello world!'
```

在本例中，`@asyncio.coroutine` 装饰器将普通的 Python 函数转换为一个协程(之后会对其实行详解)。当调用 `run_until_complete` 时，会将任务注册并在任务结束前执行循环。由于该任务是队列中的唯一任务，因此其完成后会退出循环，并返回该任务的结果。

## 6. 执行一个后台循环

通过使用 Python 标准库中的 `threading` 模块，在后台执行一个事件循环也是可能的。

```
>>> import asyncio
>>> import threading
```

```
>>>
>>> def run_loop_forever_in_background(loop):
...     def thread_func(l):
...         asyncio.set_event_loop(l)
...         l.run_forever()
...     thread = threading.Thread(target=thread_func, args=(loop,))
...     thread.start()
...     return thread
...
>>>
>>> loop = asyncio.get_event_loop()
>>> run_loop_forever_in_background(loop)
<Thread(Thread-1, started 4344254464)>
>>>
>>> loop.is_running()
True
```

注意，该示例对于初学者来说很有用，但你几乎不会应用在最终应用程序中(其原因在于，比如，停止循环将变得非常困难；`loop.stop` 将不再生效)。但对于学习来说，该示例没问题。

该循环相对让人感觉无聊。毕竟，在循环执行时，并没有完成任何工作。还未将任何任务注册到该循环中。考虑下面的示例，当把任务注册到循环并令其立刻执行时会发生什么。

```
>>> loop.call_soon_threadsafe(functools.partial(print, 'Hello world'))
Handle(functools.partial(<built-in function print>, 'Hello world'), ())
>>> Hello world
```

输出结果有点让人迷惑。首先，调用 `call_soon_threadsafe` 方法。该方法用于通知循环立刻异步执行给定函数。注意，因为很少会利用线程执行事件循环，所以在大多数情况下，仅仅使用 `call_soon` 函数就已足够。

`call_soon_threadsafe` 函数返回一个 `Handle` 对象。该对象只有一个方法：`cancel`。在合适时，完全可以取消任务。

接下来，是`>>>`提示(意味着解释器等待输入)，然后是 `Hello world`。该行由前面的函数调用打印到屏幕中，并在提示符被写到屏幕之后才打印到屏幕上。

## 13.2 协程

在 `asyncio` 中使用的大多数函数都是协程(coroutines)。协程是一种被设计用于在事件循环中执行的特殊函数。此外，如果创建了协程但并未执行它，那么将会在日志中记录一个错误。

注意：本小节的讨论基于 Python 3.4，在 Python 3.5 中可能会有变更。

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

可以通过`@asyncio.coroutine` 将一个函数装饰为协程。请考虑下面这个使用事件处理程序的`run_until_complete` 执行简单协程的示例：

```
>>> import asyncio
>>> @asyncio.coroutine
... def coro_sum(*args):
...     answer = 0
...     for i in args:
...         answer += i
...     return answer
...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(coro_sum(1, 2, 3, 4, 5))
15
```

这里创建的`coro_sum` 函数不再是一个普通的函数，而是一个协程，由事件循环调用。值得注意的是，不能再以常规方式调用该函数并返回预期结果。

```
>>> coro_sum(1, 2, 3, 4, 5)
<generator object coro at 0x104056e10>
```

协程实际上是一个由事件循环消费的特殊生成器。这也是为什么`run_until_complete` 方法可以接受的参数看起来像一个标准函数调用的原因所在。函数此时实际并未执行。由事件循环消费生成器，并最终返回结果。

底层实际上看起来类似如下代码：

```
>>> try:
...     next(coro_sum(1, 2, 3, 4, 5))
... except StopIteration as ex:
...     ex.value
...
15
```

生成器并未返回任何值。而是立刻引发了`StopIteration` 异常。`StopIteration` 被赋予函数的返回值，然后事件循环可以提取该值并正确处理该值。

## 嵌套的协程

协程提供了一种以模仿顺序编程的方式来调用其他协程的特殊机制(或者说是 Future 实例，马上会讨论到)。通过使用`yield from` 语句，一个协程可以执行另外一个协程，并由语句返回结果。这是一种以顺序方式编写异步代码的可用机制。

下面是一个使用`yield from` 关键字在一个协程中调用另一个协程的简单示例：

```
>>> import asyncio
>>> @asyncio.coroutine
... def nested(*args):
...     print('The `nested` function ran with args: %r' % (args,))
```

```
...     return [i + 1 for i in args]

...
>>> @asyncio.coroutine
... def outer(*args):
...     print('The `outer` function ran with args: %r' % (args,))
...     answer = yield from nested(*[i * 2 for i in args])
...     return answer

...
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(outer(2, 3, 5, 8))
The `outer` function ran with args: (2, 3, 5, 8)
The `nested` function ran with args: (4, 6, 10, 16)
[5, 7, 11, 17]
```

这里有两个协程，名称为 `outer` 的协程使用 `yield from` 语法调用名称为 `nested` 的协程。可以通过输出到标准输出的结果发现两个协程都已执行，最终结果在 `outer` 的结尾返回。

顺便提一下，底层所发生的事情是名称为 `outer` 的协程在遇到 `yield from` 语句时会挂起。名称为 `nested` 的协程被放入事件循环并执行。`outer` 协程在 `nested` 完成并返回结果之前不会继续执行。

有几件事值得注意。首先，`yield from` 语句返回它执行协程的结果。这就是为什么在示例中会看到为一个变量赋值的原因。

其次，为什么不直接调用函数？如果是过程式函数，直接调用并没有问题，但这是一个协程。直接调用协程会返回一个生成器，而不是返回值。可以将 `nested` 写成一个标准函数，但考虑下面这种情况，你也可能希望将其直接赋值给事件循环。

```
>>> loop.run_until_complete(nested(5, 10, 15))
The `nested` function ran with args: (5, 10, 15)
[6, 11, 16]
```

让一个协程使用 `yield from` 调用另一个协程的功能就是为了解决该问题。这提升了重用协程的可能性。

### 13.3 Future 对象与 Task 对象

由于使用 `asyncio` 完成的大多工作都是异步的，因此你在处理以异步方式执行函数的返回值时必须小心。为此，`yield from` 语句提供了一种处理方式，但另外一些时候需要其他处理方式，比如说，希望并行执行异步函数。

在顺序编程中，返回值很直接。执行一个函数，然后返回结果。但在异步编程中，函数在之前返回值，那么返回值会发生什么？并没有明确指出将返回值返回给哪一个调用者。

#### 13.3.1 Future 对象

在遇到特殊问题时的一种对应机制是使用 `Future` 对象。本质上讲，`Future` 是一个用于

通知异步函数状态的对象。这包括函数的状态——函数状态是执行中、已完成或是已取消。还包括函数的结果，或者是当函数引发异常时，返回对应的异常和回溯。

Future 是一个独立的对象，并不依赖正在执行的函数。该对象仅仅用于存储状态和结果信息，此外别无它用。

### 13.3.2 Task 对象

Task 对象是 Future 对象的子类，也是你在使用 asyncio 编程时最常使用的对象。每当一个协程在事件循环中被安排执行后，协程就会被一个 Task 对象包装。因此，在之前的示例中，当你调用 run\_until\_complete 并传递一个协程时，该协程会被包装到一个 Task 对象中并执行。Task 对象的任务是存储结果并为 yield from 语句提供值。

run\_until\_complete 方法并不是将一个协程包装到类中的唯一方式(甚至不是主要方式)。毕竟，在很多程序中，事件循环一直在执行。在这类系统中如何将任务放入事件循环？

完成这类工作的主要方式是使用 asyncio.async 方法。该方法会将协程放入事件循环，并返回对应的 Task 对象。

**注意：**如果你使用的是 Python 3.4.4 以上的版本，请使用 ensure\_future 而不是 asyncio.async。但如果你使用的是 Python 3.4.3，则继续使用 asyncio。

为了演示这一点，首先获取事件循环，然后编写一个普通的协程，如下所示：

```
>>> import asyncio  
>>>  
>>> @asyncio.coroutine  
... def make_tea(variety):  
...     print('Now making %s tea.' % variety)  
...     asyncio.get_event_loop().stop()  
...     return '%s tea' % variety  
...  
>>>
```

这里仍然使用一个普通任务，但目前为止你还没见过的写法是该任务会停止事件循环。这是解决在开始循环后(使用 run\_forever 方法)循环将无限执行的好方法。

接下来，将任务注册到事件循环。

```
>>> task = asyncio.async(make_tea('chamomile'))
```

这就是将任务注册到循环中所需的代码，但由于循环并未执行，因此该代码目前还无法执行。实际上，可以通过 done 与 result 方法查看 task 对象，如下所示：

```
>>> task.done()  
False  
>>> task.result()  
Traceback (most recent call last):
```

www.itbaizhan.cn

```
File "<stdin>", line 1, in <module>
File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/asyncio/
    futures.py", line 237, in result
        raise InvalidStateError('Result is not ready.')
asyncio.futures.InvalidStateError: Result is not ready.
```

接下来，需要开始循环。现在使用 `run_forever` 方法开始循环就没有问题了；由于调用了 `loop.stop()`，因此在任务完成后，`task` 将立即停止执行。

```
>>> loop = asyncio.get_event_loop()
>>> loop.run_forever()
Now making chamomile tea.
>>>
```

果然，开始循环后，执行任务，然后立刻停止。现在如果你检查 `task` 变量，将会得到不同的结果。

```
>>> task.done()
True
>>> task.result()
'chamomile tea'
```

当你使用 `asyncio.async` 创建一个 `Task` 对象时，将会获得一个 `Task` 对象。可以在任何时候检查该对象，从而获得该对象的状态或结果。

## 13.4 回调

`Future` 对象(以及 `Task` 对象，因为 `Task` 是 `Future` 的子类)的另一个功能是能够将回调注册到 `Future`。回调就是一个在 `Future` 完成后执行的一个函数(或协程)，该函数接受 `Future` 作为参数。

在某种程度上，回调代表了一个与 `yield from` 模型相反的模型。当一个协程使用 `yield from` 时，该协程会确保嵌套协程在其之前或同时被执行。当注册一个回调时，顺序则相反。回调被附加到原始的任务，它在任务执行之后再执行回调。

可以通过使用对象的 `add_done_callback` 方法将一个回调添加到任何 `Future` 对象。回调接受一个参数，即为 `Future` 对象本身(该对象包含状态和结果信息，如果存在底层任务，则为底层任务的状态或结果)。

请考虑下面回调的示例：

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
... def make_tea(variety):
...     print('Now making %s tea.' % variety)
...     return '%s tea' % variety
```

```
...
>>> def confirm_tea(future):
...     print('The %s is made.' % future.result())
...
>>> task = asyncio.async(make_tea('green'))
>>> task.add_done_callback(confirm_tea)
>>>
>>> loop.run_until_complete(task)
Now making green tea.
The green tea is made.
'green tea'
```

该示例首先生成一个 `make_tea` 协程，与之前示例相同，唯一的区别是本协程不会停止循环。

接下来，注意 `confirm_tea` 函数。该函数并不是一个协程，仅仅是普通函数。实际上，你并不能将一个协程作为回调发送。如果尝试执行循环，将会引发一个异常。一旦回调执行，该函数接收被注册到其中的 Future 对象(在本例中是 `task` 变量)。Future 对象包含协程的结果——在本例中是字符串'green tea'。

最后，注意对于 `add_done_callback` 的调用。这也是将 `confirm_tea` 方法作为回调赋值给 `task` 的地方。另外，还需要注意该函数被赋值给 `task`(对于一个协程的特殊调用)，而不是赋值给协程本身。如果用调用同一个协程的 `asyncio.async` 方法将另一个任务注册到循环，该任务不会得到该回调。

输出结果展示两个函数都已按照预期的顺序执行。返回值是来自 `make_tea` 的返回值。这就是 `run_until_complete` 的工作机制。

### 13.4.1 不保证成功

有一件重要的事情值得注意。Future 仅仅是被执行，但并不能保证它能够执行成功。本例仅仅是假设 `future.result()` 的结果值被正确返回，但事实可能并非如此。Task 的执行可能会引发异常，在这种情况下，尝试访问 `future.result()` 将会引发该异常。

同样地，也有可能取消任务(使用 `Future.cancel()` 方法或其他方式)。如果这么做，则任务会被标记为 `Cancelled`，会安排回调。在这种情况下，尝试访问 `future.result()` 将会引发 `CancelledError` 异常。

### 13.4.2 幕后

在内部，由 `asyncio` 通知 Future 对象已完成。Future 对象接受接下来对所有已注册到 Future 的回调，并对其调用 `call_soon_threadsafe` 函数。

需要注意的是，对于回调来说并不能保证执行顺序。完全有可能(且不会引起问题)将多个回调注册到同一个任务中。然而，你无法控制是否只执行某些回调以及回调之间的执行顺序。

### 13.4.3 带参数的回调

如你所见，回调系统的一个限制是回调接收作为位置参数的 Future 对象，但不接收其他参数。

可以通过使用 `functools.partial` 函数将其他参数发送给回调。但如果这样做，回调必须仍然接受作为位置参数的 Future。实际上，在回调被调用之前，Future 会附加到位置参数列表的结尾处。

请考虑下面接受其他参数的回调示例：

```
>>> import asyncio
>>> import functools
>>>
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
... def make_tea(variety):
...     print('Now making %s tea.' % variety)
...     return '%s tea' % variety
...
>>> def add_ingredient(ingredient, future):
...     print('Now adding %s to the %s.' % (ingredient, future.result()))
...
>>>
>>> task = asyncio.async(make_tea('herbal'))
>>> task.add_done_callback(functools.partial(add_ingredient, 'honey'))
>>>
>>> loop.run_until_complete(task)
Now making herbal tea.
Now adding honey to the herbal tea.
'herbal tea'
```

该示例的大部分内容与前面示例中的相同。唯一显著的区别是回调的注册方式是通过实例化一个带有位置参数('honey')的 `functools.partial` 对象实现的，而不是直接传递函数对象(如之前示例那样)。

同样要注意的是，所编写的 `add_ingredient` 函数接受两个位置参数，但 `partial` 仅接受一个参数。在使用 `partial` 时，Future 对象作为最后一个位置参数被发送。`add_ingredient` 的函数签名反映了这一点。

## 13.5 任务聚合

`asyncio` 模块提供了一种聚合任务的便利方法。聚合任务主要归因于两个原因。第一个原因是在一组任务中的任何任务完成后采取某些行动。第二个原因是在所有任务都完成后采取某些行动。

## 13.5.1 聚集任务

asyncio 为聚集任务目的提供的第一种机制是通过 `gather` 函数。`gather` 函数接受一系列协程或任务，并返回将那些任务聚合后的单个任务(包装其接收的任何适用协程)。

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
... def make_tea(variety):
...     print('Now making %s tea.' % variety)
...     return '%s tea' % variety
...
...
>>> meta_task = asyncio.gather(
...     make_tea('chamomile'),
...     make_tea('green'),
...     make_tea('herbal')
... )
...
...
>>> meta_task.done()
False
>>>
>>> loop.run_until_complete(meta_task)
Now making chamomile tea.
Now making herbal tea.
Now making green tea.
['chamomile tea', 'green tea', 'herbal tea']
>>> meta_task.done()
True
```

在本示例中，`asyncio.gather` 函数接收 3 个协程对象。在该函数内部将所有协程包装到一个任务中，并返回一个充当 3 个协程聚合的单独任务。

注意，`meta_task` 对象高效地对 3 个被聚集的任务进行调度。一旦开始执行循环，3 个子任务全部开始执行。

本示例中通过 `asyncio.gather` 创建的任务，返回的结果总会是一个列表，该列表包含被聚集的单个任务的结果。返回的列表中任务的顺序保证与任务聚集的顺序一致(但任务的执行并不保证按照该顺序执行)。因此，返回的字符串列表顺序与在 `asyncio.gather` 调用中协程的注册顺序保持一致。

`asyncio.gather` 还提供了针对作为整体的一组任务添加回调的机制，而不是针对每一个单独对象添加回调。如果你只想在所有任务完成后执行一次回调，但并不关心任务完成的顺序时该怎么做呢？

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
```

```
... def make_tea(variety):
...     print('Now making %s tea.' % variety)
...     return '%s tea' % variety
...
>>> def mix(future):
...     print('Mixing the %s together.' % ' '.join(future.result()))
...
>>> meta_task = asyncio.gather(make_tea('herbal'), make_tea('green'))
>>> meta_task.add_done_callback(mix)
>>>
>>> loop.run_until_complete(meta_task)
Now making green tea.
Now making herbal tea.
Mixing the green tea and herbal tea together.
['green tea', 'herbal tea']
```

当调用 `run_until_complete` 时首先会执行两个被聚集到 `meta_task` 的任务。最后，只有在这两个任务都执行完成后，才执行 `mix` 函数。这是由于只有 `meta_task` 中的所有任务完成后，`meta_task` 才算完成，因此只有在这两个单独任务都完成后才会触发回调。

你还可以看到 `mix` 函数所接收的 Future 对象是 `meta_task`，而不是单个任务。因此，其 `result` 方法返回的是这两个任务返回值连接后的列表。

### 13.5.2 等待任务

`asyncio` 模块提供的另一个工具是内置的 `wait` 协程。`asyncio.wait` 协程接受一系列协程或任务(在任务中包装任意协程)，一旦完成后就返回结果。注意该协程的签名与 `asyncio.gather` 不同。每一个协程或任务都是 `gather` 的一个单独位置参数，而 `wait` 接受一个列表作为参数。

此外，`wait` 接受一个用于在任何任务完成后返回的参数，而无须等待所有任务完成。无论该标记位是否设置，`wait` 方法总是返回两部分：第一个元素为已完成的 Future 对象；第二个元素为还未完成的部分。

请考虑下面类似之前使用 `asyncio.gather` 的一个例子：

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
... def make_tea(variety):
...     print('Now making %s tea.' % variety)
...     return '%s tea' % variety
...
>>> coro = asyncio.wait([make_tea('chamomile'), make_tea('herbal')])
>>>
>>> loop.run_until_complete(coro)
Now making chamomile tea.
Now making herbal tea.
```

# 尚学堂·百战程序员

第IV部分 其他高级主题

[www.itbaizhan.cn](http://www.itbaizhan.cn)

```
({Task(<coro>)<result='herbal tea'>, Task(<coro>)<result='chamomile tea'>}, set())
```

注意，该代码与前面的示例有一些细微区别。首先，与 `gather` 方法不同，`wait` 方法返回一个协程。该协程带有值，例如，你可以在 `yield from` 语句中使用该协程。

另一方面，你无法将回调直接附加到 `wait` 返回的协程上。如果你希望这么做，必须使用 `asyncio.async` 将该协程包装到一个任务中。

另外，返回结果也不同。`asyncio.gather` 函数将结果聚合到一个列表中并返回。而 `asyncio.wait` 的返回值是一个包含 `Future` 对象(其自身包含返回值)的两部分容器。此外，`Future` 对象被重新组织。`asyncio.wait` 协程将其分为两部分——一部分是已经完成的；另一部分是还未完成的。由于集合自身是一个未排序的结构，这意味着必须依赖 `Future` 对象来找出哪一个结果与哪一个任务对应。

## 1. 超时

也可以使得 `asyncio.wait` 协程在指定时间后返回结果，无论所有任务是否都已完成。为此，将 `timeout` 关键字参数传递给 `asyncio.wait`。

```
>>> import asyncio  
>>> loop = asyncio.get_event_loop()  
>>>  
>>> coro = asyncio.wait([asyncio.sleep(5), asyncio.sleep(1)], timeout=3)  
>>> loop.run_until_complete(coro)  
({Task(<sleep>)<result=None>}, {Task(<sleep>)<PENDING>})
```

在本例中，仅仅是使用由 `asyncio` 模块 `asyncio.sleep` 提供的一个协程。该协程仅仅等待指定的秒数，然后返回 `None`。在本例中的超时时间设置使得其中一个任务在超时之前完成(第二个任务)，而另一个任务无法完成。

第一个需要注意的区别是两部分中第二个元组现在包含一个未完成的任务；未完成的 `sleep` 协程仍然处于挂起状态。另一个已完成的协程有一个返回值(`None`)。

使用 `timeout` 并不需要等到指定的超时时间过后才完成。如果在超时时间到达之前所有的任务都执行完成，则协程将会立刻完成。

## 2. 等待任意任务

`asyncio.wait` 的一个重要功能是可以在其包含的任意 `Future` 对象完成后，即可返回协程。`asyncio.wait` 函数还接受一个 `return_when` 关键字参数。通过给该关键字传递一个特殊常量 (`asyncio.FIRST_COMPLETED`)，一旦任意任务完成后，即可完成该协程，不再需要等所有任务都完成。

```
>>> import asyncio  
>>> loop = asyncio.get_event_loop()  
>>>  
>>> coro = asyncio.wait([  
...     asyncio.sleep(3),  
...     asyncio.sleep(2),
```

```

...     asyncio.sleep(1),
... ], return_when=asyncio.FIRST_COMPLETED)
>>>
>>> loop.run_until_complete(coro)
({Task(<sleep>)<result=None>},
 {Task(<sleep>)<PENDING>, Task(<sleep>)<PENDING>})

```

在本例中，调用 `asyncio.wait` 的第一个参数是 `asyncio.sleep` 协程列表，这三个协程分别等待 3、2 和 1 秒。当该协程被调用时，会执行所有其包含的任务。只等待 1 秒的 `asyncio.sleep` 协程首先执行完成，从而使得 `wait` 协程完成。因此，返回两部分结果集，其中第一部分结果集只包含一个项(已完成的任务)，第二个结果集包含两个项(仍然挂起的任务)。

### 3. 等待异常

也可以使得在一个任务引发异常，而不是正常完成时，对于 `asyncio.wait` 的调用已经完成。在你希望尽早捕获并处理异常的情况下，这是一个非常有价值的工具。

可以使用之前使用过的 `return_from` 关键字参数触发该行为，但是这次使用 `asyncio.FIRST_EXCEPTION` 常量。

```

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>>
>>> @asyncio.coroutine
... def raise_ex_after(seconds):
...     yield from asyncio.sleep(seconds)
...     raise RuntimeError('Raising an exception.')
...
>>> coro = asyncio.wait([
...     asyncio.sleep(1),
...     raise_ex_after(2),
...     asyncio.sleep(3),
... ], return_when=asyncio.FIRST_EXCEPTION)
>>>
>>> loop.run_until_complete(coro)
({Task(<raise_ex_after>)<exception=RuntimeError('Raising an exception.',)>},
 {Task(<sleep>)<result=None>},
 {Task(<sleep>)<PENDING>})

```

在本例中，`asyncio.wait` 协程在其中一个任务引发异常后立刻停止。这意味着等待 1 秒的 `asyncio.sleep` 成功执行，因此其在返回值的第一个结果集中。`raise_ex_after` 协程也已完成，因此它也在第一个结果集中。然而，事实是该协程触发 `wait` 在等待 3 秒的协程完成之前完成，因此等待 3 秒的协程在第二个(挂起)的结果集中。

有时，所有任务都不引发异常(也是通常最常见的情况)。在这种情况下，就和正常情况一样，需要等待所有任务完成后 `wait` 才完成。

```

>>> import asyncio
>>> loop = asyncio.get_event_loop()

```

```
>>>
>>> coro = asyncio.wait([
...     asyncio.sleep(1),
...     asyncio.sleep(2),
... ], return_when=asyncio.FIRST_EXCEPTION)
>>>
>>> loop.run_until_complete(coro)
({Task(<sleep>)<result=None>, Task(<sleep>)<result=None>}, set())
```

## 13.6 队列

asyncio 模块提供了一些建立在事件循环与 Future 对象的基本代码段之上的通用模式。其中之一是基本的队列系统。

队列是一个由任务执行器处理的任务集合。Python 生态系统包含很多第三方的任务队列工具，其中最流行的一种大概就是 celery。而由 asyncio 模块提供的队列仅仅是基本的队列，并不是一个全功能的队列应用程序，如果需要，开发人员可以基于该基本队列按需求进行自定义开发。

为什么 Queue 是 asyncio 的组成部分？因为 Queue 类提供的方法被用于顺序或异步上下文中。

请首先考虑下面使用 Queue 的简单示例：

```
>>> import asyncio
>>> queue = asyncio.Queue()
>>> queue.put_nowait('foo')
>>> queue.qsize()
1
>>> queue.get_nowait()
'foo'
>>> queue.qsize()
0
```

这次更加简单，示例中并没有包含特别需要异步编程的部分。你甚至无须获取或执行事件循环。该示例就是一个非常直接的先进先出(FIFO)队列。

注意，这里使用了 `put_nowait` 与 `get_nowait` 方法。这两个方法用于立刻从队列中添加或移除项。举个例子，假如你尝试对空队列调用 `get_nowait` 方法，将会得到一个 `QueueEmpty` 异常。

```
>>> queue.get_nowait()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/asyncio/
queues.py", line 206, in get_nowait
raise QueueEmpty
asyncio.QueueEmpty
```

Queue类还提供了一个名称为get的方法。get方法在队列为空时并不会引发异常，而是耐心等待项被添加到队列中，然后再从队列中获得该项并立刻返回。与get\_nowait不同，该方法是一个协程，在异步上下文中执行。

```
>>> import asyncio  
>>> loop = asyncio.get_event_loop()  
>>> queue = asyncio.Queue()  
>>>  
>>> queue.put_nowait('foo')  
>>> loop.run_until_complete(queue.get())  
'foo'
```

在本例中，队列中已经有一项，因此get方法仍然立刻返回。如果队列中没有项，则直接调用loop.run\_until\_complete会永远处于执行状态，并阻塞解释器。

然而，可以使用asyncio.wait中的timeout参数来查看实际的执行情况。

```
>>> import asyncio  
>>> loop = asyncio.get_event_loop()  
>>> queue = asyncio.Queue()  
>>>  
>>> task = asyncio.async(queue.get())  
>>> coro = asyncio.wait([task], timeout=1)  
>>>  
>>> loop.run_until_complete(coro)  
(set(), {Task(<get>)<PENDING>})
```

此时，队列中依然为空，因此从队列中获得项的任务继续执行。你还可以查看task变量的状态。

```
>>> task.done()  
False
```

接下来，入队一项，如下所示：

```
>>> queue.put_nowait('bar')
```

你会注意到该任务并未完成，这是由于事件循环不再处于执行状态。由于任务仍然注册到该事件循环上，因此将一个用于在执行完之后停止循环的回调注册到事件循环上，就可以再次启动任务。

```
>>> import functools  
>>> def stop(l, future):  
...     l.stop()  
  
>>> task.add_done_callback(functools.partial(stop, loop))  
>>>  
>>> loop.run_forever()
```

现在，由于队列中已经包含一项，任务已完成，并且任务的结果为队列中的项('bar')。

```
>>> task.done()  
True  
>>> task.result()  
'bar'
```

## 最大队列长度

允许设置 Queue 对象的最大长度，在创建队列时，可以通过设置 maxsize 关键字参数实现这一点。

```
>>> import asyncio  
>>> queue = asyncio.Queue(maxsize=5)
```

如果设置了最大长度，Queue 将不再允许入队超过最大值的项。调用 put 方法将会等待之前的项被移除之后(并且只能是之后)将项入队。如果在队列满时调用 put\_nowait，将会引发 QueueFull 异常。

## 13.7 服务器

asyncio 模块最常见的用处是创建一个作为守护程序的服务并接受命令。asyncio 模块定义了一个 Protocol 类，用于在接收或失去连接时以及接收到数据时触发对应事件。

此外，事件循环定义了一个 create\_server 方法，用于打开一个 socket 连接，该连接允许将数据发送到事件循环并交给 Protocol 类。

请考虑下面这个简单的服务器示例，该示例不完成任何功能，仅仅是累加数字并关闭服务。

```
import asyncio

class Shutdown(Exception):
    pass

class ServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        self.transport = transport
        self.write('Welcome.')

    def data_received(self, data):
        # Sanity check: Do nothing on empty commands.
        if not data:
            return

        # Commands to this server shall be a single word, with
        # space separated arguments.
        message = data.decode('ascii')
        command = message.strip().split(' ')[0].lower()
        args = message.strip().split(' ')[1:]
```

```
# Sanity check: Verify the presence of the appropriate command.
if not hasattr(self, 'command_%s' % command):
    self.write('Invalid command: %s' % command)
    return

# Run the appropriate command.
try:
    return getattr(self, 'command_%s' % command)(*args)
except Exception as ex:
    self.write('Error: %s\n' % str(ex))

def write(self, msg_string):
    string += '\n'
    self.transport.write(msg_string.encode('ascii', 'ignore'))

def command_add(self, *args):
    args = [int(i) for i in args]
    self.write('%d' % sum(args))

def command_shutdown(self):
    self.write('Okay. Shutting down.')
    raise KeyboardInterrupt

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    coro = loop.create_server(ServerProtocol, '127.0.0.1', 8000)
    asyncio.async(coro)
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
```

该模块有点长，但一些细节值得一提。首先，`ServerProtocol` 类是 `asyncio.Protocol` 的子类。`connection_made` 与 `data_receive` 方法在基类中定义，但不实现任何功能。另外三个方法是自定义方法。

请记住，当在机器之间建立一个 `socket` 连接时，本质上发送的是字节，而不是文本字符串。这里的 `write` 方法用于进行类型转换，因此避免了在每次将数据写入通道时，将文本字符串转换为字节字符串。

该服务器的核心是 `data_received` 方法。其接受一个数据行并尝试对该数据行进行处理。它只接受两个基本命令，任何其他输入的命令都会导致错误。

最后，文件末尾的代码段实际上用于启动服务器，并在本机的特定端口上运行。这就是启动一个服务器并侦听命令所需的所有代码。

可以通过启动一个服务然后在另一个 shell 窗口中使用 `telnet` 连接到服务器来验证服务器是否能够接收命令。

```
$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome.
add 3 5
8
make_tea
Invalid command: make_tea
shutdown
Okay. Shutting down.
Connection closed by foreign host.
```

现在，你有了一个非常简单的服务器，它只能接受两个命令：add 和 shutdown。当你发起的命令服务器无法解析时，就会报错。并且该服务器能够将其自身关闭。

## 13.8 小结

就其核心而言，Python 是一个顺序语言。通过 asyncio 模块，可以使异步功能成为标准库的组成部分。

按照顺序模式编写异步代码是使得 asyncio 有价值的方式之一，但在使用 yield from 语句时在底层的处理实际上还是异步的。然而，如果你希望编写一个异步应用程序，仍然必须理解该模式的优势与弊端。

如你所见，很多方面都与顺序编程不同。你可能不知道任务的执行顺序。还可以主动取消任务。最后，代码可以使用一个回调系统注册执行，而不是按照顺序直接调用函数。上面提到的所有这些都打破了“正常的”Python 编程方法。

尽管如此，如果你有一个健壮的 Python 3 应用程序并需要一定的异步元素，asyncio 对你来说或许是一个正确的选择。

在第 14 章中，你将学习 Python 中有关代码风格的标准和最佳实践。

# 第 14 章

## 代码风格

代码被读的次数远大于被写的次数。

尽管如此，程序员在编写代码时，似乎认为他们在将来不需要维护这些代码，甚至不需要阅读它们。在这种思维下，编写出的代码在数月或者数年后重读时，就变得无法理解。

因此，作为一个程序员(使用任何语言)，你能做出最重要的事情之一就是写出易于阅读的代码。

本章探讨编写易读代码的原因，以及其中一些 Python 社区采用的标准，这些标准使得代码风格与社区能够保持一致。

### 14.1 原则

在开始讨论 Python 社区所采用的具体标准或是由其他人推荐的建议之前，考虑一些总体原则非常重要。

请记住，可读性标准的目标是提升可读性。这些规则存在的目的就是为了帮助人读写代码。

本小节讨论你需要记住的一些原则。

#### 14.1.1 假定你的代码需要维护

你很容易相信在某时自己所完成的工作在未来不需要添加内容或对其进行维护。在编写代码时，你很难预料到未来的需求，并且会低估自己引入 Bug 的倾向。然而，所编写的代码很少不需要修改且一直存在的。

如果你假设自己编写的代码会“一劳永逸”，之后无须再阅读、调试或修补，那么就会

非常容易陷入忽视其他可读性原则的境地，这仅仅是因为你相信“这次编写的代码不会需要修改，所以不用花费精力编写可读性高的代码”。

因此，对自己所写代码无须维护的直觉保持不信任才是上策。稳赚不赔的办法是赌自己将会再次见到自己编写的代码。即使你不维护，那也需要其他人维护。

## 14.1.2 保持一致性

一致性的两个方面分别为：内部一致性和外部一致性。

无论是从代码风格和代码结构层面来讲，代码都要尽量满足内部一致性。无论是哪种格式化规则，代码风格都要贯穿项目保持一致。代码结构的一致性也就是将同样类型的代码放到一起。这样项目容易把控。

代码还应该保持外部一致性。项目与代码的结构应与其他人的保持一致，如果一个新来的开发人员打开你的项目，你不应该让他的反应是：“我从来没见过像这样的东西”。社区指导原则很重要，因为这些原则是开发人员加入到你的项目时所期望的原则。类似的，请认真看待在使用特定框架时完成任务以及组织代码时所采用的标准。

## 14.1.3 考虑对象在程序中的存在方式，尤其是那些带有数据的对象

存在论(Ontology)的主要意思就是“关于存在的研究”。在哲学上(在该领域这个词很常用)存在论是关于现实与存在本质的研究，是形而上学的子集。

而对于写软件程序来说，存在论指的是关注不同“事物”在应用程序中的存在方式。如何在数据库中表示概念？或者是用类结构来表示？

这类问题最终影响你编写或组织代码的方式。是否使用继承或组合来组织两个类之间的关系？使用数据库中的哪个表完成这项功能或是这个列属于哪个表？

这些建议最终归结为“在写代码之前先思考”。尤其是思考程序希望实现的目标，以及应用程序之间如何交互。应用程序是一个对象与数据交互的世界。那么，它们之间的协作需要遵循的规则是什么？

## 14.1.4 不要做重复工作

当编写代码时，请考虑随着时间的推移重复使用的值将会变更的情况。该值是否被用于多个模块或函数中？如果有必要修改，需要花费多大的代价？

同样的原则适用于函数。在应用程序中你是否有大量的重复代码？如果这些重复代码行数较多，可以考虑将其抽象到一个函数中，如果出现修改代码的需求，则更容易管理。

另一方面，对于该原则不要过犹不及。并不是所有值都需要在模块中定义为常量(这样做有损于可读性和可维护性)。请明智判断，不断问自己这样一个问题：“如果需要变更该代码，在所有位置进行变更所需要的的成本是多少”？

## 14.1.5 让注释讲故事

代码是一个故事。在用户与程序交互的过程中，从开始到结束，代码是所发生故事的说明。程序从某一点开始(可能带有一些输入)，沿着一系列“选择自己的冒险故事”步骤到达终点，并结束(很可能带有一些输出结果)。

采用的注释风格可以是在每一些行代码之前就添加一段注释，用于解释代码的功能。如果代码是一个故事，那么注释就是故事的解释与旁白。

如果叙事式注释做得很好，读者就可以通过阅读注释了解故事，从而解析代码(例如，当尝试解决问题或维护代码时)，然后可以从零开始快速了解所需维护的代码，这样就可以专注于代码本身所表示的意义。

叙事式注释还可以帮助解释代码意图。它可以回答这样的问题：“这段代码的编写者希望完成的目标是什么？”偶尔，还可以帮助回答问题：“为什么以这种方式完成工作”？这些问题是在你阅读代码时很自然会问的问题，为这些问题提供答案将有助于了解这些内容。

因此，注释用于解释代码中不显而易见或复杂部分的原理。如果使用了有点复杂的算法，请考虑将指向解释模式文章的链接以及其他使用示例加入注释。

## 14.1.6 奥卡姆剃刀原则

通俗来讲，编写可维护代码最重要的原则就是奥卡姆剃刀原则：最简单的解决方案通常是最好的。在他的“Python 之禅”博文页面中(<https://www.python.org/dev/peps/pep-0020/>)，集合了一些编程格言(例如，在 Python 控制台中输入“import this”就可以看到这篇博文)，Tim Peters 也包括了类似下面这句格言“如果你无法向人描述你的方案，那肯定不是一个好方案”。

上述原则在代码如何运行与代码外观层面上都生效。当提到代码运行时，简单的系统更加容易维护。实现的简单化意味着更少引入复杂的 Bug，那些维护你代码的人(包括你自己)更容易凭直觉理解代码所代表的含义，并在不踩坑的前提下为程序增加代码。

至于代码的外观，请记住，尽可能使得阅读代码就好像是在了解代码所做工作的故事，而不是为了解析词汇。词汇是手段，而故事才是最终目的。写一条诸如“不要使用三元运算符”很容易。然而仅遵循这些规则(虽然有价值)并不是代码明晰的充分条件。应该重点关注以尽可能简洁的方式编写和组织代码。

## 14.2 标准

Python 社区大部分遵循所谓的 PEP 8(<https://www.python.org/dev/peps/pep-0008/>)指导原则，PEP 8 由 Guido van Rossum(Python 之父)编写并被大多数主流的 Python 项目所采用，其中包括 Python 标准库。

PEP 8 的普遍性是其强大的原因之一。该标准被大多数社区项目采纳，因此你可以预

计你遇到的大多数 Python 代码都遵循该标准。当以这种方式编写代码时，代码会更容易阅读，也更容易编写。

## 14.2.1 简洁的规则

PEP 8 中的大多数指导原则都很简单明了。部分重点如下：

- 使用 4 个空格缩进。不要使用制表符(\t)。
  - 变量应该使用下划线连接，不使用骆驼式命名风格(使用 my\_var 而不是 myVar)。  
类名称以字母开头就是骆驼式命名风格(例如：MyClass)。
  - 如果一个变量的用处是：“仅内部使用”，那么在变量名称之前加上下划线。
  - 在运算符前后加上单空格(例如， $x + y$ ，不是  $x+y$ )，也包含赋值运算符( $z = 3$  而不是  $z=3$ )。只有在关键字参数的情况下该规则不适用，在这种情况下，空格可以省略。
  - 在列表和字典中省略不必要的空白(例如：[1, 1, 2, 3, 5]而不是[ 1, 1, 2, 3, 5 ])。
- 请阅读 Python 代码风格指南，获得更多示例以及有关这些规则的更多讨论。

## 14.2.2 文档字符串

请记住，在 Python 中，如果在一个函数或类中的第一个语句是一个字符串，该字符串会自动赋值给一个特殊的 `_doc_` 变量，该变量在调用 `Help`(和一些其他的类)时会使用。

PEP 8 规定文档字符串是必须的。

```
"""Do X, Y, and Z, then return the result."""
```

该句子与作为描述的文档字符串的对比：

```
"""Does X, Y, and Z, then returns the result."""
```

如果文档字符串是一行，那么需要在类或函数体之前加空行。如果文档字符串有多行，则将结束的双引号单独放一行。

## 14.2.3 空行

空行用于逻辑分块。

PEP 8 规定“最高级”的类和函数定义之间有两个空行。

```
class A(object):  
    pass
```

```
class B(object):  
    pass
```

除了最高级之外，PEP 8 还规定类和函数的定义以一个空行分隔。

```
class C(object):
    def foo(self):
        pass

    def bar(self):
        pass
```

在函数或其他代码块中使用单空行分隔逻辑段是合理的。请考虑在逻辑段之前使用注释解释代码段的作用。

#### 14.2.4 导入

Python 允许绝对路径导入和相对路径导入。在 Python 2 中，解释器会尝试相对导入，如果找不到路径，然后再尝试使用绝对导入。

在 Python 3 中，使用特殊语法来标记相对导入——以(.)开头——“正常”的导入方式只会尝试相对路径。Python 3 的语法在 Python 2.6 以后版本中可以使用。另外，可以使用 from \_future\_ import absolute\_import 关闭隐式的相对路径导入。

如果可能，尽量使用绝对路径导入。如果必须使用相对路径，请使用显式的导入风格。如果你为 Python 2.6 或 2.7 编写代码，请考虑选择 Python 3 中的显式风格。

当导入模块时，每个模块应该单独占一行。

```
import os
import sys
```

然而，如果从同一个模块中导入多个名称，可以将这些名称分组到一行中。

```
from datetime import date, datetime, timedelta
```

此外，虽然 PEP 8 并没有强制要求，但可以考虑以包来源的方式将导入分组。对于每一组，按照字母表顺序排序。

另外，在导入时，请不要忘了使用 as 关键字给导入的内容起别名。

```
from foo.bar import really_long_name as name
```

这使得你可以简化被频繁使用的长名称或不规范命名的名称。

当导入被频繁使用的名称和原始名称无论何种原因不规范时，别名就很有价值了。

另一方面，请记住当这样做时，会在模块中掩盖原始名称，如果没必要使用别名，这样做反而会使代码变得不清晰。无论是使用何种工具，都要做到具体情况，具体分析。

#### 14.2.5 变量

如前所述，变量名称使用下划线连接，而不要使用骆驼代码风格(例如，是 my\_val 而不是 myVal)。此外，起一个具有描述性的名称同样重要。

通常情况下，使用非常短的变量名称并不合适，虽然某些情况下这样做也能接受，比

如在循环中的变量(例如, for k, v in mydict.items())。

应避免函数的命名与 Python 语言中的常用名称重复,就算是解释器允许也不行。无论在何种情况下,都不要命名某个对象为 sum 或 print。类似的,应避免 list 或 dict 之类的名称。

如果必须要命名一个与 Python 类型和关键字同名的变量,惯例是在变量名称之后加下划线;相比修改名称的拼写来说,这样做更加可行。例如,如果将一个类作为参数传递给一个函数,那么参数名称应该为 class\_,而不是 klass(一个例外是静态方法,它按照惯例使用 cls 作为第一个参数)。

## 14.2.6 注释

注释的编写应该使用英语的完整语句,注释块应放在相关的代码之前。应正确使用首字母大写和语法,以及保证拼写正确。

同时,要保证注释最新。如果代码变更,那么注释可能也需要随之变更。你应该不希望注释与代码实际要表示的意思相反,这很容易导致混淆。

模块可能包含一个注释头,通常由版本控制系统生成,其中包含文件版本的信息。这使得查看文件是否被修改变得容易,尤其是在将模块分发给别人使用时。

## 14.2.7 行长度

Python 代码风格最有争议(也是最常被拒绝使用的)的方面是对行长度的限制。PEP 8 要求行长度不超过 79 个字符,文档字符串不超过 72 个字符。

该规则让很多开发人员感到沮丧,这些开发人员认为我们生活在一个 27 寸宽屏显示器的时代。GitHub 是一个非常流行的共享代码的网站,所使用的窗口宽度是 120 个字符。

而该规则的支持者指出,很多人依然使用窄屏或 80 字符长度的终端,甚至仅仅是将代码窗口的宽度设置为小于屏幕宽度。

争论很难有一个结果。总之,无论是遵循 79 字符宽度的标准或是更宽的标准,你都应该按照项目标准的规范进行编码。当行度过长时,你应该知道如何处理代码。

使用圆括号是封装单行很长的代码的最佳方式,如下所示:

```
if (really_long_identifier_that_maybe_should_be_shorter_and  
     other_really_long_identifier_that_maybe_should_be_shorter):  
    do_something()
```

只要有可能,就应该使用该方法,而不是在换行符之前使用 \ 字符。注意,在使用诸如 and 之类的操作符时,应尽可能将其置于换行符之前。

封装函数调用也是可以的。PEP 8 列出了许多完成封装的可接受方式。一般规则是使同级别行的缩进保持一致。

```
really_long_function_name(  
    categories=[  
        x.y.COMMON_PHRASES,
```

```
x.y.FONT_PREVIEW_PHRASES,  
],  
phrase='The quick brown fox jumped over the lazy dogs.',  
)
```

当在函数调用、列表或字典中分行时，在行的结尾部分添加逗号。

### 14.3 小结

大多数时候，一年后阅读你代码的人可能就是你自己。你的记忆力并不像一开始编写代码时那么好。在编写代码时，若不留心代码的可读性与可维护性自然会使代码难以阅读和维护。

通观本书，你学会了如何使用 Python 中的多种模块、类与结构。当需要决定如何解决问题时，请记住，调试代码比写代码更有技术含量。

因此，应以代码尽可能简洁和可读为目标。一年后的你将会感谢自己。当然，你的同事以及下属也会感谢你。