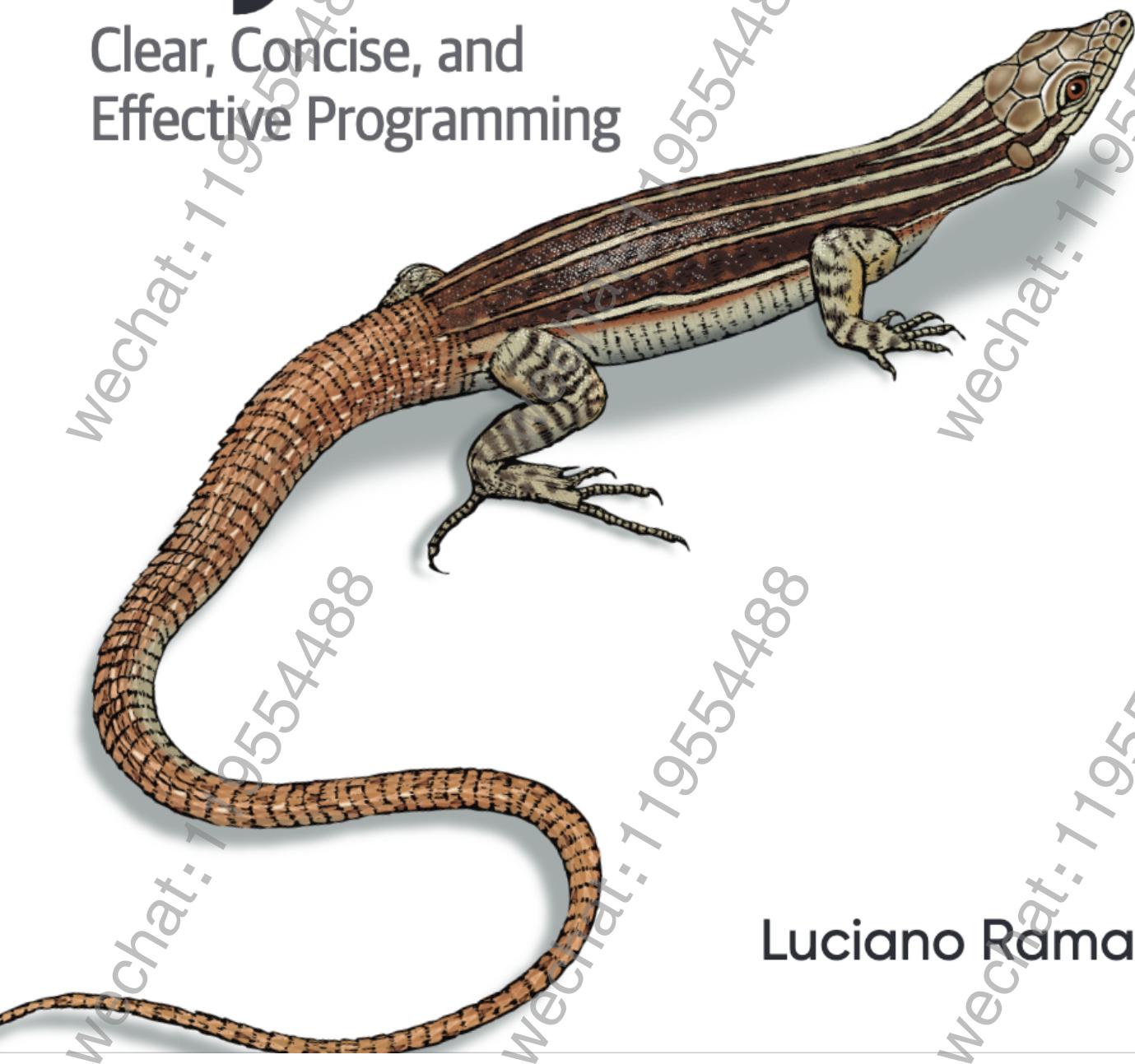


O'REILLY®

Fluent Python

Clear, Concise, and
Effective Programming



Luciano Ramalho

2nd Edition
Covers Python 3.10

wechat: 119554488

wechat: 119554488

OREILLY®

Fluent Python

Don't waste time bending Python to fit patterns you've learned in other languages. Python's simplicity lets you become productive quickly, but often this means you aren't using everything the language has to offer. With the updated edition of this hands-on guide, you'll learn how to write effective, modern Python 3 code by leveraging its best ideas.

Discover and apply idiomatic Python 3 features beyond your past experience. Author Luciano Ramalho guides you through Python's core language features and libraries and teaches you how to make your code shorter, faster, and more readable.

Complete with major updates throughout, this new edition features five parts that work as five short books within the book:

- **Data structures:** Sequences, dicts, sets, Unicode, and data classes
- **Functions as objects:** First-class functions, related design patterns, and type hints in function declarations
- **Object-oriented idioms:** Composition, inheritance, mixins, interfaces, operator overloading, protocols, and more static types
- **Control flow:** Context managers, generators, coroutines, `async/await`, and thread/process pools
- **Metaprogramming:** Properties, attribute descriptors, class decorators, and new class metaprogramming hooks that replace or simplify metaclasses

Luciano Ramalho is a principal consultant at Thoughtworks and a Python Software Foundation fellow.

"My 'go to' book when looking for detailed explanations and uses of a Python feature. Luciano's teaching and presentation are excellent. A great book for advanced beginners looking to build their knowledge."

—Carol Willing
Python Steering Council member
(2020-2021)

"This is not the usual dry coding book, but full of useful, tested examples, and just enough humor. My colleagues and I have used this amazing, well-written book to take our Python coding to the next level."

—Maria McKinley
Senior Software Engineer

PROGRAMMING / PYTHON

US \$69.99 CAN \$87.99
ISBN: 978-1-492-05635-5



9 781492 056355



Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

流畅的 Python (第2 版)

深入理解 Python 语言核心特性及底层逻辑

Luciano Ramalho 编著
陈英姿 翻译

推荐使用微信支付



灰太狼太灰 (**姿)

推荐使用支付宝



灰太狼太灰 (**姿)

码字不易,如果您觉得内容还不错,不妨捐赠一下,以示激励。

O'Reilly Media, Inc.

版权信息

C O P Y R I G H T

书 名：Fluent Python, 2nd Edition
中文字名：流畅的 Python (第 2 版)
作 者：Luciano Ramalho
出 版 社：O'Reilly Media, Inc.
出版时间：2022 年 2 月
I S B N：978-1-492-05635-5
字 数：1056 千字



码字不易,如果您觉得内容还不错,不妨捐赠一下,以示激励。

仅供个人学习使用,如涉版权问题,请自行删除。

修订记录

版本	发布日期	修订内容	文件名
v1.0.0	2024-10-10	初始版本	FluentPythonV2_CN_v1.0.0.pdf
v1.0.1	2024-12-03	优化前 5 章部分翻译内容	FluentPythonV2_CN_v1.0.1.pdf

翻译错误纠正,请提交 issue:<https://github.com/chenyz1984/FluentPython2ndCN/issues>

wechat: 119554488

前言

计划是这样的：当有人使用了你不理解的功能时，就直接毙了他们。这比学习新知识要容易得多，而且用不了多久，唯一还活着的程序员就会使用 Python 0.9.6 编写程序了，而且他们只需使用这个版本中易于理解的那一小部分功能即可。^a

——Tim Peters，传奇的核心开发者与《The Zen of Python》的作者

^a发布在 Usenet 小组 comp.lang.python 中的消息，2002 年 12 月 23 日，“Acrimony in c.l.p”。

“Python 是一种简单易学、功能强大的编程语言。”这是《Python 3.10 官方教程》的第一句话。事实的确如此，但需要注意的是：正因为 Python 语言既易学又易用，所以许多 Python 程序员只用到了其强大功能中的一小部分。

经验丰富的程序员可能在几个小时内就能编写出实用的 Python 代码。然而，当这最初的几个小时变成数周或数月后，在那些先入为主的其他编程语言影响下，许多开发人员可能会慢慢写出带有“口音”的 Python 代码。即使 Python 是您的第一门语言，也难逃此劫。因为学校或者那些入门书籍在介绍 Python 语言时，往往也会有意无意地避开这门语言所独有的特性。

另外，在试图向那些已在其他编程语言领域中拥有丰富经验的程序员介绍 Python 的时候，我还发现了一个问题：人们总是倾向于寻求自己熟悉的东西。受到其他语言影响，您大概会猜到 Python 支持规则表达式，然后就会去查阅 Python 文档。但是，如果您以前从未见过元组解包或描述符，您可能不会特地去研究它们。然后，就可能永远都失去了使用这些 Python 所独有的特性的机会。

本书并不是一本 A 到 Z 全面介绍 Python 的参考书。本书的重点是介绍 Python 所独有的语言特性。这也是一本关于 Python 语言核心和一些库的书。尽管 Python 的包索引现在已经包含了 60,000 多个库，而且其中许多库都非常实用，但我几乎不会提到 Python 标准库以外的包。

目标读者

本书的目标读者是那些正在使用 Python，同时想熟练掌握 Python3 的程序员。书中示例用 Python 3.10 做了测试，其中大多数还用 Python 3.8、3.9 做了测试。若某个示例必须使用 Python 3.10，我会标出来。

若您尚不清楚自己对 Python 的熟练程度能否跟得上本书的内容，建议您回头看看 Python 官方教程。注意，除了一些新功能之外，官方教程涵盖的内容在本书中不做说明。

非目标读者

若您是 Python 的初学者，本书内容可能会有些“超纲”。更糟的是，若在学习 Python 的过程中，过早地接触本书内容，您可能会误以为所有 Python 脚本都需要使用特殊方法和元编程（Meta-Programming）技巧。不合适地使用抽象与优化，将会适得其反。

本书是合五为一

我建议大家不要跳过第 1 章。如果您是本书的核心读者,可以跳过第 1 章,直接阅读任何一章都不会有问题。但我假定您是从头至尾按顺序阅读每一章。本书第一部分至第五部分,可被分别视为 5 本独立的书。

在介绍如何创建自己的程序之前,我通常会先介绍有哪些现成的工具可用。例如,第一部分的“[二 丰富的序列](#)”先介绍了一些现成可用的序列类型,包括一些不太受关注的类型,如 `collections.deque`。直到第三部分,我才会讲解如何定义自己的序列类型,同时说明如何使用 `collections.abc` 包 提供的 [抽象基类 \(ABCs\)](#)。而创建自己的[抽象基类 \(ABCs\)](#)更是延后到了第三部分的末尾。因为我认为在编写自己的[抽象基类 \(ABCs\)](#)之前,先熟悉如何使用 [抽象基类 \(ABCs\)](#) 是最重要的。

这种编排策略有几个优点:

- 第一,先了解有哪些现成的工具可用,可以避免“重新造轮子”。毕竟使用现成的容器类型的频率,要远高于实现自己的容器类型的频率。
- 第二,有更多的机会可以关注现成工具的高级用法,而不必讨论如何创建新工具。
- 第三,比起从零开始,继承现有的[抽象基类 \(ABCs\)](#)会更简单一些。
- 最后,我认为在见过一些实际案例之后,更容易理解抽象的概念。

当然,这种策略也存在一些缺点。比如,各章节之间的内容可能存在相互引用。我希望,当您了解我的良苦用心之后,可以容忍这种编排策略所带来的不变。

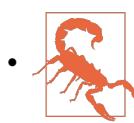
排版约定



该图标表示提示或建议。



该图标表示一般标记。



该图标表示警告或警示。

源码库

- 随书源码库(任选其一):
 - <https://gitee.com/chenyingzi/fluentpython2e-code>
 - <https://github.com/fluentpython/example-code-2e>
- CPython 实现源码:<https://github.com/python/cpython>
- typeshed 项目¹:<https://github.com/python/typeshed>

陈英姿(ShadowC)

中国,沈阳

¹ 为 Python 标准库提供[类型提示 \(Type Hints\)](#)。

录

第一部分 数据结构

第一章 Python 数据模型

1.1 本章新增内容	4
1.2 一副 Pythonic 的纸牌	5
1.3 特殊方法是如何使用的	9
1.3.1 模拟数值类型	9
1.3.2 字符串表示形式	12
1.3.3 自定义类型的布尔值	12
1.3.4 容器 (collection) API	13
1.4 特殊方法概述	14
1.5 len 为什么不是方法	15
1.6 本章小结	16
1.7 延伸阅读	16

第二章 丰富的序列

2.1 本章新增内容	20
2.2 内置序列类型概览	20
2.3 列表推导式与生成器表达式	22
2.3.1 列表推导式对可读性的影响	22
2.3.2 列表推导式与 filter 和 map 比较	23
2.3.3 笛卡尔积	24
2.3.4 生成器表达式	25
2.4 元组不仅仅是不可变列表	26
2.4.1 将元组用作记录	26
2.4.2 将元组用作不可变列表	28
2.4.3 元组与列表中的方法比较	29
2.5 序列和可迭代对象解包	30
2.5.1 用 * 获取余下的项	31
2.5.2 在函数调用和序列字面量中使用 * 解包	32
2.5.3 嵌套解包	32
2.6 序列模式匹配	33
2.6.1 用模式匹配序列实现一个解释器	37

2.7 切片	41
2.7.1 为何切片与区间都排除最后一项	41
2.7.2 切片对象	42
2.7.3 多维切片与省略号	43
2.7.4 为切片赋值	43
2.8 使用 + 和 * 处理序列	44
2.8.1 构建嵌套列表	44
2.8.2 用增量赋值运算符处理序列	46
2.8.3 一个 += 运算符赋值谜题	47
2.9 list.sort 与内置函数 sorted	48
2.10 当列表不适用时	50
2.10.1 数组	50
2.10.2 memoryview	53
2.10.3 NumPy	55
2.10.4 双端队列与其他队列	57
2.11 本章小结	59
2.12 延伸阅读	60
第三章 字典与集合	65
3.1 本章新增内容	65
3.2 字典的现代语法	66
3.2.1 字典推导式	66
3.2.2 映射解包	67
3.2.3 用 合并映射	67
3.3 用模式匹配处理映射	68
3.4 映射类型的标准 API	70
3.4.1 什么是可哈希	71
3.4.2 常用映射方法概述	72
3.4.3 插入或更新可变的值	73
3.5 缺失键的自动处理	75
3.5.1 defaultdict: 处理缺失键的另一种选择	75
3.5.2 __missing__ 方法	76
3.5.3 标准库中 __missing__ 的不一致用法	78
3.6 dict 变体	79
3.6.1 collections.OrderedDict	79
3.6.2 collections.ChainMap	79
3.6.3 collections.Counter	80
3.6.4 shelve.Shelf	80
3.6.5 子类应继承 UserDict 而不是 dict	81
3.7 不可变映射	82

3.8 字典视图	83
3.9 dict 实现方式对实践的影响	84
3.10 集合论	85
3.10.1 set 字面量	87
3.10.2 集合推导式	87
3.11 集合的实现方式对实践的影响	88
3.11.1 集合运算	88
3.12 字典视图的集合运算	90
3.13 本章小结	92
3.14 延伸阅读	92

第四章 Unicode 文本与字节序列 95

4.1 本章新增内容	96
4.2 字符问题	96
4.3 字节要点	97
4.4 基本的编码器与解码器	99
4.5 处理编码与解码问题	101
4.5.1 处理 UnicodeEncodeError	101
4.5.2 处理 UnicodeDecodeError	102
4.5.3 加载模块时的 SyntaxError	103
4.5.4 如何找出字节序列的编码	103
4.5.5 BOM:有用的鬼符	104
4.6 处理文本文件	106
4.6.1 注意默认编码	108
4.7 Unicode 字符规范化	112
4.7.1 大小写同一化	114
4.7.2 规范化文本匹配的实用函数	115
4.7.3 极端规范化:去掉变音符	116
4.8 Unicode 文本排序	119
4.8.1 用 Unicode 排序算法排序	120
4.9 Unicode 数据库	121
4.9.1 按名称查找字符	121
4.9.2 字符的数字含义	123
4.10 支持 str 和 bytes 的双模式 API	124
4.10.1 正则表达式中的 str 与 bytes	124
4.10.2 os 函数中的 str 与 bytes	125
4.11 本章小结	126
4.12 延伸阅读	127

5.1 本章新增内容	132
5.2 数据类构建器概述	132
5.2.1 数据类构建器对比	135
5.3 典型的具名元组: collections.namedtuple	136
5.4 带类型的具名元组: typing.NamedTuple	139
5.5 类型提示入门	139
5.5.1 运行时无效	140
5.5.2 变量注解语法	140
5.5.3 变量注解意义	141
5.6 @dataclass 详解	145
5.6.1 字段选项	146
5.6.2 初始化后处理: __post_init__	148
5.6.3 带类型的类属性	150
5.6.4 不用作实例字段的 Init-Only 变量	151
5.6.5 @dataclass 示例: 都柏林核心模式	151
5.7 数据类导致的代码异味	154
5.7.1 将数据类用作脚手架	155
5.7.2 将数据类用作中间表述	155
5.8 模式匹配类实例	156
5.8.1 简单类模式	156
5.8.2 关键字类模式	157
5.8.3 位置类模式	158
5.9 本章小结	159
5.10 延伸阅读	159
第六章 对象引用、可变性与垃圾回收	163
6.1 本章新增内容	163
6.2 变量不是盒子	164
6.3 同一性、相等性与别名	165
6.3.1 在 == 和 is 之间选择	167
6.3.2 元组的相对不可变性	168
6.4 默认做浅拷贝	169
6.4.1 任意对象的深拷贝与浅拷贝	170
6.5 作为引用的函数参数	172
6.5.1 不要使用可变类型作为参数的默认值	173
6.5.2 可变参数的防御性编程	175
6.6 del 与垃圾回收	177
6.7 Python 对不可变类型施加的把戏	179
6.8 本章小结	180
6.9 延伸阅读	181

第二部分 函数即对象	185
第七章 函数是一等对象	187
7.1 本章新增内容	187
7.2 将函数视为对象	188
7.3 高阶函数	189
7.3.1 map、filter 与 reduce 的替代品	190
7.4 匿名函数	191
7.5 9 种可调用对象	192
7.6 用户定义的可调用类型	193
7.7 从位置参数到仅限关键字参数	194
7.7.1 仅限位置参数	196
7.8 支持函数式编程的包	197
7.8.1 operator 模块	197
7.8.2 用 functools.partial 冻结参数	200
7.9 本章小结	201
7.10 延伸阅读	202
第八章 函数中的类型提示	205
8.1 本章新增内容	206
8.2 关于渐进式类型	206
8.3 渐进式类型实践	207
8.3.1 Mypy 初体验	207
8.3.2 让 Mypy 更加严格	208
8.3.3 默认参数值	209
8.3.4 将 “None” 设为默认值	211
8.4 类型由支持的操作来定义	211
8.5 注解中可用的类型	215
8.5.1 Any 类型	216
8.5.2 简单的类型与类	218
8.5.3 Optional 与 Union 类型	219
8.5.4 泛化容器	220
8.5.5 元组类型	222
8.5.6 泛化映射	225
8.5.7 抽象基类	226
8.5.8 Iterable	228
8.5.9 参数化泛型与 TypeVar	229
8.5.10 静态协议	233
8.5.11 Callable	237
8.5.12 NoReturn	240

8.6 注解仅限位置参数与变长参数	240
8.7 类型不完美, 测试需全面	241
8.8 本章小结	242
8.9 延伸阅读	243
第九章 装饰器与闭包	247
9.1 本章新增内容	248
9.2 装饰器基础知识	248
9.3 Python 何时执行装饰器	249
9.4 注册装饰器	250
9.5 变量作用域规则	251
9.6 闭包	254
9.7 nonlocal 声明	256
9.7.1 变量查找逻辑	257
9.8 实现一个简单的装饰器	258
9.8.1 装饰器工作原理	259
9.9 标准库中的装饰器	260
9.9.1 用 functools.cache 进行记忆化	260
9.9.2 使用 lru_cache	263
9.9.3 单分派泛化函数	264
9.10 参数化装饰器	268
9.10.1 一个参数化的注册装饰器	268
9.10.2 参数化 clock 装饰器	270
9.10.3 基于类的 clock 装饰器	272
9.11 本章小结	273
9.12 延伸阅读	273
第十章 用一等函数实现设计模式	277
10.1 本章新增内容	278
10.2 案例研究: 策略模式重构	278
10.2.1 经典策略模式	278
10.2.2 用函数实现策略模式	281
10.2.3 选择最佳策略的简单方式	284
10.2.4 找出模块中的所有策略	285
10.3 用装饰器改进策略模式	287
10.4 命令模式	288
10.5 本章小结	289
10.6 延伸阅读	290

第三部分	类与协议	293
第十一章 Pythonic 对象		295
11.1	本章新增内容	296
11.2	对象表示形式	296
11.3	再谈 Vector 类	296
11.4	后备的构造函数	299
11.5	classmethod 与 staticmethod	299
11.6	格式化显示	300
11.7	可哈希的 Vector2d	303
11.8	支持位置的模式匹配	305
11.9	第 3 版 Vector2d 的完整代码	307
11.10	Python 私有属性与“受保护”的属性	310
11.11	用 __slots__ 节省内存	311
11.11.1	简单衡量 __slot__ 节省的内存	313
11.11.2	总结 __slots__ 的问题	315
11.12	覆盖类属性 (Class Attribute)	315
11.13	本章小结	317
11.14	延伸阅读	318
第十二章 序列的特殊方法		321
12.1	本章新增内容	321
12.2	Vector 类: 用户定义序列类型	322
12.3	Vector 类第 1 版: 与 Vector2d 类兼容	322
12.4	协议与鸭子类型	324
12.5	Vector 类第 2 版: 可切片的序列	325
12.5.1	切片原理	326
12.5.2	能处理切片的 __getitem__ 方法	327
12.6	Vector 类第 3 版: 动态访问属性	329
12.7	Vector 类第 4 版: 哈希与快速等值测试	332
12.8	Vector 类第 5 版: 格式化	337
12.9	本章小结	343
12.10	延伸阅读	344
第十三章 接口、协议与抽象基类		349
13.1	类型图	349
13.2	本章新增内容	350
13.3	两种协议	351
13.4	用鸭子类型编程	352
13.4.1	Python 深入理解序列	353
13.4.2	猴子补丁: 在运行时实现协议	354

13.4.3 防御性编程与快速失败	356
13.5 大鹅类型 (Goose Typing)	358
13.5.1 子类化一个抽象基类	361
13.5.2 标准库中的抽象基类	363
13.5.3 定义并使用一个抽象基类	365
13.5.4 抽象基类语法详解	369
13.5.5 子类化抽象基类	369
13.5.6 抽象基类的虚拟子类	371
13.5.7 register 的实际使用	373
13.5.8 用抽象基类实现结构类型	374
13.6 静态协议	376
13.6.1 为函数 double 添加类型提示	376
13.6.2 运行时可检查的静态协议	377
13.6.3 运行时协议检查的局限性	380
13.6.4 支持静态协议	381
13.6.5 设计静态协议	382
13.6.6 静态协议设计最佳实践	384
13.6.7 扩展一个协议	385
13.6.8 number 模块中的抽象基类与 Numeric 协议	385
13.7 本章小结	387
13.8 延伸阅读	388
第十四章 继承的利与弊	393
14.1 本章新增内容	394
14.2 super() 函数	394
14.3 子类化内置类型很棘手	396
14.4 多重继承与方法解析顺序	398
14.5 混合类	403
14.5.1 不区分大小写的映射	403
14.6 多重继承的实际运用	405
14.6.1 抽象基类也是混合类	405
14.6.2 ThreadingMixIn 与 ForkingMixIn	405
14.6.3 Django 泛化视图混合类	406
14.6.4 Tkinter 中的多重继承	408
14.7 应对多重继承	410
14.7.1 优先使用对象组合,而不是类继承	410
14.7.2 理解不同场景下使用继承的原因	410
14.7.3 使用抽象基类显式表示接口	411
14.7.4 用显式混合类实现代码重用	411
14.7.5 为用户提供聚合类	411

14.7.6 仅子类化为扩展而设计的类	411
14.7.7 避免子类化具体类	412
14.7.8 Tkinter 的优与劣	412
14.8 本章小结	413
14.9 延伸阅读	414
第十五章 类型注解进阶 1136	417
15.1 本章新增内容	417
15.2 重载的签名	418
15.2.1 重载 max 函数	419
15.2.2 重载 max 函数的启示	423
15.3 TypedDict	423
15.4 类型校正	430
15.5 在运行时读取类型提示	432
15.5.1 运行时的注解问题	432
15.5.2 解决这个问题	434
15.6 实现泛化类	435
15.6.1 泛型的基础术语	437
15.7 型变	438
15.7.1 不变的自动售货机	438
15.7.2 协变的自动售货机	439
15.7.3 逆变的垃圾桶	440
15.7.4 型变总结	442
15.8 实现泛化静态协议	444
15.9 本章小结	446
15.10 延伸阅读	447
第十六章 运算符重载	451
16.1 本章新增内容	452
16.2 运算符重载入门	452
16.3 一元运算符	452
16.4 重载向量加法运算符 +	455
16.5 重载标量乘法运算符 *	460
16.6 将 @ 当作中缀运算符使用	462
16.7 算术运算符总结	463
16.8 丰富的比较运算符	465
16.9 增量赋值运算符	468
16.10 本章小结	472
16.11 延伸阅读	473

第四部分 控制流	477
第十七章 迭代器、生成器和经典协程	479
17.1 本章新增内容	480
17.2 单词序列	480
17.3 序列可迭代的原因:iter 函数	481
17.3.1 用 iter 处理可调用对象	483
17.4 可迭代对象与迭代器	484
17.5 为 Sentence 类实现 __iter__ 方法	487
17.5.1 Sentence 类第 2 版:经典迭代器	487
17.5.2 勿让可迭代对象变成其自己的迭代器	489
17.5.3 Sentence 类第 3 版:生成器函数	489
17.5.4 生成器工作原理	490
17.6 惰性版本的 Sentence 类	493
17.6.1 Sentence 类第 4 版:惰性生成器	493
17.6.2 Sentence 类第 5 版:惰性生成器表达式	493
17.7 何时用生成器表达式	495
17.8 等差数列生成器	496
17.8.1 用 itertools 模块生成等差数列	498
17.9 标准库中的生成器函数	499
17.9.1 用于筛选的生成器函数	500
17.9.2 用于映射的生成器函数	501
17.9.3 用于合并的生成器函数	502
17.9.4 用于扩充输入的生成器函数	504
17.9.5 用于产出输入的生成器函数	506
17.10 可迭代的规约 (Reducing) 函数	508
17.11 yield from 表达式	510
17.11.1 重新实现 chain	511
17.11.2 遍历树状结构	512
17.12 泛化可迭代类型	516
17.13 经典协程	518
17.13.1 示例:用协程计算累计平均值	519
17.13.2 让协程返回一个值	521
17.13.3 经典协程的泛型注解	524
17.14 本章小结	526
17.15 延伸阅读	526
第十八章 with、match 和 else 代码块	531
18.1 本章新增内容	531
18.2 上下文管理器与 with 块	532

18.2.1 contextlib 包中的实用工具	535
18.2.2 使用 @contextmanager	536
18.3 案例分析:lis.py 中的模式匹配	539
18.3.1 Schema 语法	540
18.3.2 导入与类型	541
18.3.3 解析器	541
18.3.4 环境	542
18.3.5 REPL	544
18.3.6 求解函数	545
18.3.7 实现闭包的 Procedure 类	552
18.3.8 使用 OR 模式	553
18.4 先这样,再那样:if 之外的 else 块	554
18.5 本章小结	556
18.6 延伸阅读	556
第十九章 Python 中的并发模型	561
19.1 本章新增内容	562
19.2 全局概览	562
19.3 术语定义	562
19.3.1 进程、线程、GIL 锁	564
19.4 并发的 Hello World	565
19.4.1 用线程实现 Spinner	565
19.4.2 用进程实现 Spinner	567
19.4.3 用协程实现 Spinner	569
19.4.4 对比几版 supervisor	572
19.5 GIL 的真实影响	573
19.5.1 小测验	574
19.6 自研进程池	576
19.6.1 基于进程的方案	577
19.6.2 理解耗时	578
19.6.3 多核素数检测器代码	579
19.6.4 实验:进程数多一些或少一些	582
19.6.5 基于线程的方案不可靠	583
19.7 多核世界的 Python	583
19.7.1 系统管理	584
19.7.2 数据科学	584
19.7.3 服务端 Web 与移动开发	585
19.7.4 WSGI 应用服务器	586
19.7.5 分布式任务队列	588
19.8 本章小结	589

19.9 延伸阅读	589
19.9.1 用线程和进程实现并发	589
19.9.2 GIL	590
19.9.3 标准库之外的并发	591
19.9.4 Python 之外的并发与水平扩展	592
第二十章 并发执行器	595
20.1 本章新增内容	595
20.2 并发网络下载	596
20.2.1 顺序下载脚本	597
20.2.2 用 concurrent.futures 实现下载	599
20.2.3 future 对象在何处	600
20.3 用 concurrent.futures 启动进程	603
20.3.1 终极版的多核素数检测器	603
20.4 实验:Executor.map 方法	606
20.5 显示下载进度并处理错误	608
20.5.1 flags2 示例中的错误处理	612
20.5.2 使用 futures.as_completed 函数	614
20.6 本章小结	616
20.7 延伸阅读	617
第二十一章 异步编程	619
21.1 本章新增内容	620
21.2 术语定义	620
21.3 asyncio 示例:探测域名	621
21.3.1 异步代码阅读技巧	622
21.4 新概念:异步可调用对象	623
21.5 用 asyncio 和 HTTPX 进行下载	624
21.5.1 原生协程的秘密:默默无闻的生成器	626
21.5.2 非此即彼的问题	626
21.6 异步上下文管理器	627
21.7 增强 asyncio 下载器	628
21.7.1 使用 asyncio.as_completed 和线程	629
21.7.2 用信号量限制网络请求	630
21.7.3 为每次下载发起多个请求	633
21.8 将任务委托给执行器	635
21.9 用 asyncio 编写服务器	636
21.9.1 FastAPI Web 服务	638
21.9.2 异步 TCP 服务器	640
21.10 异步迭代器和异步可迭代对象	645

21.10.1 异步生成器函数	645
21.10.2 异步推导式和异步生成器表达式	651
21.11 asyncio 之外的异步世界:Curio	653
21.12 异步对象的类型提示	655
21.13 异步工作原理与陷阱	656
21.13.1 围绕阻塞型调用的循环运行	656
21.13.2 I/O 密集型系统的误区	657
21.13.3 避免 CPU 密集型陷阱	657
21.14 本章小结	657
21.15 延伸阅读	658
第五部分 元编程	661
第二十二章 动态属性和特性	663
22.1 本章新增内容	663
22.2 利用动态属性进行数据转换	664
22.2.1 用动态属性访问 JSON 类数据	665
22.2.2 属性名无效的问题	668
22.2.3 用 <code>__new__</code> 灵活创建对象	669
22.3 计算的特性	671
22.3.1 第 1 步:数据驱动的属性创建	672
22.3.2 第 2 步:用特性检索链接的记录	674
22.3.3 第 3 步:用特性覆盖现有属性	677
22.3.4 第 4 步:定制的特性缓存	677
22.3.5 第 5 步:用 <code>functools</code> 缓存特性	678
22.4 用特性验证属性	680
22.4.1 <code>LineItem</code> 类第 1 版:表示订单中商品的类	680
22.4.2 <code>LineItem</code> 类第 2 版:能验证的特性 (property)	681
22.5 全面了解特性 (property)	682
22.5.1 特性 (property) 覆盖实例属性	683
22.5.2 特性的文档 (docstring)	685
22.6 定义一个特性工厂函数	686
22.7 处理属性删除操作	688
22.8 处理属性的重要属性与函数	690
22.8.1 影响属性处理的特殊属性	690
22.8.2 用于属性处理的内置函数	691
22.8.3 用于属性处理的特殊方法	691
22.9 本章小结	692
22.10 延伸阅读	693

第二十三章 属性描述符	697
23.1 本章新增内容	697
23.2 描述符示例:属性验证	698
23.2.1 LineItem 类第 3 版:一个简单的描述符	698
23.2.2 LineItem 类第 4 版:存储 (Storage) 属性的自动命名	703
23.2.3 LineItem 类第 5 版:一种新型描述符	704
23.3 覆盖型描述符与非覆盖型描述符的对比	706
23.3.1 覆盖型描述符	708
23.3.2 不含 <code>__get__</code> 方法的覆盖型描述符	709
23.3.3 非覆盖型描述符	710
23.3.4 覆盖类中的描述符	711
23.4 方法也是描述符	712
23.5 描述符用法建议	713
23.6 描述符的文档字符串与覆盖删除操作	715
23.7 本章小结	715
23.8 延伸阅读	716
第二十四章 元类编程 (Class Metaprogramming)	719
24.1 本章新增内容	719
24.2 身为对象的类	720
24.3 <code>type</code> :内置的类工厂函数	720
24.4 类工厂函数	722
24.5 介绍 <code>__init_subclass__</code>	724
24.5.1 为何 <code>__init_subclass__</code> 无法配置 <code>__slots__</code>	730
24.6 用类装饰器增强类的功能	730
24.7 何时发生:导入时与运行时	733
24.7.1 实验 1:何时求解	733
24.8 元类入门	737
24.8.1 如何用元类定制一个类	739
24.8.2 友好的元类示例	740
24.8.3 实验 2:元类的求解时机	742
24.9 用元类实现 <code>Checked</code> 类	746
24.10 元类的实际运用	750
24.10.1 可简化或取代元类的现代特性	750
24.10.2 元类是稳定的语言特性	751
24.10.3 一个类仅能有一个元类	751
24.10.4 元类应作为实现细节	752
24.11 用 <code>__prepare__</code> 实现一些高级功能	752
24.12 元类总结	754
24.13 本章小结	755

24.14 延伸阅读	755
结语	760
附录	763
附录 A 清单目录	765
A.1 术语清单	765
A.2 图片清单	779
A.3 表格清单	782
A.4 代码清单	783

wechat: 119554488

PART I

第一部分

数据结构

- 第 1 章 Python 数据模型
- 第 2 章 丰富的序列
- 第 3 章 字典与集合
- 第 4 章 Unicode 文本与字节序列
- 第 5 章 数据类构建器
- 第 6 章 对象引用、可变性与垃圾回收

wechat: 119554488

Python 数据模型

Guido (Python 之父) 对语言设计美学的感悟令人惊叹。我见过许多优秀的编程语言设计者, 他们设计出的语言在理论上精彩绝伦, 但却很少有人使用。Guido 知道如何在理论上做出一定妥协, 设计出的编程语言可以让使用者如沐春风, 这可真是难得。

——Jim Hugunin, Jython 作者, AspectJ 作者之一, .Net DLR 架构师^a

^a 摘自 “Story of Jython”——《Jython Essentials》一书的前言。

Python 的质量保障得益于它的一致性。在使用 Python 一段时间后, 您便可以根据已掌握的知识, 准确地猜测出新功能的作用。

但是, 如果您在接触 Python 之前, 有过其他面向对象语言的经验, 您可能会奇怪: 获取 `collection` 对象的大小, 为何用 `len(collection)`, 而不用 `collection.len()`? 这一点只是 Python 众多奇怪行为的冰山一角。不过当您了解背后的原因之后, 会发现这才是真正的 `Pythonic`¹。这座冰山被称为 “Python Data Model (数据模型)”, 它是一种 API, 可以使我们自己创建的对象与最地道的 Python 语言特性完美结合。

我们可将 Python 视为一个框架, 而 Data Model 就是对该框架的描述。Data Model 规范了语言自身各个组成部分的接口, 如序列、函数、迭代器、[协程 \(Coroutine\)](#)、类、上下文管理器等部分的行为。

使用框架时, 需要耗费大量时间编写方法, 以供框架调用。利用 Python Data Model 构建新类亦是如此。Python 解释器会调用特殊方法来执行基本的对象操作, 这些操作通常由特殊语法触发。这些特殊方法的名称前后都有双下划线 (__)²。例如, 为 `obj[key]` 语法提供支持的特殊方法为 `__getitem__`。为了求解 `my_collection[key]`, Python 解释器会在幕后调用 `my_collection.__getitem__(key)`。

若想让对象支持以下基本的语言结构并与之交互, 我们就需要实现特殊方法:

- 容器 (`collections`)³
- 属性 (`attribute`) 访问

¹ Pythonic 是指 Python 语言的一种编程风格, 指的是代码编写风格符合 Python 社区推崇的简洁、优雅、直观和易读的特点。遵循 `Pythonic` 的代码通常更容易理解和维护, 同时也更符合 Python 语言的设计原则。

² 这种以双下划线 (__) 为名称前缀及后缀的特殊方法, 又称为 “dunder 方法” 或 “魔术方法”。例如, `__getitem__`、`__setitem__`、`__new__`、`__init__` 等。

³ 译者注: 为了以示区别, 本书将 `collection` 翻译为 “容器”, 而将 Python 内置类型 `set` 翻译为 “集合”。

- 迭代 (包括使用 `async for` 的异步迭代)
- 运算符重载 (overloading)
- 函数和方法调用
- 字符串表示与格式化
- 使用 `await` 的异步编程
- 对象创建与销毁
- 用 `with` 或 `async with` 语句管理上下文



魔术方法和双下划线

特殊方法的专业术语为“魔术方法”。特殊方法 (如 `__getitem__`) 通常读作“dunder-`getitem`”，这是从著名作家和教师 Steve Holden 那里学到的。“dunder”的意思是“前后双下划线”。因此，特殊方法也称作“dunder 方法”。《The Python Language Reference》中的“2. Lexical analysis”中警告道：“任何时候，若不遵守文档明确说明的方式使用 `__*__` 名称，一切后果自负。”

1.1 本章新增内容

相较于第 1 版，本章内容改动较小，毕竟 Python Data Model (数据模型) 是相当稳定的。本章主要改动如下：

- “1.4 特殊方法概述”中的“表 1.1<14页>”增加了支持异步编程和其他新特性的特殊方法；
- “1.3.4 容器 (collection) API”中的“图 1.2<13页>”展示了特殊方法的使用，包括 Python 3.6 引入的抽象基类 (ABCs) :`collections.abc.Collection`。

此外，第 2 版所有章节中的字符串格式化统一都采用了 Python 3.6 中引入的 `f-string` 语法⁴，它比旧式的字符串格式化表示法 (`str.format()`方法和% 运算符) 可读性更好，也更方便。

⁴与 `f-string` 字符串格式化表示法相关的 PEP 为“[PEP 498 - Literal String Interpolation](#)”



有时候, 定义“my_fmt”格式化字符串的位置与实际执行格式化操作的位置可能不同。此时, 可以继续使用 `my_fmt.format()`。比如, `my_fmt` 的内容需要占据多行、更适合定义为常量、或者必须从配置文件或数据库中读取 `my_fmt` 内容。这些场景虽不常见, 但有时候也会遇到。

```

1 # 定义格式化字符串
2 format_string = "时间: {time} - 消息: {message}"
3
4 def log_message(message):
5     # 获取当前时间
6     current_time = get_current_time()
7     # 使用定义的格式化字符串格式化消息
8     formatted_message = format_string.format(time=
9         current_time, message=message)
10    # 记录日志
11    write_to_log(formatted_message)

```

1.2 一副 Pythonic 的纸牌

示例 1.1 虽然简单, 但展示了仅通过实现 `__getitem__` 和 `__len__` 这 2 个特殊方法, 就能获得强大的功能。

</> 示例 1.1: 一副有序的纸牌

```

1 import collections
2
3 Card = collections.namedtuple('Card', ['rank', 'suit'])
4
5 class FrenchDeck:    # 表示一副纸牌
6     ranks = [str(n) for n in range(2, 11)] + list('JQKA')
7     suits = 'spades diamonds clubs hearts'.split()
8
9     def __init__(self):
10         self._cards = [Card(rank, suit) for suit in self.suits
11                         for rank in self.ranks]
12
13     def __len__(self):
14         return len(self._cards)
15
16     def __getitem__(self, position):
17         return self._cards[position]

```

可以用 `collections.namedtuple` 构建只有属性 (attribute) 而没有自定义方法的类对象, 例如数据库中的 1 行记录。本例中使用 `collections.namedtuple` 构建用于表示一副牌中的各张纸牌的类——`Card`, 如下所示:

```

1 >>> from cards import Card
2 >>> beer_card = Card('7', 'diamonds')

```

```

3 >>> beer_card
4 Card(rank='7', suit='diamonds')

```

示例 1.1 的重点是简洁的自定义容器类 FrenchDeck(用于表示一副纸牌)。首先,与标准的 Python 容器一样,可以响应 `len()` 函数,返回一副纸牌⁵有多少张。如下所示:

```

1 >>> from cards import Card, FrenchDeck
2 >>> deck = FrenchDeck()
3 >>> len(deck)
4 52

```

其次,得益于实现了 `__getitem__` 方法,可以通过索引下标轻松地从一副纸牌中抽取某一张。例如,抽取第 1 张或最后 1 张,如下所示:

```

1 >>> from cards import Card, FrenchDeck
2 >>> deck = FrenchDeck()
3 >>> deck[0]
4 Card(rank='2', suit='spades')
5 >>> deck[-1]
6 Card(rank='A', suit='hearts')

```

最后,如果随机抽取一张纸牌,需要重新定义一个方法么? 答案: 不需要。因为 Python 已经提供了从序列中随机获取一项的函数,即 `random.choice()`。可以为 FrenchDeck 对象(一副纸牌)应用这个函数。如下所示:

```

1 >>> from cards import Card, FrenchDeck
2 >>> from random import choice
3 >>> deck = FrenchDeck()
4 >>> choice(deck)
5 Card(rank='3', suit='clubs')
6 >>> choice(deck)
7 Card(rank='8', suit='spades')
8 >>> choice(deck)
9 Card(rank='2', suit='hearts')

```

刚刚看到了使用特殊方法来利用 Python Data Model(数据模型)的 2 个优点:

- 类的使用者无需记住标准操作的方法名(如何获取项数? 用 `.size()`、`.length()`, 还是其他方法?)
- 可以充分利用 Python 标准库, 无需重新造轮子。如 `random.choice()` 函数。

只有这些么? 当然不是了, 好戏还在后面!

由于 `__getitem__` 方法将操作委托给 `self._cards` 的 `[]` 运算符, 因此 FrenchDeck 类将自动支持切片(slicing)操作。下面展示如何从一副新牌中, 抽取最上面 1 张, 再从索引位 12 开始, 步长为 13, 依次抽取 4 张 A。

```

1 >>> from cards import Card, FrenchDeck
2 >>> deck = FrenchDeck()
3 >>> deck[:3]          # 抽取最上面 3 张纸牌

```

⁵译者注: 一副纸牌指自定义容器类 FrenchDeck 的实例化对象。

```

4 [Card(rank='2', suit='spades'), Card(rank='3',
5 suit='spades'), Card(rank='4', suit='spades')]>>> deck[12::13]      # 只抽取4张A
6 [Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
7 Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]

```

在FrenchDeck类(“示例1.1<5页>”)实现了特殊方法`__getitem__`之后,这副纸牌还可以正向遍历与反向遍历。如下所示:

```

1 >>> from cards import Card, FrenchDeck
2 >>> deck = FrenchDeck()
3 >>> for card in deck:          # 正向遍历
4     ...     print(card)
5 ...
6 Card(rank='2', suit='spades')
7 Card(rank='3', suit='spades')
8 Card(rank='4', suit='spades')
9 ...
10 >>> for card in reversed(deck):    # 反向遍历
11     ...     print(card)
12 ...
13 Card(rank='A', suit='hearts')
14 Card(rank='K', suit='hearts')
15 Card(rank='Q', suit='hearts')
16 ...

```



doctest中的省略号(...)

为力求准确,本书中的Python控制台会话内容都尽可能从doctest中提取。当输出太长时,被省略的部分会用省略号(...)标记,如上述代码的最后1行。在此情况下,为了让doctest顺利通过,我加上了“# doctest: +ELLIPSIS”指令。如果在交互式控制台中试验这些示例,可以将doctest注释全部去掉。

遍历往往是隐式的。若一个容器没有实现`__contains__`方法,那么`in`运算符就会先做一次顺序扫描(隐式遍历)。本示例就是如此,因为FrenchDeck类是可遍历的,所以支持`in`运算符。如下所示:

```

1 >>> from cards import Card, FrenchDeck
2 >>> deck = FrenchDeck()
3 >>> Card('Q', 'hearts') in deck      # 执行隐式遍历, 顺序扫描一副牌中的每一项
4 True
5 >>> Card('Q', 'bearts') in deck      # 执行隐式遍历, 顺序扫描一副牌中的每一项
6 False

```

那么如何排序呢?按常规,纸牌牌面大小应该按点数(A最大、2最小),以及花色(♠>♥>♦>♣)的顺序排列。下面按此规则定义纸牌的排序函数`spades_high(card)`:梅花2返回0、黑桃A返回51,如下所示:

```

1 import collections
2
3 Card = collections.namedtuple('Card', ['rank', 'suit'])

```

```

4
5 class FrenchDeck:    ranks = [str(n) for n in range(2, 11)] + list('JQKA')      #
6     定义纸牌的点数
7     suits = 'spades diamonds clubs hearts'.split()                         # 定义纸牌的花色
8
9     def __init__(self):
10        self._cards = [Card(rank, suit) for suit in self.suits
11                      for rank in self.ranks]
12
13     def __len__(self):
14         return len(self._cards)
15
16     def __getitem__(self, position):
17         return self._cards[position]
18
19     suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)      # 花色的权重字典
20
21     def spades_high(card):      # 纸牌排序函数: 梅花2返回0, 黑桃A返回51
22         rank_value = FrenchDeck.ranks.index(card.rank)                  # 纸牌点数索引下标
23         return rank_value * len(suit_values) + suit_values[card.suit] # 返回权重, 用于排序

```

定义好 spades_high 函数后, 现在可以按牌面大小(升序)列出整副纸牌。

```

1 >>> from cards_sorted import FrenchDeck,spades_high
2 >>> deck=FrenchDeck()
3 >>> for card in sorted(deck, key=spades_high):
4     ...     print(card)
5 ...
6 Card(rank='2', suit='clubs')
7 Card(rank='2', suit='diamonds')
8 Card(rank='2', suit='hearts')
9 ... (此处省略46张纸牌)
10 Card(rank='A', suit='diamonds')
11 Card(rank='A', suit='hearts')
12 Card(rank='A', suit='spades')

```

虽然 FrenchDeck 类隐式继承 object 类, 但只从 object 类中继承了为数不多的功能。大部分功能都源自 Python Data Model (数据模型) 和组合模式。通过前面的 random.choice、reversed 和 sorted 示例可以看出, 实现了 `__len__`、`__getitem__` 这 2 个特殊方法后, FrenchDeck 的行为与标准 Python 序列一样, 都可受益于 Python 语言核心特性 (例如遍历和切片) 和标准库。`__len__`、`__getitem__` 的实现, 利用组合模式 (Composition), 将所有工作都委托给一个列表对象 (即 `self._cards`)。



如何洗牌

截至目前的设计, FrenchDeck 对象还不支持洗牌功能, 因为它是不可变的(即纸牌自身及其位置不能变化), 除非违背封装原则, 直接操作 `_cards` 属性。“[十三 接口、协议与抽象基类](#)”中的“[示例 13.6<361页>](#)”将用仅需 1 行代码的 `__setitem__` 方法, 解决了洗牌问题。

1.3 特殊方法是如何使用的

首先, 需要明确的是: 特殊方法是供 Python 解释器调用的, 而不是供用户调用的。也就是说, 没有类似 `my_object.__len__()` 这种写法, 正确的写法应该是 `len(my_object)`。如果 `my_object` 是用户定义类的实例, 当用户调用 `len(my_object)` 时, Python 解释器将在幕后调用用户实现的 `__len__()` 方法。

但是, 在处理内置类型(如 `list`、`str`、`bytearray`)或扩展(如 NumPy 数组)时, Python 解释器会走个捷径。Python 中可变长度容器⁶的底层 C 语言实现中, 有一个名为 `PyVarObject` 的结构体。此结构体中有一个名为 `ob_size` 的字段, 用于保存容器对象中的项数。因此, 如果 `my_object` 是这些内置容器类型之一的实例, 则 `len(my_object)` 会直接读取 `ob_size` 字段的值, 这比调用方法 `__len__()` 方法要快得多。

很多时候, 特殊方法都是隐式调用的。例如, `for i in x:` 语句, 其实 Python 在幕后先调用 `iter(x)`, 再调用 `x.__iter__()`(如果有)或 `__getitem__()`。在 FrenchDeck 示例中, 则调用的是 `__getitem__()`。

一般来说, 我们自己写的代码不应该频繁地直接调用特殊方法。除了涉及大量元编程之外, 大部分时间应该是在实现特殊方法, 而很少直接显式调用特殊方法。唯一例外的是 `__init__` 方法, 此方法用于在您自己的 `__init__` 实现中直接调用父类的 `__init__` 方法来初始化父类。

如果需要调用特殊方法, 则最好调用相应的内置函数(如 `len`、`iter`、`str` 等)。这些内置函数除了隐式调用对应的特殊方法之外, 通常还提供了额外的服务。而且对于内置类型来说, 调用内置函数的速度也要比直接调用特殊方法要快。

接下来几节, 会介绍特殊方法最重要的用途:

- 模拟数字(`Number`)类型
- 对象的字符串表示
- 对象的布尔值
- 实现容器类型

1.3.1 模拟数值类型

有几种特殊方法允许用户对象响应四则运算符, 如“`+`”。“[十六 运算符重载](#)”对此有详细介绍, 而此处我们只是通过一个简单示例, 进一步说明特殊方法的用途。

接下来将实现一个表示二维的向量类(`Vector`), 也就是数学与物理中使用的“欧几里得(`Euclidean`)向量”(如图 1.1 所示)。

⁶译者注: 为了以示区分, 在翻译时将 `collection` 翻译为“容器”, 而将 Python 内置类型 `set` 翻译为“集合”。



内置的 `complex` 类型可用于表示二维向量。不过,我们实现的 `Vector` 类经过扩展可以表示 n 维向量,详见“[十六 运算符重载](#)”[451页](#)”。

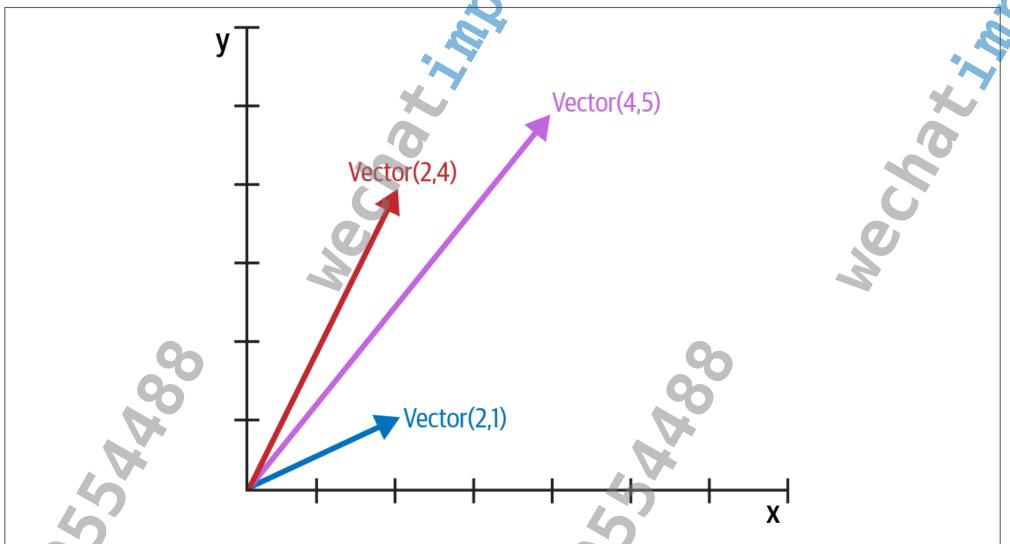


图 1.1: 二维向量加法图示: $\text{Vector}(2,4) + \text{Vector}(2,1) = \text{Vector}(4,5)$

[示例 1.2](#) 使用 `__repr__`、`__abs__`、`__add__`、`__mul__` 等特殊方法为 `Vector` 类分别实现了向量显示、向量求模、向量加法、向量标量积(即向量乘以数字)。

</> [示例 1.2: 一个简单的二维向量类](#)

```

1  # -*- coding: UTF-8 -*-
2
3 """
4 vector2d.py: 一个简单的雷, 演示一些特殊方法
5 只是演示, 一些问题做简化处理, 缺少错误处理, 尤其是 __add__ 和 __mul__ 方法。
6 本书后文还会扩充这个示例。
7
8 加法::
9   >>> v1 = Vector(2, 4)
10  >>> v2 = Vector(2, 1)
11  >>> v1 + v2
12  Vector(4, 5)
13 绝对值::
14  >>> v = Vector(3, 4)
15  >>> abs(v)
16  5.0
17 标量积::
18  >>> v * 3
19  Vector(9, 12)
20  >>> abs(v * 3)
21  15.0
22 """

```

```
23  
24  import math  
25  class Vector:  
26      def __init__(self, x=0, y=0) -> None:  
27          self.x = x  
28          self.y = y  
29  
30      def __repr__(self) -> str:  
31          return f'Vector({self.x!r}, {self.y!r})'  
32  
33      def __abs__(self):  
34          return math.hypot(self.x, self.y)  
35  
36      def __bool__(self):  
37          return bool(abs(self))  
38  
39      def __add__(self, other):  
40          x = self.x + other.x  
41          y = self.y + other.y  
42          return Vector(x, y)  
43  
44      def __mul__(self, scalar):  
45          return Vector(self.x * scalar, self.y * scalar)
```

以下控制台会话,测试示例 1.2 中的特殊方法:

```
1  >>> from vector2d import Vector  
2  >>> v1 = Vector(2,4)  
3  >>> v2 = Vector(2,1)  
4  >>> v1 + v2  
5  Vector(4, 5)  
6  >>> v = Vector(3,4)  
7  >>> abs(v)  
8  5.0  
9  >>> v * 3  
10 >>> Vector(9, 12)  
11 >>> abs(v * 3)  
12 15.0  
13 >>>
```

在上述控制台会话中,执行的 + 与 * 运算符时,Python 解释器在幕后分别调用了 Vector 类的 __add__ 与 __mul__ 方法。这 2 个方法都返回一个新的 Vector 对象,而没有修改参与运算的 Vector 对象。这符合中缀运算符的预期行为,即创建新对象,而不修改参与运算的对象(详见“[十六 运算符重载<451页>](#)”)。



示例 1.2 实现的 __mul__ 方法,仅允许 Vector 对象乘以数字,而不允许数字乘以 Vector 对象。如允许 $\text{Vector}(3,4) * 5$,但不允许 $5 * \text{Vector}(3,4)$ 。这违背了标量积交换律。此问题将在“[16.5 重载标量乘法运算符*<460页>](#)”中,用特殊方法 __rmul__ 解决。

接下来的几节,将继续讨论 `Vector` 类的其他特殊方法。

1.3.2 字符串表示形式

特殊方法 `__repr__` 供内置函数 `repr` 调用,以获取对象的字符串表示形式。若未定义 `__repr__` 方法, `Vector` 实例在 Python 控制台中将显示为 `<Vector object at 0x10e100070>` 形式。

交互式控制台和调试器会为表达式求解的结果自动调用 `repr` 函数,其处理方式与使用% 运算符处理字符串格式化中的`%r` 占位符,以及使用 `str.format` 方法处理字符串格式化中的`!r` 转换字段一样。

注意, `Vector` 类 `__repr__` 方法中的 `f-string` 使用`!r` 以标准的表示形式显示 `Vector` 对象的属性,这是推荐的做法。因为, `Vector(1,2)` 与 `Vector('1','2')` 是有区别的,后者在本示例中不可用。因为 `Vector` 类的构造函数 (`__init__`) 接受的参数是数字类型,而不是字符串。

`__repr__` 方法返回的字符串不应有歧义,最好与源码保持一致,以方便重新创建 `__repr__` 所展示的对象。鉴于此, `Vector` 类中的 `__repr__` 方法,以类似构造函数调用的形式(如 `Vector(3,4)`)返回 `Vector` 对象的字符串表示形式。

与 `__repr__` 相比, `__str__` 方法由内置函数 `str()` 调用,供 `print` 函数使用,返回对终端用户友好的字符串。

有时候, `__repr__` 方法返回的字符串已足够友好,此时则无需再定义 `__str__` 方法。继承自 `object` 类的用户类中,若未定义 `__str__` 方法,则会自动调用 `__repr__` 方法。



在 Python 中,如果必须在 `__repr__` 与 `__str__` 之间二选一,则建议选择 `__repr__`。Stack Overflow 网站中有一个问题,“[What is the difference between `__str__` and `__repr__`?](#)”,Python 专家 Alex Martelli 和 Martijn Pieters 对此做了详尽的解答。

1.3.3 自定义类型的布尔值

Python 有一个 `bool` 类型,可以在需要布尔值的地方处理对象。例如, `if` 或 `while` 语句的条件表达式,或运算符 `and`、`or`、`not` 的操作数。为了确定对象 `x` 表示的值是 `True` 或 `False`,Python 会在幕后调用 `bool(x)`,将返回 `True` 或 `False`。

默认情况下,用户定义类的所有实例都是 `True`,除非类中实现了 `__bool__` 或 `__len__` 特殊方法。具体来说,当用户调用 `bool(x)` 时,Python 将在幕后调用 `x.__bool__()` 方法;若未实现 `__bool__()`,则 Python 将继续尝试调用 `x.__len__()` 方法;若 `x.__len__()` 返回 0,则 `bool(x)` 返回 `False`,否则返回 `True`。

“[示例 1.2<10页>](#)”中实现的 `__bool__` 方法逻辑很简单。如果向量的“模”为 0,则返回 `False`,否则返回 `True`。因为 `__bool__` 方法必须返回一个布尔值,所以在 `__bool__` 方法中使用 `bool(abs(self))` 将向量的“模”转换为布尔值。在 `__bool__` 方法外部,很少需要显式调用 `bool()`,因为任何对象都可以直接在布尔值上下文⁷中使用。

注意,这个 `__bool__` 特殊方法遵守《[The Python Standard Library](#)》中“[Built-in Types](#)”一章定义的真值测试规则。

⁷布尔值上下文:指 `if` 或 `while` 语句的条件表达式,或逻辑运算符 `and`、`or`、`not` 的操作数等。

“示例 1.2<10页>”中的 `Vector.__bool__` 方法也可以像如下这样简单定义。

```
1 def __bool__(self):
2     return bool(self.x or self.y)
```

这样定义虽然可读性差,但是无需经过 `abs` 和 `__abs__` 处理,也无需计算平方与平方根。因为 `__bool__` 方法必须返回一个布尔值,因此需要用 `bool` 函数将返回值显式转换为 `bool` 类型。`or` 运算符原封不动返回 2 个操作数(`x` 与 `y`)中的其中一个:若 `x` 为 `True`,则 `x or y` 的结果为 `x`;否则,结果为 `y`(无论 `y` 为 `True` 或 `False`)。

1.3.4 容器 (collection) API

Python 语言中的基本 (essential) 容器类型⁸的接口如图 1.2 所示。图中所有的类都是抽象基类 (ABCs)。抽象基类与 `collections.abc` 模块将在“[十三 接口、协议与抽象基类](#)”中介绍。本节简要介绍 Python 中最重要的容器接口,并展示这些接口是如何由特殊方法构建的。

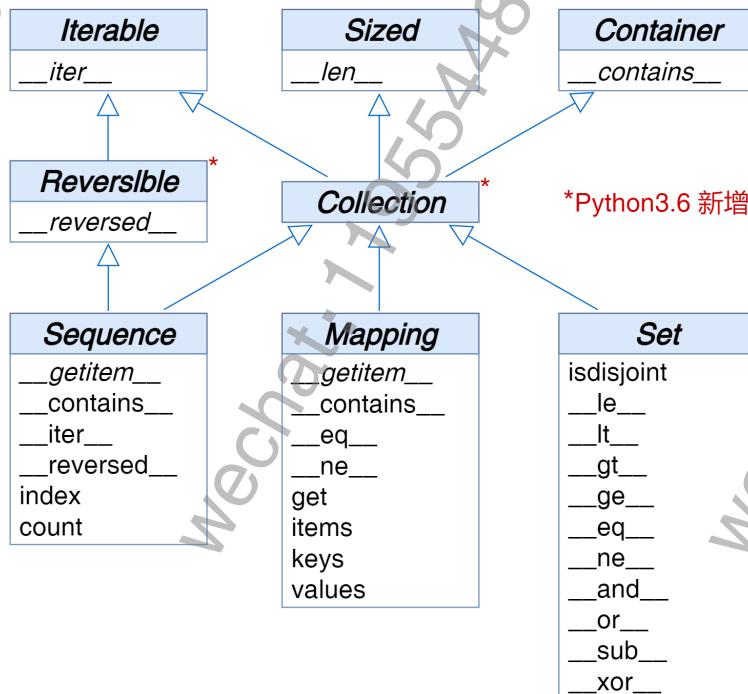


图 1.2: 基本容器类型的 UML 类图⁹

如图 1.2 所示,3 个顶层抽象基类 (ABCs) (`Iterable`、`Sized`、`Container`) 都只含 1 个特殊方法。抽象基类 (ABCs) `Collection` (Python 3.6 新增) 统一了每个容器类型都应实现的 3 个基本接口 (即 `__iter__`、`__len__`、`__contains__`):

- `Iterable` 需要支持 `for`、拆包和其他迭代方式;
- `Sized` 需要支持内置函数 `len`;
- `Container` 需要支持 `in` 运算符;

⁸译者注:本书为了区分 `collection` 与 `set`,在翻译时将 Python 内置类型 `set` 翻译为“集合”,而将 `collection` 翻译为“容器”。

⁹图 1.2 中以斜体显示的方法名称是抽象方法 (Abstract Method),必须由具体子类 (Concrete Subclass) (如 `list` 和 `dict`) 实现。其他方法由基类实现,子类可以直接继承。

Python 并不要求 **具体类** (Concrete Class) 去实际继承这些 **抽象基类** (ABCs)。只要定义的类中实现了 `__len__` 方法, 就说明这个类可满足 `Size` 接口 (即可视为 `Size` 类的子类)。

`Collection` 有 3 个很重要的专用接口:

- `Sequence`: 规范了 `list`、`str` 等内置类型的接口;
- `Mapping`: 由 `dict`、`collections.defaultdict` 等实现;
- `Set`: 是 `set` 和 `frozenset` 这 2 个内置类型的接口;

只有 `Sequence` 实现了 `Reversible`, 因为序列需要支持以任意顺序排列其中的项, 而 `Mapping` 和 `Set` 不用。



自 Python 3.7 开始, 内置 `dict` 类型也正式有顺序 (ordered) 了, 但只是保留“键 (Key)”的插入顺序, 无法随意重新排列 `dict` 中的“键 (Key)”。

抽象基类 (ABCs) `Set` 中的所有特殊方法实现的都是中缀运算符。例如, 特殊方法 `__and__` 实现了交集运算符 (&), 即 `a&b` 用于计算集合 `a` 与 `b` 的交集。

接下来的两章会详细介绍标准库中的序列 (sequences)、映射 (mappings) 和集合 (sets)。

现在, 我们来看看 Python Data model 中定义的特殊方法的主要类别。

1.4 特殊方法概述

《The Python Language Reference》中的 3. Data model 列出了 80 多个特殊方法名称, 其中半数以上用于实现算术运算符、按位运算符和比较运算符。下面几个表格概述了这些可用的特殊方法。

表 1.1 中不包含用于实现中缀运算符与核心数学函数 (如 `abs`) 的特殊方法。表中的大多数特殊方法在本书中都有所涉及, 包括近期新增的: 如异步特殊方法 `__anext__` (Python 3.5 新增)、为类定义钩子 (hook) 的 `__init_subclass__` (Python 3.6 新增)。

表 1.1: 特殊方法名称 (不含运算符、核心数学函数)

分类	方法名称
字符串 (字节) 表示形式	<code>__repr__</code> 、 <code>__str__</code> 、 <code>__format__</code> 、 <code>__bytes__</code> 、 <code>__fspath__</code>
转换为数字	<code>__bool__</code> 、 <code>__complex__</code> 、 <code>__int__</code> 、 <code>__float__</code> 、 <code>__hash__</code> 、 <code>__index__</code>
模拟容器 (collection)	<code>__len__</code> 、 <code>__getitem__</code> 、 <code>__setitem__</code> 、 <code>__delitem__</code> 、 <code>__contains__</code>
迭代 (iteration)	<code>__iter__</code> 、 <code>__aiter__</code> 、 <code>__next__</code> 、 <code>__anext__</code> 、 <code>__reversed__</code>
可调用对象或 协程 (Coroutine) 的执行	<code>__call__</code> 、 <code>__await__</code>
上下文 (context) 管理	<code>__enter__</code> 、 <code>__exit__</code> 、 <code>__aexit__</code> 、 <code>__aenter__</code>
实例构建与销毁 (析构)	<code>__new__</code> 、 <code>__init__</code> 、 <code>__del__</code>
属性 (attribute) 管理	<code>__getattr__</code> 、 <code>__getattribute__</code> 、 <code>__setattr__</code> 、 <code>__delattr__</code> 、 <code>__dir__</code>
属性 (attribute) 描述符	<code>__get__</code> 、 <code>__set__</code> 、 <code>__delete__</code> 、 <code>__set_name__</code>
抽象基类 (ABCs)	<code>__instancecheck__</code> 、 <code>__subclasscheck__</code>
类元编程 (Class Metaprogramming)	<code>__prepare__</code> 、 <code>__init_subclass__</code> 、 <code>__class_getitem__</code> 、 <code>__mro_entries__</code>

中缀运算符与数值 (numerical) 运算符由表 1.2 中的特殊方法提供支持。其中, `__matmul__`、`__rmatmul__` 和 `__imatmul__` 是 Python 3.5 新增的, 用于实现矩阵乘法的中缀运算符 @ (详见“[16.6 将 @ 当作中缀运算符使用](#)”)。

表 1.2: 运算符的符号及背后的特殊方法

运算符分类	符号	方法名称
一元数值运算符	<code>-</code> 、 <code>+</code> 、 <code>abs()</code>	<code>__neg__</code> 、 <code>__pos__</code> 、 <code>__abs__</code>
比较运算符	<code><</code> 、 <code><=</code> 、 <code>==</code> 、 <code>!=</code> 、 <code>></code> 、 <code>>=</code>	<code>__lt__</code> 、 <code>__le__</code> 、 <code>__eq__</code> 、 <code>__ne__</code> 、 <code>__gt__</code> 、 <code>__ge__</code>
算术运算符	<code>+</code> 、 <code>-</code> 、 <code>*</code> 、 <code>/</code> 、 <code>//</code> 、 <code>%</code> 、 <code>@</code> 、 <code>divmod()</code> 、 <code>round()</code> 、 <code>**</code> 、 <code>pow()</code>	<code>__add__</code> 、 <code>__sub__</code> 、 <code>__mul__</code> 、 <code>__truediv__</code> 、 <code>__floordiv__</code> 、 <code>__mod__</code> 、 <code>__matmul__</code> 、 <code>__divmod__</code> 、 <code>__round__</code> 、 <code>__pow__</code> 、 <code>__radd__</code> 、 <code>__rsub__</code> 、 <code>__rmul__</code> 、 <code>__rtruediv__</code> 、 <code>__rfloordiv__</code> 、 <code>__rmod__</code> 、 <code>__rmatmul__</code> 、 <code>__rdivmod__</code> 、 <code>__rpow__</code>
反向算术运算符	交换运算符的操作数	
增量赋值算术运算符	<code>+=</code> 、 <code>-=</code> 、 <code>*=</code> 、 <code>/=</code> 、 <code>//=</code> 、 <code>%=</code> 、 <code>@=</code> 、 <code>**=</code>	<code>__iadd__</code> 、 <code>__isub__</code> 、 <code>__imul__</code> 、 <code>__itruediv__</code> 、 <code>__ifloordiv__</code> 、 <code>__imod__</code> 、 <code>__imatmul__</code> 、 <code>__ipow__</code>
按位运算符	<code>&</code> 、 <code>l</code> 、 <code>^</code> 、 <code><<</code> 、 <code>>></code> 、 <code>~</code>	<code>__and__</code> 、 <code>__or__</code> 、 <code>__xor__</code> 、 <code>__lshift__</code> 、 <code>__rshift__</code> 、 <code>__invert__</code>
反向按位运算符	交换按位运算符的操作数	<code>__rand__</code> 、 <code>__ror__</code> 、 <code>__rxor__</code> 、 <code>__rlshift__</code> 、 <code>__rrshift__</code>
增量赋值按位运算符	<code>&=</code> 、 <code>l=</code> 、 <code>^=</code> 、 <code><<=</code> 、 <code>>>=</code>	<code>__iand__</code> 、 <code>__ior__</code> 、 <code>__ixor__</code> 、 <code>__ilshift__</code> 、 <code>__irshift__</code>



若第 1 个操作数对应的特殊方法不可用, 则 Python 会在第 2 个操作数上调用反向运算符对应的特殊方法。增量赋值是将中缀运算符与变量赋值相结合的简写形式, 如 `a+=b` 是 `a=a+b` 的简写形式。“[十六 运算符重载](#)”会详细介绍反向运算符和增量赋值。

1.5 len 为什么不是方法

此问题我在 2013 年问过核心开发人员 Raymond Hettinger, 他在回答时引用了《[The Zen of Python](#)》¹⁰中的一句话: “`rationality beats purity`”¹¹。“[1.3 特殊方法是如何使用的](#)”中提过, 当 `x` 是内置类型的实例时, `len(x)` 运行速度非常快。因为计算 CPython 内置对象的长度时, 无需调用任何方法, 而是直接读取 C 语言结构体中的字段。获取容器类型中元素的项数 (items) 是一个常见操作, `str`、`list`、`memoryview` 等各种基本的容器类型都必须高效地完成这项工作。

换句话说, `len()` 之所以不是方法, 是因为它会被 CPython 特殊处理, 就像 `abs` 函数一样, 被视作 Python Data Model (数据模型) 的一部分。但是, 借助特殊方法 `__len__`, 也可以让 `len` 适用于自定义对象。这是一种折中方案: 既满足了内置对象对速度的要求, 又保证了语言的一致性。这也体现了《[The Zen of Python](#)》中的另一句话: “`Special cases aren't special enough to break the rules.`”¹²

¹⁰在 Python 控制台中执行 “`import this`” 语句, 可以直接显示《[The Zen of Python](#)》的内容。

¹¹`rationality beats purity`: 可理解为在实践中, 实用性比纯粹性更重要。这里的“纯粹性”可理解为严格遵守编程语言的规则或规范。

¹²`Special cases aren't special enough to break the rules`: 中文译文“特殊情况不是破坏规则的理由。”



Python 源自 ABC 语言^a, 很多特性继承自 ABC。ABC 中的 # 运算符与 Python 中的 len 作用相同, 写作 #s。ABC 中的 # 也可作为中缀运算符使用, 如 x#s 表示计算 s 中有多少个 x。对应到 Python 中, 写作 s.count(x) (s 是某种序列)。

^aABC 语言是由荷兰计算机科学家 Edsger W. Dijkstra 等人在 1980 年代初开发的编程语言。其设计目标是简单性和易学性, 旨在提供一种易于理解和使用的高级编程语言, 特别适合教学和初学者。

1.6 本章小结

通过实现特殊方法, 可使自定义对象的行为如同内置类型一样, 从而写出符合 Python 风格 (社区认可的) 且更具表现力的代码。

Python 对象基本上都需要提供一个可用的字符串表示形式, 以便在调试、记录日志和向终端用户展示时使用。这就是 Python Data Model (数据模型) 中存在 `__repr__`、`__str__` 这 2 个特殊方法的原因。

模拟序列的行为 (如 “[示例 1.1<5页>](#)”) 是特殊方法常见用途之一。比如, 数据库返回的查询结果往往就是一个类似序列的容器 (collection)。“[二 丰富的序列](#)”会具体介绍如何充分利用现有的序列类型。“[十二 序列的特殊方法](#)”将介绍如何实现自己的序列类型, 届时将在 `Vector` 类 (“[示例 1.2<10页>](#)”) 的基础上创建一个多维向量类。

得益于运算符重载, Python 提供了丰富的数字类型, 除了内置的数值 (numeric) 类型之外, 还有 `decimal.Decimal` 和 `fractions.Fraction`, 这些数字类型都支持中缀运算符。数据科学库 NumPy 还提供了支持矩阵和张量的中缀运算符。“[十六 运算符重载](#)”中增强 `Vector` 类的示例将实现更多的运算符 (含反向运算符和增量赋值运算符)。

本书还将介绍 Python 数据模型其余大部分特殊方法的使用和实现。

1.7 延伸阅读

本章及本书大部分内容参考了《The Python Language Reference》中的“3. Data model”, 这是最权威的资料。

《Python in a Nutshell, 3rd, Ed》(Alex Martelli、Anna Ravenscroft、Steve Holden 合著) 对 Python Data Model (数据模型) 的讲解很精彩。书中对属性 (attribute) 访问机制的解说是我见过除 CPython 源码之外最权威的。Martelli 还经常在 [Stack Overflow](#) 中回答问题, 目前已回答 6200 多个问题。

David Beazley 著有 2 本基于 Python 3 的书, 都对 Python Data Model (数据模型) 做了详尽介绍。一本是《Python Essential Reference, 4th Ed》, 另一本是与 Brian K. Jones 合著的《Python Cookbook, 3rd Ed》。

[The Art of the Metaobject Protocol](#) (Gregor Kiczales、Jim des Rivieres、Daniel G. Bobrow 合著) 解读了元对象 (metaobject) 协议的概念, Python Data Model (数据模型) 就是元对象协议的一个例子。

杂谈

数据模型? 还是对象模型?

Python 文档中使用的术语是 “Python Data Model”, 而多数作者采用的术语是 “Python object model”。《[Python in a Nutshell, 3rd Ed](#)》以及《[Python Essential Reference, 4th Ed](#)》都对 Python Data

Model(数据模型)进行了深入解读,不过这几位作者用的术语都是“对象模型(object model)”。维基百科中,对象模型的定义是“一门特定的计算机编程语言中,对象的一般特性(properties)”。这正是“Python Data Model”所要描述的概念。本书采用“数据模型(Data Model)”这一术语,因为Python文档始终使用这个词指代Python对象模型,而且还因为《The Python Language Reference》中与本书关系最大的章节使用的标题就是“Data Model”。

麻瓜(Muggle)¹方法

按照《The Original Hacker's Dictionary》的定义,“魔法”是指“神秘莫测,或复杂到无法解释”,或者“鲜为人知的功能,可以让不可能成为可能”。

Ruby中也有类似“特殊方法”的概念,Ruby社区称之为“魔法方法”。Python社区也有很多人采用这种说法。但我认为,“特殊方法”与“魔法方法”是对立的。Python和Ruby都利用这个概念来丰富元对象(metaobject)协议。即便你我这种不会“魔法”的麻瓜(Muggle),凭借完善的文档,也可模拟出核心开发人员编写语言解释器时用到的很多功能。

Go语言就不一样了。在Go语言中,一些对象的功能确实像魔法,因为用户自己定义的类型无法模拟这些功能。例如,Go语言中的数组、字符串和映射(maps)支持使用方括号([])存取项(item),写作a[i]。但是,我们自己定义的容器类型无法使用[]表示法。更糟的是,Go语言没有可迭代接口或迭代器对象之类的概念,因此for/range句法仅支持5种“魔法”内置类型,包括数组、字符串和映射(maps)。

未来,Go语言的设计人员说不定会增强元对象(metaobject)协议。但是目前来看,与Python或Ruby相比,它的功能十分有限。

元对象(metaobject)

《The Art of the Metaobject Protocol》(以下简称AMOP)是我最喜欢的一本计算机图书。我提到这本书是因为“元对象协议”对理解Python数据模型有帮助,而且其他语言中也有类似的功能。“元对象”指构成语言自身的基本对象。在这个语境下,“协议”等同于“接口”。所以,“元对象协议”就是对象模型的高级说法,指语言核心构件的API。

一套丰富的元对象协议能让我们扩展语言,支持新的编程范式。AMOP的第一作者Gregor Kiczales后来成为面向方面编程(Aspect-Oriented Programming)的先驱,也是AspectJ(实现该范式的Java扩展)的最初作者。面向方面编程在Python这样的动态语言中实现起来更简单,一些框架就提供了这个范式,Plone内容管理系统的`zope.interface`就是一例。

¹麻瓜(Muggle),用来指代对某种特定领域或主题一无所知或不了解的人。

wechat: 119554488

丰富的序列

您可能已注意到，上述几种操作同样适用于文本、列表和表格。文本、列表和表格统称为“trains”。

FOR 命令同样也适用于“trains”。

——Leo Geurts、Lambert Meertens 和 Steven Pemberton

ABC Programmer's Handbook (第 8 页)

在 Python 诞生之前，Guido 曾为 ABC 语言贡献过代码。ABC 语言是一个历时 10 年的研究项目，旨在设计一种适合初学者的编程环境。ABC 语言中的很多理念都比较 [Pythonic](#)¹。例如，对不同类型序列的通用操作、内置元组和映射类型、源码的缩进结构、无需变量声明的强类型等。可见，Python 如此简单易用，绝非偶然。

Python 继承了 ABC 语言对序列的统一处理方式。如字符串、列表、字节序列、数组、XML 元素和数据库结果，这些序列类型在操作上具有很多共通之处，包括迭代、切片、排序和连接（concat）等。

深入理解 Python 中不同的序列类型，不但可以避免重造轮子，还可以从它们共通的接口上受到启发，在自己实现 API 时合理支持及利用现有和将来可能添加的序列类型。

本章介绍的内容大多适用于一般序列，如我们熟悉的 list 以及 Python3 中新增的 str 和 bytes。本章重点讲解 list、tuple、array 和 queue，“[四 Unicode 文本与字节序列](#)”将探讨 Unicode 字符串与 byte 字节序列相关话题。另外，本章关注的是 Python 中直接（现成）可用的序列类型；“[十二 序列的特殊方法](#)”将介绍如何创建自己的序列类型。

本章主要涵盖以下内容：

- 列表推导式和生成器表达式的基础知识；
- 元组的 2 种用法——用作记录和用作不可变列表；
- 序列解包和[序列模式（Sequence Pattern）](#)；
- 从切片读取数据和向切片写入数据；
- 专用的序列类型，如数组和队列（queue）；

¹Pythonic：可理解为具有 Python 语言风格的代码，即以 Python 方式写出的简洁、优美的代码。

2.1 本章新增内容

本章新增了“2.6 序列模式匹配”，这是在 Python 3.10 新引入的 模式匹配 (Pattern Match) 特性。

与第1版相比，对如下内容做了一些改进：

- 增加了有关序列内部机制的图表和说明；增加了容器序列与扁平序列的对比；
- 简要比较了 list 和 tuple 的性能与存储特点；
- 包含可变元素的元组需要注意的事项和应对方法；

具名元组 (named tuple) 相关话题移到了“5.3 典型的具名元组：collections.namedtuple”，与 typing.NamedTuple 和 @dataclass 放在一起探讨。

2.2 内置序列类型概览

Python 标准库中用 C 语言实现了丰富的序列类型，列举如下：

- **容器序列**：容器序列中可存放不同类型的项 (item)，其中包括嵌套容器。示例：list、tuple 和 collections.deque。
- **扁平序列**：扁平序列中可存放简单类型的项 (item)。示例：str、bytes 和 array.array。

容器序列中存放的是其所包含对象的引用，对象可以是任何类型；扁平序列中存放的是其所包含内容的值，而不是各自不同的 Python 对象。详见图 2.1。

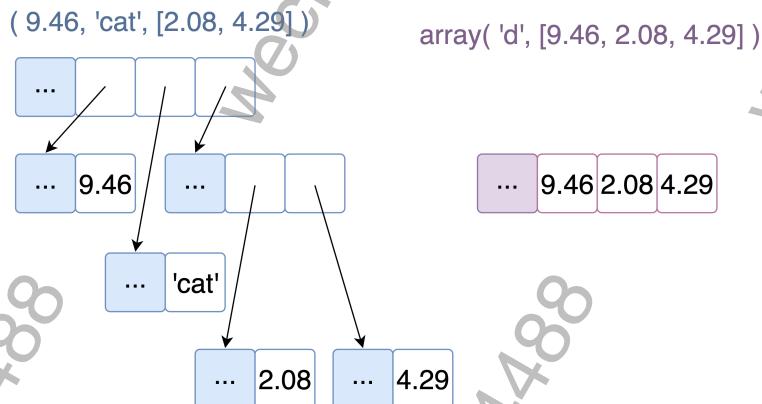


图 2.1：元组与数组的内存简图²

因此，扁平序列更加紧凑，但是只能用于存储原始机器值 (Primitive machine values)³，如 byte、int 和 float 等。

²彩色方块表示各个 Python 对象的内存标头 (header)。元组中的每一项都是指向不同 Python 对象的引用，Python 对象中还可以存放其他 Python 对象的引用，例如元组中的包含 2 项的列表。相比之下，Python 中的数组，其整体是一个对象，存放一个包含 3 个双精度数的 C 语言数组。

³“Primitive machine values (原始机器值)”是指计算机底层可以直接处理的基本数据类型，如 byte、int 和 float 等。这些数据类型通常由硬件和编程语言直接支持，并且在计算机内存中以二进制形式存储。



内存中的每个 Python 对象都有一个包含元数据的标头(header)。最简单的 Python 对象,如一个 float,其内存标头(header)中包含 1 个值字段和 2 个元数据字段。

- `ob_refcnt`: 对象的引用计数;
- `ob_type`: 指向对象类型的指针;
- `ob_fval`: 一个 C 语言 `double` 类型值, 用于存放 float 值;

在 64 位 Python 中, 每个字段占用 8 个字节。这就是为什么浮点数数组比浮点数元组更紧凑。因为, 数组是一个单一对象, 包含浮点数的原始值, 而元组由多个对象组成, 即元组本身和元组中包含的每个浮点数对象。

另外, 还可以按可变性对序列类型进行分类:

- **可变序列 (Mutable sequences)**: 例如 `list`、`bytearray`、`array.array` 和 `collections.deque`。
- **不可变序列 (Immutable sequences)**: 例如 `tuple`、`str`、`bytes`。

可变序列继承了不可变序列的所有方法, 除此之外, 还额外多实现了几个方法(如图 2.2 所示)。

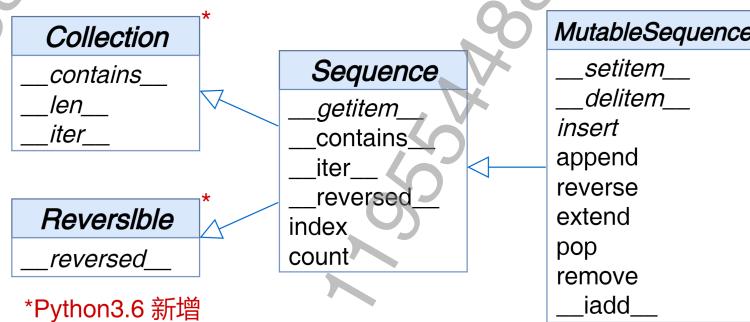


图 2.2: `collections.abc` 中部分类的简化 UML 类图⁴

内置的序列类型(`list` 与 `tuple`)实际上并不是抽象基类(ABCs) `Sequence` 与 `MutableSequence` 的子类, 而是使用这 2 个抽象基类(ABCs)注册(详见“[十三 接口、协议与抽象基类](#)”)的虚拟子类(Virtual Subclass)。因为是虚拟子类, 所以 `tuple` 与 `list` 可以顺利通过如下测试:

```

1 >>> from collections import abc
2 >>> issubclass(tuple, abc.Sequence)
3 True
4 >>> issubclass(list, abc.MutableSequence)
5 True
6 >>> issubclass(list, abc.Sequence)
7 True
  
```

记住这些序列类型的共同点: 可变序列与不可变序列; 容器序列与扁平序列。这有助于将对一种序列类型的了解延伸到不太熟悉的序列类型上。

`list` 是最基本的序列类型, 是一种可变的容器序列。这里假定您已非常熟悉 `list`, 因此接下来会跳过 `list` 的基础知识, 直接介绍[列表推导式 \(Listcomp\)](#)。列表推导式虽然功能强大, 能够方便地创建 `list`, 但其语法可能让人一开始觉得难以理解, 因此并没有得到足够的重视。掌握了列表推导式, 也就为学习[生成器表达式 \(Genexp\)](#) 打下了基础。生成器表达式的功能非常强大, 例如可以生成元素来填充任何类型的序列。列表推

⁴图 2.2 中左侧是超类; 箭头从子类指向超类, 表示继承; 以斜体显示的名称是抽象类和抽象方法(Abstract Method)。

导式与生成器表达式将在“[2.3 列表推导式与生成器表达式](#)”中探讨。

2.3 列表推导式与生成器表达式

使用[列表推导式 \(Listcomp\)](#)（针对 list）或[生成器表达式 \(Genexp\)](#)（针对其他序列类型）可以快速地构建一个序列。使用这 2 种句法写出的代码更易于理解，而且速度也更快。



为了描述简洁，许多 Python 程序员会将[列表推导式 \(Listcomp\)](#)称为“listcomps”，将[生成器表达式 \(Genexp\)](#)称为“geneexps”。

2.3.1 列表推导式对可读性的影响

请阅读[示例 2.1](#)与[示例 2.2](#)，您觉得哪个代码片段更易于理解？

</> **示例 2.1：**用 for 循环，基于字符串构建一个 Unicode 码点列表

```

1  >>> symbols = '$¢¥€¤'
2  >>> codes = []
3  >>> for symbol in symbols:
4      ...     codes.append(ord(symbol))
5  ...
6  >>> codes
7  [36, 162, 163, 165, 8364, 164]
```

</> **示例 2.2：**用列表推导式，基于字符串构建一个 Unicode 码点列表

```

1  >>> symbols = '$¢¥€¤'
2  >>> codes = [ord(symbol) for symbol in symbols]
3  >>> codes
4  [36, 162, 163, 165, 8364, 164]
```

稍微懂一点 Python 即可读懂[示例 2.1](#)，但是学会了[列表推导式 \(Listcomp\)](#)之后，您会发现[示例 2.2](#)更易于理解，因其意图更明确。

for 循环可以胜任许多任务，如遍历序列、统计序列项数、挑选序列部分项、聚合计算（计算总数和平均数）等。[示例 2.1](#) 使用 for 循环构建一个 list。

当然，若滥用[列表推导式 \(Listcomp\)](#)，会写出可读性差的代码。如果您不打算使用生成的 list 列表，就不要使用[列表推导式 \(Listcomp\)](#)语法。我曾见过一些 Python 代码，用[列表推导式 \(Listcomp\)](#)重复执行代码片段，这是不可取的。另外，[列表推导式 \(Listcomp\)](#)应保持简短。若超过 2 行，则最好将语句拆开，或者用传统的 for 循环改写。写 Python 代码如同写文章一样，尺度需要自己把握，没有硬性规定。



句法提示

Python 会忽略 []、{} 和 () 内部的换行。因此, list、[列表推导式 \(Listcomp\)](#)、tuple、dict 等结构可以拆分成多行来写, 无需使用续行转义符 \。若不小心在续行转义符后多输入一个空格, 那反而不起作用。另外, 使用这 3 种符号定义字面量时, 项与项之间使用逗号分隔, 末尾的逗号将被忽略。因此, 跨多行定义 list 字面量时, 最好在最后一项后面添加一个逗号。这样不仅能方便其他程序员为 list 添加更多项, 还可以减少代码差异给阅读带来的干扰。

列表推导式与生成器表达式的局部作用域

Python 3 中的 [列表推导式 \(Listcomp\)](#)、[生成器表达式 \(Genexp\)](#)、[集合推导式](#)、[字典推导式](#)、[for 子句](#) 中赋值的变量, 都在局部作用域内。

然而, 使用 [海象运算符 \(:=\)](#) 赋值的变量, 在推导式或生成器表达式返回后依然可以被访问, 这与函数内的局部变量行为不同。根据 “[PEP 572-Assignment Expressions](#)” 的约定, 海象运算符 (:=) 赋值的变量, 其作用域限定在函数内, 除非目标变量使用 `global` 或 `nonlocal` 声明。

```

1  >>> x = 'ABC'
2  >>> codes = [ord(x) for x in x]
3  >>> x      ❶
4  'ABC'
5  >>> codes
6  [65, 66, 67]
7  >>> codes = [last := ord(c) for c in x]
8  >>> last   ❷
9  67
10 >>> c      ❸
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13     NameError: name 'c' is not defined

```

- ❶ x 毫发无损, 绑定的值仍为 'ABC';
- ❷ last 依然可以被访问;
- ❸ c 消失了, 因为它只存在于[列表推导式 \(Listcomp\)](#) 内部;

使用[列表推导式 \(Listcomp\)](#)可以筛选或转换可迭代类型(含序列)中的项, 并以此来构建新的 list。使用 `filter` 与 `map` 这 2 个内置函数, 也可以实现此效果, 但是写出的代码, 可读性比较差。

2.3.2 列表推导式与 filter 和 map 比较

[列表推导式 \(Listcomp\)](#)涵盖了 `filter` 和 `map` 这 2 个函数的功能, 并且写出的代码不像 Python 的 `lambda` 表达式那样晦涩难懂。请看 [示例 2.3](#)。

</> [示例 2.3: 用列表推导式和 map/filter 构建列表比较](#)

```

1  >>> symbols = '$€¥€¤'
2  >>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]

```

```

3  >>> beyond_ascii
4  [162, 163, 165, 8364, 164]>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord,
5      symbols)))
6  [162, 163, 165, 8364, 164]

```

我原以为使用 map 和 filter 速度会更快,但是 Alex Martelli 告诉我事实并非如此。[随书代码库](#) 中的 `02-array-seq/listcomp_speed.py` 脚本对 [列表推导式 \(Listcomp\)](#) 与 filter/map 的速度进行了简单对比。

map 和 filter 将在本书的“[七函数是一等对象](#)”中深入讨论。下面来看一下如何用 [列表推导式 \(Listcomp\)](#) 计算笛卡尔积,即构建一个 list,其中包含 2 个或更多 list 中的所有项组成的 tuple。

2.3.3 笛卡尔积

[列表推导式 \(Listcomp\)](#) 可以根据 2 个或更多可迭代对象的笛卡尔积构建新 list。笛卡尔积的每一项是一个 tuple(由输入的各个可迭代对象中的项构成)。得到的新 list 长度等于输入的各个可迭代对象长度的乘积,如图 2.3 所示。

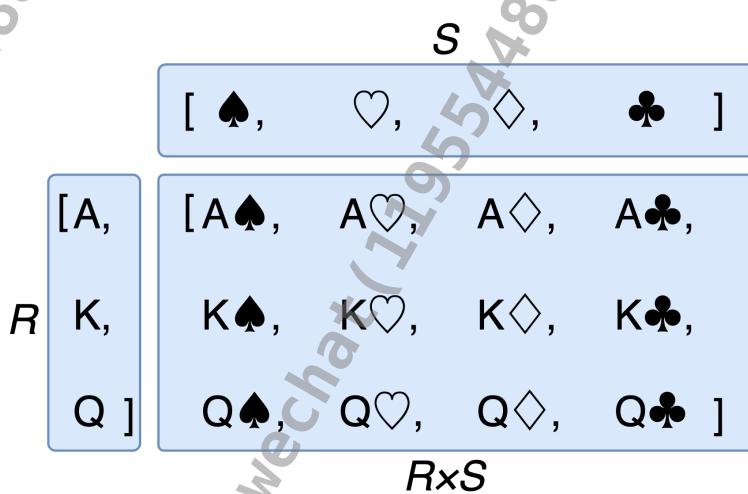


图 2.3: 扑克牌的 3 种点数与 4 种花色的笛卡尔积,是一个包含 12 种组合的序列。

假设想生成一个 list,其中包含 2 种颜色和 3 种尺寸的 T 恤衫。[示例 2.4](#) 使用列表推导式生成这个 list,得到的 list 包含 6 项。

</> [示例 2.4: 使用列表推导式生成笛卡尔积](#)

```

1  >>> colors = ['black', 'white']
2  >>> sizes = ['S', 'M', 'L']
3  >>> tshirts= [(color, size) for color in colors for size in sizes] ❶
4  >>> tshirts
5  [('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'), ('white', 'M'), ('white', 'L')]
6  >>> for color in colors: ❷
...     for size in sizes:
8  ...         print((color, size))
9  ...
10 ('black', 'S')

```

```

11 ('black', 'M')
12 ('black', 'L')('white', 'S')
13 ('white', 'M')
14 ('white', 'L')
15 >>> tshirts = [(color, size) for size in sizes for color in colors] ❸
16 >>> tshirts
17 [('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'), ('black', 'L'), ('white', 'L')]

```

- ❶ 先按颜色 (color) 再按尺寸 (size) 排列, 生成一个由 tuple 组成的 list;
- ❷ 2 个 for 循环的嵌套方式与列表推导式中 for 子句的先后顺序一致;
- ❸ 若先按尺寸 (size) 再按颜色 (color) 排列, 则只需调整 for 子句的顺序;

在“一 Python 数据模型”的示例 1.1 中, 使用以下表达式初始化一副纸牌, 这副纸牌有 52 张, 分为 13 种点数和 4 种花色, 先按花色再按点数排列。

```

1 self._cards = [Card(rank, suit) for suit in self.suits
2                         for rank in self.ranks]

```

列表推导式 (Listcomp) 的作用很单一, 只用于构建 list。若想生成其他类型的序列, 则应使用 生成器表达式 (Genexp)。“2.3.4 生成器表达式”会简要介绍如何用 生成器表达式 (Genexp) 构建除 list 以外的序列。

2.3.4 生成器表达式

可以用 列表推导式 (Listcomp) 初始化 tuple、array 和其他类型的序列, 但用 生成器表达式 (Genexp) 初始化序列可以节省更多内存。因为 生成器表达式 (Genexp) 按照迭代器协议按需逐个产出序列中的项 (item), 而不是在内存中一次性构建出完整的序列。

生成器表达式 (Genexp) 与列表推导式语法一样, 只是将列表推导式的方括号 ([]) 替换成圆括号 (())。示例 2.5 展示了用生成器表达式构建 tuple 与 array 的基本用法。

</> 示例 2.5: 用生成器表达式构建元组和数组

```

1 >>> symbols = '$€¥¤¤'
2 >>> tuple(ord(symbol) for symbol in symbols) ❶
3 (36, 162, 163, 165, 8364, 164)
4 >>> import array
5 >>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
6 array('I', [36, 162, 163, 165, 8364, 164])

```

- ❶ 若 生成器表达式 (Genexp) 是函数唯一的参数, 则无需额外再用圆括号将生成器表达式括起来;
- ❷ 因为 array.array() 构造函数需要 2 个参数, 所以必须用圆括号 () 将 生成器表达式 (Genexp) 括起来。array.array() 函数的第 1 个参数指定 array 中值的存储类型, 详见“2.10.1 数组<50页>”。

示例 2.6 用生成器表达式生成笛卡尔积, 打印 2 种颜色 3 种尺寸的 T 恤衫组合。与示例 2.4 不同, 这次 6 种 T 恤衫组成的列表不在内存中构建, 生成器表达式每次产出一项 (item), 提供给 for 循环。若 2 个 list 各有 1000 项, 则使用生成器表达式生成笛卡尔积可以节省大量内存, 因为无需在内存中一次性构建出包含 100 万

项的完整 list 提供给 for 循环。

```
</> 示例 2.6: 用生成器表达式生成笛卡尔乘积
1  >>> colors = ['black', 'white']
2  >>> sizes = ['S', 'M', 'L']
3  >>> for tshirt in (f'{c} {s}' for c in colors for s in sizes): ❶
4  ...     print(tshirt)
5  ...
6  black S
7  black M
8  black L
9  white S
10 white M
11 white L
```

- ❶ 生成器表达式逐个产出序列中的项 (item)。此示例不会在内存中一次性构建出包含 6 种 T 恤衫组合的完整 list。



“[十七 迭代器、生成器和经典协程](#)”会详细解释生成器的工作原理。本节只是简单展示如何用生成器表达式构建除列表以外的序列, 以及如何生成无需占用内存的输出。

现在, 我们继续讨论 Python 中的另一种基本序列类型: 元组 (tuple)。

2.4 元组不仅仅是不可变列表

有些 Python 入门教程将元组 (tuple) 称作“不可变列表”, 但是这一表述并未完全概括元组的特点。元组有双重用途: 既可用作不可变列表, 又可用作没有字段名的记录 (record)。后一种用法往往会被忽略, 那么就从此种用法开始讲起吧。

2.4.1 将元组用作记录

用元组保存记录 (record): 元组 (tuple) 中的每一项 (item) 保存一个字段的数据, 项的位置决定了数据的含义。

若只将元组用作不可变列表, 那么元组中的项数及顺序即变得不那么重要。但若将元组用作字段的容器 (如数据表的一行记录), 那么元组中的项数通常是固定的, 项的顺序也变得十分重要。

[示例 2.7](#) 将元组当作记录 (record) 使用。请注意, 在每个表达式中, 若对元组排序将会破坏元组携带的信息, 因为每个字段的含义都取决于字段在元组中的位置。

```
</> 示例 2.7: 将元组当作记录使用
1  >>> lax_coordinates = (33.9425, -118.408056) ❶
2  >>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66, 8014)
❷
```

```
3  >>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ('ESP', 'XDA205856')]
4      ③
5  >>> for passport in sorted(traveler_ids):
6      ...     print('%s/%s' % passport)
7      ...
8  BRA/CE342567
9  ESP/XDA205856
10  USA/31195855
11  ...
12  ...
13  USA
14  BRA
15  ESP
```

- ① 洛杉矶国际机场的经纬度；
- ② 东京市的一些信息：市名、年份、人口（单位：千）、人口变化（单位：%）、面积（单位： km^2 ）；
- ③ 一个由元组（tuple）构成的 list，元组的形式为：(country_code, passport_number)；
- ④ 在迭代列表的过程中，将 passport 变量绑定到列表中的每个元组上；
- ⑤ 用格式化运算符 % 解析元组结构，并将元组中的每个项（item）视为单独的字段；
- ⑥ for 循环知道如何获取元组中每个单独的项，此过程称为“解包”。此处，我们对元组中的第二项不感兴趣，因此将其赋值给虚拟变量 _。



一般习惯用 _ 表示虚拟变量。但是，在 match/case 语句中，_ 是通配符，用于匹配值但不绑定值（详见“[2.6 序列模式匹配](#)”）。另外，在 Python 控制台中，前一个命令的结果若不是 None，则会被赋值给 _。

看到“记录（record）”这个词语，我们往往认为它是一种带有字段名称的数据结构。在“[五 数据类构建器](#)”中，会提供 2 种创建具名字段（named fields）元组的方法。

通常没必要为了给字段指定名称而专门创建一个类。若不使用索引，只是用解包访问字段，就更没有必要了。在[示例 2.7](#)中，仅使用一个语句就将 ('Tokyo', 2003, 32_450, 0.66, 8014) 分别赋值给了 city、year、pop、chg、area。随后，在 print 函数的参数中，用% 运算符将 passport 元组中的各项分别赋值给了格式化字符串中对应的占位符（%s）。这 2 处用到的都是元组解包（tuple unpacking）。



大多数 Python 程序员都用“元组解包”这种说法，但是“可迭代对象解包（iterable unpacking）”的说法也在逐步增加，例如“[PEP 3132 —Extended Iterable Unpacking](#)”。

“[2.5 序列和可迭代对象解包<30页>](#)”中将深入介绍解包，内容包含元组、序列以及一般意义上的可迭代对象的解包。

接下来，我们将 tuple 视作不可变的 list 列表。

2.4.2 将元组用作不可变列表

Python 解释器及标准库经常将 tuple 当作不可变的 list 来使用。这么做主要有 2 点好处：

- **意图清晰**：只要在源码中见到 tuple，您就可以知道它的长度永远不会变化；
- **性能优越**：长度相同的 tuple 与 list 相比，tuple 占用的内存更少，而且 Python 可以对 tuple 做某些优化。

但是，要知道，元组的不可变性仅是针对元组中的引用而言，即元组中的引用不可删除、不可替换。倘若引用的是可变对象，在该可变对象发生改变后，元组的值也会随之变化。下边的代码通过创建 2 个元组 (a 和 b) 来演示这一现象。起初， $b == a$ ，此时 b 在内存中的初始布局如 图 2.4 所示。

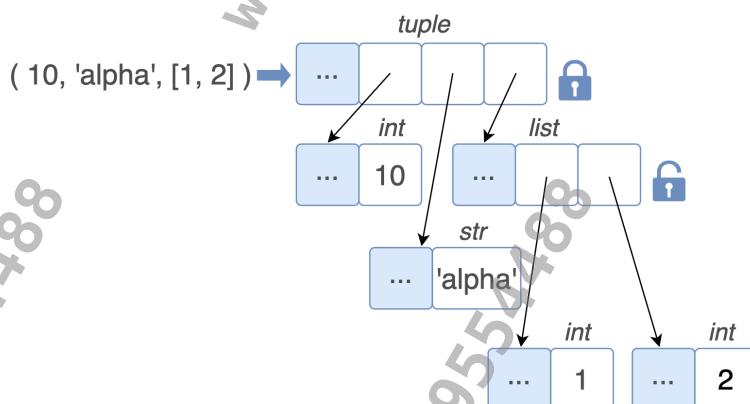


图 2.4: 元组引用可变对象⁵

我们修改 b 中的最后一项，此时 $b \neq a$ 。

```

1  >>> a = (10, 'alpha', [1,2])
2  >>> b = (10, 'alpha', [1,2])
3  >>> a == b
4  True
5  >>> b[-1].append(99)
6  >>> a == b
7  False
8  >>> b
9  (10, 'alpha', [1, 2, 99])

```

在元组 (tuple) 中存放可变对象可能导致 BUG。如 小节 3.4.1 所示，只有当对象的值不会改变时，这个对象才是 可哈希 (hashable)。不可哈希的元组不能作为字典 (dict) 的“键 (Key)”，也不能作为集合 (set) 的元素。

可以使用内置函数 hash() 定义如下所示的 fixed 函数，来判断对象的值是否是不可变的。

```

1  >>> def fixed(o):
2  ...     try:
3  ...         hash(o)
4  ...     except TypeError:
5  ...         return False

```

⁵元组的内容自身是不可变的，但是这仅仅表明元组中存放的引用始终指向同一批对象。若引用的是可变对象（如 list），那么元组的值可能随着可变对象的变化而变化。

```
6 ...     return True
7 ...>>> tf = (10, 'alpha', (1,2))
8 >>> tm = (10, 'alpha', [1,2])
9 >>> fixed(tf)
10 True
11 >>> fixed(tm)
12 False
```

小节 6.3.2 会进一步探讨元组 (tuple) 的相对不可变性。

Stack Overflow 网站中有一个问题 “[Are tuples more efficient than lists in Python?](#)”。Python 核心开发人员 Raymond Hettinger 在回答此问题时, 指出了元组在性能上的一些优势, 总结如下:

- Python 编译器在求解元组字面量时, 只需经过一次操作即可生成元组常量的字节码。而 Python 编译器在求解列表字面量时, 生成的字节码需要将每个元素都当作独立的常量推入堆栈, 然后再构建列表。
- 给定一个元组 `t`, 函数 `tuple(t)` 不会复制元组 `t`, 而是直接返回 `t` 的引用。相比之下, 给定一个列表 `l`, 函数 `list(l)` 将会创建列表 `l` 的副本。
- `tuple` 实例的长度固定, 分配的内存空间正好够用。而 `list` 实例会预分配一些内存空间, 以便随时可向 `list` 中追加元素。
- 对于元组来说, 元组中的每个项的引用 (指向实际对象的指针) 是直接存储在元组的内部数组中。这意味着元组的内部数据结构包含了一个数组, 而数组的每个元素都指向元组中的一个项。而对于列表来说, 列表中的每个项的引用是存储在另外一个地方的数组中的, 而列表本身只是存储了对这个数组的引用 (即指针)。这个数组可以在内存中的任意位置, 与列表的数据结构分开存储。当列表增长到超过当前分配的空间时, Python 需要重新分配更大的数组来容纳更多的引用, 这会导致 CPU 缓存效率降低。

2.4.3 元组与列表中的方法比较

当将元组当作列表的不可变变体使用时, 有必要了解二者 API 之间的差异。如 [表 2.1](#) 所示, 元组 (tuple) 支持所有不涉及增、删项的 list 方法, 而且元组没有 `_reversed_` 方法。即便没有 `_reversed_` 方法, `reversed(my_tuple)` 也能正常工作。

表 2.1: 列表和元组的方法及属性

方法/属性	列表	元组	描述
<code>s.__add__(s2)</code>	•	•	<code>s+s2</code> : 将 <code>s</code> 与 <code>s2</code> 拼接
<code>s.__iadd__(s2)</code>	•		<code>s+=s2</code> : 就地拼接, 即 <code>s=s+s2</code>
<code>s.append(e)</code>	•		在 <code>s</code> 最后一个元素后面追加一个元素 <code>e</code>
<code>s.clear()</code>	•		删除 <code>s</code> 的所有项

接下页

续表 2.1

方法/属性	列表	元组	描述
s.__contains__(e)	•	•	e in s
s.copy()	•		浅拷贝列表 s
s.count(e)	•	•	计算元素 e 在 s 中出现的次数
s.__delitem__(p)	•		删除列表 s 中位置 p 上的项
s.extend(it)	•		将可迭代对象 it 中的项追加到列表 s 中
s.__getitem__(p)	•	•	s[p]:获取 s 中位置 p 上的项
s.__getnewargs__()		•	支持使用 pickle 优化序列化
s.index(e)	•	•	找出元素 e 在 s 中首次出现的位置
s.insert(p, e)	•		在 s 中位置 p 之前插入元素 e
s.__iter__()	•	•	获取迭代器
s.__len__()	•	•	len(s):获取 s 中的项数
s.__mul__(n)	•	•	s * n:将 s 重复拼接 n 次
s.__imul__(n)	•		s *= n:将 s 就地拼接 n 次
s.__rmul__(n)	•	•	n * s:反向重复拼接 (详见“ 十六 运算符重载 ”)
s.pop([p])	•		移除并返回 s 中的最后 (或位置 p 上) 一项
s.remove(e)	•		将 s 中首次出现的元素 e 移除
s.reverse()	•		就地反转 s 中项的顺序
s.__reversed__()	•		获取从后向前遍历项的迭代器
s.__setitem__(p, e)	•		s[p] = e;将 s 中位置 p 的元素设置为 e (覆盖现有项)。也可覆盖一个子序列,详见“ 2.7.4 为切片赋值 ”
s.sort([key], [reverse])	•		就地对 s 中的项进行排序, key 与 reverse 为可选的关键字参数。

接下来,介绍一下 Python 编程中一种重要的惯用法,即元组 (tuple)、列表 (list) 和可迭代对象解包。

2.5 序列和可迭代对象解包

解包的特点是无需手动通过索引从序列中提取元素,这样可以减少出错的概率。解包的目标对象可以是任何可迭代对象,包括不支持索引表示法 ([]) 的迭代器。解包对可迭代对象的唯一要求是,每次只能产出一项,以提供给接收端的变量。不过也有例外,可以用星号 (*) 获取可迭代对象中余下的所有项,详见“[2.5.1 用 * 获取余下的项](#)”。

最明显的解包形式是并行赋值 (parallel assignment),即将可迭代对象中的所有项赋值给变量元组,如下所示:

```

1  >>> lax_coordinates = (33.9425, -118.408056)
2  >>> latitude, longitude = lax_coordinates # unpacking
3  >>> latitude
4  33.9425
5  >>> longitude
6  -118.408056

```

利用解包还可以不使用中间临时变量,轻松对调 2 个变量的值。

```
1 >>> b, a = a, b
```

调用函数时,在参数前面加上一个 *,利用的也是解包。

```
1 >>> divmod(20, 8)
2 (2, 4)
3 >>> t = (20, 8)
4 >>> divmod(*t)
5 (2, 4)
6 >>> quotient, remainder = divmod(*t)
7 >>> quotient, remainder
8 (2, 4)
```

上述代码还展示了解包的另一用途:让函数以一种更加方便用户使用的方式,一次性返回多个值。再举个例子:os.path.split()函数根据传入的文件系统路径构建一个元组 (path, last_part)。

```
1 >>> import os
2 >>> filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
3 >>> filename
4 ('/home/luciano/.ssh', 'id_rsa.pub')
5 >>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
6 >>> filename
7 'id_rsa.pub'
```

在解包时,若仅需使用解包得到的部分项,还可以使用下一节介绍的 * 语法。

2.5.1 用 * 获取余下的项

在定义函数时,可以使用 *args 捕获余下的所有参数,这是 Python 的一个经典特性。Python 3 将这一思想延伸到了并行赋值上。

```
1 >>> a, b, *rest = range(5)
2 >>> a, b, rest
3 (0, 1, [2, 3, 4])
4 >>> a, b, *rest = range(3)
5 >>> a, b, rest
6 (0, 1, [2])
7 >>> a, b, *rest = range(2)
8 >>> a, b, rest
9 (0, 1, [])
```

在并行赋值时,只能为一个变量(可以是任何位置)应用 * 前缀:

```
1 >>> *head, b, c, d = range(5) #末尾3项, 分别赋值给b、c、d, 其他项赋值给列表head
2 >>> head, b, c, d
3 ([0, 1], 2, 3, 4)
4 >>> a, *body, c, d = range(5) #第1项与末尾2项, 分别赋值给a、c、d, 其他项赋值给列表
5 body
6 >>> a, body, c, d
```

```

6 (0, [1, 2], 3, 4)
7 >>> a, b, c, *rest = range(5)    #前3项分别赋值给a、b、c, 其他项赋值给列表rest>>> a,
8 b, c, rest
9 (0, 1, 2, [3, 4])

```

2.5.2 在函数调用和序列字面量中使用*解包

“PEP 448 – Additional Unpacking Generalizations”为可迭代对象解包引入了更灵活的语法, “What’s New In Python 3.5”对此做了很好的概括。

在函数调用中,可以多次使用*,如下所示:

```

1 >>> def fun(a, b, c, d, *rest):
2 ...     return a, b, c, d, rest
3 ...
4 >>> fun(*[1,2], 3, *range(4,7))      #函数调用中, 多次使用*
5 (1, 2, 3, 4, (5, 6))

```

定义 list、tuple 或 set 字面量时,也可以使用*。“What’s New In Python 3.5”给出了一些示例,如下所示:

```

1 >>> *range(4), 4
2 (0, 1, 2, 3, 4)
3 >>> [*range(4), 4]
4 [0, 1, 2, 3, 4]
5 >>> [*range(4), 4, *(5, 6, 7)]
6 {0, 1, 2, 3, 4, 5, 6, 7}

```

“PEP 448”还为**引入了类似的新语法,详见“3.2.2 映射解包<67页>”。

接下来,介绍解包的另一个强大功能——处理嵌套结构。

2.5.3 嵌套解包

当元组中包含了嵌套结构,例如(a, b, (c, d)),如果目标变量的结构与元组中的值结构相匹配,Python会自动执行正确的解包操作。示例 2.8 演示了嵌套解包的具体用法。

</> 示例 2.8: 02-array-seq/metro_lat_lon.py:解包嵌套元组,获取经纬度

```

1 metro_areas = [
2     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ❶
3     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
4     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
5     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
6     ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
7 ]
8
9 def main():
10     print(f'{"":>15} | {"latitude":>9} | {"longitude":>9}')
11     for name, _, _, (lat, lon) in metro_areas: ❷
12         if lon <= 0: ❸

```

```

13     print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
14     if __name__ == '__main__':
15         main()

```

- ① 列表 metro_areas 中的每一项都是一个包含 4 个字段的元组 (tuple), 最后一个字段是由经纬度坐标组成的嵌套子元组;
- ② 将最后一个字段 (经纬度) 赋值给一个嵌套元组, 解包坐标对;
- ③ 根据判断条件 lon<=0, 输出西半球的城市;

示例 2.8 的输出如下所示:

```

1 python src/part01/metro_lat_lon.py
2
3 Mexico City | 19.4333 | -99.1333
4 New York-Newark | 40.8086 | -74.0204
5 São Paulo | -23.5478 | -46.6358

```

通常情况下,解包赋值常用于 tuple 或其他可迭代对象,而很少用于 list。然而,有一种特殊场景可以将 list 作为解包赋值的目标。这个场景就是:当数据库查询只返回 1 条记录(如应用了 limit 1 子句)时,则可以用如下代码解包返回的记录——即将返回的单条记录赋值给目标变量 record。同时,可以确保只返回 1 个结果,因为若 query_returning_single_row() 返回多条记录,此解包语句将报错。

```

1 >>> [record] = query_returning_single_row()

```

如果 record 中只有 1 个字段,还可以像下面这样,将 record 中的单个字段直接赋值给目标变量 field:

```

1 >>> [[field]] = query_returning_single_row_with_single_field()

```

上述 2 种情况,也可以用元组作为赋值目标对象。但 Python 规定单项元组(即只含 1 个元素的元组)必须在项后追加 1 个逗号 (,)。因此,上述单条记录的赋值目标元组需写成 (record,), 第 2 个单字段的赋值目标元组需写成 ((field,),)。如果忘写单项元组后面的逗号,可能会埋下不易察觉的 BUG。

接下来,将介绍更强大的序列解包方式:模式匹配 (Pattern Match)

2.6 序列模式匹配

Python 3.10 最引人注目的新特性是通过 match/case⁶ 语句实现的模式匹配 (Pattern Match)。



在“What’s New In Python 3.10”中的“PEP 634: Structural Pattern Matching”一节,Python 核心开发人员 Carol Willing 对模式匹配 (Pattern Match) 的介绍可谓精彩纷呈,建议您读一下。根据模式 (Pattern) 类型的不同,模式匹配 (Pattern Match) 将出现在本书的多个章节中,如“3.3 用模式匹配处理映射<68页>”、“5.8 模式匹配类实例<156页>”、“18.3 案例分析:lis.py 中的模式匹配<539页>”。

下面是用 match/case 模式匹配 (Pattern Match) 处理序列的第一个示例。假设您在设计一个机器人,

⁶match/case:match/case 语句实现的模式匹配 (Pattern Match) 是在“PEP 634 - Structural Pattern Matching: Specification”中提出来的。

它接受以文字和数值序列形式发送的指令,如“BEEPER 440 3”。经过拆分和解析后,得到消息[‘BEEPER’, 440, 3]。可以用示例 2.9 中的代码来处理此类消息。

</> 示例 2.9: 虚构的 Robot 类中的方法

```

1 def handle_command(self, message):
2     match message: ❶
3         case ['BEEPER', frequency, times]: ❷
4             self.beep(times, frequency)
5         case ['NECK', angle]: ❸
6             self.rotate_neck(angle)
7         case ['LED', ident, intensity]: ❹
8             self.leds[ident].set_brightness(ident, intensity)
9         case ['LED', ident, red, green, blue]: ❺
10            self.leds[ident].set_color(ident, red, green, blue)
11         case _: ❻
12             raise InvalidCommand(message)

```

- ❶ match 关键字后面的表达式(这里为 message)是匹配对象(Subject),即各个 case 子句中的模式(Pattern)尝试匹配的数据;
- ❷ case 关键字后面的是模式(Pattern)。这个模式将匹配任意含有 3 项的序列,且第 1 项必须为字符串‘BEEPER’,第 2 项和第 3 项任意,并依次绑定到变量 frequency 和 times 上。此模式(Pattern)可以匹配“BEEPER 440 3”序列指令,并将 440 赋值到变量 frequency,将 3 赋值到变量 times。
- ❸ 这个模式将匹配任意含有 2 项的序列,且第一项必须为字符串‘NECK’。
- ❹ 这个模式将匹配任意含有 3 项的序列,且第一项必须为字符串‘LED’。
- ❺ 这个模式将匹配任意含有 5 项的序列,且第一项必须为字符串‘LED’。
- ❻ 默认的 case 子句,前面所有模式(Pattern)都不匹配时,执行此子句。_是特殊的变量,稍后讲解。

表面上看,Python 中的 match/case 与 C 语言中的 switch/case 语句很像,但这只是表象。与 switch 相比,match 的一大改进是支持析构(Destructor),这是一种高级解包形式。析构对 Python 语言来说是一个新词汇,但是在支持模式匹配(Pattern Match)的语言(如 scala 和 elixir)文档中经常出现。

示例 2.10 在本书中首次使用析构,用 match/case 重写了示例 2.8 的部分代码,如下所示:

</> 示例 2.10: 02-array-seq/match_lat_lon.py 要求 Python 3.10+

```

1 metro_areas = [
2     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
3     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
4     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
5     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
6     ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
7
8
9 def main():
10    print(f'{"":>15} | {"latitude":>9} | {"longitude":>9}')
11    for record in metro_areas:
12        match record: ❶
13            case [name, _, _, (lat, lon)] if lon <= 0: ❷

```

```

14     print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
15 if __name__ == '__main__':    main()

```

- ① 这个 match 的匹配对象 (Subject) 是 record, 即 metro_areas 中的各个元组;
- ② 这个 case 子句由 2 部分组成: 模式 (Pattern) 和 if 关键字指定的卫 (guard) 语句。模式 (Pattern) 用于匹配某个值或表达式的结构, 卫 (guard) 语句用于进一步对匹配的结果进行条件判断和筛选。只有当卫 (guard) 语句为 True 时, 才会执行与该 case 子句关联的代码块。

通常来说, 需满足如下条件, 匹配对象 (Subject) 才能与序列模式 (Pattern) 相匹配:

- 匹配对象 (Subject) 是一个序列;
- 匹配对象 (Subject) 与模式 (Pattern) 的项数相等;
- 每个对应的项都相匹配, 包括嵌套的项;

例如, [示例 2.10](#) 中的模式 [name, _, _, (lat, lon)] 将匹配一个含有 4 项的序列, 并且序列中最后一项必须是一个含有两项的子序列。

序列模式 (Sequence Pattern) 可以写成元组或列表形式, 或者嵌套元组和嵌套列表形式。元组形式和列表形式没有任何区别, 因为在序列模式 (Sequence Pattern) 中, 方括号 [] 与圆括号 () 的语义是一样的。[示例 2.10](#) 中的模式 (Pattern) 写成了列表形式, 其中嵌套的子序列则写成了元组形式。这样做只是为了避免重复使用方括号 [] 或圆括号 (), 更易于理解与阅读。

序列模式 (Sequence Pattern) 可以匹配 `collections.abc.Sequence` 中的大多数实际子类 (Actual Subclass) 或虚拟子类 (Virtual Subclass) 的实例, 但 `str`、`bytes` 和 `bytearray` 除外。



在 match/case 上下文中, `str`、`bytes` 和 `bytearray` 实例不会被作为序列来处理。match 将这些类型的匹配对象 (Subject) 视为“原子”值, 如整数 987 被视为单个值, 而不是数字序列。若要将这 3 种类型的匹配对象 (Subject) 视为序列, 需要在 match 子句中对其进行类型转换, 例如以下示例中的 `tuple(phone)`。

```

1 match tuple(phone):      # 将整数值转换为数字序列
2     case ['1', *rest]:    # 北美洲与加勒比地区
3     ...
4     case ['2', *rest]:    # 非洲地区
5     ...
6     case ['3' | '4', *rest]: # 欧洲
7     ...

```

Python 标准库中的以下类型与 序列模式 (Sequence Pattern) 相兼容:

```

list  memoryview  array.array
tuple  range      collections.deque

```

与解包不同, 模式 (Pattern) 不会析构 (Destructor) 除序列以外的其他可迭代对象 (如迭代器)。

下划线 (_) 在模式 (Pattern) 中有特殊语义: 匹配相应位置上的任意一项, 但不会绑定匹配项的值。另外, 下划线 (_) 是唯一一个可在模式 (Pattern) 中多次出现的变量。

可使用 `as` 关键字将模式 (Pattern) 中的任一部分绑定到变量上, 如下所示 (将子序列 `(lat, lon)` 绑定到变量 `coord` 中):

```
1 case [ name, _, _, (lat, lon) as coord ]:
```

上述模式 (Pattern) 可以匹配序列 ['Shanghai', 'CN', 24.9, (31.1, 121.3)], 并将经维度子序列 (31.1, 121.3) 绑定到变量 coord 中。设定的变量如下所示：

绑定的变量	序列中的值
name	'Shanghai'
lat	31.1
lon	121.3
coord	(31.1, 121.3)

为序列模式 (Sequence Pattern) 添加类型信息, 可以使模式 (Pattern) 更直观具体。例如, 下面的模式 (Pattern) 与前面的示例匹配相同的嵌套序列结构。但是, 第 1 项必须是 str 实例, 嵌套的子序列中的 2 项必须是 float 实例。

```
1 case [ str(name), _, _, (float(lat), float(lon)) ]:
```



上述模式 (Pattern) 中的表达式 str(name) 与 float(lat) 看起来像函数调用: 前者将 name 转换为 str, 后者将 lat 转换为 float。但实际不是这样, 在模式 (Pattern) 上下文中, 这种语法的作用是在运行时, 检查匹配对象中项的类型。“case [str(name), _, _, (float(lat), float(lon))]” 将匹配一个 4 项序列, 其中第 1 项必须是 str, 第 4 项必须是包含一对 float 值的子序列。而且, 第 1 项中的 str 将被绑定到变量 name 上, 第 4 项子序列中的 float 值将分别绑定到变量 lat 和 lon 上。因此, 尽管 str(name) 使用了构造函数的语法, 但是在模式 (Pattern) 上下文中, 语义是完全不同的。“5.8 模式匹配类实例<156页>” 将会介绍如何在模式 (Pattern) 中使用任意的类。

此外, 若想要匹配任何以字符串开头, 并以嵌套 2 个浮点数的子序列结尾的序列 (如 ['Shanghai', 'CN', 24.9, (31.1, 121.3)]) 则可以使用如下模式 (Pattern) :

```
1 case [ str(name), *_, (float(lat), float(lon)) ]:
```

上述模式 (Pattern) 中的 *_ 将匹配序列中任意数量的项, 而且不绑定变量。若将 *_ 换成 *extra, 匹配到的若干项将以列表形式绑定到变量 extra 中。

以 if 开头的卫 (guard) 语句是可选的, 仅当模式 (Pattern) 匹配成功时才会运行该卫语句。卫 (guard) 语句可以像示例 2.10 那样, 引用模式 (Pattern) 中绑定的变量, 如下所示。

```
1 match record:
2     case [name, _, _, (lat, lon)] if lon <= 0: # if 条件中引用了模式中绑定的变
3         量 lon
4             print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

只有 case 子句中的模式 (Pattern) 匹配成功, 且 if 中的卫 (guard) 语句为 True 时, 才会执行与该 case 子句关联的 print 代码块。



使用模式 (Pattern) 进行析构 (Destructor) 具有很强的表现力, 即使只有 1 个 case 子句, match 语句也可以让代码变得更简单。Guido van Rossum 收集了一些使用 match/case 的示例, 其中包括一个标题为 “A very deep iterable and type match with extraction” 的示例。

```

1  def flatten(self) -> Rhs:  # 原始代码
2      # If it's a single parenthesized group, flatten it.
3      rhs = self.rhs
4      if (
5          not self.is_loop()
6          and len(rhs.alts) == 1
7          and len(rhs.alts[0].items) == 1
8          and isinstance(rhs.alts[0].items[0].item, Group)
9      ):
10         rhs = rhs.alts[0].items[0].item.rhs
11     return rhs
12
13 def flatten(self) -> Rhs:  # match/case 改写后的代码
14     # If it's a single parenthesized group, flatten it.
15     rhs = self.rhs
16     if not self.is_loop():
17         match rhs.alts:
18             case [Alt(items=[NamedItem(item=Group(rhs=r))])]:
19                 rhs = r
20     return rhs

```

“示例 2.10<34页>” 并不是 “示例 2.8<32页>” 的改进版本, 只是换了一种实现方式。下面的示例将说明如何用 match/case 模式匹配写出清晰、简洁、高效的 Python 代码。

2.6.1 用模式匹配序列实现一个解释器

斯坦福大学的 Peter Norvig 用 132 行优雅且易读的 Python 代码编写了 lis.py: 一个用于解析 Lisp 编程语言 Scheme 方言子集的解释器。我将 lis.py 源码更新到了 Python 3.10, 以展示模式匹配 (Pattern Match) 的用法。这一节将 Norvig 用 if/elif 和解包实现的代码与我用 match/case 重写的代码, 进行一个比较。

lis.py 中的 2 个主要函数是 parse 和 evaluate⁷。parse 函数接受的参数为包含在圆括号 () 内的 Scheme 风格表达式, 返回值是 Python 列表。两个示例如下:

```

1  >>> from lis import *
2  >>> parse('(gcd 18 45)')      # 示例1
3  ['gcd', 18, 45]
4  >>> parse('''
5  ... (define double
6  ...     (lambda (n)
7  ...         (* n 2)))
8  ... ''')

```

⁷在 Norvig 的 lis.py 源码中, 后一个函数名为 eval。为了避免与 Python 内置函数 eval 混淆, 我将 eval 函数名换成了 evaluate。

```
9 ['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

evaluate 函数的作用就是执行通过参数传入的这种列表。在上述的示例 1 中, evaluate 函数将调用 gcd 函数, 计算传入的参数 18 与 45 的最大公约数(即 9);示例 2 则定义一个名为 double 的函数, 函数参数为 n, 函数主体为表达式 (* n 2)。在 Scheme 方言中, 调用函数返回的结果是函数主体中最后一个表达式的值。

本节关注的重点是析构序列, 而对求值操作不做深入探讨。lis.py 的进一步说明见“[18.3 案例分析: lis.py 中的模式匹配<539页>](#)”。

示例 2.11 是 Norvig 实现的 evaluate 函数(有所节略, 只给出了序列模式部分):

</> 示例 2.11: lispy/py3.9/lis.py: 不使用 match/case 匹配模式

```
1 def evaluate(exp: Expression, env: Environment) -> Any:
2     "根据所处的环境, 求解表达式"
3     if isinstance(exp, Symbol): # 变量引用
4         return env[exp]
5     # ... 省略几行
6     elif exp[0] == 'quote':      # (quote exp)
7         (_, x) = exp
8         return x
9     elif exp[0] == 'if':         # (if test consequent)
10        (_, test, consequence, alternative) = exp
11        if evaluate(test, env):
12            return evaluate(consequence, env)
13        else:
14            return evaluate(alternative, env)
15    elif exp[0] == 'lambda':    # (lambda (parameters) body...)
16        (_, parms, *body) = exp
17        return Procedure(parms, body, env)
18    elif exp[0] == 'define':
19        (_, name, value_exp) = exp
20        env[name] = evaluate(value_exp, env)
21    # ... 省略余下的代码
```

注意, 每一个 elif 子句会检查列表 exp 的第 1 项, 然后解包列表并忽略列表中的第 1 项。从代码中可以看到, Norvig 使用了大量的解包。可惜, 他的代码是用 Python 2 写的, 那时还没有 match/case 模式匹配。

使用 Python 3.10 及以上版本中的 match/case 重构后的 evaluate 函数, 如示例 2.12 所示。

</> 示例 2.12: lispy/py3.10/lis.py: 使用 match/case 匹配模式(要求 Python 3.10 及以上)

```
1 def evaluate(exp: Expression, env: Environment) -> Any:
2     "根据所处的环境, 求解表达式"
3     match exp:
4         # ... 省略几行
5         case ['quote', x]: ❶
6             return x
7         case ['if', test, consequence, alternative]: ❷
8             if evaluate(test, env):
9                 return evaluate(consequence, env)
10            else:
```

```

11     return evaluate(alternative, env)
12 case ['lambda', [*parms], *body] if body: ③
13     return Procedure(parms, body, env)
14 case ['define', Symbol() as name, value_exp]: ④
15     env[name] = evaluate(value_exp, env)
16     # ... 又省略几行
17 case _:
18     raise SyntaxError(lispstr(exp))

```

- ① 模式 ['quote', x] 将匹配以 'quote' 开头的 2 项序列；
- ② 模式 ['if', test, consequence, alternative] 将匹配以 'if' 开头的 4 项序列；
- ③ 模式 ['lambda', [*parms], *body] 将匹配以 'lambda' 开头的 3 项或更多项的序列；**卫 (guard)** 语句 if body，确保 body 不为空；
- ④ 模式 ['define', Symbol() as name, value_exp] 将匹配以 'define' 开头，且后边跟一个 Symbol 实例的 3 项序列；
- ⑤ _ 提供一个兜底的**模式 (Pattern)**，用于匹配前面无法匹配的所有序列。在此示例中，若 exp 未匹配前面任何模式，则将匹配此处的兜底模式，并抛出 SyntaxError 异常。

若 match/case 中不指定兜底模式，那么当**匹配对象 (Subject)** 未匹配任何**模式 (Pattern)** 时，match 语句将悄无声息地什么都不做（有问题也不会提醒）。

为了让 lis.py 代码更易于理解，Norvig 故意没写错误检查。而使用 match/case 模式匹配，就可以添加很多错误检查，同时也不影响代码的可读性。例如，原版（[示例 2.11](#)）代码中，若要确保 'define' 模式中的 name 必须是 Symbol 实例，则需要添加一个 if 代码块、一个 isinstance 函数调用，以及一些其他代码。这样看来，显然[示例 2.12](#)比[示例 2.11](#)更简短、更安全。

1. 备选的 lambda 表达式模式

在 Scheme 方言中，lambda 表达式的语法如下所示。语法中的后缀 ... 是一种约定，表示元素可以出现 0 次或多次。

```

1 ( lambda (params...) body1 body2...)
2

```

一个简单的 'lambda' 模式可能写成：

```

1 case ['lambda', parms, *body] if body:
2

```

然后，上述这个模式可以在 parms 位置上匹配任何值，包括下面这种无效⁸的匹配对象 (Subject) 中的第 1 个 'x'。

```

1 ['lambda', 'x', ['*', 'x', 2]]
2

```

在[示例 2.12](#)中，为了确保 'lambda' 模式更安全，我使用了嵌套**序列模式 (Sequence Pattern)**（如下所示）。

⁸无效的 Scheme 方言匹配对象：在 Scheme 方言中，lambda 关键字后面紧跟的嵌套列表是函数的形参名称，即使只有 1 个元素，也要写成列表形式。若函数不接受参数，则要写成空列表，就像 Python 中的 random.random() 函数一样。

```

1   case ['lambda', [*parms], *body] if body:
2       return Procedure(parms, body, env)
3

```

在序列模式 (Sequence Pattern) 中, 每个序列只能出现一次 *。在上述嵌套序列中, 包含了 2 个序列: 外层一个、内层一个。在 parms 两侧加上 [*] 之后 (即 [*parms]), 使模式看起来更符合 Scheme 方言 语法, 而且额外还提供了结构检查。

2. 定义函数的快捷语法

Scheme 方言中还有另一种 define 语法, 可以在不使用嵌套 lambda 的情况下, 创建具名 (named) 函数, 语法如下所示:

```

1 (define (name parm...) body1 body2...)
2

```

define 关键字后面紧跟一个列表, 其中 name 是函数名称, parm... 是 0 个或多个参数名称。列表之后是函数主体, 主体包含 1 个或多个表达式。

在 match/case 语句中添加中添加如下两行, 以匹配这种语法。

```

1 case ['define', [Symbol() as name, *parms], *body] if body:
2     env[name] = Procedure(parms, body, env)
3

```

我将上述 case 子句放在示例 2.12 中 “case ['define', Symbol() as name, value_exp]:” 模式的后面。在此示例中, define 模式的放置顺序其实无关紧要, 因为匹配对象不会同时满足多个 define 模式: “case ['define', Symbol() as name, value_exp]:” 子句, 第 2 个元素必须是 Symbol 实例; 而 “case ['define', [Symbol() as name, *parms], *body] if body:” 子句, 第 2 个元素必须是一个以 Symbol 实例开头的序列。

可以想象一下, 若按 “示例 2.11<38页>” 那样不使用 match/case 模式匹配, 为了支持新 define 语法, 需要增加多少工作量。match/case 语句做的事情要比 C 语言中 switch/case 语句多出很多。

模式匹配 (Pattern Match) 是一种声明式编程风格, 即描述你想匹配什么, 而不是如何匹配。这样写出的代码结构与数据结构是一致的, 如表 2.2 所示。

表 2.2: 一些 Scheme 语法形式和处理语法的 case 序列模式

Scheme 语法形式	case 序列模式
(quote exp)	['quote', exp]
(if test conseq alt)	['if', test, conseq, alt]
(lambda (parms...) body1 body2...)	['lambda', [*parms], *body] if body
(define name exp)	['define', Symbol() as name, exp]

接下页

续表 2.2

Scheme 语法形式	case 序列模式
(define (name parms...) body1 body2...)	['define', [Symbol() as name, *parms], *body] if body

相信通过本节用 match/case 模式匹配对 Norvig 编写的 evaluate 函数进行重构之后, 您能发现使用 match/case 写出的代码更具可读性, 也更安全。



“18.3 案例分析: lis.py 中的模式匹配<539页>”还会进一步分析 lis.py, 届时将全面研究 evaluate 函数中 match/case 语句。若想深入了解 lis.py, 请阅读 Norvig 写的文章“(How to Write a (Lisp) Interpreter (in Python))”

对解包、析构 (Destructor) 和模式匹配 (Pattern Match) 的体验之旅到此结束。其他模式 (Pattern) 类型将在后续章节进行讲解。

每个 Python 程序员都知道, 序列可以通过 `s[a:b]` 语法进行切片。“2.7 切片”将会探索一些鲜为人知的切片功能。

2.7 切片

在 Python 中, 列表、元组、字符串等所有序列类型都支持切片 (slice) 操作。切片操作的强大, 要超出许多人的预期。

本章节将讨论切片 (slice) 操作的高级用法。本部分关注的是现成可用的类, 而在用户定义的类中实现切片的方法, 将在“十二 序列的特殊方法<321页>”中进行介绍, 自定义类则延后至“三 延伸阅读<293页>”。

2.7.1 为何切片与区间都排除最后一项

切片与区间 (ranges) 都排除序列的最后一项是一种 [Pythonic](#)⁹ 约定, 这与 Python、C 和很多其他语言中从 0 开始的索引相匹配。排除最后一项还可以带来如下好处:

- 在仅指定停止位置时, 容易判断切片或区间的长度。例如, `range(3)` 和 `my_list[:3]` 都只产生包含 3 项的序列。
- 同时指定起始和停止位置时, 容易计算切片或区间的长度 (做减法即可 `stop-start`)。
- 方便在索引 `x` 处, 将一个序列拆分成 2 部分而不产生重叠。直接使用 `my_list[:x]` 和 `my_list[x:]` 即可。示例如下:

```

1  >>> l = [10, 20, 30, 40, 50, 60]
2  >>> l[:2]          # 在索引位 2 处拆分
3  [10, 20]
4  >>> l[2:]
5  [30, 40, 50, 60]
6  >>> l[:3]          # 在索引位 3 处拆分
7  [10, 20, 30]
8  >>> l[3:]

```

⁹Pythonic: 可理解为具有 Python 语言风格的代码, 即以 Python 方式写出的简洁、优美的代码。

```
9 [40, 50, 60]
```

荷兰计算机科学家 Edsger W. Dijkstra 写过一篇文章,对这一风格的解释应该是最好的(见“[2.12 延伸阅读<60页>](#)”)。

接下来,深入了解 Python 是如何解析切片表示法的。

2.7.2 切片对象

众所周知,在切片时可以用 `[a:b:c]` 语法指定步长 `c`,使切片操作跳过部分项 (item)。若将步长 `c` 指定为负数,则会反向返回序列中的项 (item)。示例如下:

```
1 >>> s = 'bicycle'
2 >>> s[::3]
3 'bye'
4 >>> s[::-1]
5 'elcycib'
6 >>> s[::-2]
7 'eccb'
```

“[Python 数据模型](#)”中也有这样的例子(“[??<??页>](#)”),当时我们用 `deck[12::13]` 从一副没洗过的纸牌中抽取所有 A。

```
1 >>> deck[12::13]      # 只抽取4张A
2 [Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
3  Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

`start:stop:step` 表示法只在 `[]`(表示索引或下标运算符)内部有效,得到的结果为一个切片对象,即 `slice(start:stop:step)`。“[12.5.1 切片原理<326页>](#)”中,你会发现:为了求解表达式 `seq[start:stop:step]`,Python 会调用 `seq.__getitem__(slice(start:stop:step))`。即使不想自己实现序列类型,了解切片对象也有一定的好处。如,当代码中包含大量切片操作时,可以通过为切片命名,从而提高代码的可读性(如示例 2.13 所示)。

</> **示例 2.13:** 从纯文本形式的发票中,提取商品信息

```
1 invoice = """
2 ... 0.....6.....40.....52..55.....
3 ... 1909 Pimoroni PiBrella           $17.50 3 $52.50
4 ... 1489 6mm Tactile Switch x20      $4.95  2 $9.90
5 ... 1510 Panavise Jr. - PV-201       $28.00 1 $28.00
6 ... 1601 PiTFT Mini Kit 320x240      $34.95 1 $34.95
7 ...
8 >>> SKU = slice(0, 6)
9 >>> DESCRIPTION = slice(6, 40)
10 >>> UNIT_PRICE = slice(40, 52)
11 >>> QUANTITY = slice(52, 55)
12 >>> ITEM_TOTAL = slice(55, None)
13 >>> line_items = invoice.split('\n')[2:]
14 >>> for item in line_items:
```

```

15 ...     print(item[UNIT_PRICE], item[DESCRIPTION])
16 ...$17.50      Pimoroni PiBrella
17 $4.95       6mm Tactile Switch x20
18 $28.00      Panavise Jr. - PV-201
19 $34.95      PiTFT Mini Kit 320x240

```

“12.5 Vector 类第 2 版：可切片的序列<325页>”在介绍自定义容器类型时，还会再次讨论切片对象。从用户角度出发，切片还包括 2 个额外的功能：多维切片和省略号（...）表示法。

2.7.3 多维切片与省略号

[] 运算符还可接受以逗号分隔的多个索引或切片。负责处理 [] 运算符的特殊方法 `__getitem__` 与 `__setitem__` 将从 `a[i, j]` 接收的索引当作元组。即为了求解 `a[i, j]`，Python 会在幕后调用 `a.__getitem__((i, j))`。

例如，在 NumPy 包中，`numpy.ndarray` 表示的二维数组可以用 `a[i, j]` 语法获取数组中的元素，还可以用表达式 `a[m:n, k:l]` 获取二维切片，详见“2.10.2 memoryview”中的“示例 2.22<55页>”。

除了 `memoryview` 之外，Python 内置的序列类型都是一维的，因此只支持单个索引或切片，不支持索引或切片元组（如 `a[i, j]` 或 `a[m:n, k:l]`）。

Python 将省略号（...）¹⁰识别为一个标记。它是 `Ellipsis` 对象的别名，是 `ellipsis` 类的唯一实例。因此可将其作为参数传递给函数，也可以作为切片规范的一部分，如 `f(a, ..., z)` 或 `a[i: ...]`。Numpy 在处理多维数组切片时，将 ... 视为一种快捷语法。例如，若 `x` 为四维数组，则 `x[i, ...]` 是 `x[i, :, :, :]` 的快捷语法。更多相关信息，请参阅 Numpy 官网 `NumPy quickstart`。

在写作本书时，我还未发现 Python 标准库采用 `Ellipsis` 或多维索引和多维切片。若您发现了，请告诉我。此种语法的存在，是为了给用户定义的类型和扩展（如 Numpy）提供支持。

切片除了可以从序列中提取信息（构建新序列）之外，还可以就地更改可变序列（即不构建新序列）。

2.7.4 为切片赋值

将切片表示法用在赋值语句的左侧，或者作为 `del` 语句的目标，可以就地移植、切除或以其他方式更改可变序列。示例如下：

```

1  >>> l = list(range(10))
2  >>> l
3  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4  >>> l[2:5] = [20, 30]          ❶
5  >>> l
6  [0, 1, 20, 30, 5, 6, 7, 8, 9]
7  >>> del l[5:7]               ❷
8  >>> l
9  [0, 1, 20, 30, 5, 8, 9]
10 >>> l[3::2] = [11, 22]        ❸
11 >>> l
12 [0, 1, 20, 11, 5, 22, 9]
13 >>> l[2:5] = 100             ❹

```

¹⁰省略号（...）：这里的省略号是指 3 个英文句点（...），而不是标点符号 …（Unicode U+2026）。

```

14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>TypeError: can only assign an iterable
16 >>> l[2:5] = [100]          ❸
17 >>> l
18 [0, 1, 100, 22, 9]

```

- ❶ 将序列中索引位 2~5 (排除最后 1 项) 的元素, 替换为 “20,30”。即将元素 “2,3,4” 替换为 “20,30”。
- ❷ 删除序列中索引位 5~7 (排除最后 1 项) 的元素, 即删除元素 “6,7”。
- ❸ 将序列中从索引位 3 开始, 步长为 2 的元素替换为 “11,22”。即将 30 与 8 分别替换为 11 与 22。
- ❹ 若赋值目标是一个切片, 则赋值语句右侧必须是一个可迭代对象, 即使只有 1 项。
- ❺ 若赋值目标是一个切片, 则赋值语句右侧必须是一个可迭代对象, 即使只有 1 项。

许多 Python 教程都会介绍如何用运算符 + 和 * 执行序列的拼接操作, 但是这两个运算符还有一些细节需要特别注意 (详见 “[2.8 使用 + 和 * 处理序列](#)”)。

2.8 使用 + 和 * 处理序列

通常, 用 + 运算符拼接的 2 个序列必须是同类型序列, 而且都不会被更改, 拼接后将得到一个同类型的新序列。若想多次拼接同一个序列, 可以用运算符 * 将序列乘以一个整数。同样, 结果也是一个新创建的序列。

```

1 >>> l = [1, 2, 3]
2 >>> k = [10, 20, 30]
3 >>> l + k
4 [1, 2, 3, 10, 20, 30]
5 >>> l * 5
6 [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
7 >>> 5 * 'abcd'
8 'abcdabcdabcdabcdabcd'

```

运算符 + 与 * 始终创建一个新的对象, 绝不更改操作数。



注意, $a * n$ 这种表达式, 若序列 a 中包含可变项, 则结果可能会令人出乎意料。例如, 用 $my_list = [[]] * 3$ 初始化一个嵌套列表, 得到的结果是一个列表没错。但是, 嵌套的 3 个引用将指向同一个列表对象, 这可能与您的预期相悖。

“[2.8.1 构建嵌套列表](#)” 将说明使用 * 初始化嵌套列表时的陷阱。

2.8.1 构建嵌套列表

有时候, 需要用一定数量的子列表来初始化一个列表。例如, 将一些学生分配到团队列表中, 或表示游戏棋盘上的方格。解决这些问题的最佳方式是使用 [列表推导式 \(Listcomp\)](#) (如 “[示例 2.14<44页>](#)” 所示)。

</> [示例 2.14: 列表推导式:一个列表中嵌套 3 个长度为 3 的子列表](#)

```

1 >>> board = [['_'] * 3 for i in range(3)] ❶
2 >>> board

```

```

3  [['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
4  >>> board[1][2] = 'X' ❷
5  >>> board
6  [['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]

```

- ❶ 用列表推导式创建一个列表, 内部嵌套 3 个包含 3 项的子列表。
 ❷ 将索引位 [1][2] (即第 2 行第 3 列) 的项, 修改为 X。

示例 2.15 中的做法看似省事, 但却隐藏了一个陷阱。

</> 示例 2.15: 错误做法: 在一个列表中 3 次引用同一个列表对象

```

1  >>> weird_board = [['_'] * 3] * 3 ❶
2  >>> weird_board
3  [['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
4  >>> weird_board[1][2] = '0' ❷
5  >>> weird_board
6  [['_', '_', '0'], ['_', '_', '0'], ['_', '_', '0']]

```

- ❶ 外部列表中的 3 个引用都指向了同一个列表对象。
 ❷ 预期将索引位 [1][2] (即第 2 行第 3 列) 的项修改为 '0'。但实际上所有行的第 3 列都被更改为了 '0'。

示例 2.15 的问题在于, 其本质上与以下代码的行为类似:

```

1  row = ['_'] * 3
2  weird_board = []
3  for i in range(3):
4      weird_board.append(row) # 同一个 row, 向 weird_board 追加了 3 次。

```

此外, 示例 2.14 中❶处的列表推导式等价于以下代码:

```

1  >>> board = []
2  >>> for i in range(3):
3      ...     row = ['_'] * 3 ❶
4      ...     board.append(row)
5
6  >>> board
7  [['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
8  >>> board[1][2] = 'X' ❷
9  >>> board
10 [['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]

```

- ❶ 每次迭代构建一个新 row, 追加到 board 中。
 ❷ 将索引位 [1][2] (即第 2 行第 3 列) 的项, 修改为 X。



若暂时不理解本节所讲的内容也没关系。关于引用与可变对象的机制和陷阱, 详见“[六 对象引用、可变性与垃圾回收<163页>](#)”。

当前,讨论的是如何使用普通运算符`+`和`*`处理序列,除此之外还有增量赋值运算符`+=`与`*=`。根据目标序列是否可变,增量赋值运算符`+=`与`*=`可产生不同的效果(详见“[2.8.2 用增量赋值运算符处理序列](#)”)

2.8.2 用增量赋值运算符处理序列

根据第一个操作数的不同,增强赋值运算符`+=`和`*=`的表现也截然不同。为了简化讨论,本节重点关注增量加法运算符`(+=)`。但是,相关概念同样适用于`*=`和其他增量赋值运算符。

促使`+=`起作用的特殊方法是`__iadd__`(就地相加)。但是,若未实现`__iadd__`,Python会转而调用`__add__`。以表达式“`a += b`”为例:

- 若可变序列(如`list`、`bytearray`、`array.array`)`a`实现了特殊方法`__iadd__`,那么Python将调用`a.__iadd__`方法,就地更改序列`a`(行为类似于`a.extend(b)`);
- 若可变序列`a`未实现特殊方法`__iadd__`,那么表达式“`a += b`”等同于“`a = a + b`”,即先求解表达式“`a + b`”得到新对象,再将新对象绑定到`a`上;
- 对于不可变序列(如`tuple`)`a`,无论`a`是否实现了特殊方法`__iadd__`,都无法就地更改序列`a`。表达式“`a += b`”等同于“`a = a + b`”,即先求解表达式“`a + b`”得到新对象,再将新对象绑定到`a`上;

也就是说,变量`a`绑定的对象身份是否发生变化,取决于可变序列`a`有没有实现特殊方法`__iadd__`。通常,对于可变序列,建议实现`__iadd__`特殊方法,使增量赋值运算符可以就地更改可变序列,以节省内存。但是,对于不可变序列,显然无法就地更改。

上述内容同样也适用于通过特殊方法`__imul__`实现的`*=`运算符。关于特殊方法`__iadd__`与`__imul__`的讨论,详见“[十六 运算符重载](#)”。下面用`*=`运算符分别处理一个可变序列和一个不可变序列,观察序列`id`是否发生变化?

```

1  >>> l = [1, 2, 3]
2  >>> id(l)
3  2133297795008      ❶
4  >>> l *= 2
5  >>> l
6  [1, 2, 3, 1, 2, 3]
7  >>> id(l)
8  2133297795008      ❷
9  >>> t = (1, 2, 3)
10 >>> id(t)
11 2133297807232      ❸
12 >>> t *= 2
13 >>> id(t)
14 2133297282208      ❹

```

❶ 列表`l`最初的`id`。

❷ 为列表`l`应用增量运算符`*=`之后,列表`l`的`id`未发生变化,还是同一个列表对象,只是增加了几项。

❸ 元组`t`最初的`id`。

❹ 为元组`t`应用增量运算符`*=`之后,元组`t`的`id`发生改变,创建了新的元组。

重复拼接不可变序列（如 tuple）效率低下¹¹，因为 Python 解释器必须赋值整个目标序列，创建新序列并包含要拼接的项，而不是简单地在原序列上就地追加新项。

本节所讲的内容是 `+=` 运算符的常规用法。“2.8.3 一个 `+=` 运算符赋值谜题”将展示 `+=` 运算符的一个极端情况，以元组（tuple）为例说明“不可变”的真正含义。

2.8.3 一个 `+=` 运算符赋值谜题

</> 示例 2.16：一个增量赋值的谜题

```
1 >>> t = (1, 2, [30, 40])
2 >>> t[2] += [50, 50]
3 >>> t
```

阅读示例 2.16 中的代码，从以下选项中选出正确答案：

- A、`t` 的值变为 `(1, 2, [30, 40, 50, 60])`；
- B、触发“`TypeError: 'tuple' object does not support item assignment`”异常；
- C、A 和 B 都不对；
- D、A 和 B 都对；

看到选项时，您可能很确定答案是 B。但是，正确答案却是 D，即“A 和 B 都对”。在 Python3.11.4 的控制台中，上述表达式的输出如示例 2.17 所示¹²。

</> 示例 2.17：增量赋值谜题的执行结果

```
1 >>> t = (1, 2, [30, 40])
2 >>> t[2] += [50, 50]
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 TypeError: 'tuple' object does not support item assignment
6 >>> t
7 (1, 2, [30, 40, 50, 50])
```

<https://pythontutor.com> 是一个优秀的在线工具，可以用图解的方式描述 Python 的运行机制。图 2.5 包含了 2 幅截图，分别展示了示例 2.17 中元组 `t` 的初始和最终状态。

看一下 Python 为表达式 `s[a] += b` 生成的字节码（见示例 2.18），您一定会恍然大悟。

</> 示例 2.18：增量赋值谜题的执行结果

```
1 >>> import dis
2 >>> dis.dis('s[a] += b')
3 1           0 LOAD_NAME           0 (s)
4           3 LOAD_NAME           1 (a)
5           6 DUP_TOP_TWO
```

¹¹字符串是个例外。因为现实中经常用增量赋值运算符拼接字符串，因此 CPython 对字符串拼接做了优化。内存为 `str` 实例分配空间时，会额外预留空间。因此，字符串拼接时，无需每次都复制整个字符串。

¹²某些读者指出，本例中可用“`t[2].extend([50,60])`”替代“`t[2] += [50, 50]`”，就不会报错了。但是，此段代码的本意是为了展示 `+=` 运算符的这种迷惑行为。

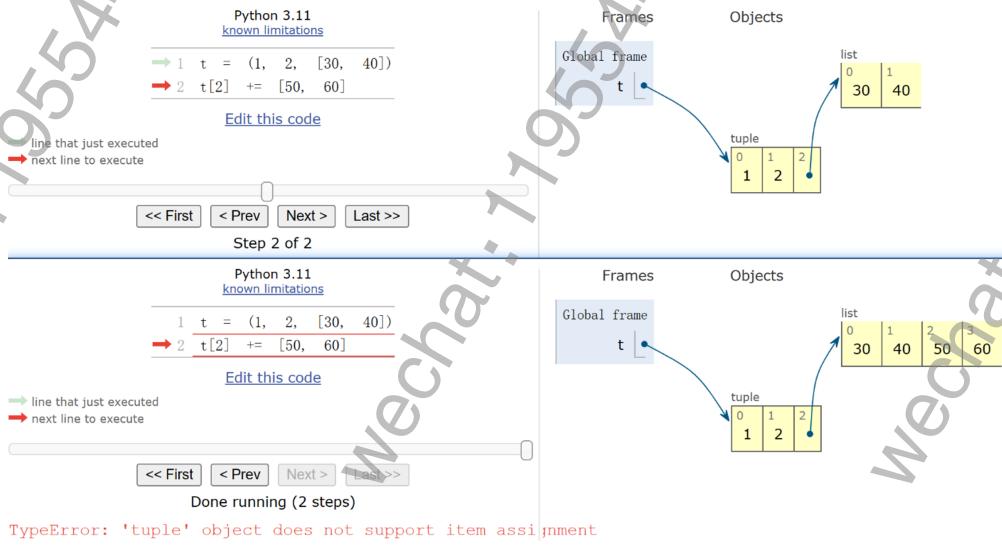
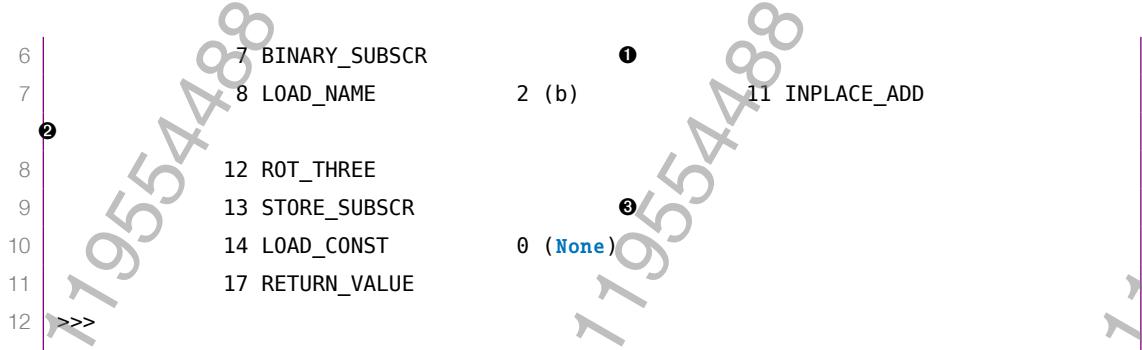


图 2.5: 元组增量赋值谜题中,元组 t 的初始和最终状态。



- ① 将 $s[a]$ 的值放在栈顶 (TOS);
- ② 执行 $TOS += b$ 。若 TOS 引用一个可变对象 (在示例 2.17 中是一个列表), 则操作成功;
- ③ 赋值 $s[a] = TOS$ 。若 s 是不可变对象 (在示例 2.17 中是 t 元组), 则操作失败;

此种情况相当极端,在我使用 Python 的 20 年间,从未见过有人受到这个奇怪行为的影响。

我从这个谜题中得到了 3 个启示:

- 不要在元组中存放可变的项 (如 list);
- 增量赋值不是原子操作。如示例 2.17 中,部分操作 (这里为赋值) 执行完毕后,又抛出了异常;
- 检查 Python 字节码并不难,从中可以看出 Python 在背后都做了什么;

在揭露使用 + 和 * 运算符拼接序列的微妙细节之后,介绍一下序列的另一个基本操作:排序。

2.9 list.sort 与内置函数 sorted

`list.sort()` 方法可以无需创建列表的副本,而就地排序列表。该方法返回值为 `None`,是 Python 的一个约定¹³。例如,`random.shuffle(s)` 函数,将就地混排可变序列 s ,并返回 `None`。

¹³就地更改对象的函数 (或方法),应该返回 `None`,以提示用户该方法 (或函数) 就地更改了对象,而未创建新的对象。



返回 `None` 表示就地更改的约定存在一个缺点, 即这种函数(或方法)无法级联调用。相反, 返回新对象的函数(或方法)(例如 `str` 的所有方法)可以以流式接口(fluent interface)的方式进行级联调用, 如 `str.strip().lower()`。

与 `list.sort()` 相反, 内置函数 `sorted(it)` 接受任何可迭代对象作为参数, 包括不可变序列和生成器(见“[十七 迭代器、生成器和经典协程<479页>](#)”)。无论传入什么类型的可迭代对象, `sorted(it)` 函数始终返回排序后, 所创建的新列表。

`list.sort()` 与 `sorted(it)` 均接受两个可选的关键字参数:

- `reverse`: 值为 `True` 时, 以降序排序。默认为 `False`。
- `key`: 一个只接受一个参数的函数名。Python 会将该参数指定的函数应用到每一项上, 将函数执行结果作为排序依据。例如, 排序字符串时, 指定 `key=str.lower` 会将所有项都用 `lower` 函数转为小写后, 再排序。而 `key=len`, 则按序列中每个项的字符长度来排序。默认为恒等函数(dentity function), 即比较项的本身。



`min()` 和 `max()` 等内置函数, 以及标准库中的其他函数(如 `itertools.groupby()`、`heapq.nlargest()`)也有可选的关键字参数 `key`。

下面举例说明如何使用 `list.sort()`、`sorted(it)` 函数以及关键字参数。这些示例也表明了 Python 的排序算法是稳定的(即能够保留比较时相等的两项的相对顺序)。

```
1  >>> fruits = ['grape', 'raspberry', 'apple', 'banana']
2  >>> sorted(fruits)
3  ['apple', 'banana', 'grape', 'raspberry'] ❶
4  >>> fruits
5  ['grape', 'raspberry', 'apple', 'banana'] ❷
6  >>> sorted(fruits, reverse=True)
7  ['raspberry', 'grape', 'banana', 'apple'] ❸
8  >>> sorted(fruits, key=len)
9  ['grape', 'apple', 'banana', 'raspberry'] ❹
10 >>> sorted(fruits, key=len, reverse=True)
11 ['raspberry', 'banana', 'grape', 'apple'] ❺
12 >>> fruits
13 ['grape', 'raspberry', 'apple', 'banana'] ❻
14 >>> fruits.sort()
15 >>> fruits
16 ['apple', 'banana', 'grape', 'raspberry'] ❻
```

- ❶ 得到一个按字母表顺序升序排序的新字符串列表;
- ❷ 查看原来的列表, 可知 `sorted` 函数并未修改原来的列表;
- ❸ 得到一个按字母表顺序降序排序的新字符串列表;
- ❹ 得到一个按项的字符串长度增序排序的新字符串列表。排序算法很稳定, 尽管 `grape` 与 `apple` 长度都是 5, 但仍按原列表中的顺序排列;

- ⑤ 同④,但是按字符串长度降序排序。因为排序算法稳定,所以 grape 仍按原始列表中的顺序排在 apple 之前;
- ⑥ 到目前为止,原始列表 fruits 的顺序始终未改变;
- ⑦ fruits.sort() 函数将列表 fruits 就地排序,返回 None;
- ⑧ 原始列表 fruits 的顺序发生了改变;



默认情况下,Python 按字符代码的字典顺序排序字符串。这意味着,ASCII 大写字母会排在小写字母之前。而非 ASCII 字符可能会以不合理的方式排序。“[4.8 Unicode 文本排序<119页>](#)”将讲解如何以人类预期的方式正确排序文本。

一旦序列被排序,就可以非常高效地对其进行搜索。Python 标准库中的[bisect 模块](#)已提供了即时可用的二进制搜索算法。其中的[bisect.insort](#) 函数还可以确保排序后的序列始终保持有序。您可以在<https://www.fluentpython.com/extra/ordered-sequences-with-bisect/>“Managing Ordered Sequences with Bisect”中找到关于[bisect 模块](#)的图解介绍。

本章目前所介绍的内容大都适用于一般意义上的序列,不仅限于列表和元组。Python 程序员有时会过度使用 list 类型,因为它太方便了。但是,在处理大型数字列表时,更推荐使用数组。本章余下内容将介绍列表与元组的替代方案——数组、memoryview、Numpy、双端队列和其他队列。

2.10 当列表不适用时

虽然 list 类型简单灵活,但是针对具体的需求,或许会有更好的选择。例如,在处理上百万个浮点值时,使用数组可以节省大量内存。另外,若经常需要在列表两端增加和删除项时,使用[deque](#) (双端队列)¹⁴更合适。



若在代码中需要经常检查容器中是否存在某一项(例如 `item in my_collection`),则 `my_collection` 应考虑使用 set 类型,尤其是项数较多的时候。Python 对 set 类型的成员检查做了优化,速度更快。set 也是可迭代对象,但不是序列,因为 set 中的项是无序的(详见“[三 字典与集合<65页>](#)”)。

2.10.1 数组

若一个序列中只包含数值,那么使用 `array.array` 数组会更高效。`array.array` 数组支持所有可变序列的操作(包括`.pop`、`.insert` 和 `.extend`)。此外,还包含快速加载项的方法(如`.frombytes`)和快速保存项的方法(如`.tofile`)。

如图 2.6 所示,一个由 float 值构成的数组并不保存完整的 float 实例,而只保存代表其机器值的压缩字节——类似于 C 语言中的 double 数组。创建 array 数组对象时,需要提供一个类型码(`typecode`),即用于确定底层 C 使用什么类型存储数组中的项。例如, `b` 是 C 语言中的 signed char 类型码,即取值范围 $-128 \sim 127$ 之间的整数。如果用 `array('b')` 创建数组,那么此数组中的每个项都用 1

`array('d', [9.46, 2.08, 4.29])`

...	9.46	2.08	4.29
-----	------	------	------

图 2.6: float 值构成的数组

¹⁴`collections.deque`是一种更高效的 FIFO(先进先出)数据结构。

个字节存储，并且均被解析为 1 个整数。对于大型的数字序列而言，这样做可以节省大量内存。另外，Python 不允许向数组中添加与指定类型不同的值。

示例 2.19 创建一个包含 1000 万个随机 float 值的数组，将这些 float 值存入文件，再从文件中读取这些 float 值。

</> 示例 2.19：创建、保存、加载一个大型浮点型数组

```

1  >>> from array import array      ❶
2  >>> from random import random
3  >>> floats = array('d', (random() for i in range(10**7)))  ❷
4  >>> floats[-1]                ❸
5  0.9654076063623321
6  >>> fp = open('floats.bin', 'wb')
7  >>> floats.tofile(fp)         ❹
8  >>> fp.close()
9  >>> floats2 = array('d')       ❺
10 >>> fp = open('floats.bin', 'rb')
11 >>> floats2.fromfile(fp, 10**7) ❻
12 >>> fp.close()
13 >>> floats2[-1]              ❼
14 0.9654076063623321
15 >>> floats2 == floats        ❽
16 True

```

- ❶ 从 array 包导入 array 类型；
- ❷ 从一个可迭代对象（本例为一个生成器表达式）中创建一个双精度浮点数（类型代码为 'd'）数组 floats；
- ❸ 查看浮点数数组 floats 中最后 1 个数；
- ❹ 将 floats 数组存入二进制文件 floats.bin；
- ❺ 创建一个存放双精度浮点数（类型代码为 'd'）的空数组 floats2；
- ❻ 从二进制文件 floats.bin 中读取 1000 万个数到浮点数数组 floats2 中；
- ❼ 查看浮点数数组 floats2 中最后一个数；
- ❽ 确认两个浮点数数组 floats 与 floats2 的内容是否一致；

从示例 2.19 可以看出，array.tofile 与 array.fromfile 使用起来并不难。二者的速度也非常快，经我试验用 array.fromfile 从 array.tofile 创建的二进制文件中加载 1000 万个双精度浮点数，用时大约 0.1 秒。这比从文本文件中读取快了近 60 倍，而且无需使用内置函数 float 解析读取到的每一行。array.tofile 保存文件的速度约比一行一个浮点数写入文本文件快 7 倍。此外，保存 1000 万个双精度浮点数的二进制文件约占 80,000,000 字节（一个双精度浮点数占 8 字节，零开销），而保存相同数据量的文本文件占 181,515,739 字节。

若想使用数值数组表示二进制数据（如光栅图像），Python 有专门的类型：bytes 和 bytearray（详见“[四 Unicode 文本与字节序列](#)”）。

最后，比较一下 list 与 array 的功能（见表 2.3）。

表 2.3: list 与 array 的方法与属性¹⁶

方法/属性	list	array	
s.__add__(s2)	•	•	s + s2 拼接
s.__iadd__(s2)	•	•	s += s2 就地拼接
s.append(e)	•	•	在数组 s 尾部追加元素 e
s.byteswap()		•	交换数组中所有项目的字节, 以进行字节序转换
s.clear()	•		删除所有项
s.__contains__(e)	•	•	e in s
s.copy()	•		列表浅拷贝
s.__copy__()		•	为 copy.copy 提供支持
s.count(e)	•	•	统计元素 e 在对象 s 中出现的次数
s.__deepcopy__(()		•	为优化 copy.deepcopy 提供支持
s.__delitem__(p)	•	•	删除对象 s 中位置 p 处的项
s.extend(it)	•	•	向 s 追加可迭代对象 it 中的项
s.frombytes(b)		•	向 s 追加字节序列 b 中的项(解析为压缩机器值)
s.fromfile(f, n)		•	向 s 追加二进制文件 f 中的 n 项(解析为压缩机器值)
s.fromlist(l)		•	向 s 追加列表 l 中的项(一旦触发 TypeError, 则一项也不追加)
s.__getitem__(p)	•	•	s[p] 获取位置 p 上的项或切片
s.index(e)	•	•	查找元素 e 在对象 s 中首次出现的位置
s.insert(p, e)	•	•	在对象 s 的位置 p 之前插入元素 e
s.itemsize		•	数组中每一项的字节长度
s.__iter__()	•	•	获取迭代器
s.__len__()	•	•	len(s) 项数
s.__mul__(n)	•	•	s * n 将 s 重复拼接 n 次
s.__imul__(n)	•	•	s *= n 将 s 就地重复拼接 n 次
s.__rmul__(n)	•	•	n * s 反向重复拼接(详见“ 十六 运算符重载 ”)
s.pop([p])	•	•	删除并返回位置 p 处的项(默认为最后一项)
s.remove(e)	•	•	删除 s 中首次出现的元素 e
s.reverse()	•	•	就地反转 s 中项的顺序
s.__reversed__()	•		获取 s 中从后向前, 遍历项的迭代器
s.__setitem__(p, e)	•	•	s[p] = e 将 s 中位置 p 处的项或切片赋值为 e
s.sort([key], [reverse])	•		就地排序 s 中项的顺序
s.tobytes()		•	将数组 s 转换为 bytes 对象
s.tofile(f)		•	将项的压缩机器值存入二进制文件 f
s.tolist()		•	将数组 s 转换为 list 列表
s.typecode		•	获取数组 s 中项的类型(单字符的 C 语言类型)

¹⁶省略了由 object 实现的方法和弃用的数组方法。



从 Python 3.10 开始, array 类型不再提供就地排序方法 `s.sort()`。若需要对数组进行排序, 请使用内置函数 `sorted(s)` 重新构建排序后的数组。若为了确保增加新项后, 数组仍然是有序的, 请使用 `bisect.insort` 函数。示例如下:

```

1  >>> import array
2  >>> import bisect
3  >>> a = array.array('i',[9, 20, 1, 4, 100, 45])
4  >>> a = array.array(a.typecode, sorted(a)) # 对数组进行排序
5  >>> a
6  array('i', [1, 4, 9, 20, 45, 100])
7  >>> bisect.insort(a, 30) # 增加新项, 数组仍然有序
8  >>> a
9  array('i', [1, 4, 9, 20, 30, 45, 100])

```

经常使用数组的人, 如果不知道 `memoryview`, 可谓遗憾终生。下一小节, 将介绍 `memoryview`。

2.10.2 memoryview

内置的 `memoryview` 类是一种共享内存的序列类型, 可让你在不复制字节的情况下处理数组切片。它的灵感来源于 NumPy 库 (详见 “[2.10.3 NumPy](#)”)。NumPy 的主要作者 Travis Oliphant 在回答问题 “[什么时候适合使用 `memoryview`?](#)” 时说道:

`memoryview` 本质上是 Python 中通用的 NumPy 数组(不含数学功能)结构。它允许你在数据结构(如 PIL 图像、SQLite 数据库、NumPy 数组等)之间共享内存, 而无需事先复制。这个特性对大型数据集非常重要。

`memoryview.cast` 方法使用与 `array` 模块类似的表示法, 允许您根据需要以不同的方式读取或写入底层内存中的多个字节, 而无需移动字节的 bits。这对于需要以不同数据类型或不同字节序读取或写入内存的场景非常有用。`memoryview.cast` 将返回另一个 `memoryview` 对象, 且新对象始终与原对象共享相同的内存。

[示例 2.20](#)展示了如何以不同维度的矩阵视图 ($2 \times 3, 3 \times 2$), 展现一个 6 字节数组:

</> [示例 2.20](#): 分别以 $1 \times 6, 2 \times 3, 3 \times 2$ 矩阵视图, 处理 6 字节内存

```

1  >>> from array import array
2  >>> octets = array('B', range(6)) ❶ 构建一个6字节数组
3  >>> m1 = memoryview(octets) ❷ 根据数组, 构建一个memoryview对象
4  >>> m1.tolist()
5  [0, 1, 2, 3, 4, 5]
6  >>> m2 = m1.cast('B', [2, 3]) ❸ 将1x6的memoryview对象转换为2x3
7  >>> m2.tolist()
8  [[0, 1, 2], [3, 4, 5]]
9  >>> m3 = m1.cast('B', [3, 2]) ❹ 将1x6的memoryview对象转换为3x2
10 >>> m3.tolist()
11 [[0, 1], [2, 3], [4, 5]]
12 >>> m2[1,1] = 22 ❺ 修改memoryview对象m2中的项

```

```

13  >>> m3[1,1] = 33          ❶ 修改memoryview对象m3中的项
14  >>> octets
15  array('B', [0, 1, 2, 33, 22, 5])  ❷ 多个memoryview对象之间共享内存

```

- ❶ 构建一个 6 字节的数组 octets, 存储类型为 'B';
- ❷ 根据 6 字节数组构建一个 memoryview 对象 m1, 然后导出为列表;
- ❸ 根据 ❷ 中的 memoryview 对象 m1, 以 2 行 3 列的形式, 构建一个新的 memoryview 对象 m2;
- ❹ 根据 ❷ 中的 memoryview 对象 m1, 以 3 行 2 列的形式, 构建一个新的 memoryview 对象 m3;
- ❺ 将 m2 的第 2 行第 2 列设置为 22, 即将 m2 中的 4 修改为 22;
- ❻ 将 m3 的第 2 行第 2 列设置为 33, 即将 m3 中的 3 修改为 33;
- ❼ 显示原数组 m1, 证明 octets、m1、m2、m3 之间的内存是共享的。

memoryview 的这种强大功能也可用于搞破坏。如示例 2.21 所示, 将展示如何用 memoryview 破坏一个 16 位整数数组中某项的一个字节。

</> 示例 2.21: 修改一个 16 位整数数组中某一项的字节, 以改变该项的值。

```

1  >>> import array
2  >>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
3  >>> memv = memoryview(numbers) ❷
4  >>> len(memv)
5
6  >>> memv[0] ❸
7  -2
8  >>> memv_oct = memv.cast('B') ❹ memv_oct 一个元素占一个字节
9  >>> memv_oct[0], memv_oct[1], memv_oct[2], memv_oct[3]
10 (254, 255, 255, 255)
11 >>> memv_oct.tolist() ❺
12 [254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
13 >>> memv_oct[5] = 4 ❻ 修改memv_oct中第6个字节为4
14 >>> numbers
15 array('h', [-2, -1, 1024, 1, 2]) ❼

```

- ❶ 构建一个包含 5 个 16 位有符号整数 (typecode 为 'h') 的数组 numbers; 数组中每个元素占 2 个字节;
- ❷ 根据数组 numbers, 构建一个 memoryview 对象 memv;
- ❸ memoryview 对象 memv 中的 5 项, 与原数组 numbers 一样;
- ❹ 根据 memv 对象构建新的 memoryview 对象 memv_oct (类型代码 'B')。memv_oct 中每个元素占 1 个字节;
- ❺ 导出 memoryview 对象 memv_oct 中的元素, 显示为一个包含 10 个字节的列表, 方便查看;
- ❻ memv_oct[5] = 4 修改 memv_oct 中第 6 (索引位为 5) 个字节为 4。
- ❼ 因为 numbers 与 memv_oct 共享内存, 所以 memv_oct[5]=4 相当于修改 numbers[3] 的高位字节为 4 (numbers[3] 低位字节为 memv_oct[4])。注意 numbers 的变化¹⁷: 对于一个 2 字节无符号整数, 高位字节为 4 (二进制 00000100), 则此整数对应的二进制值为 00000100 00000000, 十进制值就是 1024

¹⁷ 数组 numbers 变化的原因: 具体来说, 由于 numbers 是一个有符号短整型 (2 字节) 数组, 原始 numbers 数组中的第 3 个元素 (numbers[2] = 0) 在内存中的表示为 00000000 00000000, 而被修改后的 numbers 数组中的第 3 个元素 (numbers[2] = 1024) 在内存中的表示为 00000100 00000000。这里高位字节由 00000000 变为 00000100, 导致整数的值从 0 变为 1024。

(详见图 2.7)。

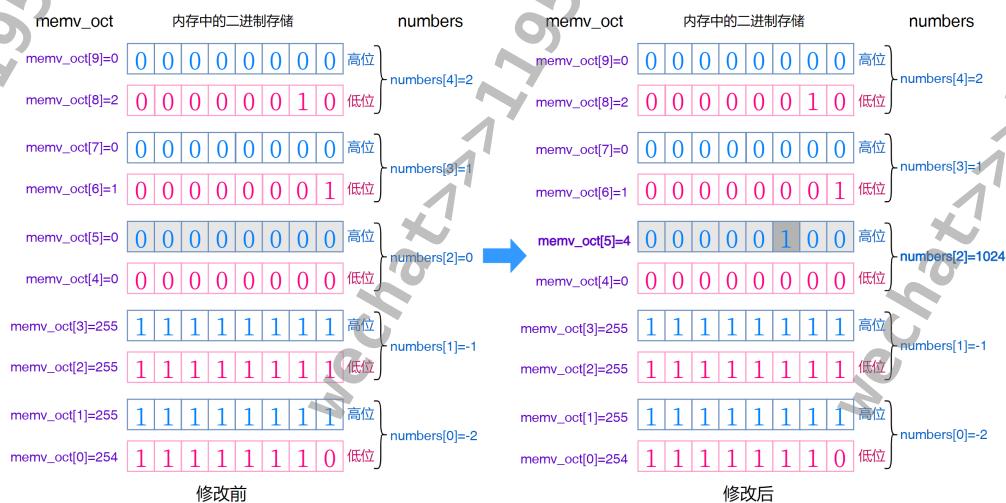


图 2.7: 通过 memoryview 修改内存中的字节

如果要对数组做一些高级的数值处理(如矩阵处理),则应该使用 NumPy 库。时不我待,现在就开始吧。

2.10.3 NumPy

本书重点关注的是如何充分利用 Python 标准库中已有的功能。但是,NumPy 实在是太优秀了,值得单列出来讲解一下。

得益于 NumPy 的优秀,Python 成为科学计算(需要处理高级数组和矩阵运算)领域的主流语言。NumPy 实现了多维同构数组和矩阵类型,不仅能保存数字,还能保存用户定义的记录,并提供高效的元素层面操作。

在 NumPy 基础之上编写的 SciPy 库提供了线性代数、数值微积分、统计学中的许多科学计算方法。SciPy 速度快、运算可靠,因为它利用了 Netlib Repository 中广泛使用的 C 和 Fortan 语言代码库。换句话说,SciPy 为科学家提供了两全其美的解决方案,既有交互式提示,又有高级 Python API,以及用 C 和 Fortran 优化的工业级数值运算函数。

示例 2.22 简单演示用 NumPy 对二维数组做一些基本操作的用法。

</> 示例 2.22: numpy.ndarray 中行与列的基本操作

```

1  >>> import numpy as np      ❶
2  >>> a = np.arange(12)       ❷
3  >>> a
4  array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
5  >>> type(a)
6  <class 'numpy.ndarray'>
7  >>> a.shape              ❸
8  (12,)
9  >>> a.shape = 3, 4        ❹
10 >>> a
11 array([[ 0,  1,  2,  3],
12        [ 4,  5,  6,  7],
13        [ 8,  9, 10, 11]])
14 >>> a[2]                  ❺

```

```

15 array([ 8,  9, 10, 11])
16 >>> a[2, 1]          ⑥
17 9
18 >>> a[:, 1]          ⑦
19 array([1, 5, 9])
20 >>> a.transpose()    ⑧
21 array([[ 0,  4,  8],
22        [ 1,  5,  9],
23        [ 2,  6, 10],
24        [ 3,  7, 11]])

```

- ① 导入安装好的 NumPy 包 (NumPy 不在 Python 标准库中,需要单独安装)。习惯将 NumPy 导入为 np;
- ② 构建一个从整数 0~11 的 numpy.ndarray 对象,然后查看该对象的内容;
- ③ 查看数组 a 的维度。这是个包含 12 个元素的一维数组;
- ④ 为数组 a 增加一个维度,使其变为 3 行 4 列的二维数组;
- ⑤ 获取索引位 2 上的行,即第 3 行;
- ⑥ 获取索引位 (2,1) 上的元素,即第 3 行第 2 列上的元素;
- ⑦ 获取索引位 1 上的列,即第 2 列;
- ⑧ 对数组 a 执行转置 (行列转换)操作,将创建一个新数组。

NumPy 还支持一些高级操作。如加载、保存和操作 numpy.ndarray 对象的所有元素。

```

1 >>> import numpy
2 >>> data = numpy.random.randint(10**14, 10**15, dtype=np.int64, size=(10**7,)) ①
3 >>> float_data = numpy.round(data / (10**9), 8)                          ②
4 >>> np.savetxt("floats-10M-lines.txt", float_data, fmt='%.8f', delimiter='\n',
5               newline='\n')
6 >>> floats = numpy.loadtxt('floats-10M-lines.txt')                         ③
7 array([722017.39029803, 697107.90326653, 898128.62672448])
8 >>> floats *= .5               ⑤
9 >>> floats[-3:]               ④
10 array([361008.69514901, 348553.95163327, 449064.31336224])
11 >>> from time import perf_counter as pc          ⑥
12 >>> t0 = pc(); floats /= 3; pc() - t0          ⑦
13 0.009845599997788668
14 >>> numpy.save('floats-10M', floats)           ⑧
15 >>> floats2 = numpy.load('floats-10M.npy', 'r+') ⑨
16 >>> floats2 *= 6
17 >>> floats2[-3:]               ⑩
18 memmap([722017.39029803, 697107.90326653, 898128.62672448])

```

- ① 生成 1000 万个范围在 $10^{14} \sim 10^{15}$ 之间的整数数组 data;
- ② 将数组 data 中的每个项除以 10^9 后,四舍五入保留 8 位小数。然后,将数组 data 写入文件 floats-10M-lines.txt,每个浮点数占一行;
- ③ 从文本文件 floats-10M-lines.txt 载入 1000 万个浮点数到 numpy 数组 floats 中;
- ④ 用序列切片表示法查看数组 floats 中最后 3 个数;

- ⑤ 将数组 floats 中的每个元素都乘以 0.5, 然后查看最后 3 个数;
- ⑥ 导入性能测量计时器包 perf_counter (自 Python3.3 起可用);
- ⑦ 将数组 floats 中的每个元素都除以 3。1000 万个元素耗时不到 10ms;
- ⑧ 将数组 floats 保存到二进制文件 floats-10M.npy 中;
- ⑨ 以内存映射文件的形式将文件 floats-10M.npy 的内容加载到另一个数组 floats 中。即使数组因太大而无法完全放入内存, Python 也能高效地处理数组切片;
- ⑩ 将数组 floats2 中的每个元素都乘以 6 之后, 查看最后 3 个元素;

NumPy 和 SciPy 这两个库的功能异常强大, 为很多优秀的工具 (如 Pandas、scikit-learn) 提供了坚实的基础。Pandas 实现的高效数组类型可以保存非数值型数据, 此外还支持多种格式 (包括 csv、xls、SQL 转储、HDF5 等) 的导入和导出。scikit-learn 是目前使用最广泛的机器学习工具集。NumPy 和 SciPy 这两个库中的函数大多是用 C 或 C++ 实现的, 由于它们释放了 Python 的 GIL (全局解释器锁), 因此可以充分利用 CPU 的所有核心。Dask 项目支持跨机器集群并行处理 NumPy、Pandas 和 scikit-learn 操作。这些包虽然不是本书的重点, 但是如果不能快速了解一下 NumPy, 您就无法看到 Python 序列的全貌。

在了解了扁平序列 (标准数组、NumPy 数组) 之后, 现在换个话题, 一起来看看 list 列表替代品: 队列。

2.10.4 双端队列与其他队列

借助列表的.append(e) 与.pop([p]) 方法, 可将列表当作栈或队列¹⁸来使用。但是, 在列表头部 (即索引位为 0) 插入和删除项时, 需要在内存中移动列表, 所以会产生额外的开销。

`collections.deque`类实现了一种线程安全的双端队列, 旨在可以快速地在队列两端插入和删除项。在构建双端队列时, 可以通过可选参数 maxlen 将队列设置为 “有界队列 (即长度固定) ”。当有界 deque 对象被填满后, 从一端添加新项, 将自动从另一个端丢弃一项。示例 2.23 展示了可对 deque 对象执行的一些基本操作。

</> 示例 2.23: 处理一个 deque 对象

```

1  >>> from collections import deque
2  >>> dq = deque(range(10), maxlen=10) ❶ 构建一个10项的有界队列
3  >>> dq
4  deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
5  >>> dq.rotate(3)                  ❷ 将右侧3项移到左侧
6  >>> dq
7  deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
8  >>> dq.rotate(-4)                ❸ 将左侧4项移到右侧
9  >>> dq
10 >>> dq
11 >>> dq.appendleft(-1)           ❹ 左侧追加1项, 挤掉右侧1项
12 >>> dq
13 >>> dq
14 >>> dq.extend([11, 22, 33])    ❺ 向右侧依次追加迭代对象中的项, 挤掉左侧3项
15 >>> dq
16 >>> dq
17 >>> dq.extendleft([10, 20, 30, 40]) ❻ 向左侧依次追加迭代对象中的项, 挤掉右侧4项。
18 >>> dq

```

¹⁸通过.append() 与.pop(0) 实现先进先出 (FIFO) 行为。

19 `deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)`

- ① 可选参数 maxlen=10, 限制 deque 实例中最多可以存放 10 项。 maxlen 也是 deque 实例的只读属性;
- ② rotate(n) 轮转 deque 实例中的项。当 n>0 时, 从 deque 实例右端取 n 项放到左端; 当 n<0 时, 从左端取 n 项放到右端;
- ③ rotate(-4) 将 deque 实例中左侧的 4 项 (即 7,8,9,0) 放到队列右侧;
- ④ 用 appendleft(-1) 方法向已满 (即 len(d) == d maxlen) 的 deque 对象左侧添加一项, 将导致另一端会丢弃一项 (0 被丢弃);
- ⑤ 向右侧依次追加迭代对象中的项, 将左侧的 3 项 (即 -1, 1, 2) 挤出队列;
- ⑥ 向左侧依次追加迭代对象中的项, 将右侧的 4 项 (即 9, 11, 22, 33) 挤出队列;

表 2.4 为 list 与 deque 可用的方法对比。从表中可见, deque 实现了大部分 list 方法, 并增加了一些针对 deque 设计的专用方法, 如 popleft 和 rotate。但这也有一个隐藏的代价: 从 deque 中删除项的速度并不快。append 和 popleft 操作是原子操作, 因此 deque 可以安全地用作多线程应用程序中的先进先出队列, 而无需加锁。

表 2.4: list 与 deque 实现的方法²⁰

方法	list	deque	
<code>s.__add__(s2)</code>	•		<code>s + s2</code> 拼接
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> 就地拼接
<code>s.append(e)</code>	•	•	将元素 e 追加到右端 (放在最后一个元素后面)
<code>s.appendleft(e)</code>		•	将元素 e 追加到左端 (放在第一个元素前面)
<code>s.clear()</code>	•	•	删除所有项
<code>s.__contains__(e)</code>	•		<code>e in s</code>
<code>s.copy()</code>	•		列表浅拷贝
<code>s.__copy__()</code>		•	为 <code>copy.copy</code> (浅拷贝) 提供支持
<code>s.count(e)</code>	•	•	计算元素 e 在序列 s 中出现的次数
<code>s.__delitem__(p)</code>	•	•	删除位置 p 上的项
<code>s.extend(i)</code>	•	•	将可迭代对象 i 中的项逐个追加到 s 的右侧
<code>s.extendleft(i)</code>		•	将可迭代对象 i 中的项逐个追加到 s 的左侧
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> 获取位置 p 上的项或切片
<code>s.index(e)</code>	•		查找元素 e 在列表 s 中首次出现的位置
<code>s.insert(p, e)</code>	•		在位置 p 前面插入元素 e
<code>s.__iter__()</code>	•	•	获取迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code> 获取项数
<code>s.__mul__(n)</code>	•		<code>s * n</code> 重复拼接
<code>s.__imul__(n)</code>	•		<code>s *= n</code> 就地重复拼接
<code>s.__rmul__(n)</code>	•		<code>n * s</code> 反向重复拼接 (详见 “ 十六 运算符重载 ”)
<code>s.pop()</code>	•	•	删除并返回最后 (右侧) 一项。列表还通过参数 p, 删除位置处 p 的项。

续表 2.4

方法	list	deque	
s.popleft()		•	删除并返回第一(左侧)项。
s.remove(e)	•	•	删除 s 中第一个出现的元素 e
s.reverse()	•	•	就地反转 s 中各项的顺序
s.__reversed__()	•	•	获取从后向前遍历的迭代器
s.rotate(n)		•	将队列中的 n 项,从一端移到另一端。n>0,从右向左;n<0,从左向右。
s.__setitem__(p, e)	•	•	s[p] = e 设置位置 p 的元素为 e,覆盖现有的项或切片
s.sort([key], [reverse])	•		就地排序列表

除了 deque 双端队列之外,Python 标准库的如下几个包中还实现了一些其他队列:

- **queue 包**

提供了几个可用于线程间安全通信的同步队列类: SimpleQueue、Queue、LifoQueue 和 PriorityQueue。除了 SimpleQueue 之外, 其他几个类都可以通过指定构造参数 maxsize, 将队列设置为“有界”。但是, 与有界 deque 双端队列不同的是, 这些有界队列在队列被填满后, 不会为了腾出空间而丢弃队列中的项, 而是阻塞插入新项直至其他线程从队列中取出一项。利用这种特性, 可以限制当前的活动线程数量。

- **multiprocessing 包**

multiprocessing 包实现了自己的无界 SimpleQueue 队列和有界 Queue 队列。这与 queue 包中的队列类似, 只不过是专门针对进程间通信的队列。multiprocessing 包还为任务管理提供了专用的multiprocessing.JoinableQueue队列。

- **asyncio 包**

asyncio 包提供了 Queue、LifoQueue、PriorityQueue 和 JoinableQueue 这些队列, 以及受 queue 和 multiprocessing 块启发的 API。但这些 API 用于管理异步编程中的任务。

- **heapq 包**

与前 3 个模块相比, heapq 未实现任何队列, 只是提供了 heappush 和 heappop 等函数, 以便可以将可变队列当作堆队列或优先级队列使用。

至此, 我们结束了对 list 类型替代方案的概述, 也结束了对一般序列类型的探索。余下的 str 和二进制序列将在“[四 Unicode 文本与字节序列](#)”中介绍。

2.11 本章小结

要想编写出简洁、高效和地道的 Python 代码, 必须要掌握标准库中的各种序列类型。

Python 序列可以按可变性分为: 可变序列和不可变序列, 也可以按内存模型分为: 扁平序列和容器序列。扁平序列更紧凑、更快、更易用, 但仅限于存储原子数据, 如数字、字符和字节。容器序列则更灵活, 但在存放可变对象时可能会产生意想不到的结果。因此, 在将容器序列与嵌套数据结构结合使用时, 要格外谨慎。

需要特别注意的是, Python 中没有绝对意义上的不可变容器序列类型: 即使是“不可变”的元组, 当包含可变项(如列表或用户定义对象)时, 它的值也可能被改变。

²⁰省略了由 object 实现的方法

列表推导式和生成器表达式是构建和初始化序列的最佳手段。如果您对它们还不熟悉,请花点时间掌握它们的基本用法。这并不难,很快得心应手。

Python 中的元组有两种作用:作为无字段名称的 record 记录使用和作为不可变列表使用。将元组用作不可变列表时,请记住,只有当元组中的所有项都是不可变对象时,才能保证元组值是固定的。对一个元组调用 `hash(t)` 是判断其值是否是固定值的快速方法。如果 `t` 中包含可变项,则会引发 `TypeError`。

当元组用作 record 记录时,元组解包是提取元组字段的最安全、可读性最高的方法。除了解包元组之外,* 还能在许多上下文中用于解包列表和可迭代对象,它的一些用途由 Python 3.5 的 “[PEP 448- Additional Unpacking Generalizations](#)” 中。Python 3.10 引入了用 `match/case` 语法表示的序列模式匹配 (Pattern Match),以支持更强大的解包,即所谓的 “[析构 \(Destructor\)](#)”。序列切片是 Python 最受欢迎的语法特性,它的强大远超您的想象。用户定义的序列也可以支持 NumPy 中使用的多维切片和省略号 (..., 以 3 个英文句点表示) 表示法。通过切片赋值来修改可变序列是一种极具表现力的方式。

重复拼接 (如 `seq * n`) 非常方便,只要小心谨慎,还可以用于初始化内含不可变项的嵌套列表。针对可变序列的增量赋值 (如 `+=` 或 `*=`),会就地更改 (序列需要实现 `__iadd__` 或 `__imul__` 方法) 可变序列;而针对不可变序列的增量赋值,则会产生新的不可变序列。

`x.sort()` 方法和内置的 `sorted(x)` 函数使用方便灵活,这要归功于可选的 `key` 参数。`key` 参数的值是一个只接受一个参数的函数 (用于计算排序标准的函数) 名。顺便说一下,内置的 `min` 和 `max` 函数也有可选参数 `key`。

除了列表和元组,Python 标准库还提供了 `array.array`。虽然 NumPy 和 SciPy 并不是标准库的一部分,但如果您需要对大数据集进行数值处理,那么哪怕学习这些库中的一小部分,也会让您受益匪浅。

最后,我们简单介绍了多功能、线程安全的 `collections.deque`。并在 “[表 2.4<58页>](#)” 中对 `deque` 与 `list` 的 API 进行了对比,并提到了 Python 标准库实现的其他队列。

2.12 延伸阅读

《[Python Cookbook, 3rd Ed](#)》(David Beazley、Brian K. Jones 合著) 一书的第 1 章 “[Data Structures](#)” 中,有许多以序列为重点的示例,包括 “[示例 1.11. Naming a Slice](#)”,从中我学到了将分片赋值给变量以提高可读性的技巧,如 “[示例 2.13<42页>](#)” 所示。

《[Python Cookbook,2nd Ed](#)》是为 Python 2.4 编写的,但其中的许多代码都能在 Python 3 中使用,而且第 5 章和第 6 章中的许多示例都涉及序列。该书由 Alex Martelli、Anna Ravenscroft 和 David Ascher 编辑,其中汇集了数十位 Python 爱好者的贡献。第 3 版从头开始重写,更侧重于语言的语义,尤其是 Python 3 中的变化,而旧版则强调语用 (即如何将语言应用到实际问题中)。尽管第 2 版中的一些解决方案已过时,但老实说,我认为手头同时拥有 2 版《[Python Cookbook](#)》是值得的。

Python 官方的 “[Sorting Techniques](#)” 有几个使用 `sorted` 和 `list.sort` 的高级示例。

“[PEP 3132 -Extended Iterable Unpacking](#)” 是阅读并行赋值左侧 `*extra` 语法新用法的权威资料。如果您想了解 Python 的发展历程,可以看一下 BUG 跟踪工单 “[Missing *-unpacking generalizations](#)”,该工单建议对可迭代解包符号进行增强。其讨论结果被汇集成了 “[PEP 448 -Additional Unpacking Generalizations](#)”。

如 “[2.6 序列模式匹配<33页>](#)” 所述,在 “[What's New In Python 3.10](#)” 中,Carol Willing 写的 “[Structural Pattern Matching](#)” 以大约 1400 字的篇幅 (打印为 PDF 时,不足 5 页) 很好地介绍了模式匹配 (Pattern Match) 新特性。“[PEP 636 - Structural Pattern Matching: Tutorial](#)” 也不错,但篇幅较长。“[PEP 636](#)” 的

“Appendix A –Quick Intro”一节,内容比 Willing 写的介绍要简短,因为它省略了模式匹配 (Pattern Match) 优势的介绍。如果您需要更多论据来说服自己或他人模式匹配 (Pattern Match) 对 Python 有好处,请阅读长达 22 页的“PEP 635 –Structural Pattern Matching: Motivation and Rationale”。

Eli Bendersky 的博文“Less copies in Python with the buffer protocol and memoryviews”中包含一个关于 memoryview 的简短教程。

市面上有许多介绍 NumPy 的书籍,有些书名甚至未提及“NumPy”一词。《Python Data Science Handbook》(Jake VanderPlas 著)与《Python for Data Analysis, 2nd Ed》(Wes McKinney 著)这两本书都值得一读。

“NumPy 建立在向量之上”。这是《From Python to NumPy》(Nicolas P. Rougier 著)一书的开篇语。向量化运算将数学函数应用于数组中的所有元素,无需用 Python 编写显式循环。它们可以并行操作,在现代 CPU 中通过特殊的矢量指令,利用多核 CPU 或委托 GPU 执行(取决于特定的库)。Rougier 书中的第 1 个示例显示,用生成器方法将一个 Pythonic 类重构为调用几个 NumPy 向量函数的精简函数后,速度提高了 500 倍。

要了解如何使用 `deque`(和其他容器),请参阅 Python 文档中“collections —Container datatypes”中的示例和实用技巧。

Edsger W. Dijkstra 本人在题为“Why numbering should start at zero”的简短备忘录中,对排除区间与切片中最后 1 项的 Python 约定,做了最佳辩护。备忘录的主题是数学表示法,但也与 Python 相关,因为 Dijkstra 严谨而幽默地解释了为何像 2, 3, ..., 12 这样的序列应该用 $2 \leq i < 13$ 表示。所有其他可能合理的约定写法均被否定了,因为不能统一所有用户的想法。标题指的是从 0 开始的索引,但备忘录实际上是关于为何“ABCDE”[1:3] 得到的是“BC”而不是“BCD”,以及为何 `range(2, 13)` 会得到 2, 3, 4, ..., 12。顺便说一下,虽然该备忘录是手写的,但字迹漂亮、清晰可度,甚至有人根据备忘录的字迹创建了一种字体。

杂谈

元组的本质

2012 年,我在 PyCon US 上展示了关于 ABC 语言 的海报。在创建 Python 之前,Guido van Rossum 曾开发过 ABC 语言 的解释器,所以他来看了我的海报。交谈中,提到了 ABC 语言 的 compound 类型,这显然是 Python 元组的前身。compound 也支持并行赋值,并可用作字典(或 ABC 语言 中的 Table 类型)中的复合 Key。不过,compound 类型不是序列,它不可迭代、不能通过索引检索字段、更不能对它进行切片。只能将 compound 类型作为一个整体来处理,或者通过并行赋值提取其中的各个字段,仅此而已。

我与 Guido 讲,这些限制正凸显了 compound 类型的主要用途——即用作没有字段名的记录(record)。他回应道“将元组表现为序列的特性实际上是一种 hack^a”。

这是一种务实的做法,正是因为 Python 更务实,所以才比 ABC 语言 更成功。从语言实现者的角度来看,让元组表现为序列的成本很低。这样做虽然将记录这种用途弱化了,但得了一种不可变列表——即使它不像 frozenlist 那样有明确的名称。

扁平序列与容器序列

为了强调不同序列类型采用的内存模型,我使用了“容器序列”和“扁平序列”这 2 个术语。“容器”一词来自文档《3. Data model》:“一种对象中包含对其他对象的引用,这种对象称为“容器 (collection)”。^b

为了表达准确,我将“容器”与“序列”连在一起,因为 Python 中有些容器不是序列,如 `dict` 和 `set`。

容器类型可以嵌套, 其中可以包含任何类型的对象, 甚至是容器自身。

而扁平序列是不可嵌套的序列类型, 其内部只能包含简单的原子类型, 如 int、float 或 char。

采用“扁平序列”这个术语是为了与“容器序列”进行区分。

在官方文档中, 除了前面提到的“容器”之外, collections.abc 中还有一个名为 Container 的抽象基类 (ABCs), 这个类只有一个方法, 即 in 运算符背后的特殊方法 `__contains__`。字符串与数组不是传统意义上的容器, 却是 Container 类的虚拟子类 (Virtual Subclass)。因为字符串与数组都实现了 `__contains__` 方法。这表明人类经常用同一个词指代不同的事物, 本书使用“容器”一词指代可包含其他对象引用的对象, 而用 Container 指代 collections.abc.Container 类。

大杂烩列表

Python 入门教程通常会强调列表中可以包含不同类型的对象, 但在实践中, 这个特性却并不是很有用。我们将一些项放入列表中, 是为了稍后处理它们, 这意味着所有的项都应该至少支持某些共同的操作 (也就是说, 不管它们在基因上是不是 100% 的鸭子, 它们都应该“嘎嘎”叫)。例如, 在 Python 3 中, 若列表中的项不可以比较, 那么就不能对列表进行排序操作 (如下所示):

```

1  >>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
2  >>> sorted(l)
3  Traceback (most recent call last):
4  File "<stdin>", line 1, in <module>
5  TypeError: '<' not supported between instances of 'str' and 'int'
6

```

与列表不同, 元组内通常包含不同类型的项。这也很容易理解: 如果元组是数据库表中的 1 行记录, 则表中的每个字段很可能具有不同的数据类型。

聪明的参数 key

list.sort、sorted、max 和 min 都有 1 个可选参数 key, 这是个好主意。使用参数 key 既简单又高效。说它简单, 是因为只需定义一个单参数函数, 此函数可以检索或计算您想用来对对象进行排序的任何标准; 这比编写一个返回 -1、0、1 的双参数函数要容易得多。说它高效, 是因为参数 key 指定的函数只在处理序列中各项时调用 1 次, 而双参数比较函数在每次排序算法需要比较两个项时都要调用。当然, Python 也需要在排序时比较 key, 但这种比较是在优化的 C 代码中完成的, 而不是在您编写的 Python 函数中完成的。

顺便说一下, 使用参数 key, 即使序列中参杂了数字和类似数字的字符串, 也可以正确排序。我们只需通过设置 key 参数, 将所有项视为 int 或 str 即可。

```

1  >>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
2  >>> sorted(l, key=int)      # 将所有项都视为 int
3  [0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
4  >>> sorted(l, key=str)      # 将所有项都视为 str
5  [0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
6

```

Oracle、Goole 和 Timbot 阴谋

sorted(list) 与 list.sort 使用的排序算法是 Timsort。这是一种自适应算法, 可以根据数据的排序方式

在插入排序法和归并排序法之间切换。这样做更高效,这是因为现实中的数据,某些项往往往往已具有一定顺序。

Timsort 于 2002 年首次在 CPython 中使用。自 2009 年起, Timsort 也被用于标准 Java 和 Android 中的数组排序,这一事实在 Oracle 将 Timsort 的部分相关代码作为 Goole 侵犯 Sun 知识产权的证据时,才广为人知^c。2021 年,美国最高法院裁定 Goole 使用 Java 代码为“合理使用”。

Timsort 是由 Tim Peters 发明的,他是一位多产的 Python 核心开发人员,以至于人们认为他是一个 AI,人称“Timbot”。您可以在“[Python Humor](#)”中了解这一阴谋论。《Python 之禅》(import this) 也是 Tim 写的。

^ahack 意味着这种行为并不是元组的本质目的,而是一种暂时性的修改或扩展。这种 hack 使得元组在处理一些序列操作时更加方便,但在设计和使用元组时,应该将元组视为一种记录,而不是序列。

^b为了以示区分,本书在翻译时将 collection 翻译为“容器”,将 Python 内置类型 set 翻译为“集合”。

^c详见由威廉·阿尔苏普 (William Alsup) 法官于 2012 年颁布的命令。

wechat: 119554488

字典与集合

基本上, Python 就是包裹在一堆语法糖 (Syntactic Sugar) 中的字典。

—Lalo Martins, 早期的数字游民、Python 专家

所有的 Python 程序中都会用到字典。即使不直接在我们的代码中使用, 也会间接用到字典。因为 `dict` 类型是实现 Python 的基石。Python 的一些核心结构 (如类和实例属性、模块的命名空间以及函数关键字参数等) 在内存中, 都是以字典的形式存在的。此外, `__builtins__.__dict__` 存储了所有的内置类型、对象和函数。

由于字典的重要作用, Python 对其做了高度优化, 而且仍在不断改进。Python 字典能如此高效, 这要归功于“哈希表”。

除了字典之外, 内置类型中的 `set` 和 `frozenset` 也基于哈希表。与其他语言相比, Python 为这两种类型提供了更丰富的 API 和运算符。具体而言, Python 集合实现了集合论中的所有基本运算, 如并集、交集、子集测试等。有了它们, 我们就可以用更具描述性的方式来表达算法, 避免了大量的嵌套循环和测试条件。

本章涵盖如下内容:

- 构建及处理 `dict` 和映射的现代语法, 包括增强的解包和模式匹配;
- 映射 (mapping) 类型的常用方法;
- 缺失 (missing) key 的特殊处理;
- 标准库中的 `dict` 变体;
- `set` 与 `frozenset` 类型;
- 哈希表对集合与字典行为的影响;

3.1 本章新增内容

第 2 版中的改动, 大多是对映射类型新功能的介绍。

- “[3.2 字典的现代语法](#)” 介绍了增强的解包语法和合并映射的不同方法, 包括受 `dict` 支持的 `!` 和 `!=` 运算符 (Python 3.9 引入);

- “3.3 用模式匹配处理映射”说明如何使用 match/case 模式匹配 (Python 3.10 引入) 语法处理映射;
- “3.6.1 collections.OrderedDict”将关注 dict 与 OrderedDict 之间微小却不可忽略的差异。毕竟自 Python 3.6 开始,dict 也能保留 key 的插入顺序;
- 新增的“3.8 字典视图”与“3.12 字典视图的集合运算”,将讲解 dict.keys、dict.items 和 dict.values 返回的视图对象;

dict 和 set 的底层实现仍然依赖于哈希表,但 dict 代码有两个重要的优化,可以节省内存并保留 dict 中 key 的插入顺序。“3.9 dict 实现方式对实践的影响”和“3.11 集合的实现方式对实践的影响”总结了如何合理利用 dict 与 set 类型。

3.2 字典的现代语法

接下来的章节将介绍用于构建、解包和处理映射的高级语法功能。其中一些特性在语言中并不是新的,但对您来说可能是新的。还有一些需要 Python 3.9 (如 | 操作符) 或 Python 3.10 (如 match/case)。先从我们熟知的功能讲起。

3.2.1 字典推导式

自 Python 2.7 起,列表推导式和生成器表达式的语法经过改造,可以适用于字典推导式(以及后文中的集合推导式)。字典推导式可以从任何可迭代对象中获取 Key:Value 对,并构建 dict 实例。

示例 3.1 展示了如何使用字典推导式,根据同一个元组列表建立 2 个 dict 实例。

```
</> 示例 3.1: 字典推导式示例
1  >>> dial_codes = [
2      ...     (880, 'Bangladesh'),
3      ...     (55, 'Brazil'),
4      ...     (86, 'China'),
5      ...     (91, 'India'),
6      ...     (62, 'Indonesia'),
7      ...     (81, 'Japan'),
8      ...     (234, 'Nigeria'),
9      ...     (92, 'Pakistan'),
10     ...     (7, 'Russia'),
11     ...     (1, 'United States'),
12     ... ]
13
14  >>> country_dial = {country: code for code, country in dial_codes} ❶
15
16  >>> country_dial
17  {'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91, 'Indonesia': 62, 'Japan':
18  ...     : 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United States': 1}
19
20  >>> {code: country.upper()
21  ...     for country, code in sorted(country_dial.items())
22  ...     if code < 70}
23  {55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'} ❷
```

❶ 构建由多个元组构成的列表

❷

❶ 像 dial_codes 这样的可迭代 Key:Value 对可以直接传递给 dict 构造函数;

- ② 这里,将 dial_codes 项中 Key 和 Value 的位置:以 country 为 Key,以 code 为 Value;
- ③ 按国家名称排序 country_dial,再次对调 Key 和 Value,将 Value 转换为大写形式,并筛选出 code<70 的项;

在习惯了列表推导式之后,理解字典推导式很容易。即使暂时不理解也没关系,推导式语法已经被广泛使用,熟练掌握是迟早的事。

3.2.2 映射解包

从 Python 3.5 开始,“PEP 448 -Additional Unpacking Generalizations”在两方面增强了对映射解包的支持。

首先,在函数调用时,可以为多个参数都应用 ** 符号来解包映射(如示例 3.2 所示)。但是,要求映射中的所有 key 都是字符串类型,并且在所有的参数中是唯一的(因为关键字参数不可以重复)。

</> 示例 3.2: **解包映射

```
1 >>> def dump(**kwargs):
2     ...     return kwargs
3 ...
4 >>> dump(**{'x': 1}, y=2, **{'z': 3})
5 {'x': 1, 'y': 2, 'z': 3}
```

其次,**可以在 dict 的字面量中使用,同样也可以多次使用(如示例 3.3 所示)。

</> 示例 3.3: dict 字面量中的 **

```
1 >>> {'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
2 {'a': 0, 'x': 4, 'y': 2, 'z': 3} # 后面的 x:4, 覆盖了前面的 x:1
```

在此情况下,允许键(Key)重复。但是,后面的键(Key)会覆盖前面的键(Key)。比如示例 3.3 中 x 映射的值。

此种句法也可用于合并映射,但是合并映射还有其他方式(见“3.2.3 用 | 合并映射”)。

3.2.3 用 | 合并映射

从 Python 3.9 开始,支持使用并集运算符 | 与 |= 来合并映射。

运算符 |= 将创建一个新的映射(如示例 3.4 所示)。

</> 示例 3.4: 用 | 合并映射

```
1 >>> d1 = {'a': 1, 'b': 3}
2 >>> d2 = {'a': 2, 'b': 4, 'c': 6}
3 >>> d1 | d2          # 创建了新的映射
4 {'a': 2, 'b': 4, 'c': 6}
```

通常,运算符 | 创建的新映射类型与左操作数(如示例 3.4 中的 d1)相同。但是,当涉及用户定义的类型时,也可能与右操作数的映射类型相同。这背后,涉及到了运算符重载规则(详见“十六 运算符重载”)。

`|=` 运算符, 将就地更新现有 (即左操作数) 的映射。续前例 (示例 3.4), 当时映射 `d1` 未发生变化。但是, 用 `|=` 运算符合并映射后, 就地更新了映射 `d1` (如示例 3.5 所示)。

```
</> 示例 3.5: 用 |= 合并映射
1 >>> d1          # 续上例, 映射 d1 未变化
2 {'a': 1, 'b': 3}
3 >>> d1 |= d2
4 >>> d1          # 就地更新了左操作数 d1
5 {'a': 2, 'b': 4, 'c': 6}
```



如果您要维护 Python 3.8 或更早版本的 Python 代码, 则 “PEP 584-Add Union Operators To dict” 的 “Motivation” 一节, 总结了合并映射的其他方法。

3.3 用模式匹配处理映射

match/case 语句的匹配对象 (Subject) 可以是映射。映射的模式 (Pattern) 看起来像 dict 字面量, 但是它能匹配 `collections.abc.Mapping` 的任何实际子类或虚拟子类¹的实例。

在 “丰富的序列” 中, 只重点讲解了序列模式 (Pattern)。其实, 不同类型的模式 (Pattern) 可以组合和嵌套。借助析构 (Destructor), 模式匹配 (Pattern Match) 成为处理嵌套映射与序列等结构化记录的一种强大工具。我们经常需要从 JSON API 和具有半结构化 schema 的数据库 (如 MongoDB, EdgeDB 或 PostgreSQL) 中读取这类记录。示例如示例 3.6 所示, 函数 `get_creators` 有一些简单的类型注解 (Type Annotations), 作用是明确表明参数类型是 dict, 返回值类型是 list。

```
</> 示例 3.6: creator.py:get_creators() 函数从出版物记录中提取创作者姓名
```

```
1 def get_creators(record: dict) -> list:
2     match record:
3         case {'type': 'book', 'api': 2, 'authors': [*names]}: ❶
4             return names
5         case {'type': 'book', 'api': 1, 'author': name}: ❷
6             return [name]
7         case {'type': 'book'}: ❸
8             raise ValueError(f"Invalid 'book' record: {record!r}")
9         case {'type': 'movie', 'director': name}: ❹
10            return [name]
11         case _: ❺
12             raise ValueError(f'Invalid record: {record!r}'')
```

❶ 匹配含有`'type': 'book'`, `'api': 2`, 并且`'authors'` 键将被映射到一个序列对象, 以列表的形式返回序列中的项。即将序列中的项, 保存在列表 `names` 中。

❷ 匹配含有`'type': 'book'`, `'api': 1`, 并且`'author'` 键将被映射到任何对象, 以列表的形式返回匹配到的对象。

¹ 虚拟子类: 虚拟子类是调用抽象基类的 `register()` 方法注册的类, 详见 “13.5.6 抽象基类的虚拟子类”。

即将匹配的对象,保存在列表 name 中。

- ③ 其他含有'type': 'book' 的映射都是无效的,引发 ValueError。
- ④ 匹配含有'type': 'movie', 并且'director' 键将被映射到单个对象,以列表的形式返回匹配到的对象。即将匹配的对象,保存在列表 name 中。
- ⑤ 其他对象均无效,引发 ValueError。

示例 3.6 展示了处理半结构化数据(例如 JSON 记录)的一些有效实践:

- 包含一个描述记录类型的字段(如'type': 'movie')。
- 包含一个标识 schema 版本的字段(如'api': 2),以便于公开 API 的版本更迭。
- 包含一个处理特定无效记录(例如“book”)的 case 子句,以及兜底的 case 子句。

下面在 doctest 中测试一下 get_creators 函数(如示例 3.7 所示)。

```
</> 示例 3.7: 测试 creator.py:get_creators() 函数
1 >>> b1 = dict(api=1, author='Douglas Hofstadter',
2 ...     type='book', title='Gödel, Escher, Bach')
3 >>> get_creators(b1)
4 ['Douglas Hofstadter']
5 >>> from collections import OrderedDict
6 >>> b2 = OrderedDict(api=2, type='book',
7 ...     title='Python in a Nutshell',
8 ...     authors='Martelli Ravenscroft Holden'.split())
9 >>> get_creators(b2)
10 ['Martelli', 'Ravenscroft', 'Holden']
11 >>> get_creators({'type': 'book', 'pages': 770})
12 Traceback (most recent call last):
13 ...
14 ValueError: Invalid 'book' record: {'type': 'book', 'pages': 770}
15 >>> get_creators('Spam, spam, spam')
16 Traceback (most recent call last):
17 ...
18 ValueError: Invalid record: 'Spam, spam, spam'
```

注意,模式(Pattern)中键(Key)的顺序无关紧要,如示例 3.7 所示,即使 b2 是一个 OrderedDict,也可以作为映射模式(Pattern)的匹配对象(Subject)。

与“2.6 序列模式匹配”中的序列模式(Sequence Pattern)不同,映射模式(Mapping Pattern)只需部分匹配即可视作匹配成功。

倘若想将多出的“Key:Value”对捕获到单个 dict 中,可以在一个变量前加上**。但是,此变量必须放在模式(Pattern)最后(如示例 3.8 所示)。**_这种画蛇添足的写法是无效的。

```
</> 示例 3.8: 将多出的 Key:Value 对捕获到一个 dict 中
```

```
1 >>> food = dict(category='ice cream', flavor='vanilla', cost=199)
2 >>> match food:
3 ...     case {'category': 'ice cream', **details}: ❶ 将余下的 K:V 对捕获到字典
4 ...         details 中
5 ...             print(f'Ice cream details: {details}')
```

```

5 ...
6 Ice cream details: {'flavor': 'vanilla', 'cost': 199}

```

“3.5 缺失键的自动处理”中讲解的 collections.defaultdict 等映射,通过 `__getitem__` 检索键(即 `d[key]`²)始终都是成功的。这是因为对于 collections.defaultdict, `__getitem__` 找不到搜索的键时,会使用指定的默认值自动创建对应的项(详见“3.5 缺失键的自动处理”)。而对于模式匹配 (Pattern Match) 而言,仅当匹配对象 (Subject) 在运行 match 语句之前,已经含有所需的键 (Key),才能成功匹配。



模式匹配 (Pattern Match) 不会自动处理缺失的键 (Key), 因为模式匹配 (Pattern Match) 始终使用 `d.get(key, sentinel)` 的形式来检索匹配对象 (Subject) 中的项。其中, `sentinel` 是特殊的标记值, 不会出现在用户数据中。

讲完映射的句法与结构之后,接下来讲解映射的 API(详见“3.4 映射类型的标准 API”)。

3.4 映射类型的标准 API

collections.abc 模块提供的抽象基类 (ABCs) `Mapping` 和 `MutableMapping`,描述了 `dict` 与类似类型的接口(如图 3.1 所示)。

抽象基类 (ABCs) 的主要价值在于对映射的标准接口进行文档化和正式化,并在需要广义上的映射对象时,为 `isinstance` 提供测试标准(如示例 3.9 所示)。

```

</> 示例 3.9: isinstance 测试
1 1>>> my_dict = {}
2 2>>> isinstance(my_dict, abc.Mapping)
3 3> True
4 4>>> isinstance(my_dict, abc.MutableMapping)
5 5> True

```



将 `isinstance` 与 ABC 一起使用,通常比检查函数参数是否属于具体 (concrete) 的 `dict` 类型更好。因为这样就可以使用其他的映射类型。此问题将在“十三 接口、协议与抽象基类”中详细探讨。

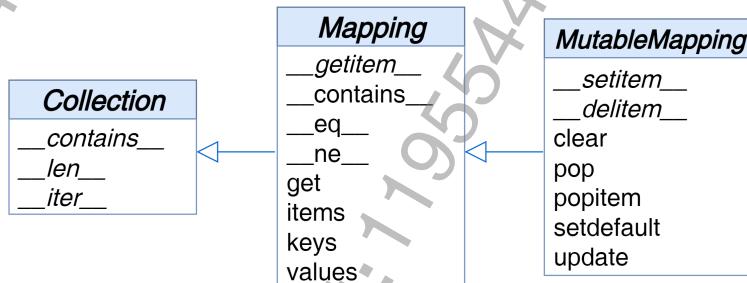


图 3.1: `MutableMapping` 及其在 `collections.abc` 中的超类的简化 UML 类图

²`d[key]` 句法的底层是通过 `__getitem__` 实现的。

如果要实现自定义的映射,可以通过扩展 `collections.UserDict` 类或使用组合方式包装 `dict`,而不是直接子类化这些 **抽象基类 (ABCs)**。在标准库中, `collections.UserDict` 类和所有具体的映射类,都在其实现中封装了基本的 `dict` 数据结构,而 `dict` 本身是建立在哈希表上的。因此,这些类都有一个共同的限制,即键 (Key) 必须是“**可哈希 (hashable)**”(值不必是可哈希)的。“3.4.1 什么是可哈希”会讲述关于“**可哈希 (hashable)**”的相关概念。

3.4.1 什么是可哈希

Python 术语表 中包含对“**可哈希**”的定义,摘录(部分有改动)如下:

如果一个对象在其生命周期内有一个永不改变的“**哈希码**^a”(依托 `__hash__()` 方法),并且可以与其他对象进行等值比较(依托 `__eq__()` 方法),那么这个对象就是“**可哈希 (hashable)**”的。相等的可哈希对象,必须拥有相同的哈希码。

^a哈希码:映射对象中的项由“键: 值”对构成。为了避免与映射对象中的“值”相混淆,本书用“哈希码”代替“哈希值”。

数值类型以及不可变的扁平类型(如 `str`、`bytes`)均是**可哈希 (hashable)**的。若一个不可变的容器类型中,其所含的对象也全是**可哈希 (hashable)**的,那么容器类型自身也是**可哈希 (hashable)**的。`frozenset` 对象总是**可哈希 (hashable)**的(如示例 3.10 中的 `tf`),因为根据定义, `frozenset` 包含的每个元素都必须是**可哈希 (hashable)**的。而在 `tuple` 中,仅当其所有的项都是**可哈希 (hashable)**的, `tuple` 对象自身才是**可哈希 (hashable)**的(如示例 3.10 中的 `tt`、`tl`)。

</> 示例 3.10: 可哈希的 `frozenset` 与 `tuple`

```

1  >>> tt = (1, 2, (30, 40))          # 可哈希的 tuple
2  >>> hash(tt)
3  8027212646858338501
4  >>> tl = (1, 2, [30, 40])        # 不可哈希的 tuple
5  >>> hash(tl)
6  Traceback (most recent call last):
7  File "<stdin>", line 1, in <module>
8  TypeError: unhashable type: 'list'
9  >>> tf = (1, 2, frozenset([30, 40]))  # 总是可哈希的 frozenset
10 >>> hash(tf)
11 -4118419923444501110

```

根据 Python 版本、设备架构的不同,对象的哈希码可能会有所不同。另外,若出于安全考量,而在哈希计算过程中**加盐 (Salted)**³,也会导致哈希码发生变化。正确实现的对象,只能保证其哈希码在一个 Python 进程中保持不变。

用户自定义类型默认是**可哈希 (hashable)**的,因为其哈希码取自其 `id()`。并且,继承自 `object` 类的 `__eq__()` 方法,也只是比较对象的 ID。若对象实现了考虑其内部状态的自定义 `__eq__()` 方法,则只有当它的 `__hash__()` 总返回同一个哈希码时,对象才是**可哈希 (hashable)**的。在实践中,这要求 `__eq__()` 与 `__hash__()` 仅考虑那些在对象生命周期中,永远不会改变的实例属性。

现在,让我们来回顾一下 Python 中最常用的映射类型(即 `dict`、`defaultdict` 和 `OrderedDict`)的 API。

³一些安全隐患与可采用的应对方案,参见:PEP 466 Secure and interchangeable hash algorithm。

3.4.2 常用映射方法概述

映射类型的基本 API 非常丰富。显示了 dict 及其 2 个受欢迎的变体 (defaultdict 和 OrderedDict) 实现的方法。defaultdict 与 OrderedDict 都定义在 collections 模块中。

表 3.1: 映射类型 (dict, collections.defaultdict, collections.OrderedDict) 实现的方法⁵

方法	dict	defaultdict	OrderedDict	
d.clear()	•	•	•	删除所有项
d.__contains__(k)	•	•	•	k in d
d.copy()	•	•	•	浅 (Shallow) 拷贝
d.__copy__()		•		为 copy.copy(d) 提供支持
d.default_factory ¹		•		由 __missing__ 调用的可调用函数, 用于设置缺失值。
d.__delitem__(k)	•	•	•	del d[k]
d.fromkeys(it, [initial])	•	•	•	用可迭代对象 it 中的键, 创建新的映射。可以为每个项指定一个可选的初始值 (默认为 None)。
d.get(k, [default])	•	•	•	通过键 (k) 获取映射中的项。若键 (k) 缺失, 则返回默认值或 None。
d.__getitem__(k)	•	•	•	d[k], 通过键 (k) 获取映射中的项。
d.items()	•	•	•	获取映射中所有项的键值 (Key:Value) 对。
d.__iter__()	•	•	•	获取键 (Key) 的迭代器。
d.keys()	•	•	•	获取映射中的所有键 (Key)
d.__len__()	•	•	•	len(d), 获取映射中项的数量。
d.__missing__(k)		•		当 __getitem__ 找不到键 (Key) 时, 会调用此方法。
d.move_to_end(k, [last])			•	将 k 移动到 d 的开头或末尾 (last 默认为 True)。
d.__or__(other)	•	•	•	为 d1 d2 提供支持, 用于合并 d1, d2 (Python 3.9)。
d.__ior__(other)	•	•	•	为 d1 = d2 提供支持, 以就地将 d2 合并到 d1 中 (Python 3.9)。
d.pop(k, [default])	•	•	•	如果 k 存在, 则移除并返回对应的值; 如果 k 不存在, 则返回默认值或 None (Python 3.9)。
d.popitem() ²	•	•	•	移除并返回最后插入的项, 以 (key, value) 的形式返回。
d.__reversed__()	•	•	•	为 reverse(d) 提供支持, 按插入顺序反向返回所有键 (Key) 的迭代器。
d.__ror__(other) ³	•	•	•	为 other dd 提供支持, 反向合并运算符 (Python 3.9)。
d.setdefault(k, [default])	•	•	•	若 k 在字典 d 中, 则返回 d[k]; 否则设置 d[k] = default 并返回它。
d.__setitem__(k, v)	•	•	•	d[k] = v, 将键 k 对应的值, 设置为 v。
d.update(m, [**kwargs])	•	•	•	用映射或可迭代对象中的“键: 值”对, 来更新映射 d。
d.values()	•	•	•	获取映射 d 中所有项的所有值。

d.update(m) 用 **鸭子类型** (Duck Typing) 处理参数 m (详见“12.4 协议与鸭子类型”): 首先, 检查 m 是否有 keys() 方法。若有, 则假定 m 是映射; 否则, update() 方法会退而求其次, 开始迭代参数 m, 并假定 m 中

¹d.default_factory 不是方法, 而是终端用户在实例化 defaultdict 时设置的一个可调用属性。

²OrderedDict.popitem(last=False) 会移除第一个插入的项 (先进先出)。在 Python 3.10b3 中, dict 或 defaultdict 不支持 last 关键字参数。

³反向运算符, 详见“十六 运算符重载”

的所有项都是键值对 ((Key, Value))。多数 Python 映射的构造函数,其内部逻辑都与 update() 方法一致。言外之意,构造函数可以从其他映射或可产生键值对的可迭代对象中初始化(构造)新的映射对象。

一个微妙的映射方法是 setdefault()。当需要就地更新映射中某个项的值时, setdefault() 可以避免键的查找。详见“3.4.3 插入或更新可变的值”。

3.4.3 插入或更新可变的值

根据 Python 的 Fail-Fast (快速失败) 原则,当使用 `d[k]` 访问 `dict` 中不存在的键 `k` 时,Python 会立即引发错误。Python 程序员知道,当获取默认值比处理 `KeyError` 更方便时, `d.get(k, default)` 可以是 `d[k]` 的替代方案。但是,在希望获取一个可变值并对其进行更新时,还有一种更好的方式(详见示例 3.13 中的 `setdefault`)。

考虑一个用于索引文本的脚本,生成一个映射。其中,每个键是文本中的单词,而键对应的值是该单词出现的位置列表,如示例 3.11 所示⁴。

</> 示例 3.11: 用示例 3.12 的脚本,处理《Python 之禅》的部分输出

```
1 $ python3 index0.py zen.txt
2 a [(19, 48), (20, 53)]
3 Although [(11, 1), (16, 1), (18, 1)]
4 ambiguity [(14, 16)]
5 and [(15, 23)]
6 are [(21, 12)]
7 aren [(10, 15)]
8 at [(16, 38)]
9 bad [(19, 50)]
10 be [(15, 14), (16, 27), (20, 50)]
11 beats [(11, 23)]
12 Beautiful [(3, 1)]
13 better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11), (17, 8), (18, 25)]
14 ...
```

示例 3.12 中的脚本不太完美,目的只是提醒您 `dict.get` 并非处理缺失键的最佳方式。此脚本⁵ 是根据 Alex Martelli 的一个示例修改而来。

</> 示例 3.12: index0.py: 用 `dict.get` 获取并更新单词出现的位置列表

```
1 """构建一个索引, 将单词映射到出现位置的列表。"""
2 import re
3 import sys
4
5 WORD_RE = re.compile(r'\w+')
6
7 index = {}
8 with open(sys.argv[1], encoding='utf-8') as fp:
9     for line_no, line in enumerate(fp, 1):
10         for match in WORD_RE.finditer(line):
```

⁴ 每行对应一个单词及其出现的位置列表,每个位置的格式为:(行号,列号)。

⁵ 原脚本出自 Martelli 的演讲“Re-learning Python”的第 41 张幻灯片。此脚本其实是演示 `dict.setdefault`,如示例 3.13 所示。

```

11     word = match.group()
12     column_no = match.start() + 1
13     # 本代码不完美, 仅用作演示。
14     occurrences = index.get(word, []) ①
15     occurrences.append(location) ②
16     index[word] = occurrences ③
17
18 # 按字母顺序显示索引
19 for word in sorted(index, key=str.upper): ④
20     print(word, index[word])

```

- ① 获取单词出现的位置列表,若未找到,则返回 []。
- ② 将新找到的位置,追加到 occurrences 中。
- ③ 将更新后的 occurrences,放入字典 index 中。此处的 index[word] 隐含一次对字典 index 的搜索操作。
- ④ sorted 函数的参数 key,并不是调用 str.upper 方法,只是传入了这个方法的引用,以便 sorted 使用它来排序各个单词,并规范化输出⁶。

示例 3.12 中,处理 occurrences 用了 3 行代码。若换成 dict.setdefault 则只需 1 行代码。示例 3.13 更接近于 Alex Martelli 的代码。

</> 示例 3.13: 用 dict.setdefault 获取字典的值,并更新。

```

1 """构建一个索引, 将单词映射到出现位置的列表。"""
2
3 import re
4 import sys
5
6 WORD_RE = re.compile(r'\w+')
7
8 index = {}
9 with open(sys.argv[1], encoding='utf-8') as fp:
10     for line_no, line in enumerate(fp, 1):
11         for match in WORD_RE.finditer(line):
12             word = match.group()
13             column_no = match.start() + 1
14             location = (line_no, column_no)
15             index.setdefault(word, []).append(location) ①
16
17 # 按字母顺序显示索引
18 for word in sorted(index, key=str.upper):
19     print(word, index[word])

```

- ① 获取单词出现的位置列表。若未找到,则设为空列表 [];setdefault 会返回该列表,可以直接更新它,而无需再次搜索。

换句话说,示例 3.4.3 中的第 1 行与第 2~4 行的作用一样。

```
1 my_dict.setdefault(key, []).append(new_value) ①
```

⁶Python 中将方法/函数当作一等对象来使用,详见“[函数是一等对象](#)”。

```
2
3 if key not in my_dict:
4     my_dict[key] = []
5 my_dict[key].append(new_value)
```

但是,后一种写法至少要在字典中搜索 2 次 Key,若未找到则搜索 3 次。而 setdefault 只需搜索 1 次。“3.5 缺失键的自动处理”将探讨与缺失键相关的话题:即如何处理任何查找(而不仅仅是插入)中的缺失键。

3.5 缺失键的自动处理

有时搜索的键 (Key) 不一定存在,为了以防万一,可以人为设置一个值 (Value),以便于某些情况的处理。人为设置值 (Value) 主要包括 2 种方案:

- 一种是用 defaultdict 代替普通的 dict;
- 另一种,是定义 dict 的子类或其他映射类型的子类,并实现 __missing__ 方法。

下面将介绍这 2 种方案。

3.5.1 defaultdict: 处理缺失键的另一种选择

当 d[k] 句法找不到搜索的键 k 时,collections.defaultdict 实例会按需创建具有默认值的项。[示例 3.14](#) 使用 defaultdict 类,以更优雅的方式重新实现[示例 3.13](#)中的单词索引编制任务。

[示例 3.14](#)的实现原理是:实例化 defaultdict 对象时,提供一个可调用对象。当 __getitem__ 遇到不存在的键时,会调用这个可调用对象,以便生成一个默认值。举个例子,假设用 dd = defaultdict(list) 创建一个 defaultdict 对象。若 dd 中没有 new-key 键,则 dd['new-key'] 表达式将执行以下步骤:

1. 调用 list() 创建一个新列表 []。
2. 将列表 [] 插入 dd,作为键 'new-key' 的值。即向 dd 中插入项(键值对为: 'new-key': [])
3. 返回列表 [] 的引用。

用于生成默认值的可调用对象,存放在实例属性 default_factory 中。

</> [示例 3.14: index_default.py: 用 defaultdict 类代替 setdefault 方法](#)

```
1 """构建一个索引, 将单词映射到出现位置的列表。"""
2 import collections
3 import re
4 import sys
5
6 WORD_RE = re.compile(r'\w+')
7
8 index = collections.defaultdict(list) ①
9 with open(sys.argv[1], encoding='utf-8') as fp:
10     for line_no, line in enumerate(fp, 1):
11         for match in WORD_RE.finditer(line):
12             word = match.group()
13             column_no = match.start() + 1
14             location = (line_no, column_no)
```

```

15     index[word].append(location) ②
16
17 # 按字母顺序显示索引
18 for word in sorted(index, key=str.upper):
19     print(word, index[word])

```

- ❶ 创建一个 `collections.defaultdict` 对象, 将实例属性 `default_factory` 设置为可调用对象 `list`。
 ❷ 若单词 `word` 最初不在 `index` 中, 将调用 `default_factory` 来生成缺失值。本例则是用 `list()` 生成一个空列表 `[]`, 然后分配给 `index[word]` 并返回。因此, `append(location)` 操作, 始终都是成功的。
 若未提供 `default_factory`, 则会因缺少键而引发常见的 `KeyError` 异常。



`collections.defaultdict` 的 `default_factory` 属性, 仅用于为 `__getitem__` 提供默认值。例如, 若 `dd` 是一个 `collections.defaultdict` 对象, 若该对象中不含键 `k`, 则 `dd[k]` 将调用 `default_factory` 创建默认值。但是, `dd.get(k)` 依然返回 `None`, 而且 `k in dd` 也返回 `False`。

通过调用 `default_factory` 使 `defaultdict` 工作的机制是 `__missing__` 特殊方法, 也是接下来将讨论的一个特性。

3.5.2 `__missing__` 方法

映射对象处理缺失键的底层逻辑是在特殊方法 `__missing__` 中实现的。`dict` 基类本身未定义这个方法, 但是 `dict` 知道此方法。若您继承了 `dict` 类, 并提供了特殊方法 `__missing__`, 那么 `dict.__getitem__` 在找不到键时, 将会调用 `__missing__` 方法, 而不是引发 `KeyError` 异常。

假设你想要一个映射, 在查询时将键转换为 `str` 字符串。一个具体的案例是物联网设备代码库⁷。该代码库用 `Board` 类表示可编程开发板 (如 Raspberry Pi 或 Arduino); 用 `Board.pins` 属性表示开发板上的通用 I/O 引脚, 该属性是一个 I/O 引脚标识符到引脚对象的映射。I/O 引脚标识符可以是纯数值, 或者也可以是类似 “AO” 或 “P9_12”的字符串。为了保持一致, 可能希望 `Board.pins` 中的所有键都是 `str` 字符串。但有时也想通过数值来查找引脚 (如 `my_board.pins[13]`), 以便于初学者通过数字点亮开发板上 13 号引脚的 LED。[示例 3.15](#) 展示了这个映射的工作方式。

</> 示例 3.15: 搜索非字符串键时, `StrKeyDict0` 将未找到的键转换为字符串

```

1  >>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
2  >>> d['2']          # 测试用 d[key] 表示法, 检索映射中的项
3  'two'
4
5  >>> d[4]
6  'four'
7  >>> d[1]
8  ...
9  KeyError: '1'
10

```

⁷ 例如, 已停止开发的 Pingo.io 库: <https://github.com/pingo-io/pingo-py>。

```

11  >>> d.get('2')          # 测试用 d.get(key) 表示法, 检索映射中的项
12  'two'
13  >>> d.get(4)
14  'four'
15  >>> d.get(1, 'N/A')
16  'N/A'
17
18  >>> 2 in d            # 测试 in 运算符
19  True
20  >>> 1 in d
21  False

```

示例 3.16 实现了可通过以上测试(示例 3.15)的 StrKeyDict0 类。



继承 collections.UserDict 类(而不是 dict 类), 是创建用户自定义映射类型的最佳方式(如“3.6.5 子类应继承 UserDict 而不是 dict”中的示例 3.21)。示例 3.16 中 StrKeyDict0 继承 dict 类, 只是为了说明内置的 dict.__getitem__ 方法支持 __missing__。

</>示例 3.16: StrKeyDict0 在查找键时,将非字符串键转换为字符串。

```

1  class StrKeyDict0(dict):
2
3      def __missing__(self, key):
4          if isinstance(key, str):
5              raise KeyError(key)
6          return self[str(key)]
7
8      def get(self, key, default=None):
9          try:
10              return self[key]
11          except KeyError:
12              return default
13
14      def __contains__(self, key):
15          return key in self.keys() or str(key) in self.keys()

```

- ❶ StrKeyDict0 继承 dict。
- ❷ 检查 key 是否是 str 字符串类型。若是, 并且找不到此键, 则引发 KeyError 异常。
- ❸ 将 key 转换为 str 字符串类型, 并查找该字符串键。
- ❹ get 方法通过使用 self[key], 将键搜索委托给 __getitem__, 以让 __missing__ 发挥作用。
- ❺ 若引发 KeyError 异常, 说明 __missing__ 也找不到键, 因此返回 default。
- ❻ 先搜索未经转换的键(实例可能包含非 str 字符串键), 再搜索字符串形式的键。

此处, 请思考: 为何实现特殊方法 __missing__ 时, 要做 isinstance(key, str) 测试?

若不对此测试, 则 __missing__ 方法可处理任何类型的键, 即键是否是 str 类型都无所谓。但前提是 str(key) 得到的必须是已存在的键。若 str(key) 不是已存在的键, 将会陷入无限递归: 即 __missing__ 方法的

最后一行 `self[str(key)]`, 将调用 `__getitem__(str(key))` 方法, 转而又调用 `__missing__` 方法。

在此示例中, 为了一致性的行为, 还需要 `__contains__` 方法。因为 `k in d` 操作会调用 `__contains__` 方法, 但从 `dict` 继承的方法不会回退到调用 `__missing__` 方法。在我们实现的 `__contains__` 方法中, 有一个微妙的细节: 我们并未按常规的 Pythonic 风格 (`k in my_dict`) 检查映射中的键。因为 `str(key) in self` 将递归调用 `__contains__` 方法。为了避免此问题, 我们显式地在 `self.keys()` 中查找键。

在 Python 3 中, 即使对于非常大的映射, 像 `k in my_dict.keys()` 这样的键查找也是高效的, 因为 `dict.keys()` 返回的是一个类似于集合的视图 (详见 “3.12 字典视图的集合运算”)。尽管如此, `k in my_dict` 也可以做同样的工作, 并且更快, 因为它避免了通过属性查找来找到 `.keys` 方法。

在示例 3.16 中, `__contains__` 方法使用 `self.keys()` 还有一个具体的原因。即为了保证结果正确, 必须使用 `key in self.keys()` 检查未经修改的键, 因为 `StrKeyDict0` 不会强制要求字典中的所有键都是 `str` 类型。这个简单的示例只有一个目标, 即让搜索 “更友好”, 而不是强制键的类型。



用户自定义的类, 若继承了标准库中的映射, 在实现 `__getitem__`、`get` 或 `__contains__` 方法时, 不一定要回落到 `__missing__` 方法。具体原因见 “3.5.3 标准库中 `__missing__` 的不一致用法”。

3.5.3 标准库中 `__missing__` 的不一致用法

考虑如下不同场景, 以及各场景查找缺失键的不同行为:

- `dict` 子类

`dict` 子类若仅实现了 `__missing__` 方法, 而未实现其他方法。在此情况下, 可能只有 `d[k]` 会调用 `__missing__` 方法。而 `d[k]` 底层是由继承自 `dict` 类的 `__getitem__` 方法实现。

- `collections.UserDict` 子类

同样, 若 `UserDict` 子类仅实现了 `__missing__` 方法, 而未实现其他方法。继承自 `UserDict` 的 `get` 方法, 会调用 `__getitem__`。这意味着, `d[k]` 与 `d.get(k)` 在查找缺失键时, 会调用 `__missing__` 方法。

- `abc.Mapping` 子类, 以最简单的方式实现 `__getitem__` 方法。

定义一个精简的 `abc.Mapping` 子类。子类中实现了 `__missing__` 方法与必要的抽象方法。其中, `__getitem__` 方法不调用 `__missing__`。此子类永远不会触发 `__missing__` 方法。

- `abc.Mapping` 子类, 让 `__getitem__` 调用 `__missing__` 方法。

定义一个精简的 `abc.Mapping` 子类。子类中实现了 `__missing__` 方法与必要的抽象方法。其中, `__getitem__` 方法会调用 `__missing__`。此子类的 `d[k]`、`d.get(k)` 与 `k in d` 在查找缺失键时, 将触发 `__missing__` 方法。

请参考随书代码库中的 `missing.py` 文件, 以获得这些场景的演示。

以上描述的 4 种场景, 都假定为最小实现。若在自定义的子类中实现了 `__getitem__`、`get`、`__contains__` 方法, 则可以根据实际需求决定这些方法是否使用 `__missing__`。本节的要点是为了说明: 在继承标准库中的映射, 以使用 `__missing__` 方法时, 要小心谨慎。因为, 不同的基类对 `__missing__` 方法的使用方式是不同的。

不要忘记, `setdefault` 和 `update` 的行为也会受到键查找的影响。最后, 根据 `__missing__` 的逻辑, 您可能需要在 `__setitem__` 中实现特殊逻辑, 以避免不一致或令人惊讶的行为。相关的示例, 详见 “3.6.5 子类应继承 `UserDict` 而不是 `dict`” 中的示例 3.21。

至此,我们已介绍了 dict 与 defaultdict 两种映射类型。除此之外,标准库中还提供了其他的映射类型。下面将逐一讨论。

3.6 dict 变体

在本节中,将概述标准库中除 defaultdict(见3.5.1)之外的其他映射类型。

3.6.1 collections.OrderedDict

自 Python 3.6 起,内置的 dict 类型也会保留键的顺序。因此,使用 OrderedDict 的最常见原因是编写与早期 Python 版本兼容的代码。话虽如此,dict 与 OrderedDict 之间仍有一些细微差异。Python 文档中列出了这些差异,摘录如下(根据使用频率,顺序有所调整)。

- OrderedDict 的等值操作,会检查键的顺序是否匹配。
- OrderedDict 的 popitem() 方法具有不同的签名。它接受一个可选参数,用于指定弹出的项。
- OrderedDict 多出一个 move_to_end() 方法,可以高效地将元素移到某一段。
- 常规的 dict 主要用于执行映射操作,插入顺序是次要的。
- OrderedDict 的目的是方便重新排序,其空间效率、迭代速度与更新操作的性能是次要的。
- 从算法上讲,OrderedDict 处理频繁重排序的效果,要比 dict 更好。因此,OrderedDict 适合跟踪最近的访问(例如,LRU 缓存)。

3.6.2 collections.ChainMap

ChainMap 实例保存一系列可以作为一个整体进行搜索的映射。查找操作按每个输入映射在构造函数调用中出现的顺序执行,一旦在某个映射中找到指定的键,立即结束查找。例如:

</> 示例 3.17: 在 ChainMap 的多个映射中查找键

```

1 >>> d1 = dict(a=1, b=3)
2 >>> d2 = dict(a=2, b=4, c=6)
3 >>> from collections import ChainMap
4 >>> chain = ChainMap(d1, d2)           # 先查找映射 d1, 再查找映射 d2。
5 >>> chain['a']
6 1
7 >>> chain['c']
8 6

```

ChainMap 实例不会复制输入的映射,而是保存对它们的引用。对 ChainMap 的更新或插入操作,只会影响第一个输入映射。延续示例 3.17:

</> 示例 3.18: 更新 ChainMap

```

1 >>> chain['c'] = -1
2 >>> d1
3 {'a': 1, 'b': 3, 'c': -1}
4 >>> d2

```

```
5 { 'a': 2, 'b': 4, 'c': 6}
```

ChainMap 适用于实现具有嵌套作用域的语言解释器。ChainMap 中的每个映射都表示一个作用域上下文, 包含从最内层的封闭作用域到最外层作用域。collections 文档中的ChainMap objects一节, 列举了多个使用 ChainMap 的示例, 其中包括受 Python 变量查找规则启发而编写的代码段, 如下所示:

</> 示例 3.19: ChainMap 模拟 Python 变量查找规则

```
1 import builtins
2 pylookup = ChainMap(locals(), globals(), vars(builtins))
```

“18.3.4 环境”的示例 18.14 显示了一个 ChainMap 子类, 用于实现 Scheme 语言子集的解释器。

3.6.3 collections.Counter

collections.Counter 是一种用于对其中的键 (Key) 进行计数的映射类型。更新映射中的现有键, 键的计数会随之增加。collections.Counter 可用于统计可哈希 (hashable) 对象的实例数量, 或者作为 multiset (多重集, 本节稍后讨论) 使用。Counter 实现了用于合并计数的 + 和 - 运算符, 以及其他有用的方法, 如 most_common([n]) 将返回一个包含前 n 个计数最大的项及其计数的有序元组列表 (参见:Counter 文档)。

示例 3.20 中, 用 collections.Counter 统计单词中的字母数量。

</> 示例 3.20: 用 Counter 统计单词中的字母数量

```
1 >>> ct = collections.Counter('abracadabra')
2 >>> ct
3 Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
4 >>> ct.update('aaaaazzz')
5 >>> ct
6 Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
7 >>> ct.most_common(3)
8 [('a', 10), ('z', 3), ('b', 2)]
```

请注意, 'b' 与 'r' 两个键并列第三位, 但是 ct.most_common(3) 仅显示前 3 项。

若将 collections.Counter 用作 multiset (多重集), 则假定每个键都是集合 (set) 中的一个元素, 计数值则是元素在集合中出现的次数。

3.6.4 shelve.Shelf

标准库中的 shelve 模块, 为字符串键到 Python 对象 (以 pickle 二进制格式序列化的) 的映射, 提供了持久性存储。您可能觉得 shelve 这个名称有些奇怪, 但是当您意识到 pickle jars (泡菜坛) 是放在架子 (shelve) 上的, shelve 这个名字就显得合理了。

模块级函数 shelve.open 会返回一个 shelve.Shelf 实例——一个由 dbm 模块支持的简单键值 DBM 数据库。shelve.Shelf 具有以下特征:

- shelve.Shelf 是 collections.abc.MutableMapping 的子类, 所以它提供了我们期望的映射类型的基本方法。

- 此外, `shelve.Shelf` 提供了一些其他的 I/O 管理方法, 如 `sync` 和 `close`。
- `shelve.Shelf` 实例是一个上下文管理器。因此, 可以使用 `with` 块来确保 `Shelf` 实例在使用后, 会被自动关闭。
- 每当为键分配一个新值时, 键和值都会被保存。
- `shelve.Shelf` 中的键必须是字符串类型。
- `shelve.Shelf` 中的值必须是 `pickle` 模块可以序列化的对象。

`shelve`、`dbm` 和 `pickle` 模块的文档, 提供了更多详细信息和一些注意事项。



Python 的 `pickle` 在最简单的情况下很容易使用, 但有几个缺点。在采用涉及 `pickle` 的解决方案之前, 请阅读 Ned Batchelder 的《Pickle 的 9 个缺陷》一文。在此文中, Ned 提到了可以采用的其他序列化格式。

`OrderedDict`、`ChainMap`、`Counter` 和 `Shelf` 可以直接使用, 也可以通过创建子类来实现自定义功能。而 `UserDict` 则不同, 它被设计成仅作为一个基类来扩展使用, 我们需要创建其子类来定义具体的功能。

3.6.5 子类应继承 `UserDict` 而不是 `dict`

最好通过扩展 `collections.UserDict` (而不是 `dict`) 来创建新的映射类型。当尝试扩展“3.5.2 `_missing_` 方法”中的示例 3.16, 以确保添加到映射中的键都被存储为 `str` 字符串时, 可意识到这一点。

子类应继承 `UserDict` (而不是 `dict`) 的主要原因是: 内置的 `dict` 在实现时, 对某些方法进行了简化。这会迫使我们在继承 `dict` 时, 需要重写这些方法。而直接继承 `UserDict` 类, 则可以避免这些问题⁸。

请注意, `UserDict` 并未继承自 `dict`, 而是使用了组合: `UserDict` 类内部有一个 `dict` 实例 (名为 `data`), 用于保存实际的项 (“键: 值” 对)。与示例 3.16 (见“3.5.2 `_missing_` 方法”) 相比, 这可以避免在编写特殊方法 (如 `_setitem_`) 时, 出现不必要的递归, 并简化了 `_contains_` 方法的编码。

由于 `StrKeyDict` (示例 3.21) 继承了 `UserDict`, 因此其实现过程比 `StrKeyDict0` (见“3.5.2 `_missing_` 方法”中的示例 3.16) 更简洁, 而且功能更丰富: `StrKeyDict` 将所有的键都存储为 `str` 字符串, 避免了在用包含非字符串键的数据构建或更新实例时, 不会出现意外。

</> 示例 3.21: `StrKeyDict` 在插入、更新和查找时, 始终将非字符串键转换为 `str` 字符串。

```

1 import collections
2
3 class StrKeyDict(collections.UserDict):
4
5     def __missing__(self, key): ❶
6         if isinstance(key, str):
7             raise KeyError(key)
8         return self[str(key)]
9
10    def __contains__(self, key):
11        return str(key) in self.data ❸
12

```

⁸子类继承 `dict` 与其他内置类型的具体问题, 将在“14.3 子类化内置类型很棘手”中讨论。

```
13     def __setitem__(self, key, item):
14         self.data[str(key)] = item ④
```

- ❶ StrKeyDict 扩展（继承）UserDict。
- ❷ __missing__ 与 [示例 3.16](#)（见“[3.5.2 __missing__ 方法](#)”）完全相同。
- ❸ __contains__ 更简单：可以假定存储的所有键都是 str，并且可以只检查 self.data 即可，而不用像 StrKeyDict0() 那样调用 self.keys()。
- ❹ __setitem__ 将 key 转换为 str 类型。委托给 self.data 属性后，此种方法更容易被覆盖（重写）。

由于 UserDict 扩展了 abc.MutableMapping，因此使 StrKeyDict 成为功能完整的映射的其余方法都是从 UserDict、MutableMapping 或 Mapping 继承的。尽管 MutableMapping 与 Mapping 是抽象基类（ABCs），但是也有一些有用的具体方法。其中，以下方法值得关注：

- MutableMapping.update

这个强大的方法可以直接调用，但 __init__ 也可以使用它从其他映射、（键、值）对可迭代对象和关键字参数中加载实例。因为它用 self[key] = value 语法向映射添加项，所以它最终会调用子类实现的 __setitem__。

- Mapping.get

在 StrKeyDict0 中（见“[3.5.2 __missing__ 方法](#)”中的 [示例 3.16](#)），我们必须自己实现 get 方法，以返回与 __getitem__ 相同的结果。但在 [示例 3.21](#) 中，我们继承了 Mapping.get，其实现与 StrKeyDict0.get 完全相同（请参阅 [Python 源代码](#)）。



Antoine Pitrou 撰写了 [PEP 455 –Adding a key-transforming dictionary to collections](#)，并提交了一个补丁以使用 TransformDict 增强 collections 模块。TransformDict 比 StrKeyDict 更通用，可在应用转换之前保留所提供的键。PEP 455 于 2015 年 5 月被拒绝（参见 Raymond Hettinger 的 [rejection message](#)）。为了试验 TransformDict，我将 Pitrou 的补丁从 [issues18986](#) 提取到一个独立模块中（参见 [随书代码库](#) 的 `03-dict-set/transformdict.py`）。

我们知道 Python 中有不可变序列类型，那么是否有不可变映射类型呢？其实，标准库中并没有真正的不可变映射类型，但有一个替代方案可用（见“[3.7 不可变映射](#)”）。

3.7 不可变映射

Python 标准库中提供的映射类型都是可变的，但有时也需要防止用户意外更改映射。“[3.5.2 __missing__ 方法](#)”提到的 [硬件设备库 Pingo](#) 中有个具体的用例：board.pins 映射表示设备上的物理 GPIO 引脚，因此需要防止被人无意修改。毕竟软件不能更改硬件，否则 board.pins 表示的 GPIO 引脚会与设备上的物理引脚不一致。

types 模块提供了一个包装类 MappingProxyType，它接受一个映射对象，并返回一个 mappingproxy 实例。该实例是原始映射对象的只读动态代理。这意味着，对原始映射对象的更新将体现在 mappingproxy 实例上，但是不能通过 mappingproxy 实例更改映射对象。请参阅 [示例 3.22](#) 对 MappingProxyType 类的简要演示。

</> 示例 3.22: MappingProxyType 根据 dict 对象构建只读的 mappingproxy 实例

```

1  >>> from types import MappingProxyType
2  >>> d = {1: 'A'}
3  >>> d_proxy = MappingProxyType(d)      ❶
4  >>> d_proxy
5  mappingproxy({1: 'A'})
6  >>> d_proxy[1]           ❷
7  'A'
8  >>> d_proxy[2] = 'x'      ❸
9  Traceback (most recent call last):
10 File "<stdin>", line 1, in <module>
11 TypeError: 'mappingproxy' object does not support item assignment
12 >>> d[2] = 'B'
13 >>> d_proxy           ❹
14 mappingproxy({1: 'A', 2: 'B'})
15 >>> d_proxy[2]
16 'B'
17 >>>

```

- ❶ 通过 MappingProxyType 创建字典对象 d 的只读代理(即 d_proxy)。
- ❷ 通过 d_proxy 可以查看字典对象 d 中的项。
- ❸ 无法通过 d_proxy 更改字典对象。
- ❹ d_proxy 是动态的,可反映 d 的变化。

以下是在硬件编程场景中的实际应用:在 Broad 的具体子类 (Concrete Subclass) 中,构造函数将引脚对象存储到一个私有映射中。通过 MappingProxyType 将此私有映射实现为一个公开的.pin 属性。将实现为 mappingproxy 实例的.pin 属性暴露给 API 客户端使用。这样,客户端就无法意外地添加、删除或更改引脚。

接下来,我们将介绍视图。通过视图可对 dict 执行一些高性能操作,而无需进行不必要的数据复制。

3.8 字典视图

dict 实例方法.keys()、.values() 和.items() 分别返回名为 dict_keys、dict_values 和 dict_items 的类的实例。这些字典视图是 dict 实现中使用的内部数据结构的只读投影。它们避免了 Python 2 中等效方法的内存开销。Python 2 中的等效方法返回的是重复了 (duplicating) 目标字典中已有数据的列表。此外,Python 3 中的字典视图,还取代了在 Python 2 中返回迭代器的旧方法。

示例 3.23 展示了所有字典视图支持的一些基本操作。

</> 示例 3.23: .values() 方法返回 dict 对象的值视图

```

1  >>> d = dict(a=10, b=20, c=30)
2  >>> values = d.values()
3  >>> values
4  dict_values([10, 20, 30])      ❶
5  >>> len(values)             ❷
6  3
7  >>> list(values)           ❸

```

```

8 [10, 20, 30]
9 >>> reversed(values)      ④
10 <dict_reversevalueiterator object at 0x10e9e7310>
11 >>> values[0]           ⑤
12 Traceback (most recent call last):
13 File "<stdin>", line 1, in <module>
14 TypeError: 'dict_values' object is not subscriptable

```

- ① 通过字典视图 (Dictionary View) 对象的字符串形式,查看字典视图的内容。
- ② 查询字典视图 (Dictionary View) 的长度。
- ③ 字典视图 (Dictionary View) 是一个可迭代对象,方便构建列表。
- ④ 字典视图 (Dictionary View) 实现了 `__reversed__` 方法,返回一个自定义迭代器。
- ⑤ 不能用 `[]` 运算符获取视图中的项。

字典视图 (Dictionary View) 对象是一个动态代理。如果源字典被更新了,你可以通过现有的视图对象立即看到这些变化。继续示例 3.23。

</> 示例 3.24: 字典视图可反映字典的变化

```

1 >>> d['z'] = 99
2 >>> d
3 {'a': 10, 'b': 20, 'c': 30, 'z': 99}
4 >>> values
5 dict_values([10, 20, 30, 99])

```

`dict_keys`、`dict_values` 和 `dict_items` 是内部类,不能通过 `__builtins__` 或标准库中的任何模块获取。尽管可以得到实例,但是在 Python 代码中不能自己手动创建它们的实例。

</> 示例 3.25: 不能手动创建字典视图的实例

```

1 >>> values_class = type({}.values())
2 >>> v = values_class()
3 Traceback (most recent call last):
4 File "<stdin>", line 1, in <module>
5 TypeError: cannot create 'dict_values' instances

```

`dict_values` 类是最简单的字典视图 (Dictionary View),它只实现了 `__len__`、`__iter__` 和 `__reversed__` 特殊方法。除此之外,`dict_keys` 和 `dict_items` 还实现了多个集合方法,基本与 `frozenset` 类相当。在介绍完集合之后,再进一步讨论 `dict_keys` 和 `dict_items`,详见“3.12 字典视图的集合运算”。

现在,让我们看一下 `dict` 底层实现方式所告知的一些规则和技巧。

3.9 dict 实现方式对实践的影响

Python 用哈希表实现 `dict`,因此字典效率非常高。理解这个设计对实践的影响非常重要:

- 字典的键 (Key) 必须是可哈希 (hashable) 的对象。如“3.4.1 什么是可哈希”所说,必须正确实现 `__hash__` 与 `__eq__` 方法。

- 通过键访问项的速度非常快。对于一个包含数百万个键的 dict 对象, Python 可通过计算键的哈希码, 并从哈希表中派生一个索引偏移量, 直接定位到键的位置。可能还需要尝试几次才能找到匹配的条目, 这可能带来一点开销。
- CPython 3.6 中, dict 的内存布局更为紧凑, 顺带的副作用是: 键的顺序可以得以保留。Python 3.7 开始, dict 正式支持保留键的顺序。
- 尽管 dict 采用了新的紧凑布局, 但不可避免地仍会有明显的内存开销。对于容器来说, 最紧凑的内部数据结构是指向项的指针数组⁹。与此相比, 哈希表需要为每个条目存储更多的数据, 并且 Python 需要将至少 $\frac{1}{3}$ 的哈希表行留空, 以保证高效。
- 为了节省内存, 应避免在 `__init__` 方法之外创建实例属性。

关于实例属性的最后一个技巧, 来自于 Python 的默认行为: Python 将实例属性存储在一个特殊的 `__dict__` 属性中, 此属性是依附到每个实例¹⁰的一个 dict。自从 Python 3.3 实现了 “PEP 412 -Key-Sharing Dictionary” 之后, 类的实例可以共用一个随类一起存储的公共哈希表。若每个新实例都与 `__init__` 返回的首个类实例拥有相同的属性名称, 那么这些新实例的 `__dict__` 属性将共享一个公共哈希表。每个新实例的 `__dict__` 属性, 仅以指针数组的形式存储新实例的属性值。在 `__init__` 之后添加实例属性, 会强制 Python 仅为该实例的 `__dict__` 创建一个新的哈希表¹¹。根据 PEP 412, 这种优化可以将面向对象程序的内存使用量减少 10% ~ 20%。

关于紧凑布局和键共享优化的细节相当复杂。要了解更多信息, 请阅读本书网站 fluentpython.com 的 “Internals of sets and dicts”。

现在, 让我们深入了解集合 (set)。

3.10 集合论

`set` (集合) 并非 Python 中的新鲜事务, 但仍然未得到充分利用。`set` 类型及其不可变形式的 `frozenset`, 最初作为模块出现在 Python 2.3 标准库中, 并在 Python 2.6 中被提升为内置类型。



本书使用“集合”一词指代 `set` 与 `frozenset`。若讨论内容仅涉及 `set` 类, 则使用等宽字体, 写作 `set`。

`set` 是一组唯一的对象。`set` 的基本作用是去除重复项。

</> 示例 3.26: `set` (集合) 的作用

```

1  >>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
2  >>> set(l)
3  {'eggs', 'spam', 'bacon'}
4  >>> list(set(l))
5  ['eggs', 'spam', 'bacon']

```

⁹元组就是这样存储的。

¹⁰除非类中定义了 `__slots__` 属性, 详见 “[11.11 用 __slots__ 节省内存](#)”。

¹¹为每个实例单独创建新的哈希表, 是 Python 3.3 之前所有实例的默认行为。



若既想去除重复项, 又想保留每个项首次出现的位置顺序。则使用普通的 dict 即可实现, 如下所示:

</> 示例 3.27: 用 dict 去除重复项, 并保留每个项首次出现的位置顺序。

```

1 >>> dict.fromkeys(l).keys()
2 dict_keys(['spam', 'eggs', 'bacon'])
3 >>> list(dict.fromkeys(l).keys())
4 ['spam', 'eggs', 'bacon']

```

set(集合) 中的元素必须是可哈希 (hashable) 的。set 类型是不可哈希 (hashable) 的, 因此不能用嵌套的 set 实例来构建 set 对象。但是, frozenset 是可哈希 (hashable) 的, 所以 set 对象中可以包含 frozenset 类型的元素。

除了强制唯一性之外, set 类型还以中缀运算符的形式实现了许多集合运算。因此, 给定集合 a 与 b, a|b 可计算二者的并集; a&b 可计算二者的交集; a-b 可计算二者的差集; a^b 可计算二者的对称差集。巧妙地使用集合运算, 可以减少 Python 程序的行数和执行时间。同时, 通过移除一些循环和条件逻辑, 还可以使代码更易于阅读和理解。

举个例子, 假设你有一个包含大量 E-Mail 的集合 (haystack), 还有一个包含较少 E-Mail 的集合 (needles)。您想统计 needles 中有多少 E-Mail 出现在 haystack 中。借助集合的交集运算符 (&), 只需用一行代码即可实现此需求 (如 [示例 3.28](#) 所示)。

</> 示例 3.28: 统计 needles 中 E-Mail 在 haystack 中出现的次数 (二者均为集合类型)。

```
found = len(needles & haystack)      # 借助 & 交集运算符,
```

若没有交集运算符 (&), 您必须编写更多的代码 (如 [示例 3.29](#)), 才能完成与 [示例 3.28](#) 相同的任务。

</> 示例 3.29: 统计 needles 中 E-Mail 在 haystack 中出现的次数 (效果与 [示例 3.28](#) 一致)

```

1 found = 0
2 for n in needles:
3     if n in haystack:
4         found += 1

```

[示例 3.28](#) 的运行速度比 [示例 3.29](#) 稍快。不过, [示例 3.29](#) 处理的 needles 与 haystack 可以是任何可迭代对象, 而 [示例 3.28](#) 要求二者均为 set(集合)。然而, 即使一开始二者不是 set(集合), 也可以随时在代码中快速构建它们 (如 [示例 3.30](#) 所示)。

</> 示例 3.30: 统计 needles 中 E-Mail 在 haystack 中出现的次数 (支持任何可迭代类型)

```

1 found = len(set(needles) & set(haystack))
2 # 另一种方式:
3 found = len(set(needles).intersection(haystack))

```

当然, 像 [示例 3.30](#) 那样构建 set(集合), 会有一定的额外开销。但是, 若 needles 或 haystack 中的任何一个已经是 set(集合)。那么, [示例 3.30](#) 中的第二种方式可能比 [示例 3.29](#) 开销更小。

上述任何一个示例都能在大约 0.3 毫秒内, 在由 10,000,000 个项组成的 haystack 中, 搜索出 1,000 个元

素。即每个元素的搜索时间接近 0.3 微秒。

除了极快的成员资格测试（得益于底层哈希表）之外，内置类型 `set` 和 `frozenset` 还提供了丰富的 API，用于创建新集合或更改现有集合。在讨论这些操作之前，先说明一下语法。

3.10.1 set 字面量

`set`（集合）字面量的语法，看起来几乎与集合的数学表示法几乎相同，例如 `1, 1, 2` 等。唯有一点例外：没有用于表示空集的字面量符号，必须将空集写作 `set()`。



语法陷阱

务必记住，要创建一个空集合（`set`），应该使用不带参数的构造函数，即 `set()`。倘若写成，则将创建一个空 `dict`——这一点在 Python 3 中没有改变。

在 Python 3 中，集合的标准字符串表示形式始终使用... 表示法（空集合除外）：

```

1  >>> s = {1}
2  >>> type(s)
3  <class 'set'>
4  >>> s
5  {1}
6  >>> s.pop()
7  1
8  >>> s
9  set()

```

像 `1, 2, 3` 这样的 `set` 字面量语法，比调用构造函数（例如 `set([1,2,3])`）更快、更易读。后一种形式速度较慢，是因为为了求解 `set([1,2,3])`，Python 必须先查找 `set` 名称，以获取构造函数 `set()`；然后，构建一个列表 `[1,2,3]`；最后，再将列表 `[1,2,3]` 传给构造函数 `set()`。相比之下，要处理像 `1, 2, 3` 这样的字面量，Python 只需要运行专门的 `BUILD_SET` 字节码¹²。

没有特殊的语法来表示 `frozenset` 字面量——它们必须通过调用构造函数来创建。Python 3 中，`frozenset` 的标准字符串表示法，就像一个 `frozenset` 构造函数调用。请注意控制台会话中的输出：

```

1  >>> frozenset(range(10))
2  frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

```

说到语法，列表推导式的思想也适用于构建集合（`set`）。

3.10.2 集合推导式

集合推导式（`setcomps`）早在 Python 2.7 中就被添加了，同时加入的还有“3.2.1 字典推导式”中的字典推导式。示例 3.31 展示了如何使用集合推导式。

¹²这可能很有趣，但不是非常重要。只有在求解一个 `set`（集合）字面量时，才有加速效果。并且每个 Python 进程最多发生一次加速——即在模块首次编译时。若您好奇，可以从 `dis` 模块导入 `dis` 函数，并用它来反汇编 `set` 字面量（如 `dis('1')`）和 `set` 调用（如 `dis('set([1])')`）的字节码。

</> 示例 3.31：构建一个集合，元素为 Unicode 名称中含有 ‘SIGN’ 的 Latin-1 字符。

```

1  >>> from unicodedata import name
2  >>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')} ❶
3  {'$', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©', '°', '+', '÷', '±',
   , '>', '¬', '®', '%'}

```

❶ 从 unicodedata 导入 name 函数，以获取字符名称。

❷ 将代码范围为 32~255，且名称中含有 ‘SIGN’ 一词的字符放入集合。

不同的 Python 进程得到的输出顺序也不同，原因与 “3.4.1 什么是可哈希” 中提到的哈希加盐 (Salted) 有关。

语法讲解完毕，现在让我们看一下集合的行为。

3.11 集合的实现方式对实践的影响

set 与 frozenset 类型都是通过哈希表实现的。这种设计带来了以下影响：

- 集合元素必须是可哈希 (hashable) 对象，如 “3.4.1 什么是可哈希” 所说，必须实现正确的 `__hash__` 和 `__eq__` 方法，
- 成员测试很高效。对于一个包含数百个元素的集合，通过计算元素的哈希码并派生出一个索引偏移量，即可直接定位到该元素。稍微搜索几次，即可找到匹配的元素。即使穷尽搜索，开销也不大。
- 与存放元素 (项) 指针的低级数组相比，集合有显著的内存开销。尽管低级数组的内存结构更为紧凑，但当在较大的数组中搜索元素时，搜索速度会明显下降。而集合则使用更高效的算法（如哈希表）来进行快速的搜索和查找。
- 集合中元素的顺序取决于插入顺序，但这种方式并不实用，也不可靠。例如，若两个不同的元素，但具有相同的哈希码，则它们的位置取决于哪个元素先被加入。
- 向集合添加元素后，可能会改变现有元素的顺序。这是因为，如果哈希表已满 $\frac{2}{3}$ 以上，算法的效率就会下降。所以，Python 可能需要随着哈希表的增长，而移动和调整其大小。当这种情况发生时，元素会被重新插入，它们的相对排序可能会发生变化。

要了解更多信息，请阅读本书网站 fluentpython.com 的 “Internals of sets and dicts”。

现在，让我们来讲解一下集合提供的丰富运算。

3.11.1 集合运算

图 3.2 概述了可用于可变集合与不可变集合的方法¹³。其中，许多是重载运算符（如 `&` 和 `>=`）的特殊方法。

表 3.2 列出了集合的数学运算符以及对应的 Python 运算符或方法。注意，有些运算符和方法（如 `&=`、`difference_update` 等）会对目标集合执行就地更改。这样的运算符，在集合的数学世界中毫无意义。并且，在 `frozenset` 中也未实现这些就地更改的运算符。

¹³ 斜体名称是抽象类和抽象方法；为简洁起见，省略了反向运算符方法。

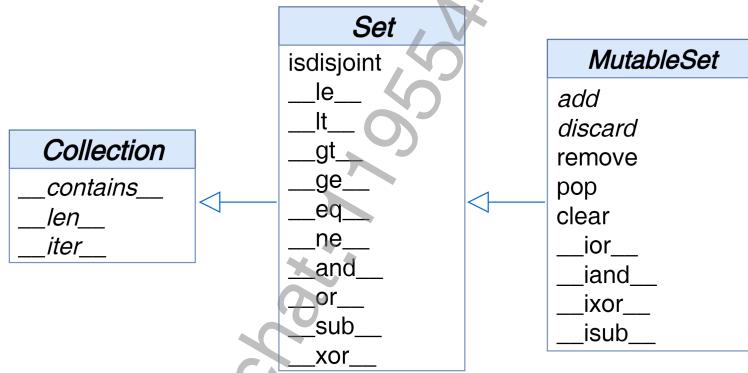
图 3.2: MutableSet 及其在 collections.abc 中的超类的简化 UML 类图¹³

表 3.2: 集合的数学运算符 (要么生成一个新集合, 要么就地更新可变集合。)

数学符号	Python 运算符	方法	说明
$s \cap z$	$s \& z$	<code>s.__and__(z)</code>	集合 s 与 z 的交集
	$z \& s$	<code>s.__rand__(z)</code>	反向 & 运算符
		<code>s.intersection(it, ...)</code>	s 与根据可迭代对象 it 等构建的集合的交集
	$s \&= z$	<code>s.__iand__(z)</code>	将 s 就地更新为 s 与 z 的交集
		<code>s.intersection_update(it, ...)</code>	用 s 与根据可迭代对象 it 等构建的集合的交集就地更新 s
$s \cup z$	$s \mid z$	<code>s.__or__(z)</code>	集合 s 与 z 的并集
	$z \mid s$	<code>s.__ror__(z)</code>	反向 运算符
		<code>s.union(it, ...)</code>	s 与根据可迭代对象 it 等构建的集合的并集
	$s \mid= z$	<code>s.__ior__(z)</code>	将 s 就地更新为 s 与 z 的并集
		<code>s.update(it, ...)</code>	用 s 与根据可迭代对象 it 等构建的集合的并集就地更新 s
$s \setminus z$	$s - z$	<code>s.__sub__(z)</code>	集合 s 与 z 的相对补集 (或差集)
	$z - s$	<code>s.__rsub__(z)</code>	反向 - 运算符
		<code>s.difference(it, ...)</code>	s 与根据可迭代对象 it 等构建的集合的差集
	$s -= z$	<code>s.__isub__(z)</code>	将 s 就地更新为 s 与 z 的差集
		<code>s.difference_update(it, ...)</code>	用 s 与根据可迭代对象 it 等构建的集合的差集就地更新 s
$s \Delta z$	$s ^ z$	<code>s.__xor__(z)</code>	对称差集 (s & z 的补集)
	$z ^ s$	<code>s.__rxor__(z)</code>	反向 ^ 运算符
		<code>s.symmetric_difference(it)</code>	s 与 set(it) 的补集
	$s ^= z$	<code>s.__ixor__(z)</code>	使用 s 与 z 的对称差集就地更新 s
		<code>s.symmetric_difference_update(it, ...)</code>	用 s 与根据可迭代对象 it 等构建的集合的对称差集就地更新 s



表 3.2 中的中缀运算符要求两个操作数都是集合, 其他方法则接受一个或多个可迭代参数。例如, 为了计算 4 个集合 a, b, c, d 的并集, 可以调用 `a.union(b,c,d)`。其中, a 必须是集合, 而 b, c, d 可以是任何类型的可迭代对象 (对象中的元素需要是可哈希 (hashable) 的)。若要用 4 个可迭代对象创建一个新集合 (而不是更新现有集合), 可以写成 `*a,*b,*c,*d`。这要归功于 Python 3.5 PEP 448 –Additional Unpacking Generalizations。

表 3.3 列出了 `set` (集合) 谓词——即返回 `True` 或 `False` 的运算符和方法。

表 3.3: 返回布尔值的集合比较运算符和方法

数学符号	Python 运算符	方法	说明
$S \cap Z = \emptyset$		<code>s.isdisjoint(z)</code>	集合 s 与 z 不相交 (二者没有交集)
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	元素 e 是集合 s 的 成员
$S \subseteq Z$	<code>s <= z</code>	<code>s.__le__(z)</code>	s 是 z 的 子集
		<code>s.issubset(it)</code>	s 是由可迭代对象 it 构建的集合的 子集
$S \subset Z$	<code>s < z</code>	<code>s.__lt__(z)</code>	s 是 z 的 真子集
$S \supseteq Z$	<code>s >= z</code>	<code>s.__ge__(z)</code>	s 是 z 的 超集
		<code>s.issuperset(it)</code>	s 是由可迭代对象 it 构建的集合的 超集
$S \supset Z$	<code>s > z</code>	<code>s.__gt__(z)</code>	s 是 z 的 真超集

除了源自数学集合论的运算符与方法之外, Python 中的集合类型还实现了一些其他实用的方法, 如表 3.4 所示。

表 3.4: 集合的其他实用方法

方法	<code>set</code>	<code>frozenset</code>	说明
<code>s.add(e)</code>	•		将元素 e 添加到 s 中
<code>s.clear()</code>	•		删除 s 中的所有元素
<code>s.copy()</code>	•	•	浅拷贝 s
<code>s.discard(e)</code>	•		如果元素 e 存在, 则将其从 s 中移除
<code>s.__iter__()</code>	•	•	获取遍历集合 s 的迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code>
<code>s.pop()</code>	•		从 s 中移除并返回一个元素。若 s 为空, 则引发 <code>KeyError</code> 。
<code>s.remove(e)</code>	•		从 s 中删除元素 e 。若 e 不在 s 中, 则引发 <code>KeyError</code> 。

对集合功能的概述到此结束。“3.12 字典视图的集合运算”将兑现“3.8 字典视图”的承诺, 探讨与 `frozenset` 行为非常相似的 2 种字典视图。

3.12 字典视图的集合运算

“3.8 字典视图”中描述的 `.keys()` 与 `.items()` 这两个 `dict` 方法, 返回的视图对象 (即 `dict_keys` 与 `dict_items`) 与 `frozenset` 类型极为相似, 如表 3.5 所示。

表 3.5: frozenset、dict_keys、dict_items 实现的方法

方法	frozenset	dict_keys	dict_items	说明
s.__and__(z)	•	•	•	s & z (s 与 z 的交集)
s.__rand__(z)	•	•	•	反向 & 运算符
s.__contains__(e)	•	•	•	e in s
s.copy()	•			浅拷贝 s
s.difference(it, ...)	•			s 与可迭代对象 it 等的差集
s.intersection(it, ...)	•			s 与可迭代对象 it 等的交集
s.isdisjoint(z)	•	•	•	s 与 z 不相交 (没有共同元素)
s.issubset(it)	•			s 是可迭代对象 it 的子集
s.issuperset(it)	•			s 是可迭代对象 it 的超集
s.__iter__()	•	•	•	获取遍历 s 的迭代器
s.__len__()	•	•	•	len(s)
s.__or__(z)	•	•	•	s z (s 与 z 的并集)
s.__ror__(z)	•	•	•	反向 运算符
s.__reversed__()		•	•	获取逆序 (反方向) 遍历 s 的迭代器
s.__rsub__(z)	•	•	•	反向 - 运算符
s.__sub__(z)	•	•	•	s - z (s 与 z 的差集)
s.symmetric_difference(it)	•			s & set(it) 的补集
s.union(it, ...)	•			s 与可迭代对象 it 等的并集
s.__xor__(z)	•	•	•	s ^ z (s 与 z 的对称差集)
s.__rxor__(z)	•	•	•	反向 ^ 运算符

需要特别注意的是, dict_keys 与 dict_items 实现了一些特殊方法, 以支持强大的集合运算符, 包括: & (交集), | (并集)、- (差集) 和 ^ (对称差集)。

例如, 使用 & 可以轻松获取 2 个字典都包含的键 (如 [示例 3.32](#) 所示):

</> 示例 3.32: 用 & 获取两个字典中都有的键

```

1 >>> d1 = dict(a=1, b=2, c=3, d=4)
2 >>> d2 = dict(b=20, d=40, e=50)
3 >>> d1.keys() & d2.keys()
4 {'b', 'd'}

```

注意, & 运算符的返回值是一个 set 对象。更妙的是, [字典视图 \(Dictionary View\)](#) 中的集合运算符与 set 实例兼容。如 [示例 3.33](#) 所示。

</> 示例 3.33: 字典视图的集合元素符与 set 实例兼容

```

1 >>> s = {'a', 'e', 'i'}
2 >>> d1.keys() & s
3 {'a'}
4 >>> d1.keys() | s

```

5 { 'a', 'c', 'b', 'd', 'i', 'e'}



仅当 dict 中的所有值都是可哈希 (hashable) 的, 才可将 dict_items 视图当作集合来使用。倘若 dict 中有不可哈希的值, 对 dict_items 视图做集合运算, 将引发 TypeError: unhashable type 'T' 异常。其中, T 是违规(即不可哈希)值的类型。相反, dict_keys 视图始终可被当作集合来使用。因为根据 dict 的实现原理, dict 中的每个键都是可哈希 (hashable) 的。

在检查代码中 dict 的内容时, 将集合运算符与字典视图 (Dictionary View) 结合使用, 可以节省大量的循环和 if 条件语句。繁重的工作都可以交给 C 语言实现的 Python, 来高效完成。

本章至此结束。

3.13 本章小结

dict (字典) 是 Python 的基石。多年来, 我们熟悉的字面量语法 `k1: v1, k2: v2` 得到了增强, 现已支持用 `**` 解包、模式匹配 (Pattern Match) 以及字典推导式。

除了基本的 dict 之外, Python 标准库中的 collections 模块还提供了方便易用的专用映射, 如 defaultdict、ChainMap 和 Counter。有了新的 dict 实现之后, OrderedDict 就不像以前那样有用了。但仍应保留在标准库中, 一方面是为了实现向后兼容, 另一方面 OrderedDict 还有一些特性是 dict 所不具备的。例如, OrderedDict 的比较运算符 (==) 会将字典中键的顺序纳入考虑范围之内。collections 模块中的 UserDict 是一个易于使用的基类, 可用于创建自定义映射。

多数映射中都拥有 2 个强大的方法: `setdefault` 与 `update`。`setdefault` 方法可以更新持有可变值的项 (例如, list 值), 从而避免再次搜索相同的键。`update` 方法可以批量插入或覆盖映射中的项。这些项可以来自其他映射, 可以来自提供 (键, 值) 对的可迭代对象, 也可以来自关键字参数。映射构造函数内部也使用 `update` 方法, 以便从映射、可迭代对象或关键字参数来初始化实例。自 Python 3.9 起, 还可以用 `!=` 运算符更新映射, 以及用 `|` 运算符根据两个映射的合集, 创建一个新的映射。

映射的 API 中的 `__missing__` 方法是一个巧妙的钩子方法, 利用 `__missing__` 方法可以自定义句法 `d[k]` (调用 `__getitem__`) 在找不到键时的行为。

`collections.abc` 模块中的抽象基类 (ABCs) `Mapping` 与 `MutableMapping` 定义了标准接口, 可用于运行时的类型检查。`types` 模块中的 `MappingProxyType` 为映射包装了一层不可变的外壳, 以映射被意外更改。此外, 也有用于 `Set` 与 `MutableSet` 的抽象基类 (ABCs)。

字典视图 (Dictionary View) 是 Python 3 的一大亮点, 它消除了 Python 2 中 `.keys()`、`.values()`、`.items()` 方法的内存开销。即 Python 3 中的这些方法, 无需再构建列表, 重复目标 dict 实例中的数据。此外, `dict_keys` 与 `dict_items` 类支持 `frozenset` 中最有用的运算符和方法。

3.14 延伸阅读

在 Python 标准库文档的 “`collections—Container datatypes`” 中, 包含了几种映射类型的示例和实践技巧。`collections` 模块的 Python 源码 (`Lib/collections/init.py`), 对于想要创建新的映射类型, 或理解现有映射

类型逻辑来说,是一个很好的参考。David Beazley 和 Brian K. Jones 所著的《Python Cookbook》第 3 版 (O'Reilly) 第 1 章中有 20 个关于数据结构的经典案例,其中大部分都巧妙地使用了 dict。

Greg Gandenberger 在 “Python Dictionaries Are Now Ordered. Keep Using OrderedDict” 一文中,提倡继续使用 collections.OrderedDict,理由是“显式优于隐式”、向后兼容性、以及一些工具和库都假定 dict 键的顺序无关紧要。

Guido van Rossum 在 “PEP 3106 -Revamping dict.keys(), .values() and .items()” 中介绍了 Python 3 的字典视图 (Dictionary View) 功能。他在摘要中指出,这个想法来自 Java Collections Framework。

PyPy 是第一个实现 Raymond Hettinger 关于紧凑字典提案的 Python 解释器,详见博客文章 “Faster, more memory efficient and more ordered dictionaries on PyPy”,文中承认 PHP7 也采用了类似的布局,详见 “PHP's new hashtable implementation”。当创作者引用先前的技术成果时,这总是非常好的。

在 PyCon 2017 上,Brandon Rhodes 做了题为 “The Dictionary Even Mightier” 的演讲,这是他经典动画演讲 “The Mighty Dictionary” (包括哈希碰撞的动画) 的续集。另一个关于 Python dict 内部最新且更深入的视频是 Raymond Hettinger 的 “Modern Dictionaries”。视频中提到,最初 Raymond Hettinger 向 CPython 核心开发团队建议实现紧凑字典,但无果而终。后来,成功说服 PyPy 团队,引起 CPython 团队的关注,而后由 INADA Naoki 在 CPython 3.6 中实现。有关详细信息,请查看 CPython 源码 Objects/dictobject.c 中的注释,以及设计文档 Objects/dictnotes.txt。

在 “PEP 218 -Adding a Built-In Set Object Type” 中,记录了为 Python 添加集合 (set) 的原因。在 PEP 218 被批准时,尚未为 set 提供专门的字面量语法。set 字面量最初是为 Python 3 创建的,随后被向后移植到了 Python 2.7,同时还引入了字典推导式与集合推导式。在 2019 年的 PyCon 大会上,我做了题为 “Python Set Practice:learning from Python's set types” 的演讲,介绍了集合在实际程序中的用例,涵盖了集合的 API 设计,以及 uintset 的实现。uintset 是一个使用位向量 (而非哈希表) 实现的存放整数元素的集合类,其灵感来自 Alan Donovan 和 Brian Kernighan (Addison-Wesley) 所著的《The Go Programming Language》第 6 章中的一个示例。

IEEE 的 Spectrum 杂志有一篇关于 Hans Peter Luhn 的报道,他是一位多产的发明家,曾为一种打孔卡牌申请了专利,这种卡牌可以根据现有原料选择鸡尾酒配方,还有其他各种发明,包括 哈希表! 请参阅 “Hans Peter Luhn and the Birth of the Hashing Algorithm”。

杂谈

语法糖 (Syntactic Sugar)

我的朋友 Geraldo Cohen 曾说过,Python “简单而正确”。编程语言纯粹主义者喜欢认为语法并不重要。

语法糖将诱发分号癌^a。

——Alan Perlis

^a关于“语法糖将诱发分号癌”的说法,是指过多地使用语法糖可能导致代码复杂性增加。因此,在使用语法糖时,应该权衡其可读性和代码复杂性之间的平衡。

语法是编程语言的用户界面,因此在实践中很重要。

在发现 Python 之前,我们用 Perl 与 PHP 进行了一些 Web 编程。这些语言中的映射语法非常好用,每当我必须用 Java 或 C 时,我都会非常想念它^a。

良好的映射字面量语法需要使用方便,如便于配置、便于实现表驱动设计、便于存放原型设计与测试数据。Go 语言的设计人员就从动态语言中学到了这一点。Java 由于缺乏在代码中表达结构化数据的好方法,导致 Java 社区不得不采用冗长且复杂的 XML 作为其数据格式。

JSON 是 XML 的一种替代方案,去繁从简,并取得了巨大成功。在很多情况下,可以取代 XML。简洁明了的列表与字典语法,使 JSON 成为一种出色的数据交换格式。

PHP 和 Ruby 模仿 Perl 的哈希语法,使用 `=>` 将键与值连接起来^a。而 JavaScript 和 Python 一样,只使用一个字符(即`:`)来连接键和值。既然一个字符就够了,为什么还要用两个字符呢?

JSON 来自 JavaScript,但它也恰好几乎是 Python 语法的完全子集。除了 `true`、`false` 与 `null` 的拼写不一致之外,JSON 与 Python 几乎完全兼容。

Armin Ronacher 在 Twitter 上说,他喜欢破解 Python 的全局命名空间,为 Python 的 `True`、`False`、`None` 添加兼容 JSON 的别名。这样,就可以将 JSON 直接粘贴到 Python 控制台中,像下面这样。

```

1  >>> true, false, null = True, False, None
2  >>> fruit = {
3  ...     "type": "banana",
4  ...     "avg_weight": 123.2,
5  ...     "edible_peel": false,
6  ...     "species": ["acuminata", "balbisiana", "paradisiaca"],
7  ...     "issues": null,
8  ... }
9  >>> fruit
10 {'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
11 'species': ['acuminata', 'balbisiana', 'paradisiaca'], 'issues': None}

```

现在,大家都使用 Python 的 `dict` 与 `list` 语法来交换数据。如今,语法越来越便利,还能方便地保留插入顺序。

真是简单又正确。

^a指 Perl 与 PHP 中的映射语法

^b译者注:从 Ruby 1.9 开始,若哈希的键是 `Symbol` 类型,也可以使用`:`字符来连接键与值。

Unicode 文本与字节序列

人类使用文本, 而计算机使用字节。

——Esther Nam 与 Travis Fischer,
《Python 中的字符编码和 Unicode》^a

^aPyCon 2014, “Character Encoding and Unicode in Python” (幻灯片, [视频](#)) 演讲第 12 张幻灯片。

Python 3 中明确区分了人类文本字符串与原始字节序列。将字节序列隐式转换为 Unicode 文本已成为过去。本章将讨论 Unicode 字符串、二进制序列以及用于在它们之间进行转换的编码。

在 Python 编程工作中, 你可能会误以为不需要深入了解 Unicode。然而, 这种想法是不切实际的。不管怎样, str 类型和 bytes 类型之间的差异是我们必须面对的。而且, 作为额外的收获, 你会发现专用二进制序列类型 (Python 3 引入) 所提供的功能, 有些是 Python 2 中“全能” str 类型所没有的。

本章涵盖以下主题:

- 字符、码点、字节表示。
- 二进制序列的独特特征: bytes, bytearray 与 memoryview。
- 完整 Unicode 编码与传统字符集的编码。
- 避免编码错误与处理编码错误。
- 处理文本文件的最佳实践。
- 默认编码的陷阱与标准 I/O 的问题。
- 通过规范化, 进行安全的 Unicode 文本比较。
- 用于规范化、大小写同一化、强制去除读音符的实用函数。
- 用 locale 与 pyuca 库正确排序 Unicode 文本。
- Unicode 数据库中的字符元数据。
- 能处理 str 与 bytes 的双模式 API。

4.1 本章新增内容

Python 3 对 Unicode 的支持已经非常全面和稳定, 因此最值得注意的新增内容是 “4.9.1 按名称查找字符”, 描述了一个用于搜索 Unicode 数据库的实用程序——从命令行查找带圈数字和微笑猫。

一个值得一提的小变化是: 自 Python 3.6 以来, Windows 对 Unicode 的支持变得更好更简单了。详见 “4.6.1 注意默认编码”。

让我们从字符、码点和字节, 这些基本概念开始。



在第 2 版中, 我扩充了有关 `struct` 模块的部分, 并将其发布在配套网站 “fluent-python.com/” 的 “*Parsing binary records with struct*”。

在该网站, 您还可以找到 “*Building Multi-Character Emojis*”, 描述如何通过组合 Unicode 字符来制作国旗、彩虹旗、不同肤色的人以及不同的家庭图标。

4.2 字符问题

“字符串 (`string`)” 的概念很简单: 字符串就是一串字符。问题在于 “字符 (`character`)” 的定义。

在 2021 年, 对 “字符 (`character`)” 的最佳定义是 Unicode 字符。因此, 我们 [从 Python 3 的 `str` 中获取的项是 Unicode 字符, 这与从 Python 2 的 `unicode` 对象中的项一样。而从 Python 2 的 `str` 中获取的项是原始字节 \(Raw Bytes \)](#)。

Unicode 标准明确将字符标识与特定的字节表示形式分开。

- 字符的标识, 即 [码点 \(`Code Point` \)](#)¹, 是一个 0~1,114,111 (十进制) 之间的数字, 在 Unicode 标准中现实为 4~6 个十六进制数字, 前缀为 “U+”, 从 U+0000 到 U+10FFFF。例如, 字母 A 的码点为 U+0041; 欧元符号 € 的码点为 U+20AC; G-clef 音乐符号 ♩ 的码点为 U+1D11E。在 Python 3.10.0b4 中使用的 Unicode 13.0.0 标准中, 约 13% 的有效码点对应着字符。
- 表示字符的实际字节取决于所使用的 [编码](#)。编码是在 [码点 \(`Code Point` \)](#) 与字节序列之间转换时使用的算法。例如, 字母 A (U+0041) 在 UTF-8 编码中使用单个字节 `\x41` 表示, 而在 UTF-16LE 编码中使用字节序列 `\x41\x00` 表示。再如, 欧元符号 € (U+20AC) 在 UTF-8 编码中需要 3 个字节, 即 `\xe2\x82\xac`。而在 UTF-16LE 中, 同一个码点被编码为 2 个字节, 即 `\xac\x20`。

将码点转换为字节的过程, 称作 “[编码 \(`Encoding` \)](#)”; 而将字节转换为码点的过程, 称作 “[解码 \(`Decoding` \)](#)”。如 [示例 4.1](#) 所示。

</> [示例 4.1: 编码与解码](#)

```

1  >>> s = 'café'
2  >>> len(s)           ❶
3  4
4  >>> b = s.encode('utf8') ❷
5  >>> b
6  b'caf\xc3\xa9'       ❸
7  >>> len(b)           ❹

```

¹ Unicode 码点 (`Code Point`) 是指每个字符在 Unicode 编码表中所对应的唯一数字标识。

```

8 5
9 >>> b.decode('utf8')      ⑤
10 'café'
11 >>> b[3:5].decode('utf8') ⑥
12 'é'
13 >>> b[3:5].decode('utf8').encode('utf8')
14 b'\xc3\xaa'

```

- ① 字符串“café”有4个Unicode字符。
- ② 用UTF-8将str对象编码(Encoding)为bytes对象。
- ③ bytes(字节序列)字面量以字符b开头。
- ④ 字节序列b有5个字节(在UTF-8中,字符“é”的码点被编码为2个字节)。
- ⑤ 用UTF-8将bytes对象解码(Decoding)为str对象。
- ⑥ 字符“é”的码点被编码为2个字节,所以解码字符“é”也需要切片2个字节。



若记不住.decode()与.encode()的区别,可将字节序列视作晦涩难懂的机器核心转储,将Unicode字符串视作“人类可读”的文本。因此,将字节序列解码为str,以获得人类可读的文本;而将str编码为字节序列,以用于存储或传输。

虽然Python 3的str类型基本相当于Python 2的unicode类型,只是换了个新名称。但是,Python 3中的bytes类型并不是简单地将旧的str换个名称那么简单,而且还有关系紧密的bytarray类型。因此,在讨论编码与解码问题之前,有必要先介绍一下二进制序列类型(即bytes与bytarray)。

4.3 字节要点

新的二进制序列类型在很多方面都与Python 2中的str不同。首先要知道的是,Python内置了两种基本的二进制序列类型:不可变bytes类型(Python 3引入)与可变类型bytarray(Python 2.6²引入)。Python文档有时使用通用术语“字节字符串”来指代bytes与bytarray。我一般不用这个令人困惑的通用术语。

bytes或bytarray中的每一项都是0~255之间的整数,而不是像Python 2 str中那样的单字符字符串。然而,二进制序列的切片也还是二进制序列——包括长度为1的切片,如示例4.2所示。

</>示例4.2:包含5个字节的bytes与bytarray

```

1 >>> cafe = bytes('café', encoding='utf_8') ①
2
3 >>> cafe
4 b'caf\xc3\xaa'
5
6 >>> cafe[0]      ②
7 99
8 >>> cafe[:1]     ③
9 b'c'
10 >>> cafe_arr = bytarray(cafe)
11 >>> cafe_arr     ④
12 bytarray(b'caf\xc3\xaa')

```

²Python 2.6和2.7也有bytes类型,但它只是str类型的别名。

```
11 >>> cafe_arr[-1:] ❸
12 bytearray(b'\xa9')
```

- ❶ 指定编码 (encoding), 可以根据 str 对象构建 bytes 对象 cafe。
- ❷ bytes 对象 cafe 中的每项, 都是 range(256) 内的整数。
- ❸ bytes 对象的切片, 还是 bytes 对象——即使是只有一个字节的切片。
- ❹ bytearray 没有字面量语法: 显示为 bytearray() 调用形式, 参数为一个 bytes 字面量。
- ❺ bytearray 对象的切片, 也还是 bytearray 对象。



my_bytes[0] 返回的是一个 int 整数, 而 my_bytes[:1] 返回的是一个长度为 1 的字节序列。若您不理解此种行为, 只能说明您已习惯了 Python 的 str 类型, 即 str[0]==str[:1]。除此之外, 对于 Python 中的其他序列类型, 一个项不能等同于长度为 1 的切片。

虽然二进制序列实际上是整数序列, 但是它们的字面量表示法表明其中含有 ASCII 文本。因此, 根据每个字节值的不同, 使用了四种不同的显示方式:

- 对于十进制代码为 32 ~ 126 的字节 (从空格到 ~), 使用 ASCII 字符本身。
- 对于制表符、换行符、回车符和 \ 对应的字节, 使用转义序列 \t、\n、\r 和 \\。
- 若字节序列中同时出现字符串分隔符'与", 则整个序列使用'分隔。其中的'被转义为 \³。
- 对于其他字节值, 使用十六进制转义序列 (例如, \x00 是空字节)。

这就是为什么在 [示例 4.2](#) 中你会看到 b'caf\xc3\xa9': 前 3 个字节 b'caf' 在可打印的 ASCII 范围内, 而后 2 个字节 b'\xc3\x9a' 不在可打印的 ASCII 范围。

bytes 和 bytearray 不支持 str 的格式化方法 (format、format_map) 和依赖于 Unicode 数据的 str 方法 (包括 casefold、isdecimal、isidentifier、isnumeric、isprintable 和 encode)。其他 str 方法均受 bytes 和 bytearray 类型的支持。这意味着您可以使用熟悉的字符串方法, 如 endswith、replace、strip、translate、upper 等, 来处理二进制序列——只需要指定 bytes 类型的参数。此外, 如果正则表达式编译自二进制序列, 而不是字符串, 则 re 模块中的正则表达式函数也可用于处理二进制序列。自 Python 3.5 起, % 运算符又能支持二进制序列了⁴。

二进制序列有一个名为 fromhex 的类方法 (str 没有此方法), 可通过解析由空格分隔的十六进制数字对来构建二进制序列:

```
1 >>> bytes.fromhex('31 4B CE A9')
2 b'1K\xce\x9a'
```

构建 bytes 或 bytearray 实例, 还可以调用各自的构造函数, 并传入以下参数:

- 一个 str 对象与关键字参数 encoding。
- 一个可迭代对象, 其中的项为 0 ~ 255 之间的整数。
- 一个实现了缓冲区协议的对象 (如 bytes、bytearray、memoryview、array.array)。构造函数将源对象

³ 小知识: Python 默认用作字符串分隔符的 ASCII 单引号, 在 Unicode 标准中的名称实际上是 APOSTROPHE (撇号)。真正的单引号是不对称的, 左边是 U+2018, 右边是 U+2019。

⁴ % 运算符在 Python 3.0 ~ 3.4 中不起作用, 这给处理二进制数据的开发人员带来了极大的痛苦。详见 “PEP 461 – Adding % formatting to bytes and bytearray”。

中的字节序列复制到新创建的二进制序列中。



在 Python 3.5 之前, 还可以为 bytes 或 bytearray 构造函数指定一个整数参数, 使用空字节创建一个对应长度的二进制序列。这种函数签名在 Python 3.5 中被弃用, 在 Python 3.6 中正式移除。详见 “[PEP 467 –Minor API improvements for binary sequences](#)”。

用类缓冲 (buffer-like) 对象构建二进制序列是一种底层操作, 可能涉及类型转换, 如 [示例 4.3](#) 所示。

</> [示例 4.3](#): 用数组中的原始数据构建 bytes 对象

```
1 >>> import array
2 >>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
3 >>> octets = bytes(numbers) ❷
4 >>> octets
5 b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ 指定 typecode 为 'h', 创建一个短整型 (16 bits) 数组。
- ❷ 将 numbers 中的字节序列复制给 octets。octets 是 numbers 的副本。
- ❸ 用于表示 5 个短整数的字节序列 (10 个字节)。

用类缓冲 (buffer-like) 对象构建 bytes 或 bytearray 对象, 始终都会复制源对象 (如内存) 中的字节序列。与之相反, memoryview 对象可在二进制数据结构之间共享内存中的字节序列 (详见 “[2.10.2 memoryview](#)”)。在对 Python 二进制序列进行简要探讨之后, 接下来研究如何在二进制序列与 str 之间相互转换。

4.4 基本的编码器与解码器

Python 发行版自带了超过 100 多个编解码器 (Encoder/Decoder), 用于文本与字节之间的相互转换。每个编解码器都有一个名称 (如 utf_8), 并且通常还有几个别名 (如 utf8、utf-8 或 U8)。可将这些名称指定给 open()、[str.encode\(\)](#)、[bytes.decode\(\)](#) 等函数的 encoding 参数。[示例 4.4](#) 展示了用 3 种不同的编解码器, 对同一段文本进行编码的效果。

</> [示例 4.4](#): 用 3 种不同的编解码器, 编码同一段文本。

```
1 >>> for codec in ['latin_1', 'utf_8', 'utf_16']:
2 ...     print(codec, 'El Niño'.encode(codec), sep='\t')
3 ...
4 latin_1 b'El Ni\xf1o'
5 utf_8 b'El Ni\xc3\xb1o'
6 utf_16 b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

[图 4.1](#) 展示了用不同的编解码器对字母 “A” 到 G-clef 音乐符号 $\text{\textgreek{gamma}}$ 等字符进行编码后得到的字节序列。请注意, 最后 3 种是可变长度的多字节编码。

[图 4.1](#) 中的 * 表示, 某些编码 (如 ASCII 与多字节的 GB2312) 不能表示所有 Unicode 字符。然而, UTF 编码的设计初衷就是可以处理所有 Unicode 码点 (Code Point)。

[图 4.1](#) 中展示的是一些比较典型的编码, 介绍如下:

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
À	U+0041	41	41	41	41	41	41	41 00
Ź	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
Ѐ	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Ѓ	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
Ӄ	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
Ӄ	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
Ӄ	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

图 4.1: 12 个字符 (码点) 及不同编码的字节表示 (* 表示该字符不能用该编码表示)

- latin1 (即 iso8859_1)

一种很重要的编码,是其他编码 (cp1252 与 Unicode) 的基础。注意,latin1 与 cp1252 的字节值是一样的,甚至连 码点 (Code Point) 也相同。

- cp1252

Microsoft 创建的 latin1 超集,添加了有用的符号,如大括号 {} 与 € (欧元)。一些 Windows 应用将此编码称为“ANSI”,但此编码并不是 ANSI 标准。

- cp437

IBM PC 最初的字符集,包含框图符号。与后出现的 latin1 不兼容。

- gb2312

用于编码简体中文的旧标准。亚洲语言广泛使用的几种多字节编码之一。

- utf-8

目前,Web 最常用的 8 位编码标准。根据 W3Techs 发布的《Usage statistics of character encodings for websites》报告,截至 2021 年 7 月,97% 的网站都在使用 UTF-8。在 2014 年 9 月,本书英文第 1 版出版时,这一比例是 81.4%。

- utf-16le

UTF 16 位编码方案的一种形式。所有 UTF-16 编码都支持通过转义序列 (称为“代理对 (Surrogate Pair)”) 表示超过 U+FFFF 的 码点 (Code Point)。



早在 1996 年,UTF-16 就取代了最初的 16 位 Unicode 1.0 编码——UCS-2。UCS-2 只支持 U+FFFF 以下的 码点 (Code Point),尽管 UCS-2 早在上世纪就已被淘汰,但许多系统仍在使用。截至 2021 年,超过 57% 的已分配 码点 (Code Point) 高于 U+FFFF,其中包括最重要的表情符号 (emoji)。

在完成对常见编码的概述后,下面将探讨编码与解码操作中所涉及的问题。

4.5 处理编码与解码问题

尽管存在一个通用的 `UnicodeError` 异常, 但 Python 在报告错误时往往会更具体: 要么是 `UnicodeEncodeError` (将 str 编码为二进制序列时), 要么是 `UnicodeDecodeError` (将二进制序列解码为 str 时)。当源编码意外时, 加载 Python 模块也可能引发 `SyntaxError`。我们将在接下来的章节中介绍如何处理这些错误。



当收到 Unicode 错误时, 首先要注意异常的具体类型。是 `UnicodeEncodeError`、`UnicodeDecodeError`, 还是其他提到编码问题的错误 (如 `SyntaxError`)? 要解决问题, 首先要了解问题。

4.5.1 处理 `UnicodeEncodeError`

多数非 UTF 编解码器仅能处理 Unicode 字符的一小部分子集。在将文本转换为字节时, 若目标编码中未定义某个字符, 则会引发 `UnicodeEncodeError` 异常, 除非将 `errors` 参数传递给编码方法或函数, 来做特殊处理。错误处理器的行为如 [示例 4.5](#) 所示。

```
</> 示例 4.5: 编码到字节: 成功与错误处理
1  >>> city = 'São Paulo'
2  >>> city.encode('utf_8')          ❶
3  b'S\xc3\xaa Paulo'
4  >>> city.encode('utf_16')
5  b'\xff\xfeS\x00\xe3\x00\x00 \x00P\x00a\x00u\x00l\x00o\x00'
6  >>> city.encode('iso8859_1')      ❷
7  b'S\xe3 Paulo'
8  >>> city.encode('cp437')          ❸
9  Traceback (most recent call last):
10  File "<stdin>", line 1, in <module>
11  File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode
12      return codecs.charmap_encode(input,errors,encoding_map)
13 UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in position 1:
14      character maps to <undefined>          ❹
15  >>> city.encode('cp437', errors='ignore') ❺
16  b'So Paulo'
17  >>> city.encode('cp437', errors='replace') ❻
18  b'S?o Paulo'
19  >>> city.encode('cp437', errors='xmlcharrefreplace') ❼
20  b'S&#227;o Paulo'
```

❶ UTF 编码可处理任何 str 对象。

❷ iso8859_1 编码也适用于 “São Paulo” 字符串。

❸ cp437 无法编码 “a” (带波形符的 “a”)。默认的错误处理程序为 ‘strict’, 即引发 `UnicodeEncodeError` 异常。

❹ `error='ignore'` 处理程序会跳过无法编码的字符; 这通常是个非常糟糕的做法, 会导致数据悄无声息的丢失。

- ⑤ error='replace' 处理程序会用? 替换无法编码的字符; 数据也会丢失, 但用户会得到一个提示, 说明出了问题。
 - ⑥ error='xmlcharrefreplace' 处理程序会将无法编码的字符替换为 XML 实体。如果无法使用 UTF, 并且又不能丢失数据, 这是唯一的选择。



编解码器的错误处理是可扩展的。可以通过向 `codecs.register_error` 函数传递一个名称和错误处理函数，为参数 `error` 注册额外的字符串。请参阅 `codecs.register_error` 文档。

据我所知,ASCII 是所有编码的共同子集,因此如果文本仅由 ASCII 字符组成,则所有编码都有效。Python 3.7 新增了一个布尔值方法 `str.isascii()`,用于检查 Unicode 文本是否 100% 由 ASCII 字符构成。如果是,则可用任何编码将文本转换为字节序列,而不会引发 `UnicodeEncodeError` 异常。

4.5.2 处理 UnicodeDecodeError

并非所有字节都包含有效的 ASCII 字符, 也并非每个字节序列都是有效的 UTF-8 或 UTF-16; 因此, 在将二进制序列转换为文本时, 如果假定使用了这些编码之一, 则会在遇到无法识别的字节时引发 `UnicodeDecodeError`。

另外,许多传统的8位编码(如cp1252、iso8859_1和koi8_r)能够解码任何字节流(包括随机噪声),而且不会报错。因此,若程序采用了错误的8位编码,它将会默默地解码出垃圾信息。



乱码字符被称为“鬼符 (gremlin)”或 mojibake (文字化け——日语, 意为“转换后的文本”)。

说明了使用错误的编解码器会产生“鬼符 (gremlin)”或引发 `UnicodeDecodeError` 异常。

</> 示例 4.6: 将字节序列解码为 str(有些成功,有些需要错误处理器)

```
1  >>> octets = b'Montr\xe9al'          ①
2  >>> octets.decode('cp1252')          ②
3  'Montréal'
4  >>> octets.decode('iso8859_7')       ③
5  'Montréal'
6  >>> octets.decode('koi8_r')          ④
7  'Montr\x89al'
8  >>> octets.decode('utf_8')           ⑤
9  Traceback (most recent call last):
10  File "<stdin>", line 1, in <module>
11  UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5: invalid
12  continuation byte
13  >>> octets.decode('utf_8', errors='replace') ⑥
14  'Montréal'
```

- ① 用 latin1 对单词'Montréal' 进行编码, '\xe9' 是字符 'é' 的字节序列。

- ② 用 cp1252 (Windows 1252) 进行解码是可行的,因为它是 latin1 的超集。
- ③ ISO-8859-7 适用于希腊文,无法正确解码 '\xe9' 字节,但不会出错。
- ④ KOI8-R 适用于俄文。现在, '\xe9' 字节表示西里尔字母 “И”。
- ⑤ utf_8 编解码器检测到 octets 不是有效的 UTF-8 二进制序列,引发 `UnicodeDecodeError` 异常。
- ⑥ 使用 `errors='replace'` 错误处理器后,UTF-8 无法解码的字节 '\xe9' 被替换为 '߿' (码点 U+FFFD)。'߿' 是用于表示未知字符的官方 Unicode 替换字符 (REPLACEMENT CHARACTER)。

4.5.3 加载模块时的 `SyntaxError`

Python 3 源代码默认使用 UTF-8 编码,而 Python 2 默认使用 ASCII 编码。若加载一个包含非 UTF-8 数据的.py 模块,并且未声明编码,则会收到如下消息:

```
1 SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line 1, but no
   encoding declared; see https://python.org/dev/peps/pep-0263/ for details
```

GNU/Linux 和 macOS 系统中大都使用 UTF-8,因此当在这类系统中打开一个在 Windows (cp1252 编码) 上创建的.py 文件,可能会遇到上述错误。这种错误,在 Windows 系统中也可能会发生,因为所有平台的 Python 3 源码的默认编码都是 UTF-8。

要解决此问题,请在文件顶部添加一个神奇的编码注释,如 [示例 4.7](#) 所示:

</> [示例 4.7: ola.py: 打印葡萄牙语 “Hello, World!”](#)

```
1 # coding: cp1252
2
3 print('Olá, Mundo!')
```



现在, Python 3 的源代码不再局限于 ASCII,而是默认使用更出色的 UTF-8 编码。对于使用传统编码(如“cp1252”)的源代码,最好的“修复”方法就是将它们转换为 UTF-8,而不必再费心处理编码注释。如果你的编辑器不支持 UTF-8,那么是时候换一个了。

假设您有一个文本文件,其内容可能是源码或者诗句,但是您不知道它的编码。此时,该如何检测文本的实际编码方案?答案将在[“4.5.4 如何找出字节序列的编码”](#)揭晓。

4.5.4 如何找出字节序列的编码

如何自己找到字节序列的编码标准?简单来说,不能。这只能由别人来告诉你。

有些通信协议和文件格式(如 HTTP 和 XML),可通过头部信息明确指明内容的编码。若字节流中包含大于 127 的字节值,则该字节流用的肯定不是 ASCII 编码。另外,按照 UTF-8 与 UTF-16 构建方式,其可用的字节序列也受到了限制。

Leo 猜测 UTF-8 解码的技巧

(以下几段摘自技术审校 Leonardo Rochael 在本书草稿中留下的注释。)

按照 UTF-8 的设计方式,一段随机字节序列,甚至来自非 UTF-8 编码的非随机字节序列,几乎不会被 UTF-8 解码为乱码。相反,在遇到此种情况时,通常都会引发 `UnicodeDecodeError` 异常。

其原因是 UTF-8 转义序列从不使用 ASCII 字符,并且这些转义序列具有位模式,这使得随机数据很难意外成为有效的 UTF-8。

因此,如果能将某些包含代码(十进制)>127的字节解码为 UTF-8,那么它很可能就是 UTF-8 编码。

在处理巴西的在线服务(其中一些服务与传统后端相连)时,我有时迫不得已,首先尝试通过 UTF-8 解码;当遇到 `UnicodeDecodeError` 异常时,再通过 `cp1252` 解码来处理此异常。这种解码策略虽然不够优雅,但却很有效。

不过,考虑到人类语言也有自己的规则和限制,只要假定字节流是人类可读的纯文本,就有可以通过试探与分析字节流特征来找出其编码。例如,如果 `b'\x00'` 字节经常出现,那么很可能是 16 位或 32 位编码,而不是 8 位编码方案,因为纯文本中不可能包含空字符(`b'\x00'`);

当字节序列 `b'\x20\x00'` 频繁出现时,它更可能是 UTF-16LE 编码中的空格字符(UTF+0020),而不是晦涩难懂的 U+2000 EN QUAD 字符——鬼知道它是什么!

“`Chardet: 通用字符编码检测器`”就是这么工作的,它可识别 30 多种编码。`Chardet` 是一个可在程序中使用的 Python 库,不过也提供了一个命令行实用工具——`chardetect`。如下是 `chardetect` 对本章源文件的检测报告。

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

尽管二进制序列编码的文本通常不包含明确的编码提示,但 UTF 格式可以在文本内容的开头添加一个“字节序标记(Byte-Order Mark)”(详见“[4.5.5 BOM:有用的鬼符](#)”)。

4.5.5 BOM:有用的鬼符

示例 4.4(4.4 节)中,您可能已注意到在 UTF-16 编码的序列开头有几个额外的字节。在这里,它们又出现了:

```
1  >>> u16 = 'El Niño'.encode('utf_16')
2
3  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

我指的是 `b'\xff\xfe'`。这是一个 BOM,即字节序标记(Byte-Order Mark),表示执行编码时使用的是 Intel CPU 小端(little-endian)字节序。

在小端(little-endian)设备中,各个码点的最低有效字节在前,最高有效字节在后。例如,字母‘E’的码点是 U+0045(十进制 69),在字节偏移 2 和 3 处⁵被编码为 69 与 0。

```
1  >>> list(u16)
2  [255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

⁵字节偏移的 0、1 处(即 255,254)为字节序标记(Byte-Order Mark)。

而在大端 (big-endian) CPU 上, 编码顺序正好反过来: 码点的最高有效字节在前, 最低有效字节在后。如, 字母 'E' 将被编码为 0 和 69。

为了避免混淆, UTF-16 编码会在要编码的文本前加上特殊的不可见字符 ZERO WIDTH NO-BREAK SPACE (U+FEFF)。在小端 (little-endian) 设备中, 此字符被编码为 b'\xff\xfe' (十进制 255,254)。因为按照设计, Unicode 标准中没有 U+FFFE 字符, 因此在小端字节序编码的设备上, 字节序列 b'\xff\xfe' 必定是字符 ZERO WIDTH NO-BREAK SPACE (即 U+FEFF)。所以, 编码解码器才能知道使用的哪种字节序。

UTF-16 有两个变种:

- UTF-16LE, 显式指明使用小端 (little-endian) 字节序;
- UTF-16BE, 显式指明使用大端 (big-endian) 字节序;

若直接使用这两个变种, 则在编码的字节序列中不会生成 BOM。如下所示, 为字母 'E' 的编码序列 (码点 U+0045, 即十进制 69)。

```
1 >>> u16le = 'El Niño'.encode('utf_16le')
2 >>> list(u16le)      # 小端字节序: 低有效位在前, 高有效位在后。
3 [69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
4 >>> u16be = 'El Niño'.encode('utf_16be')
5 >>> list(u16be)      # 大端字节序: 高有效位在前, 低有效位在后。
6 [0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

若存在 BOM, 那么 UTF-16 编码解码器会将开头的 ZERO WIDTH NO-BREAK SPACE 字符去掉, 只提供文件中真正的文本内容。根据 Unicode 标准, 若文件是 UTF-16 编码且未含 BOM, 则应假定它是 UTF-16BE (big-endian)。但是, Intel x86 架构是小端 (little-endian) 设备, 因此有很多文件用的是不带 BOM 的小端字节序 UTF-16 编码。

字节序仅对一个字 (word) 占多个字节的编码 (如 UTF-16 与 UTF-32) 有影响。UTF-8 的优势是, 无论设备使用哪种字节序, 生成的字节序列始终一致, 因此不需要 BOM。尽管如此, 某些 Windows 应用程序 (尤其是记事本) 还是会在 UTF-8 文件中添加 BOM。Excel 也依赖 BOM 来检测 UTF-8 文件, 否则 Excel 会假定内容是使用 Windows 代码页 (code page) 编码。这种带有 BOM 的 UTF-8 编码, 在 Python 的编解码器注册表中被称为 UTF-8-SIG。字符 U+FEFF 在 UTF-8-SIG 编码中被表示为 3 个字节的序列 b'\xef\xbb\xbf'。因此, 如果一个文件以这 3 个字节开头, 很可能是带有 BOM 的 UTF-8 编码文件。



Caleb 关于 UTF-8-SIG 的提示

本书技术审校 Caleb Hattingh 建议始终使用 UTF-8-SIG 编码解码器读取 UTF-8 文件。这样做是无害的, 因为不管文件中有没有 BOM, UTF-8-SIG 编码解码器都能正确读取文件中的内容, 并且不会返回 BOM 本身。在编写本书时, 我建议用 UTF-8, 以实现通用的互操作性。举个例子, 在 Unix 系统中, 若 Python 脚本以注释 (#!/usr/bin/env python3) 开头, 则可将该文件用作可执行文件。为此, 文件的前 2 个字节必须是 b'#!'。而有了 BOM, 这个约定就被打破了。若您需要导出带有 BOM 的数据, 供其他应用使用, 则应使用 UTF-8-SIG。但是, 请注意 Python 的编解码器文档中说:“在 UTF-8 中, 不鼓励使用 BOM, 通常应避免使用 BOM。”

下面换个话题, 讨论 Python 3 中处理文本文件的方式。

4.6 处理文本文件

处理文本 I/O 的最佳实践是“Unicode 三明治⁶”(如图 4.2 所示)。根据此原则,我们应尽早(如打开文件进行读取时)将输入的 bytes 解码为 str。三明治中的“馅”即是应用程序的业务逻辑,其中的文本处理完全由 str 对象完成。决不能在其他处理过程中进行解码或编码。在输出时, str 应尽可能晚地被编码为 bytes。大多数的 Web 框架都是这样工作的,在使用框架的过程中,我们很少接触到 bytes。例如,在 Django 中,视图应该输出 Unicode 字符串(str);Django 框架本身负责将响应编码为 bytes,默认使用 UTF-8 编码。

在 Python 3 中,我们可以轻松地采纳“Unicode 三明治”的建议,因为内置函数 open() 会在以文本模式读取文件时进行必要的解码,并在以文本模式写入文件时进行必要的编码。所以,调用 my_file.read() 方法得到的,以及传给 my_file.write(text) 方法的都是 str 对象。

因此,处理文本文件其实很简单。但是,若依赖默认编码处理文本文件,就可能会遇到麻烦。

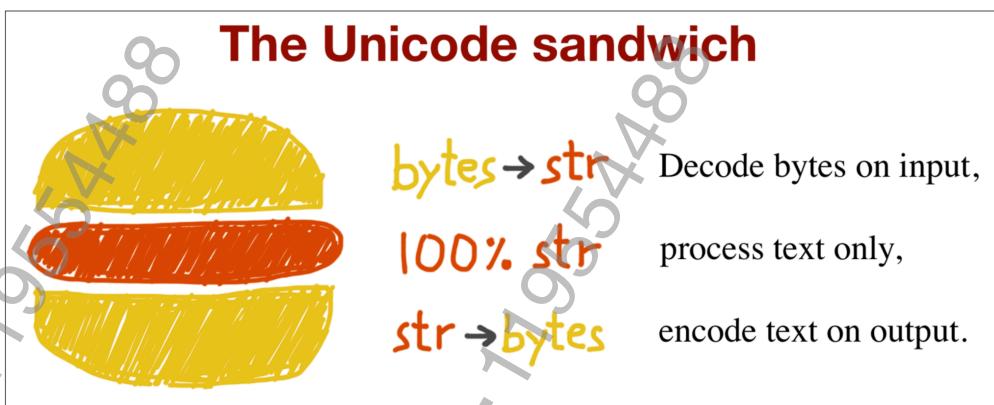


图 4.2: Unicode 三明治:当前文本处理的最佳实践。

看一下 [示例 4.8](#) 中的控制台会话。您能发现其中的 bug 么?

</> [示例 4.8](#): 一个平台的编码问题

```

1  >>> open('cafe.txt', 'w', encoding='utf_8').write('café')
2
3  >>> open('cafe.txt').read()
4  'cafÃ©'

```

[示例 4.8](#) 中的 bug 是:我在写文件时指定了 UTF-8 编码,但在读取文件时却没有这样做,因此 Python 假定使用 Windows 系统的默认编码 (code page 1252),导致文件的最后一个字节被解码为字符“Ã©”,而不是“é”。

我在 64 位 Windows 10 (build 18363) 中用 Python 3.8.1 运行 [示例 4.8](#)。在新版 GNU/Linux 或 macOS 中运行同样的语句则没有问题。因为,GNU/Linux 或 macOS 中的默认编码是 UTF-8,让人误以为一切正常。若在打开文件进行写入时,省略了参数 encoding,则 Python 会使用区域设置中的默认编码,并且也会使用相同的编码正确读取文件。但是这样一来,脚本生成的字节内容就会因平台而异。即使在同一个平台中,由于区域设置不同,也会产生兼容性问题。

⁶我第一次看到“Unicode 三明治”这个术语,是 Ned Batchelder 在 PyCon 2012 做的演讲“Pragmatic Unicode”上。



需要在多台设备或多种场合下运行的代码,绝不能依赖 encoding 默认值。打开文件时,应始终明确指定参数 encoding。因为不同的设备使用的默认编码可能会不同,有时隔一天也会发生变化。

示例 4.8 中一个奇怪的细节是:第一个语句中的 write 函数汇报写入了 4 个字符,但是下一行读取时得到了 5 个字符。示例 4.9 在示例 4.8 的基础上增加了一些代码,对此问题以及其他细节做了说明。

</> 示例 4.9: 仔细分析在 Windows 中运行的示例 4.8,找出问题并修正。

```
1  >>> fp = open('cafe.txt', 'w', encoding='utf_8')
2  >>> fp
3  <_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
4  >>> fp.write('café')      ❷
5
6  >>> fp.close()
7  >>> import os
8  >>> os.stat('cafe.txt').st_size
9  5
10 >>> fp2 = open('cafe.txt')
11 >>> fp2
12 <_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
13 >>> fp2.encoding      ❸
14 'cp1252'
15 >>> fp2.read()        ❹
16 'cafÃo'
17 >>> fp3 = open('cafe.txt', encoding='utf_8')    ❷
18 >>> fp3
19 <_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
20 >>> fp3.read()        ❸
21 'café'
22 >>> fp4 = open('cafe.txt', 'rb')                 ❹
23 >>> fp4
24 <_io.BufferedReader name='cafe.txt'>
25 >>> fp4.read()          ❽
26 b'caf\xc3\xa9'          ❽
```

- ❶ 默认情况下,open 使用文本模式,并返回一个具有特定编码的 TextIOWrapper 对象。
- ❷ TextIOWrapper 的 write 方法会返回写入的 Unicode 字符数。
- ❸ os.stat 显示该文件有 5 个字节;UTF-8 将 “é” 编码为 2 个字节,即 0xc3 和 0xa9。
- ❹ 打开文本文件,未显式指定 encoding 参数,将返回一个 TextIOWrapper 对象。该对象使用区域 (locale) 设置中的默认编码。
- ❺ TextIOWrapper 对象有一个 encoding 属性。可以检查该属性,以查看 TextIOWrapper 对象的编码:本例为 cp1252。
- ❻ 在 Windows cp1252 编码中,字节 0xc3 是 “Ã”(带浪线的 A),0xa9 是版权符号 ©。
- ❼ 使用正确的编码,打开同一文件。
- ❽ 结果复合预期,得到 4 个 Unicode 字符,即 ‘café’。

- ❾ 'rb' 标志以二进制模式打开文件, 以供读取。
- ❿ 返回的对象是 BufferedReader, 而不是 TextIOWrapper。
- ⓫ 读取返回的字节序列, 结果与预期相符。



除非想确定文件编码, 否则不要以二进制模式打开文本文件。即便真想确定文件编码, 也应使用 `Chardet`, 而不应重复造轮子 (参见 “[4.5.4 如何找出字节序列的编码](#)”)。一般来说, 二进制模式只用于打开二进制文件, 如光栅图像。

[示例 4.9](#) 中的问题, 与打开文本文件时所依赖的默认设置有关。此默认设置有多个来源, 详见下节。

4.6.1 注意默认编码

有几项设置会影响 Python 中 I/O 的默认编码, 请参阅 [示例 4.10](#) 中的脚本 `default_encodings.py`。

</> [示例 4.10: 探索编码默认设置](#)

```

1 import locale
2 import sys
3
4 expressions = """
5     locale.getpreferredencoding()
6     type(my_file)
7     my_file.encoding
8     sys.stdout.isatty()
9     sys.stdout.encoding
10    sys.stdin.isatty()
11    sys.stdin.encoding
12    sys.stderr.isatty()
13    sys.stderr.encoding
14    sys.getdefaultencoding()
15    sys.getfilesystemencoding()
16 """
17
18 my_file = open('dummy', 'w')
19
20 for expression in expressions.split():
21     value = eval(expression)
22     print(f'{expression:>30} -> {value!r}')

```

[示例 4.10](#) 在 GNU/Linux (Ubuntu 14.04 至 19.10) 与 macOS (10.9 至 10.14) 上的输出完全相同, 表明这些系统都是使用 UTF-8 (如下所示)。

```

1 $ python3 default_encodings.py
2 locale.getpreferredencoding() -> 'UTF-8'
3     type(my_file) -> <class '_io.TextIOWrapper'>
4     my_file.encoding -> 'UTF-8'
5     sys.stdout.isatty() -> True
6     sys.stdout.encoding -> 'utf-8'

```

```
7     sys.stdin.isatty() -> True
8     sys.stdin.encoding -> 'utf-8'
9     sys.stderr.encoding -> 'utf-8'
10    sys.getdefaultencoding() -> 'utf-8'
11    sys.getfilesystemencoding() -> 'utf-8'      sys.stderr.isatty() -> True
```

但是,在 Windows 系统中,其输出结果如 [示例 4.11](#) 所示。

</> [示例 4.11](#): Windows 10 PowerShell 上的默认编码 (cmd.exe 上的输出相同)

```
1  > chcp
2  Active code page: 437
3  > python default_encodings.py ❶
4  locale.getpreferredencoding() -> 'cp1252' ❷
5      type(my_file) -> <class '_io.TextIOWrapper'>
6      my_file.encoding -> 'cp1252' ❸
7      sys.stdout.isatty() -> True ❹
8      sys.stdout.encoding -> 'utf-8' ❺
9      sys.stdin.isatty() -> True
10     sys.stdin.encoding -> 'utf-8'
11     sys.stderr.isatty() -> True
12     sys.stderr.encoding -> 'utf-8'
13     sys.getdefaultencoding() -> 'utf-8'
14     sys.getfilesystemencoding() -> 'utf-8'
```

- ❶ chcp 显示控制台当前的活动代码页 (code page):437
- ❷ 运行 default_encodings.py, 并将结果输出到控制台。
- ❸ locale.getpreferredencoding() 是最重要的设置。
- ❹ 文本文件默认使用 locale.getpreferredencoding() 编码。
- ❺ 将结果发送到控制台, 因此 sys.stdout.isatty() 返回 True。
- ❻ 现在, sys.stdout.encoding 的值与 chcp 汇报的控制台代码页 (code page) 不同!

自从本书第 1 版出版以来, Windows 自身以及 Python for Windows 对 Unicode 的支持都有所改善。[示例 4.11](#) 曾经在 Windows 7 上的 Python 3.4 中报告了 4 种不同的编码。在过去, stdout、stdin 和 stderr 的编码与 chcp 命令报告的活动代码页相同。但现在它们全都是 utf-8, 这要归功于 Python 3.6 中实施的 “[PEP 528 - Change Windows console encoding to UTF-8](#)”, 以及 cmd.exe 中的 PowerShell 对 Unicode 的支持 (自 2018 年 10 月 Windows 1809 起)⁷。当标准输出流写入控制台时, chcp 和 sys.stdout.encoding 会显示不同的内容, 这是很奇怪的, 但很好的是, 现在我们可以在 Windows 上打印 Unicode 字符串而不会出现编码错误。除非用户将输出重定向到文件中, 我们很快就会看到这一点。然而, 这并不意味着你所有喜欢的表情符号都会在控制台中出现, 这还取决于控制台所使用的字体。

另一个变化是同在 Python 3.6 中实现的 “[PEP 529 - Change Windows filesystem encoding to UTF-8](#)”, 它将文件系统编码 (用于表示目录和文件名称) 从微软专有的 MBCS 更改为 UTF-8。

但是, 如果将 [示例 4.10](#) 的输出重定向到一个文件, 则会出现这样的情况:

```
Z:\>python default_encodings.py > encodings.log
```

⁷ 来源: [Windows Command-Line: Unicode and UTF-8 Output Text Buffer](#)

然后, `sys.stdout.isatty()` 的值变为 `False`; `sys.stdout.encoding` 由 `locale.getpreferredencoding()` 进行设置, 在该机器上为 '`cp1252`'; 但 `sys.stdin.encoding` 与 `sys.stderr.encoding` 仍为 `utf-8`。



在 [示例 4.12](#) 我用 “`backslashN{}`” 转义来处理 Unicode 字面量, 括号内是字符的官方名称。这样做可能比较麻烦, 但是更完全。若字符名称不存在, Python 会引发 `SyntaxError` 异常——这比写入一个十六进制数(易出错, 且不易察觉)要友好的多。而且, 有时您可能想在注释中说明字符代码的意思, 直接使用 `\N{}` 很容易被接受。

这意味着, [示例 4.12](#) 的脚本在将输出打印到控制台中, 可以正常工作。但是, 将输出重定向到文件时, 就可能遇到问题。

</> [示例 4.12: stdout_check.py](#)

```

1  import sys
2  from unicodedata import name
3
4  print(sys.version)
5  print()
6  print('sys.stdout.isatty():', sys.stdout.isatty())
7  print('sys.stdout.encoding:', sys.stdout.encoding)
8  print()
9
10 test_chars = [
11     '\N{HORIZONTAL ELLIPSIS}',      # cp1252 中存在, cp437 中不存在
12     '\N{INFINITY}',                # cp437 中存在, cp1252 中不存在
13     '\N{CIRCLED NUMBER FORTY TWO}', # cp437 与 cp1252 中都不存在
14 ]
15
16 for char in test_chars:
17     print(f'Trying to output {name(char)}:')
18     print(char)

```

[示例 4.12](#) 显示了 `sys.stdout.isatty()` 与 `sys.stdout.encoding` 的值, 以及如下 3 个字符。

- ‘...’ HORIZONTAL ELLIPSIS, 存在于 CP 1252 中, 但不存在于 CP 437 中。
- ‘∞’ INFINITY, 存在于 CP 437 中, 但不存在于 CP 1252 中。
- ‘㉚’ CIRCLED NUMBER FORTY TWO, CP 1252 与 CP 437 中均不存在。

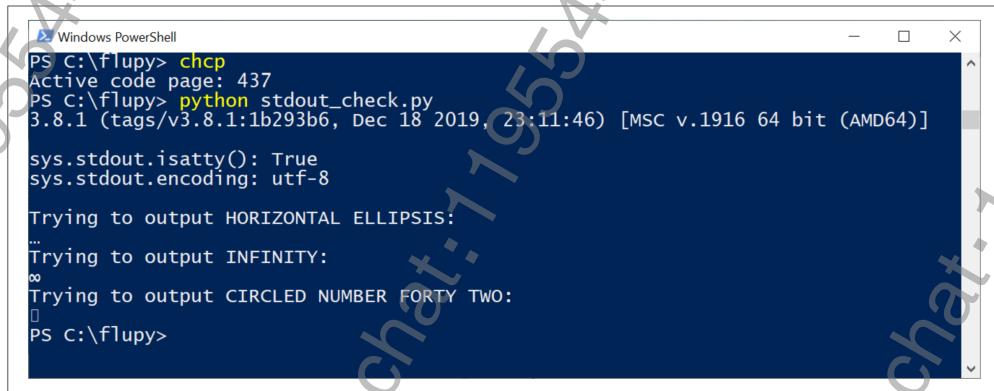
当我在 PowerShell 或 cmd.exe 上运行 `stdout_check.py` 时, 输出效果如 [图 4.3](#) 所示。

尽管 `chcp` 报告活动代码为 437, 但 `sys.stdout.encoding` 为 `UTF-8`, 因此 `HORIZONTAL ELLIPSIS` 和 `INFINITY` 都能正确输出。`CIRCLED NUMBER FORTY TWO` 被一个矩形取代, 但没有出错。据推测, 它被识别为一个有效字符, 但控制台字体中没有显示它的字形。

但是, 当我将 `stdout_check.py` 的输出重定向到一个文件时, 得到的结果如 [图 4.4](#) 所示。

[图 4.4](#) 演示的第一个问题是提及字符 “`\u221e`” 的 `UnicodeEncodeError`。, 因为 `sys.stdout.encoding` 是 “`cp1252`”——一个没有 `INFINITY` 字符的代码页。

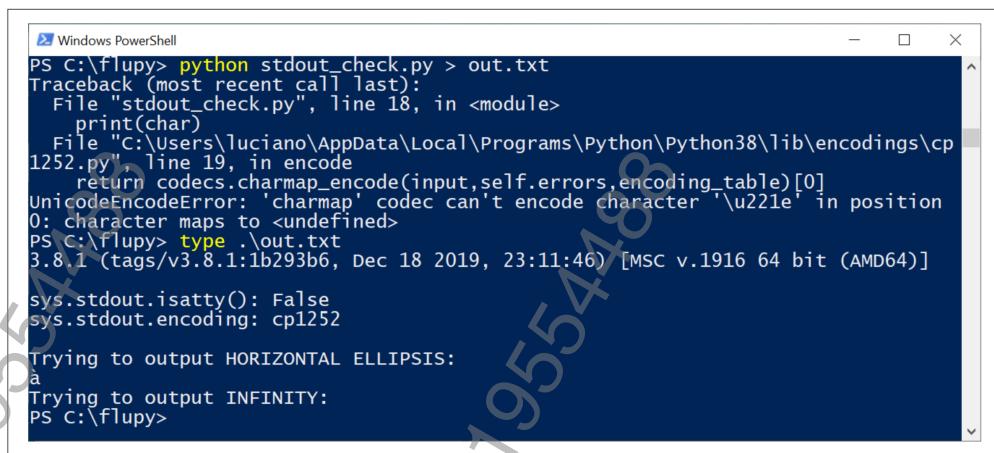
使用 `type` 命令 (或 VS Code、Sublime Text 等 Windows 编辑器) 读取 `out.txt`, 您会发现 `HORIZONTAL ELLIPSIS` 被显示为 “`à`” (LATIN SMALL LETTER A WITH GRAVE)。事实证明, CP 1252 中的字节值 0x85 表



```
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
□
PS C:\flupy>
```

图 4.3: PowerShell 上运行 stdout_check.py



```
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp
1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position
0: character maps to <undefined>
PS C:\flupy> type ..\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
a
Trying to output INFINITY:
PS C:\flupy>
```

图 4.4: PowerShell 上运行 stdout_check.py, 重定向输出到文件

示“...”，但在 CP 437 中却表示“à”。可见，活动代码页似乎确实很重要，尽管不是很合理，也没有多少用。但却初步解释了 Windows 中糟糕的 Unicode 体验。



我使用了一台为美国市场配置的笔记本电脑，运行 Windows 10 OEM 来进行这些实验。其他国家本地化的 Windows 版本可能有不同的编码配置。例如，在巴西，Windows 控制台默认使用代码页 850，而不是 437。

为了解决这个令人抓狂的默认编码问题，让我们最后看一下 [示例 4.11](#) 中的不同编码。

- 如果在打开文件时省略 encoding 参数，则默认值将由 `locale.getpreferredencoding()` 提供（[示例 4.11](#) 中为 ‘cp1252’）。
- 在 Python 3.6 之前，`sys.stdout/stdin/stderr` 的编码是由 `PYTHONIOENCODING` 环境变量设置的。现在，除非将 `PYTHONLEGACYWINDOWSSTUDIO` 设置为非空字符串，否则该变量将被忽略。否则，交互式环境下的标准 I/O 使用 UTF-8 编码；重定向到文件的 I/O，则使用由 `locale.getpreferredencoding()` 定义的编码。
- `sys.getdefaultencoding()` 被 Python 内部用于二进制数据与 str 之间的隐式转换。不支持更改此设置。
- `sys.getfilesystemencoding()` 用于对文件名（而非文件内容）进行编码/解码。它在 `open()` 获得文件名的 str 参数时使用；如果文件名作为字节参数给出，它将原封不动地传递给操作系统 API。



在 GNU/Linux 和 macOS 上, 所有这些编码都默认设置为 UTF-8, 而且几年来一直如此, 因此 I/O 可以处理所有 Unicode 字符。而在 Windows 系统中, 不仅同一系统中使用不同的编码, 而且这些编码通常都是"cp850" 或"cp1252" 等仅支持 ASCII 的代码页, 另外还有 127 个不同编码的字符。因此, Windows 用户更容易遇到编码错误, 除非他们格外小心。

总而言之, 最重要的编码设置是 `locale.getpreferredencoding()` 返回的编码: 它是打开文本文件的默认编码, 也是 `sys.stdout/ stdin/stderr` 重定向到文件时的默认编码。然而, [locale.getpreferredencoding 文档](#) 中 (部分) 写道:

```
locale.getpreferredencoding(do_setlocale=True)
```

根据用户偏好, 返回文本数据使用的编码。用户偏好在不同的系统上有不同的表达方式, 在某些系统上可能无法通过编程实现, 因此该函数只是返回一个猜测的编码 ...

因此, 关于编码默认值的最佳建议是: 不要依赖默认编码。

如果你能遵循“Unicode 三文治”的建议, 在程序中始终明确指定编码, 就能避免很多痛苦。可惜的是, 即使将 `bytes` 正确转换为 `str`, `Unicode` 仍有一些不尽如人意的地方。接下来的“[4.7 Unicode 字符规范化](#)”与“[4.8 Unicode 文本排序](#)”所讨论的主题, 对 ASCII 来说很简单, 但对 `Unicode` 来说却是相当复杂。

4.7 Unicode 字符规范化

由于 `Unicode` 中有组合字符 (附加在前一个字符上的分音符和其他标记, 在打印时显示为一个字符, 如 é 由字母 'e' 和分音符 " " 组成), 所以字符串比较会相对复杂。

例如, 单词 “café” 的构成方式可能包含两种, 分别编码为 4 个码点、5 个码点。但是, 显示结果看起来完全一样。

```
1  >>> s1 = 'café'
2  >>> s2 = 'cafe\u0301N{COMBINING ACUTE ACCENT}'
3  >>> s1, s2
4  ('café', 'café')
5  >>> len(s1), len(s2)
6  (4, 5)
7  >>> s1 == s2
8  False
```

将 `COMBINING ACUTE ACCENT` (`U+0301`) 放在字母 “e”的后面, 就会得到字符 “é”。在 `Unicode` 标准中, 像 “é” 与 “e\u0301” 这样的序列, 被称为“规范等价物 (Canonical Equivalents,)”, 应用程序应该将它们视为相同的字符。但 Python 看到的是两个不同的码点序列, 所以认为二者是不同的。

此问题的解决方案是使用 `unicodedata.normalize()` 函数。该函数的第一个参数是“`NFC`”、“`NFD`”、“`NFKC`”、“`NFKD`”四者之一。

先讲解前两个:“`NFC`”与“`NFD`”。`NFC` (Normalization Form C) 将码点 (Code Point) 组合起来, 以生成最短的等效字符串。而 `NFD` 对码点进行分解, 将合成字符扩展为基本字符与单独的组合字符。这两种规范化方式都能得到复合预期的比较效果, 如下所示。

```

1  >>> from unicodedata import normalize
2  >>> s1 = 'café'
3  >>> s2 = 'cafe\N{COMBINING ACUTE ACCENT}'
4  >>> len(s1), len(s2)
5  (4, 5)
6  >>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
7  (4, 4)
8  >>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
9  (5, 5)
10 >>> normalize('NFC', s1) == normalize('NFC', s2)
11 True
12 >>> normalize('NFD', s1) == normalize('NFD', s2)
13 True

```

键盘驱动程序通常会生成组合字符,因此用户输入的文本默认采用 NFC 格式。但是,为了安全起见,最好在保存之前使用 normalize('NFC', user_text) 对字符串进行规范化。NFC 也是 W3C 在《Character Model for the World Wide Web: String Matching》中推荐的规范化形式。

NFC 有时会将一些单字符规范化为另一个单字符。例如,电阻单位 Ω (U+2126,名称 OHM SIGN) 会被规范化为大写希腊字母 Ω (U+03A9, Unicode 名称为 GREEK CAPITAL LETTER OMEGA)。二者在视觉上是一样的,但在比较时并不相等。因此,要规范化以防止出现意外。

```

1  >>> from unicodedata import normalize, name
2  >>> ohm = '\u2126'                      # 电阻单位: Ω (欧 姆)
3  >>> name(ohm)
4  'OHM SIGN'
5  >>> ohm_c = normalize('NFC', ohm)      # 将电阻单位字符规范化为大写希腊字母 Ω (Omega)
6  >>> name(ohm_c)
7  'GREEK CAPITAL LETTER OMEGA'
8  >>> ohm == ohm_c
9  False
10 >>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
11 True

```

另外两种规范化形式是 NFKC 与 NFKD,其中字母 K 表示“兼容性 (Compatibility)”。这两种规范化形式较为严格,会影响到所谓的“兼容性字符”。尽管 Unicode 的目标是为每个字符提供一个“统一的”码点,但为了与现有标准兼容,某些字符会出现多次。例如,尽管 Unicode 中已有了希腊字母 “ μ ” (U+03BC, 名称为 GREEK SMALL LETTER MU),但是为了便于与 latin1 相互转换,Unicode 还是添加了微符号 “ μ ” (U+0085, 名称为 MICRO SIGN)。因此,微符号 “ μ ” (MICRO SIGN) 被视为“兼容性字符”。

在 NFKC 与 NFKD 形式中,兼容字符经过兼容性分解,被替换成一个或多个字符。即使这样会存在一些格式损失,但仍是“首选”的表示形式。在理想情况下,格式化应是外部标记语言的职责,而不应由 Unicode 处理。举两个例子,二分之一 $\frac{1}{2}$ (U+00BD) 经过兼容性分解后,得到 3 个字符序列,即 “1/2”;“微”符号 “ μ ” (U+00B5) 经过兼容性分解后,得到小写字母 “ μ ” (U+03BC)⁸。

以下是 NFKC 规范化的效果:

⁸“微”符号 “ μ ” (U+00B5) 是兼容性字符,而欧姆符号 “ Ω ” (U+2126) 却不是,真是奇怪。最终结果是, NFC 不会改动“微”符号,而是将欧姆符号 “ Ω ” (U+2126) 转为大写希腊字母 “ Ω ” (Omega, U+03A9)。而 NFKC 和 NFKD 会将欧姆和微符号都更改为希腊字符。

```

1  >>> from unicodedata import normalize, name
2  >>> half = '\N{VULGAR FRACTION ONE HALF}'
3  >>> print(half)
4 ½
5  >>> normalize('NFKC', half)
6  '1/2'
7  >>> for char in normalize('NFKC', half):
8      ...     print(char, name(char), sep='\t')
9  ...
10 1 DIGIT ONE
11 0 FRACTION SLASH
12 2 DIGIT TWO
13 >>> four_squared = '4²'
14 >>> normalize('NFKC', four_squared)
15  '42'
16 >>> micro = 'μ'
17 >>> micro_kc = normalize('NFKC', micro)
18 >>> micro, micro_kc
19  ('μ', 'μ')
20 >>> ord(micro), ord(micro_kc)
21  (181, 956)
22 >>> name(micro), name(micro_kc)
23  ('MICRO SIGN', 'GREEK SMALL LETTER MU')

```

尽管 '1/2' 替代 $\frac{1}{2}$ 是可接受的，“微”符号 μ 也确实是小写希腊字母 ‘μ’。但是，将 ‘4²’ 转换成 ‘42’ 就改变原意了。应用程序可将 ‘4²’ 保存为 ‘4²’，但是 `normalize` 函数对格式一无所知。因此，NFKC 或 NFKD 就可能会损失或曲解信息，但二者可以为搜索和索引提供方便的中间表示形式。

遗憾的是，对于 Unicode 来说，看似简单的事情往往会更加复杂。 $\frac{1}{2}$ (U+00BD, VULGAR FRACTION ONE HALF) 经 NFKC 规范化后，1 与 2 之间用的是字符 ‘/’ (U+2044, FRACTION SLASH)，而不是 ‘/’ (U+002F, SOLIDUS)，即我们熟悉的 ASCII 字符 ‘/’ (十进制代码 47)。因此，若用户搜索由 3 个 ASCII 字符序列构成的 ‘1/2’，则不会找到规范化之后的 Unicode 序列 (即找不到 $\frac{1}{2}$)。



NFKC 和 NFKD 规范化会导致数据丢失，因此仅可用于搜索和索引等特殊情况，而不可用于文本的永久存储。

当为搜索或索引准备文本时，还有一种比较有用的操作，即 4.7.1 所介绍的“大小写同一化”。

4.7.1 大小写同一化

大小写同一化本质上是将所有文本转换为小写，并进行一些额外的转换。此特性由 `str.casefold()` 方法提供支持。

对于仅包含 latin1 字符的字符串 `s`，`s.casefold()` 生成的结果与 `s.lower()` 相同。只有两个例外：“微”符号 μ 将变成小写希腊字母 “μ” (在多数字体中，二者看起来一样)；德语中的 Eszett 或 “sharp s” (ß) 将变成 “ss”。

```
1  >>> micro = 'μ'
2  >>> name(micro) 'MICRO SIGN'
3  >>> micro_cf = micro.casefold()
4  >>> name(micro_cf)
5  'GREEK SMALL LETTER MU'
6  >>> micro, micro_cf
7  ('μ', 'μ')
8  >>> eszett = 'ß'
9  >>> name(eszett)
10 'LATIN SMALL LETTER SHARP S'
11 >>> eszett_cf = eszett.casefold()
12 >>> eszett, eszett_cf
13 ('ß', 'ss')
```

`str.casefold()` 与 `str.lower()` 返回不同结果的码点有近 300 个。

与 Unicode 相关的其他问题一样, 大小同一化也很复杂, 有很多语言层面的特殊情况, 但 Python 核心团队努力提供了一个解决方案, 希望能适用于大多数用户。

在接下来的几节中, 我们将运用规范化知识来开发几个实用函数。

4.7.2 规范化文本匹配的实用函数

由前文可知, 我们可以安全的使用 NFC 与 NFD 规范化形式进行 Unicode 字符串比较, 二者可得到一致而准确的结果。对多数应用程序来说, NFC 是最好的规范化形式; 若进行不区分大小写的比较, 则应使用 `str.casefold()`。

如需处理多语言文本, 则建议在您的工具箱中增加如 [示例 4.13](#) 所示的工具函数:`nfc_equal` 与 `fold_equal`。

</> [示例 4.13: normeq.py: 规范化 Unicode 字符串比较](#)

```
1 """
2 规范化 Unicode 字符串的实用函数, 确保得到准确的比较结果。
3 使用 NFC 规范化形式, 区分大小写:
4   >>> s1 = 'café'
5   >>> s2 = 'cafe\u0301'
6   >>> s1 == s2
7   False
8   >>> nfc_equal(s1, s2)
9   True
10  >>> nfc_equal('A', 'a')
11  False
12 使用NFC规范化形式, 不区分大小写:
13  >>> s3 = 'Straße'
14  >>> s4 = 'strasse'
15  >>> s3 == s4
16  False
17  >>> nfc_equal(s3, s4)
18  False
19  >>> fold_equal(s3, s4)
20  True
```

```

21     >>> fold_equal(s1, s2)
22     True      >>> fold_equal('A', 'a')
23     True
24     """
25     from unicodedata import normalize
26     def nfc_equal(str1, str2):
27         return normalize('NFC', str1) == normalize('NFC', str2)
28
29     def fold_equal(str1, str2):
30         return (normalize('NFC', str1).casefold() == normalize('NFC', str2).casefold())

```

除了 Unicode 规范化和不区分大小写 (它们都是 Unicode 标准的一部分) 之外, 有时还需要进行更深层次的转换, 例如将 “café” 改为 “cafe”。下一节将了解何时以及如何进行这样的转换。

4.7.3 极端规范化: 去掉变音符

Google 搜索的秘诀有很多, 但其中一个显然是忽略变音符 (如重音符号、音节等)。删除变音符号并不是一种正确的规范化形式, 因为它往往会影响单词的含义, 并可能在搜索时产生误报。但它也有助于应对生活中的一些事实: 人们有时会懒惰或不了解变音符号的正确使用, 而且拼写规则会随着时间的推移而变化, 这意味着重音在生活语言中可能时有时无。

除了搜索之外, 去掉变音符号还可以提高 URL 的可读性。至少在基于拉丁语的语言中是这样的, 比如维基百科中有关圣保罗市 (São Paulo) 的 URL:

https://en.wikipedia.org/wiki/S%C3%A3o_Paulo

%C3%A3 部分是 URL 编码的 UTF-8 表示方式, 代表单个字母 “ã” (带波浪符的 “a”)。下面的拼写虽然不正确, 但更容易识别:

https://en.wikipedia.org/wiki/Sao_Paulo

要从 str 中删除所有变音符号, 可以使用 [示例 4.14](#) 中的函数。

</> [示例 4.14: simplify.py: 去除所有组合标记的函数](#)

```

1 import unicodedata
2 import string
3
4 def shave_marks(txt):
5     """去除所有变音符"""
6     norm_txt = unicodedata.normalize('NFD', txt) ❶
7     shaved = ''.join(c for c in norm_txt if not unicodedata.combining(c)) ❷
8     return unicodedata.normalize('NFC', shaved) ❸

```

- ❶ 将所有字符分解为基本字符和组合标记。
- ❷ 过滤掉所有组合标记。
- ❸ 重新组合所有字符。

[示例 4.15](#) 演示了 [示例 4.14](#) 中函数 shave_marks 的效果。

</> [示例 4.15: \[示例 4.14\]\(#\) 中函数 shave_marks 的使用示例](#)

```

1  >>> order = ' "Herr Voß: • % cup of Etker™ caffè latte • bowl of açai. " '
2  >>> shave_marks(order)
3  ' "Herr Voß: • % cup of Etker™ caffè latte • bowl of açai. " ' ❶
4  >>> Greek = 'Ζέφυρος, Ζέφυρο'
5  >>> shave_marks(Greek)
6  'Ζέφυρος, Ζέφυρο' ❷

```

- ❶ 仅替换了字母“è”、“ç”和“í”。
 ❷ “é”和“é”均被替换。

示例 4.14 中的函数 `shave_marks` 虽然可以正常工作,但可能做得太过了。通常情况下,移除变音符是为了将拉丁文本转换为纯 ASCII 文本。但 `shave_marks` 还会转换非拉丁字符(如希腊字母),这类字符永远不会因为失去变音符号而变成 ASCII 字符。因此,仅当基字符在拉丁字母表中时,移除附加标记(如变音符)才有意义。详见 [示例 4.16](#)。

</> [示例 4.16](#): 从拉丁字符中删除组合标记的函数⁹

```

1  import unicodedata
2  import string
3
4  def shave_marks_latin(txt):
5      """移除基字符为拉丁字符的变音符"""
6      norm_txt = unicodedata.normalize('NFD', txt) ❶
7      latin_base = False
8      preserve = []
9
10     for c in norm_txt:
11         if unicodedata.combining(c) and latin_base: ❷
12             continue # 忽略(移除)拉丁基字符的变音符
13             preserve.append(c) ❸
14         # 若不是组合字符,那么就是新的基字符
15         if not unicodedata.combining(c): ❹
16             latin_base = c in string.ascii_letters
17             shaved = ''.join(preserve) ❺
18             return unicodedata.normalize('NFC', shaved)

```

- ❶ 将所有字符分解为基本字符和组合标记。
 ❷ 当基字符是拉丁字符时,跳过组合标记,即移除组合标记。
 ❸ 否则,保留当前字符。
 ❹ 检测新的基字符,并确定其是否为拉丁字符。
 ❺ 重新组合所有字符。

更激进的做法是将西方文本中的常用符号(如花引号、长破折号、列表项符号等¹⁰)替换为 ASCII 对应符号。这也是 [示例 4.17](#) 中函数 `asciiize` 的作用。

⁹省略了 `import` 模块导入部分,因此此代码片段是[示例 4.14](#)中 `simplify.py` 的一部分。

¹⁰Curly quotes (花引号“”):是在英语等西方语言中使用的引号形式。

Em dashes (长破折号—):通常用于表示一个短暂的停顿,或引出一个附加的说明或强调。在 LaTeX 中,可以使用两个连字符(–)来表示 Em dash。

Bullets (列表项符号•):Bullets 是用来表示列表项的符号,常用于有序或无序列表中。

- ① 构建“字符到字符 (char-to-char)”的替换映射表。
 - ② 构建“字符到字符串 (char-to-string)”的替换映射表。
 - ③ 合并映射表。
 - ④ 函数 `dewinize` 不影响 ASCII 或 latin1 文本, 仅替换 Microsoft 在 cp1252 中为 latin1 额外添加的字符。
 - ⑤ 先调用函数 `dewinize` 函数, 再去掉变音符。
 - ⑥ 将德语 Eszett (ß) 替换为 “ss” (此处未使用 `str.casefold()`, 因为我们想保留大小写)。
 - ⑦ 应用 NFKC 规范化, 将字符和与其兼容的码点 (Code Point) 组合起来。

示例 4.18 演示了函数 `asciize` 的效果。

```
</> 示例 4.18: 示例 4.17 中函数 asciiize 的使用示例

1 >>> order = ' "Herr Voß: • ½ cup of Etker™ caffè latte • bowl of açai." '
2 >>> dewinize(order)
3 'Herr Voß: - ½ cup of Etker(TM) caffè latte - bowl of açai.'
4 >>> asciiize(order)
5 'Herr Voss: - 1/2 cup of Etker(TM) caffe latte - bowl of acai.'
```

- ① 函数 `dewinize` 可替换 Curly quotes (花引号“”、列表项符号 (•) 和商标符号 (™))。
 - ② 函数 `asciize` 中, 调用函数 `dewinize`。不仅去除了变音符号(`), 还将'ß'替换为了ss。

¹¹此代码片段是示例 4.14 中 simplify.py 的一部分。



不同的语言有不同的变音符号去除规则,例如,德语会将“ü”改为“ue”。我们定义的函数 `asciize` 功能暂时不够完善,可能不适合您的语言。但是,对葡萄牙语的处理效果,还是可以接受的。

总而言之, `simplify.py` 中函数做的事情远远超出了标准规范化的范围。它们会对文本进行深度处理,很有可能改变文本原意。只有了解目标语言、目标用户以及转换后的用途,才能确定要不要做这么深入的规范化。

我们对 Unicode 文本规范化的讨论到此结束。接下来,探讨 Unicode 文本的排序问题。

4.8 Unicode 文本排序

Python 通过逐一比较序列中的每一项,来对任何类型的序列进行排序。对于字符串,这意味着逐一比较每个字符的码点 (Code Point)。不幸的是,这对于使用非 ASCII 字符的人来说,会产生不可接受的结果。

下面对巴西种植的水果列表 `fruits` 进行排序。

```
1 >>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
2 >>> sorted(fruits)
3 ['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

不同区域采用的排序规则有所不同,葡萄牙语等语言按拉丁字母表进行排序,重音符与变音符在排序时几乎没什么区别¹²。因此,“cajá”按“caja”来排序,并且必须位于常规词“caju”之前,但是位于常规词“caja”之后。

排序后的列表 `fruits` 应如下所示:

```
1 ['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

在 Python 中对非 ASCII 文本进行排序的标准方法是使用 `locale.strxfrm` 函数。根据 `locale` 模块文档的描述,该函数可“将字符串转换为可用于本地化比较的字符串”。

在使用 `locale.strxfrm` 函数之前,必须先为应用程序设置一个合适的区域设置 (locale),并确保操作系统支持该设置。示例 4.19 也许可以做到这一点。

</> 示例 4.19: `locale_sort.py`: 使用 `locale.strxfrm` 函数作为排序键

```
1 import locale
2 my_locale = locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
3 print(my_locale)
4 fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
5 sorted_fruits = sorted(fruits, key=locale.strxfrm)
6 print(sorted_fruits)
```

在安装了 `pt_BR.UTF-8` 本地语言的 GNU/Linux (Ubuntu 19.10) 上运行 示例 4.19,我得到了正确的结果:

```
1 'pt_BR.UTF-8'
2 ['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

¹²只有在极少数情况下(当变音符号是两个单词之间的唯一区别时),变音符号才会影响排序——在此情况下,带有变音符号的单词会被排在不带变音符号的常规词之后。

因此,在使用 `locale.strxfrm` 作为排序键之前,需要调用 `setlocale(LC_COLLATE, "your_locale")`。不过,有一些注意事项:

- 区域设置是全局生效,因此不建议在库中调用 `locale.setlocale` 函数。应用程序或框架应在进程启动时设定区域设置 (`locale`),并且之后不再更改。
- 区域设置 (`locale`) 必须已安装在系统中,否则 `locale.setlocale` 会引发 `locale.Error: unsupported locale setting` 异常。
- 必须知道如何拼写区域设置 (`locale`) 名称。
- 操作系统制造商必须正确实施区域设置 (`locale`)。我在 Ubuntu 19.10 上取得了成功,但在 macOS 10.14 上却失败了。在 macOS 上,调用 `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` 成功返回了字符串" `pt_BR.UTF-8`"。但是, `sorted(fruits, key=locale.strxfrm)` 却产生了与 `sorted(fruits)` 相同的错误结果。我还在 macOS 上尝试了 `fr_FR`、`es_ES` 和 `de_DE` 区域设置,但 `locale.strxfrm` 均未生效¹³。

因此,标准库提供的国际化排序方案是有效的,但是似乎只在 GNU/Linux 上可得到良好的支持。即便如此,也还是要依赖区域设置 (`locale`),这会为部署带来麻烦。幸运的是,有一个更简单的方案可用,即 PyPI 提供的 `pyuca` 库。

4.8.1 用 Unicode 排序算法排序

James Tauber (多产的 Django 贡献者) 一定是受到了 Unicode 字符排序这一痛点的困扰,因此开发了 `pyuca` 库——单纯使用 Python 实现了 Unicode 排序算法 (Unicode Collation Algorithm, UCA)。示例 4.20 展示了这个库使用起来多么简单。

</> 示例 4.20: 使用 `pyuca.Collator.sort_key` 方法

```

1  >>> import pyuca
2  >>> coll = pyuca.Collator()
3  >>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
4  >>> sorted_fruits = sorted(fruits, key=coll.sort_key)
5  >>> sorted_fruits
6  ['açaí', 'acerola', 'atemoia', 'cajá', 'caju']

```

这么排序很简单,并且可以在 GNU/Linux、macOS 和 Windows 上运行——至少对于这个简短示例来说,排序结果是正确的。

`pyuca` 库 不考虑区域设置 (`locale`)。若要自定义排序,可向 `Collator()` 构造函数提供自定义排序表的路径,默认使用的排序表是项目自带的 `allkeys.txt`,这是 Unicode.org 网站提供的“Default Unicode Collation Element Table”的副本。

¹³同样,我未找到解决方案,不过我发现其他人也报告了同样的问题。本书的技术审校 Alex Martelli 在他的 Macintosh (macOS 10.9) 上使用 `setlocale` 和 `locale.strxfrm` 没有任何问题。可见,结果因人而异。



PyICU: Miro 对 Unicode 排序的建议

(技术审校 Miroslav Šedivý 是一位精通多种语言的 Unicode 方面的专家。这是他写的关于 `pyuca` 的文章)

`pyuca` 有一种排序算法, 不遵循各个语言的排序顺序。例如, 德语中的 Ä 位于 A 和 B 之间, 而在瑞典语中则在 Z 之后。此种情况, 则建议使用 `PyICU`, 这个库的工作方式与 `locale` 类似, 但无需更改进程的语言环境。若您想更改土耳其语中 ī/İ 的大小写, 也需要使用 `PyICU`。`PyICU` 中包含一个必须编译的扩展, 因此在某些系统中可能比 `pyuca` (仅需 Python) 更难安装。

顺便说一下, 排序表 (`allkeys.txt`) 是构成 Unicode 数据库 (下一节的主题) 的众多数据文件之一。

4.9 Unicode 数据库

Unicode 标准提供了一个完整的数据库 (以多个结构化文本文件的形式存在), 其中不仅包括将 **码点** (Code Point) 映射到字符名称的映射表, 还包括各个字符的元数据以及字符之间的关系。例如, Unicode 数据库记录了字符是否可打印、是否是字母、是否是十进制数字, 或是否是其他数字字符, 这也是 `str` 方法 `isprintable`、`isalpha`、`isdecimal` 和 `isnumeric` 的工作原理。`str.casefold()` 还使用 Unicode 表中的信息。



`unicodedata.category(char)` 函数返回 `char` 在 Unicode 数据库中的类别 (以 2 个字母表示)。使用高级的 `str` 方法判断字符类别更简单。例如, 若 `label` 中的每个字符都属于如下类别之一: `Lm`、`Lt`、`Lu`、`LI` 或 `Lo`, 则 `label.isalpha()` 返回 `True`。关于 `Lm`、`Lt`、`Lu`、`LI` 或 `Lo` 这些类别代码的含义, 请参阅维基百科 “[Unicode character property](#)” 的 “[General Category](#)” 部分。

4.9.1 按名称查找字符

`unicodedata` 模块 中提供了一些可用于获取字符元数据的函数, 包括 `unicodedata.name()`, 该函数返回字符在 Unicode 标准中的正式名称。图 4.5¹⁴演示了该函数。

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ä')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♛')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

图 4.5: 在 Python 控制台中探索 `unicodedata.name()`

¹⁴这是一个图片,而不是代码块。因为我撰写本书时 O'Reilly 的数字出版工具链还不支持表情符号。

可以使用 `unicodedata.name()` 函数构建应用程序,让用户通过名称搜索字符。图 4.6 演示了 `cf.py` 命令行脚本,该脚本将一个或多个单词作为参数,并列出官方 Unicode 名称中包含这些单词的字符。`cf.py` 的完整源代码见 [示例 4.21](#)。

```
$ ./cf.py cat smiling
U+1F638 😺 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 😻 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😻 SMILING CAT FACE WITH HEART-SHAPED EYES
```

图 4.6: 使用 `cf.py` 查找微笑的猫



不同操作系统和应用程序对表情符号的支持差异很大。近年来,macOS 终端对表情符号的支持最好,其次是现代 GNU/Linux 图形终端。Windows cmd.exe 和 PowerShell 现在虽然支持 Unicode 输出,但在我撰写本书时(即 2020 年 1 月)仍然无法显示表情符号——至少不能“开箱即用”。本书技术审校 Leonardo Rochael 告诉我,微软推出了一款新的开源 Windows Terminal,它对 Unicode 的支持可能优于旧版的微软控制台。但我还没来得及尝试。

在 [示例 4.21](#) 中,请注意 `find` 函数中的 `if` 语句使用了 `issubset()` 方法来快速测试查询集中的所有单词是否都出现在根据字符名称建立的单词列表中。得益于 Python 丰富的集合 API,我们不需要嵌套的 `for` 循环和另一个 `if` 语句,即可实现这种检查。

</> [示例 4.21: cf.py:字符查找工具](#)

```
1  #!/usr/bin/env python3
2
3  import sys
4  import unicodedata
5
6
7  START, END = ord(' '), sys.maxunicode + 1          ❶
8
9  def find(*query_words, start=START, end=END):        ❷
10    query = {w.upper() for w in query_words}           ❸
11
12    for code in range(start, end):                     ❹
13      char = chr(code)                                ❺
14      name = unicodedata.name(char, None)              ❻
15      if name and query.issubset(name.split()):        ❼
16        print(f'U+{code:04X}\t{char}\t{name}')          ❽
17
18  def main(words):
19    if words:
20      find(*words)
21    else:
22      print('Please provide words to find.')
23
24  if __name__ == '__main__':
25    main(sys.argv[1:])
```

- ❶ 设置要搜索的 码点 (Code Point) 范围的默认值。
- ❷ 函数 `find` 接受 `query_words` 和可选的仅关键字参数来限制搜索范围, 以方便测试。
- ❸ 将 `query_words` 转换为一组大写字符串。
- ❹ 获取 `code` 对应的 Unicode 字符。
- ❺ 获取字符的名称。若 码点 (Code Point) 不对应任何字符, 则返回 “None”。
- ❻ 若字符名称存在, 则将名称拆分成单词列表。然后, 检查 `query` 集合是不是该列表的子集。
- ❼ 以 `U+9999` 格式打印带有 码点 (Code Point) 的行、字符及其名称。

`unicodedata` 模块 还提供了一些其他有趣的函数。小节 4.9.2 将介绍其中的几个函数, 用于从具有数字含义的字符中获取信息。

4.9.2 字符的数字含义

`unicodedata` 模块 包含一些函数, 用于检查 Unicode 字符是否表示数字。若表示数字, 则检查其对人类可读的具体数值(而不是码点编号)。示例 4.22 展示了 `unicodedata.name()` 和 `unicodedata.numeric()` 函数, 以及 `str.isdecimal()` 和 `str.isnumeric()` 方法的使用。

```
</> 示例 4.22: Unicode 数据库中数字字符的元数据示例
1 import unicodedata
2 import re
3
4 re_digit = re.compile(r'\d')
5
6 sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'
7
8 for char in sample:
9     print(f'U+{ord(char):04x}', ❶
10           char.center(6), ❷
11           're_dig' if re_digit.match(char) else '-',
12           'isdig' if char.isdigit() else '-',
13           'isnum' if char.isnumeric() else '-',
14           f'{unicodedata.numeric(char):.2f}', ❸
15           unicodedata.name(char), ❹
16           sep='\t') ❺
```

- ❶ 格式为 `U+0000` 的 码点 (Code Point)。
- ❷ 在长度为 6 的字符串中, 居中显示字符。
- ❸ 如果字符与 `r'\d'` 正则表达式匹配, 则显示 `re_dig`。
- ❹ 如果 `char.isdigit()` 为 `True`, 则显示 `isdig`。
- ❺ 如果 `char.isnumeric()` 为 `True`, 则显示 `isnum`。
- ❻ 数字格式为: 总长度为 5, 小数点后保留 2 位小数。
- ❼ Unicode 标准中的字符名称。

如果您的终端字体包含所有这些字形, 则运行 示例 4.22 会得到 图 4.7。

图 4.7 的第 6 列是在字符上调用 `unicodedata.numeric(char)` 的结果。这表明 Unicode 知道表示

图 4.7: macOS 终端显示数字字符及其元数据;re_dig 表示该字符正则表达式 `r'\d'` 匹配。

数字的符号的数值。因此,如果您想创建一个支持泰米尔 (Tamil) 数字或罗马数字的电子表格应用程序,那就试试吧。

图 4.7 表明,正则表达式 `r'\d'` 可匹配阿拉伯数字 “1” 与梵文数字 “3”,但是不能匹配 `isdigit` 方法认为是数字的其他字符。可见,`re` 模块对 Unicode 的支持并不充分。PyPI 有一个新的 `regex` 模块,旨在取代 `re` 模块,可提供更好的 Unicode 支持¹⁵。我们将在 “4.10 支持 str 和 bytes 的双模式 API” 中回过头来继续讨论 `re` 模块。

在本章中,我们使用了几个 `unicodedata` 函数,但还有许多函数我们没有涉及。请参阅标准库中 “`unicodedata` 模块文档” 进一步学习。

接下来,我们将快速了解双模式 API,这些 API 提供了可接受 `str` 或 `bytes` 参数的函数,并可根据参数类型进行特殊处理。

4.10 支持 str 和 bytes 的双模式 API

Python 标准库中有一些可接受 `str` 或 `bytes` 类型参数的函数。它们会根据参数类型的不同,而有不同的行为。可在 `re` 和 `os` 模块中找到相关示例。

4.10.1 正则表达式中的 str 与 bytes

如果使用 `bytes` 构建正则表达式,则诸如 `\d` 和 `\w` 这样的模式只能匹配 ASCII 字符;相反,如果以 `str` 的形式给出这些模式,则它们可以匹配 ASCII 以外的 Unicode 数字或字母。示例 4.23 和 图 4.8 比较了 `str` 和 `bytes` 模式如何匹配字母、ASCII 数字、上标和泰米尔数字。

</> 示例 4.23: ramanujan.py: 比较简单 str 和 bytes 正则表达式的行为

```

1 import re
2
3 re_numbers_str = re.compile(r'\d+')
4 re_words_str = re.compile(r'\w+')
5 re_numbers_bytes = re.compile(rb'\d+')
6 re_words_bytes = re.compile(rb'\w+')
7
8 text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef")

```

❶❷❸

¹⁵对于示例 4.22 来说, `regex` 模块 在数字识别方面的表现并不比 `re` 模块好多少。

```

9      " as 1729 = 1³ + 12³ = 9³ + 10³.")
10     text_bytes = text_str.encode('utf_8')      ❸
11
12     print(f'Text\n {text_str!r}')            ❹
13     print('Numbers')
14     print(' str :', re_numbers_str.findall(text_str)) ❽
15     print(' bytes:', re_numbers_bytes.findall(text_bytes)) ❾
16     print('Words')
17     print(' str :', re_words_str.findall(text_str)) ❽
18     print(' bytes:', re_words_bytes.findall(text_bytes)) ❾

```

- ❶ 前两个正则表达式都是 str 类型。
- ❷ 前两个正则表达式都是 bytes 类型。
- ❸ 要搜索的 Unicode 文本,包含 1729 的泰米尔 (Tamil) 数字 (逻辑行一直到右括号符号为止)。
- ❹ 该字符串在编译时与前一个字符串拼接起来 (参见 《Python 语言参考》 中的 “2.4.2. String literal concatenation”)
- ❺ 使用 bytes 正则表达式进行搜索时,需要一个 bytes 字符串。
- ❻ str 模式 `r'\d+'` 会匹配泰米尔 (Tamil) 语和 ASCII 数字。
- ❼ bytes 模式 `rb'\d+'` 仅匹配数字的 ASCII 字节。
- ❽ str 模式 `r'\w+'` 可匹配字母、上标、泰米尔 (Tamil) 语和 ASCII 数字。
- ❾ bytes 模式 `rb'\w+'` 仅匹配字母和数字的 ASCII 字节。

```

$ python3 ramanujan.py
Text
'Ramanujan saw களைக் as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
 str : ['களைக்', '1729', '1', '12', '9', '10']
 bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
 str : ['Ramanujan', 'saw', 'களைக்', 'as', '1729', '1³', '12³', '9³', '10³']
 bytes: [b'Ramanujan', b'saw', b'களைக்', b'as', b'1729', b'1', b'12', b'9', b'10']
$ 

```

图 4.8: 示例 4.23 ramanujan.py 脚本得到的结果截图

示例 4.23 只是一个简单的例子,其目的是说明一个问题:您可以在 str 和 bytes 上使用正则表达式。但在 bytes 上使用正则表达式时,ASCII 范围之外的字节将被视为非数字和非单词字符。

对于 str 正则表达式,有一个 `re.ASCII` 标志可以使 `\w`、`\W`、`\b`、`\B`、`\d`、`\D`、`\s` 和 `\S` 只匹配 ASCII 字符。详见 `re` 模块文档。

另一个重要的 str 与 bytes 双模式 API 模块是 `os` 模块。

4.10.2 os 函数中的 str 与 bytes

GNU/Linux 内核对 Unicode 的支持不够完善。因此,在实际应用中,你可能会遇到一些由字节序列组成的文件名,这些字节序列在任何编码方案中都无法解码为字符串。尤其是在用客户端连接多种操作系统的文件服务器时,此问题尤为常见。

为了解决此问题,所有接受文件名或路径名的 `os` 模块 函数都将参数视为 `str` 或 `bytes`。如果使用 `str` 参数调用此类函数,参数将自动使用 `sys.getfilesystemencoding()` 指定的编解码器进行转换,并且操作系统响应也将使用相同的编解码器进行解码。这也几乎就是我们想要的行为,符合 Unicode 三明治(见 4.6 节的图 4.2)最佳实践。

但是,如果您必须处理(可能是为了修正)无法以上述方式自动处理的文件名时,则可以将 `bytes` 参数传递给 `os` 函数以获取 `bytes` 返回值。如此一来,您便可以处理任何文件名或路径名(不管其中包含多少鬼符),如示例 4.24 所示。

</> 示例 4.24: 分别将 `str` 参数、`bytes` 参数传入 `listdir`,并查看结果

```
1 >>> os.listdir('.')
2 ['abc.txt', 'digits-of-π.txt']
3 >>> os.listdir(b'.')
4 [b'abc.txt', b'digits-of-\\xcf\\x80.txt']
```

❶ 第 2 个文件名是“`digits-of-□.txt`”(带有希腊字母 pi)。

❷ 如果给定一个 `bytes` 参数, `listdir` 会以 `bytes` 形式返回文件名: `b'\\xcf\\x80'` 是希腊字母 pi 的 UTF-8 编码。

为了帮助手动处理文件名或路径名的 `str` 或字节序列,`os` 模块提供了特殊的编码和解码函数 `os.fsecode(name_or_path)` 和 `os.fsdecode(name_or_path)`。自 Python 3.6 起,这两个函数可接受的参数为:`str`、`bytes` 或实现 `os.PathLike` 接口的对象。

Unicode 是一个非常复杂且深奥的主题。现在,是时候结束对 `str` 和 `bytes` 的探索了

4.11 本章小结

本章一开始,我们就否定了“1字符 == 1字节”的概念。随着世界采用 Unicode,我们需要将文本字符串的概念与在文件中表示它们的二进制序列分开,Python 3 强化了这种分离。

在简要概述了二进制序列数据类型(字节、字节数组和内存视图)之后,我们开始讲解编码和解码。通过示例,介绍了一些重要的编解码器。随后,介绍了如何防止和处理 Python 源文件中因错误编码而导致的 `UnicodeEncodeError`、`UnicodeDecodeError` 和 `SyntaxError`。

然后,我们探讨了在没有元数据的情况下进行编码检测的理论和实践:理论上讲,这是不可能做到的;但实际上,Chardet 软件包对一些常用编码的检测效果相当不错。随后,我们介绍了字节序标记,这是 UTF-16 和 UTF-32 文件中常见的唯一编码提示(有时在 UTF-8 文件中也能找到)。

在接下来的“4.6 处理文本文件”中,演示了虽然打开文本文件很简单,但是却隐藏了一个陷阱:在打开文件时,参数 `encoding` 并不是必需的(但应该指定)。若未指定 `encoding` 参数,你可能会遇到在不同平台上无法兼容的“纯文本”生成问题。然后,我们介绍了 Python 默认使用的不同编码设置,以及如何检测它们。对于 Windows 用户来说,一个悲哀的事实是,这些设置通常在同一台机器上有不同的值,而且这些值互不兼容;相比之下,GNU/Linux 和 macOS 用户则要幸福多了,几乎所有地方都默认使用 UTF-8 编码。

Unicode 为某些字符提供了多重表示方式,因此规范化是文本匹配的先决条件。除了解释规范化和大小写同一化(大小写不敏感)外,还介绍了一些实用函数,可以根据自己需要进行改写,比如去掉所有重音符的转换函数。然后,我们还介绍了如何利用标准库中的 `locale` 模块 对 Unicode 文本进行正确排序(但有一些注

意事项), 以及不依赖于区域设置 (locale) 的替代排序方案——外部 `pyuca` 包。

最后, 得益于 Python 的强大功能, 我们利用 Unicode 数据库编写了一个命令行实用程序, 只需 28 行代码就能按 Unicode 字符的名称搜索字符。我们还浏览了其他 Unicode 元数据, 并简要了解了双模式 API, 其中有些函数可以接受 `str` 或 `bytes` 参数, 从而根据不同的参数类型产生不同的结果。

4.12 延伸阅读

Ned Batchelder 在 2012 年美国 PyCon 上的演讲 “Pragmatic Unicode, or, How Do I Stop the Pain?” 非常出色。Ned 在提供幻灯片和视频的同时, 还提供了完整的文字记录。

“Character encoding and Unicode in Python: How to (╯°□°)╯︵ ┻━┻ with dignity” (幻灯片, 视频) 是 Esther Nam 和 Travis Fischer 在 PyCon 2014 上发表的精彩演讲。在该演讲中, 我找到了本章精辟的题记: “人类使用文本, 而计算机使用字节”。

Lennart Regebro (本书第 1 版的技术审校之一) 在短文 “Unconfusing Unicode: What Is Unicode?” 中提出 “Useful Mental Model of Unicode (UMMU)” 这一概念。Unicode 是一个复杂的标准, 因此 Lennart 的 UMMU 是一个很好的切入点。

Python 文档中的 “Unicode HOWTO” 部分, 从多个不同的角度对本章所涉及的主题做了介绍。涵盖了历史简介、语法细节、编解码器、正则表达式、文件名, 以及 Unicode I/O 最佳实践 (即 Unicode 三明治)。每个部分都提供了大量的参考资料链接。“《Dive Into Python 3》” (Mark Pilgrim 著) 的 “第 4 章字符串” 也精彩地介绍了 Python 3 对 Unicode 的支持。此外, 该书的 “第 15 章” 介绍了将 `Chardet` 库从 Python 2 移植到 Python 3 的过程, 这是一个很有价值的案例。因为, 从旧的 `str` 类型切换到新的 `bytes` 类型, 是大多数迁移过程中引起困扰的主要原因, 而这也正是一个用于检测编码的库 (如 `Chardet`) 的核心关注点。

若您很熟悉 Python 2, 但对 Python 3 较为陌生, 可以阅读 Guido van Rossum 写的 “What’s New in Python 3.0”。该文简要列出了 Python 3 的 15 点变化, 并提供了大量的参考链接。Guido 开门见山的说道: “你自己以为熟悉的二进制数据和 Unicode 知识全都变了。”Armin Ronacher 的博客文章 “The Updated Guide to Unicode on Python” 深入浅出地强调了 Python 3 中 Unicode 的一些陷阱 (Armin 并不是 Python 3 的忠实粉丝)。

《Python Cookbook,3rd ed》¹⁶ (O’Reilly) 的 “第 2 章. 字符串与文本” 中, 有几个经典示例谈到了 Unicode 规范化、文本清洗, 以及在字节序列上执行面向文本的操作。第 5 章介绍了文件与 I/O, 其中的 “5.17. 将字节写入文本文件” 指出, 任何文本文件的底层都有一个二进制流。该二进制流在需要时, 可以直接访问。该书的 “6.11. 读写二进制结构数组” 还用到了 `struct` 模块。

Nick Coghlan 的 “Python Notes” 博客有 2 篇与本章密切相关的文章: “Python 3 and ASCII Compatible Binary Protocols” 与 “Processing Text Files in Python 3”。强烈推荐阅读。

Python 支持的编码列表, 参见 `codecs` 模块文档 中的 “Standard Encodings” 一节。如果您需要以编程方式获取该列表, 请参阅 CPython 源代码 中的 `/Tools/unicode/listcodecs.py` 脚本。

《Unicode Explained》(Jukka K. Korpela 著) 与 《Unicode Demystified》(Richard Gillam 著) 这两本书虽然不是专门针对 Python 的, 但是为我学习 Unicode 相关概念提供了很大的帮助。《Programming with Unicode》(Victor Stinner 著) 是一本可免费阅读的自出版图书 (Creative Commons BY-SA)。该书涵盖了

¹⁶ 《Python Cookbook,3rd ed》由 David Beazley 和 Brian K. Jones 编写, 其中文版《Python Cookbook 中文版 (第 3 版)》链接为 <https://item.jd.com/13897579.html>

Unicode 的一般性话题,还介绍了主流操作系统与一些编程语言(包括 Python)相关的工具和 API。

W3C 网站中的“Case Folding: An Introduction¹⁷”与“Character Model for the World Wide Web: String Matching”页面涵盖了规范化相关的概念。前者是一个简单的入门指南,而后者则是以枯燥的标准用语编写的工作草案——与“Unicode® Standard Annex #15—UNICODE NORMALIZATION FORMS”的风格一致。Unicode.org 网站的“Frequently Asked Questions: Normalization”一节更具可读性,Mark Davis¹⁸写的“NFC FAQ”也不错。

2016 年,纽约现代艺术博物馆 (Museum of Modern Art, MoMA) 收藏了最初的 176 个表情符号。这些表情符号由 Shigetaka Kurita 在 1999 年为日本移动运营商 NTT DOCOMO 设计的。追溯历史,Emojipedia 发表了《Correcting the Record on the First Emoji Set》,称日本软银 (SoftBank) 于 1997 年在手机上部署了已知最早的表情符号集。SoftBank 的那套表情包有 90 个表情符号,现已纳入 Unicode 标准。例如,U+1F4A9 (💩,PILE OF POO)。Matthew Rothenberg 创建的 emojitracker.com 实时更新 Twitter 上表情符号的使用次数。在我写下这段话时,Twitter 最流行的表情符号是“😂,FACE WITH TEARS OF JOY (U+1F602)”,使用量超过 3,313,667,315 次。

杂谈

源代码中的非 ASCII 名称:您应该使用它们吗?

Python 3 允许在源代码中使用非 ASCII 标识符:

```
1  >>> ação = 'PBR' # ação = stock
2  >>> ε = 10**-6 # ε = epsilon
```

有些人不喜欢使用非 ASCII 标识符。其中最常见的理由是,坚持使用 ASCII 标识符可以使代码易于阅读和编辑。然而,这个观点忽略了一个关键点:你希望你的源代码能够被其预期受众阅读和编辑,而这个受众可能并不是“每个人”。对于国际化的项目或者涉及多语言的开源项目来说,使用英文作为标识符可以方便来自不同国家和文化背景的开发者参与。如此一来,使用 ASCII 字符就足够满足需求了,而无需使用更复杂的 Unicode 字符作为标识符。

但如果您是巴西的一名教师,您的学生可能更喜欢阅读用葡萄牙语命名的变量和函数(当然,拼写要正确)。而且,他们在本地化键盘上输入变音符和重读元音也不会有任何困难。

既然 Python 可以解析 Unicode 名称,UTF-8 也是源代码的默认编码方式。那么,我认为没有必要像 Python 2 中那样,用不带重音符号的葡萄牙语编码标识符(除非要在 Python2 中运行代码)。如果标识符使用葡萄牙语命名,并且省略了重音符号,那么对任何人来说,代码的可读性都不会有所改善。

这是我作为一个讲葡萄牙语的巴西人的观点,但我相信它适用于不同的国界和文化:应该选择能让团队更容易阅读代码的人类语言,然后使用正确的字符拼写。

什么是纯文本 (Plain Text)

对于每天处理非英语文本的人来说,“纯文本”并不仅仅指“ASCII”。Unicode 术语表对纯文本的定义如下:

¹⁷W3C 中“Case Folding: An Introduction”页面的大部分内容已被“Character Model for the WWW: String Matching”所取代。

¹⁸Mark Davis 是多个 Unicode 算法的作者,在撰写本文时他还担任 Unicode Consortium 的主席。

仅由给定标准的一系列 码点 (Code Point) 组成的计算机编码文本, 没有其他格式或结构信息。

此定义的前半句很贴切, 但我不认同逗号后面的部分。HTML 是纯文本格式的一个很好的例子, 它包含格式和结构信息。但它仍然是纯文本, 因为 HTML 文件中的每个字节都表示一个文本字符(通常使用 UTF-8)。纯文本中的字节不具有非文本含义, 与诸如.png 或.xls 文档不同。在这些文件中, 大多数字节代表打包的二进制值(如 RGB 值和浮点数)。在纯文本中, 数字被表示为数字字符的序列。

我在编写本书时, 使用的是一种纯文本格式——AsciiDoc, 它是 O'Reilly 出色的 *Atlas 图书出版平台* 工具链的一部分。AsciiDoc 源文件是 UTF-8(而不是 ASCII)的纯文本。否则, 写这一章会非常痛苦。尽管名字叫 AsciiDoc, 但它确实是一款很棒的工具。

Unicode 的世界正在不断地扩张, 但有些边缘场景仍缺少工具支持。比如, 本书使用的字体中可能没有我想使用的字符。这就是为什么我必须在本章的一些示例中使用图片, 而不使用代码块的原因。不过, Ubuntu 与 macOS 终端可以正确显示大多数的 Unicode 文本, 包括“mojibake”一词的日语字符: 文化字け。

str 的码点 (Code Point) 在 RAM 中如何表示?

Python 官方文档回避了“如何在内存中存储字符串的 码点 (Code Point) ”的问题。这实际上是一个实现细节。理论上, 这并不重要: 无论内部表示形式如何, 每个 str 都必须在输出时被编码成字节。

在内存中, Python 3 将每个 str 存储为一个 码点 (Code Point) 序列, 每个 码点 (Code Point) 使用固定的字节数, 以便高效地直接访问任何字符或切片。

自 Python 3.3 起, 当创建一个新的 str 对象时, 解释器会检查其中的字符, 并选择适合该特定 str 的最经济的内存布局: 如果只有 latin1 范围内的字符, 则该 str 的每个 码点 (Code Point) 只使用 1 个字节。否则, 根据 str 的不同, 每个 码点 (Code Point) 可能使用 2 个或 4 个字节。这只是一个简要说明, 完整信息请查看“[PEP 393 –Flexible String Representation](#)”。

Python 3 中的灵活字符串表示方式, 与 Python 3 中整数类型(int)的工作方式类似。若一个整数在一个 机器字 (Machine Word) 中放得下, 则存储在一个机器字中; 否则, 解释器会切换到变长表示方式, 类似于 Python 2 中的长整型(long type)。这种聪明的做法能得到推广, 真是让人欣喜!

然而, 我们总是可以依靠 Armin Ronacher 来发现 Python 3 中的问题。他向我解释了为什么在实践中这样做并不是一个很好的想法: 在一个原本全是 ASCII 字符的文本中添加一个 RAT 字符(U+1F400), 内存中存储各个字符的数组会立刻变大。原本, 每个字符只占 1 字节, 而添加 RAT 字符后, 每个字符都占 4 字节。而除了 RAT 字符之外, 每个字符只需 1 字节就足够了。此外, 由于 Unicode 字符的组合方式很多, 按位置快速检索任意字符就没那么容易了; 从 Unicode 文本中提取切片也是幼稚的做法, 往往结果都是错误的, 会产生乱码。随着表情符号的流行, 这些问题只会越来越严重。

wechat: 119554488

数据类构建器

数据类就像孩子。作为一个起点,它们是可以的,但要作为一个成年对象参与整个系统的工作,它们需要承担一些责任。

——Martin Fowler 与 Kent Beck^a

^a摘自《重构:改善既有代码的设计》第 1 版,第 3 章,“代码中的异味,数据类”一节,第 87 页。

Python 提供了几种方法来构建一个简单的类。这种简单的类只是字段的集合,几乎没有额外的功能。这种模式 (Pattern) 被称为“数据类”,而 dataclasses 是支持这种模式的包之一。本章将介绍 3 种不同的类构建器,以作为编写数据类的快捷方式。

- `collections.namedtuple`

最简单的数据类构建方式,从 Python 2.6 开始提供。

- `typing.NamedTuple`

构建数据类的另一种方式,需要为数据类中的字段添加 [类型提示 \(Type Hints\)](#) ——从 Python 3.5 开始。在 Python 3.6 中,增加了 class 语法。

- `@dataclasses.dataclass`

一个类装饰器,从 Python 3.7 开始提供。与前两种方式相比,可定制的内容更多,增加了大量选项,可实现更复杂的功能。

在介绍了这些类构建器之后,我们将进一步讨论为什么称之为“数据类 (Data Class)”的编码模式 (Pattern) 可能是一种 [代码异味 \(Code Smell\)](#):即指出这种模式可能是面向对象设计不佳的一个征兆。



`typing.TypedDict` 看起来像是另一种数据类构建器。它的语法与 `typing.NamedTuple` 类似。并且，在 Python 3.9 的 `typing` 模块文档中，二者紧挨在一起。

但是，`TypedDict` 不会构建可以实例化的具体类（Concrete Class）。它只是一种语法，用于为函数参数和变量编写类型提示。这些参数和变量将映射的值用作记录，将映射的键用作字段名。详见“[15.3 TypedDict](#)”。

5.1 本章新增内容

本章是《流畅的 Python》第 2 版中新增的章节。“[5.3 典型的具名元组: collections.namedtuple](#)”原来在第 1 版第 2 章中，除此之外，本章所有内容都是新增的。

首先，我们将对 3 个类构建器做一个高层次的概述。

5.2 数据类构建器概述

[示例 5.1](#) 是一个简单的类，用于表示一个地理坐标的经纬度。

```
</> 示例 5.1: class/coordinates.py
1 class Coordinate:
2
3     def __init__(self, lat, lon):
4         self.lat = lat
5         self.lon = lon
```

`Coordinate` 类负责保存纬度和经度属性。编写 `__init__` 样板代码很快就会过时，特别是如果你的类有多个属性：每个属性都要写 3 次。而且，这些样板代码并不能给我们带来我们期望从 Python 对象中获取的基本功能（如 [示例 5.2](#) 所示）。

```
</> 示例 5.2: 简单类的功能粗糙
1 >>> from coordinates import Coordinate
2 >>> moscow = Coordinate(55.76, 37.62)
3 >>> moscow
4 <coordinates.Coordinate object at 0x107142f10> ❶
5 >>> location = Coordinate(55.76, 37.62)
6 >>> location == moscow ❷
7 False
8 >>> (location.lat, location.lon) == (moscow.lat, moscow.lon) ❸
9 True
```

❶ 从 `object` 继承的 `__repr__`，并没有什么用处。

❷ `==` 没有意义。因为从 `object` 继承而来的 `__eq__` 方法，是比较对象 ID 的。

❸ 比较两个地理坐标，需要显式地逐一比较每个属性。

本章介绍的数据类构建器会自动提供必要的 `__init__`、`__repr__` 和 `__eq__` 方法, 以及其他有用的功能。



本章讨论的类构建器都不依赖于继承来完成工作。`collections.namedtuple` 与 `typing.NamedTuple` 构建的类都是元组的子类。`@dataclass` 是一个类装饰器, 不会以任何方式影响类的层次结构。它们各自使用不同的元编程技术将方法和数据属性注入到正在构造的类中。

示例 5.3 是一个使用 `namedtuple` 构建的 `Coordinate` 类。`namedtuple` 是一个工厂函数 (Factory Function), 它可使用您指定的名称和字段来构建 `tuple` 的子类。

</> 示例 5.3: 用 `namedtuple` 构建 `Coordinate` 类

```
1 >>> from collections import namedtuple
2 >>> Coordinate = namedtuple('Coordinate', 'lat lon')
3 >>> issubclass(Coordinate, tuple)
4 True
5 >>> moscow = Coordinate(55.756, 37.617)
6 >>> moscow
7 Coordinate(lat=55.756, lon=37.617) ❶
8 >>> moscow == Coordinate(lat=55.756, lon=37.617) ❷
9 True
```

❶ 有用的 `__repr__`。

❷ 有意义的 `__eq__`。

新出现的 `typing.NamedTuple` 提供了与 `collections.namedtuple` 相同的功能, 不过为每个字段添加了类型提示 (Type Hints) (如示例 5.4 所示)。

</> 示例 5.4: `typing.NamedTuple` 为字段添加了类型提示

```
1 >>> import typing
2 >>> Coordinate = typing.NamedTuple('Coordinate',
3 ... [('lat', float), ('lon', float)])
4 >>> issubclass(Coordinate, tuple)
5 True
6 >>> typing.get_type_hints(Coordinate)
7 {'lat': <class 'float'>, 'lon': <class 'float'>}
```



`typing.NamedTuple` 也可以通过关键字参数指定数据类的字段, 如下所示:

```
1 Coordinate = typing.NamedTuple('Coordinate', lat=float, lon=float)
```

这样可读性更强, 而且还能以 `**fields_and_types` 的形式, 提供字段与字段类型的映射。

从 Python 3.6 开始, 也可以在 `class` 语句中使用 `typing.NamedTuple`, 并按照 “PEP 526 – Syntax for Variable Annotations” 语法规编写类型提示 (Type Hints)。这样可读性更强, 并且还可以轻松覆盖方法或添加

新方法。如 [示例 5.5](#), 用 class 语句再次定义了 Coordinate 类, 拥有一对 float 类型的属性 (经纬度), 以及一个自定义的 `__str__` 方法, 用于显示格式为 “55.8°N, 37.6°E” 的地理坐标。

```
</> 示例 5.5: typing_namedtuple/coordinates.py
1 from typing import NamedTuple
2
3 class Coordinate(NamedTuple):    # 在 class 语句中使用 typing.NamedTuple
4     lat: float
5     lon: float
6
7     def __str__(self):
8         ns = 'N' if self.lat >= 0 else 'S'
9         we = 'E' if self.lon >= 0 else 'W'
10        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```



在 class 语句中, 虽然 `NamedTuple` 出现在超类的位置上。但实际上, 它并不是超类。`typing.NamedTuple` 使用 [元类 \(MetaClass\)^a](#) 这一高级功能来自定义用户类的创建。看一下这个:

```
1 >>> issubclass(Coordinate, typing.NamedTuple)
2 False
3 >>> issubclass(Coordinate, tuple)
4 True
```

^a元类 (MetaClass) 是 “[二十四类元编程 \(Class Metaprogramming\)](#)” 中的主题之一。

在通过 `typing.NamedTuple` 生成的 `__init__` 方法中, 字段参数的顺序与在 class 语句中出现的顺序相同。

与 `Typing.NamedTuple` 一样, `dataclass` 装饰器也支持用 “[PEP 526](#)” 语法来声明实例属性。`dataclass` 装饰器读取 [变量注解 \(Variable Annotation\)](#), 并自动为构建的类生成方法。作为对比, 请查看借助 `dataclass` 装饰器编写的 `Coordinate` 类 (如 [示例 5.6](#) 所示)。

```
</> 示例 5.6: dataclass/coordinates.py
1 from dataclasses import dataclass
2
3 @dataclass(frozen=True)
4 class Coordinate:
5     lat: float
6     lon: float
7
8     def __str__(self):
9         ns = 'N' if self.lat >= 0 else 'S'
10        we = 'E' if self.lon >= 0 else 'W'
11        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°{we}'
```

请注意, [示例 5.5](#) 与 [示例 5.6](#) 中的类主体完全相同, 区别在于 class 语句本身。`@dataclass` 装饰器不依

赖于继承或元类 (MetaClass)。因此,如果你使用这些机制,也不会受到影响。示例 5.6 中的 Coordinate 类是 object 的子类。

5.2.1 数据类构建器对比

不同的数据类构建器有许多共同点,总结如表 5.1 所示。

表 5.1: 比较 3 个数据类构建器的部分功能 (x 代表此数据类的实例)

功能特性	namedtuple	NamedTuple	dataclass
可变实例	NO	NO	YES
class 语句语法	NO	YES	YES
构造字典	x._asdict()	x._asdict()	dataclasses.asdict(x)
获取字段名	x._fields	x._fields	[f.name for f in dataclasses.fields(x)]
获取默认值	x._field_defaults	x._field_defaults	[f.default for f in dataclasses.fields(x)]
获取字段类型	N/A	x._annotations_	x._annotations_
更改后,创建新实例	x._replace(...)	x._replace(...)	dataclasses.replace(x, ...)
运行时,创建新类	namedtuple(...)	NamedTuple(...)	dataclasses.make_dataclass(...)



通过 `typing.NamedTuple` 与 `@dataclasses.dataclass` 构建的类都包含一个名为 `_annotations_` 的类属性,该属性用于保存字段的类型提示 (Type Hints)。然而,直接访问 `_annotations_` 属性来检索字段类型信息并不推荐。相反,建议使用函数 `inspect.get_annotations(MyClass)` (Python ≥ 3.10) 或 `typing.get_type_hints(MyClass)` (Python 3.5~3.9) 来获取字段类型信息。因为这些函数提供了额外的服务,例如可以解析类型提示 (Type Hints) 中的前向引用 (Forward Reference),详细内容见“15.5.1 运行时的注解问题”。

现在,我们来讨论一下这些主要功能。

1. 可变实例

这些类构建器之间的关键区别是:`collections.namedtuple` 与 `typing.NamedTuple` 构建的是 `tuple` 的子类,因此实例是不可变的。默认情况下,`@dataclasses.dataclass` 生成的数据类是可变的。但该装饰器接受一个关键字参数 `frozen` (如“5.2 数据类构建器概述”的示例 5.6 所示)。当 `frozen=True` 时,将生成一个不可变的数据类,若在实例初始化后试图为字段赋值,将引发异常。

2. class 语句语法

只有 `typing.NamedTuple` 和 `@dataclasses.dataclass` 支持常规的 class 语句语法,从而可以更轻松地将方法和文档字符串添加到正在创建的类中。

3. 构造字典

两个具名元组变体 (即 `collections.namedtuple` 与 `typing.NamedTuple`) 都提供了一个实例方法 (`_asdict`),用于根据数据类实例中的字段构造 `dict` 对象。而 `dataclasses` 模块也提供了函数 `dataclasses.asdict` 来执行此操作。

¹类装饰器与元类都将在“二十四 类元编程 (Class Metaprogramming)”中探讨。这两种机制都提供了超越继承的功能,方便你定制类的行为。

4. 获取字段名与默认值

这三种数据类构建器都支持获取字段名与可能配置的默认值。对于具名元组（即 `collections.namedtuple` 与 `typing.NamedTuple`）数据类，这些元数据位于在类属性 `._fields` 和 `._fields_defaults` 之中。对于用 `dataclass` 装饰器构建的类，这些元数据可以用 `dataclasses` 模块的 `fields` 函数获取。`fields` 函数将返回一个由 `Field` 对象 构成的元组，`Field` 对象具有多个属性，包括 `name` 与 `default` 等。

5. 获取字段类型

用 `typing.NamedTuple` 与装饰器 `@dataclasses.dataclass` 构建的数据类中，有一个类属性 `__annotations__`，该属性值是一个字段名到字段类型的映射。如前所述，不能直接读取 `__annotations__`，而要使用 `inspect.get_annotations(MyClass)`（Python ≥ 3.10 ）或 `typing.get_type_hints(MyClass)`（Python 3.5~3.9）²。

6. 更改后，创建新实例

给定一个具名元组（`collections.namedtuple` 与 `typing.NamedTuple`）实例 `x`，`x._replace(**kwargs)` 会返回一个新实例，并根据关键字参数 `kwargs` 替换实例中的一些属性值。模块级函数 `dataclasses.replace(x, **kwargs)` 对 `dataclass` 装饰器构建的数据类也会执行同样的操作。

7. 运行时，创建新类

虽然具名元组的（`typing.NamedTuple` 与 `@dataclasses.dataclass`）`class` 语句语法可读性更高，但该语句是 **硬编码**（Hardcoded）。当编写 **框架**（Framework）时，可能需要在运行时动态创建数据类。为此，可以使用默认的函数调用语法，即 `collections.namedtuple(...)` 与 `typing.NamedTuple(...).dataclasses` 模块提供的 `make_dataclass` 函数也能实现同样的目的。

在概述了数据类构建器的主要功能之后，下面将逐一讨论这 3 个数据类构建器。先从最简单的 `collections.namedtuple` 开始。

5.3 典型的具名元组：`collections.namedtuple`

`collections.namedtuple()` 是一个 **工厂函数**（Factory Function），用于构建具有字段名、类名、详细 `__repr__` 信息的增强 `tuple` 子类。`collections.namedtuple()` 构建的数据类，可以在任何适用于 `tuple` 的场景下使用。事实上，为了方便，**Python 标准库** 中许多返回 `tuple` 的函数，现在都会返回具名元组。而且，完全不影响用户的代码。



由 `namedtuple` 构建的类，其实例所占用的内存与 `tuple` 完全相同。因为，字段名都存储在 `namedtuple` 构建的类中。

示例 5.7 展示了如何定义一个具名元组来保存有关城市的信息。

```
</>示例 5.7: 定义与使用一个具名元组
1  >>> from collections import namedtuple
2  >>> City = namedtuple('City', 'name country population coordinates')
3  >>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
```

①

②

²关于使用 `__annotations__` 的最佳实践，详见 <https://docs.python.org/3/howto/annotations.html#annotations-howto>

```

4  >>> tokyo
5  City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
6  139.691667))
7  >>> tokyo.population
8  36.933
9  >>> tokyo.coordinates
10 (35.689722, 139.691667)
11 >>> tokyo[1]
12 'JP'

```

- ❶ 创建具名元组需要 2 个参数:类名、字段名列表。后者(字段名列表)可以是可生成多个字符串的可迭代对象,或者是以空格分隔的单个字符串。
- ❷ 字段值必须以单独的位置参数传递给构造函数(相比之下,tuple 构造函数只接受单个可迭代对象)
- ❸ 可以根据字段名称或位置,访问数据类的字段。

作为 tuple 的子类,City 从 tuple 中继承了一些有用的方法,如用于比较运算符的 `__eq__` 和 `__lt__`。`__lt__` 还可用于排序由 City 实例构成的列表。

除了从 tuple 继承的属性和方法之外,具名元组还提供一些额外的属性和方法。示例 5.8 展示了最有用的属性:类属性 `_fields`、类方法 `_make(iterable)`,以及实例方法 `_asdict()`。

</> 示例 5.8: 具名元组最有用的属性及方法(续示例 5.7)

```

1  >>> City._fields
2  ('name', 'country', 'population', 'location') ❶
3  >>> Coordinate = namedtuple('Coordinate', 'lat lon')
4  >>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889, 77.208889))
5  >>> delhi = City._make(delhi_data) ❷
6  >>> delhi._asdict() ❸
7  {'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
8  'location': Coordinate(lat=28.613889, lon=77.208889)}
9  >>> import json
10 >>> json.dumps(delhi._asdict()) ❹
11 '{"name": "Delhi NCR", "country": "IN", "population": 21.935,
12 "location": [28.613889, 77.208889]}'

```

- ❶ 类属性 `_fields` 是一个元组,存储该类的字段名称。
- ❷ 类方法 `_make()` 方法根据可迭代对象构建 City;与 `City(*delhi_data)` 作用相同。
- ❸ 实例方法 `_asdict()` 返回根据具名元组实例构建的 dict 对象,键为具名元组的字段名,值为具名元组的字段值。
- ❹ 实例方法 `_asdict()` 可将数据序列化为 JSON 格式。



在 Python 3.7 之前, `_asdict` 方法返回一个 `OrderedDict`。从 Python 3.8 开始,它返回一个简单的 dict——因为 Python 3.8 开始,dict 可以保留键的插入顺序。若必须使用 `OrderedDict`, [_asdict\(\) 文档](#) 建议使用 `OrderedDict(x._asdict())` 根据 `x._asdict()` 结果自行创建。

从 Python 3.7 开始, namedtuple 只接受 defaults 关键字参数。参数值是一个可产生 N 项的可迭代对象, 可以为具名元组最右边的 N 个字段提供 N 个默认值。示例 5.9 展示了如何定义一个具名元组 Coordinate, 并为字段 reference 指定默认值。

</> 示例 5.9: 构建具名元组, 为字段指定默认值 (续示例 5.8)

```

1 >>> Coordinate = namedtuple('Coordinate', 'lat lon reference', defaults=['WGS84'])
2 >>> Coordinate(0, 0)
3 Coordinate(lat=0, lon=0, reference='WGS84')
4 >>> Coordinate._field_defaults
5 {'reference': 'WGS84'}

```

“5.2.1 数据类构建器对比”中的“class 语句语法”中提到, 用 typing.NamedTuple 与 @dataclasses.dataclass 支持的 class 语法, 可以方便我们编写方法。collections.namedtuple 也可以为具名元组增加方法, 但这只是一个技巧 (hack)。若您对此技巧不感兴趣, 可跳过下面的附注栏。

为具名元组 (collections.namedtuple) 注入方法

回顾以下“— Python 数据模型”中的示例 1.1 是如何构建 Card 类(扑克牌类)的。

```
1 Card = collections.namedtuple('Card', ['rank', 'suit'])
```

在第一章的后面, 我写了一个用于排序扑克牌的 spades_high 函数。若能将这一逻辑封装在 Card 的一个方法中就更好了。然而, Card 类不是用 class 语句构建的, 为其添加 spades_high 方法需要一些技巧(hack)。需要先定义函数 spades_high, 然后将此函数赋值给一个类属性(如 ?? 所示)。

```

1 >>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0) ❶
2 >>> def spades_high(card): ❷
3 ...     rank_value = FrenchDeck.ranks.index(card.rank)
4 ...     suit_value = card.suit_values[card.suit]
5 ...     return rank_value * len(card.suit_values) + suit_value
6 ...
7 >>> Card.overall_rank = spades_high ❸
8 >>> lowest_card = Card('2', 'clubs')
9 >>> highest_card = Card('A', 'spades')
10 >>> lowest_card.overall_rank() ❹
11 0
12 >>> highest_card.overall_rank()
13 51

```

- ❶ 为每种花色附加一个包含值的类属性。
- ❷ spades_high 将成为一个方法; 第一个参数不必命名为 self。无论如何, 当 spades_high 被作为方法调用时它会获取接收者。
- ❸ 将函数 spades_high 作为名为 overall_rank 的方法注入(附加)到 Card 类中。
- ❹ 方法注入成功。

为了可读性和将来的维护, 最好在 class 语句中为 collections.namedtuple 数据类编写方法。不过, 学会这个可行的技巧^a也不是件坏事, 因为不一定什么时候, 这个小技巧可能就会派上用场。

我们稍微偏离了主题, 目的是为了展示动态语言的强大之处。

⁸如果你了解 Ruby, 你就会知道方法注入 (injecting methods) 是 Ruby 爱好者中众所周知, 但又颇具争议的技术。在 Python 中, 方法注入并不常见, 因为它不适用于任何内置类型——str、list 等。我认为 Python 的这一限制是一件幸事。

接下来, 我们来介绍具名元组的变体——typing.NamedTuple。

5.4 带类型的具名元组:typing.NamedTuple

“5.3 典型的具名元组:collections.namedtuple”一节, [示例 5.9](#) 中具有默认字段值的 Coordinate 类, 还可以使用 typing.NamedTuple 构建, 如 [示例 5.10](#) 所示。

</> [示例 5.10](#): typing_namedtuple/coordinates2.py

```
1 from typing import NamedTuple
2
3 class Coordinate(NamedTuple):
4     lat: float
5     lon: float
6     reference: str = 'WGS84'
```

- ❶ 每个实例字段都必须指定 [类型提示 \(Type Hints\)](#)。
- ❷ reference 字段指定了 [类型提示 \(Type Hints\)](#) 与字段默认值。

用 typing.NamedTuple 创建的类, 与 collections.namedtuple 生成的类非常相似。它们都有相同的方法和从 tuple 继承的方法。唯一的区别是 typing.NamedTuple 创建的类包含一个类属性 `_annotations_`, 它用于指定类中字段的 [类型提示 \(Type Hints\)](#)。然而, 这个属性在代码实际运行时是被 Python 忽略的, 因为 [类型提示 \(Type Hints\)](#) 仅在编译时使用, 不会影响运行时的行为和属性访问。

鉴于 typing.NamedTuple 的主要功能是 [类型提示 \(Type Hints\)](#), 在继续探索数据类构建器之前, 先简要介绍一下 [类型提示 \(Type Hints\)](#)。

5.5 类型提示入门

[类型提示 \(Type Hints\)](#) 又称“[类型注解 \(Type Annotations\)](#)”, 是一种声明函数参数、返回值、变量和属性的预期类型的方法。关于类型提示, 您需要了解的是: 在 Python 字节码编译器和解释器中, 并不强制执行 [类型提示 \(Type Hints\)](#) 检查。



以上是 [类型提示 \(Type Hints\)](#) 的简要介绍, 足以理解 typing.NamedTuple 和 `@dataclass` 声明中使用的注释语法和含义即可。我们将在“[八 函数中的类型提示](#)”中介绍函数签名的类型提示, 并在“[十五 类型注解进阶 1136](#)”介绍更高级的 [类型提示 \(Type Hints\)](#)。本章节, 只涉及简单内置类型(如 str、int、float 等)的 [类型提示 \(Type Hints\)](#), 但这些类型可能是数据类中最常见的字段类型。

5.5.1 运行时无效

可将 Python 类型提示视为“可供 IDE 和类型检查器验证类型的文档”。这是因为 [类型提示 \(Type Hints\)](#) 不会影响 Python 程序的运行时行为 (如 [示例 5.11](#) 所示)。

```
</>示例 5.11: Python 在运行时,不强制执行类型提示检查

>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     lon: float
...
>>> trash = Coordinate('Ni!', None)
>>> print(trash)
Coordinate(lat='Ni!', lon=None) ❶
```

❶ 我已经说过:Python 在程序运行时,不执行类型检查!

如果在 Python 模块中输入 [示例 5.11](#) 的代码,它将运行并显示一个无意义的坐标,没有错误或警告:

```
1 $ python3 nocheck_demo.py
2 Coordinate(lat='Ni!', lon=None)
```

[类型提示 \(Type Hints\)](#) 主要用于支持第三方类型检查器,例如 [Mypy](#) 或 [PyCharm IDE](#) 内置类型检查器。这些是静态分析工具:它们只检查“静态”的 Python 源代码,而不检查运行中的代码。

要看到 [类型提示 \(Type Hints\)](#) 的效果,你必须用一种工具(比如 [linter](#))来对你的代码进行检查。例如,以下是 Mypy 对之前示例的检查结果:

```
$ mypy nocheck_demo.py
2 nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
3 incompatible type "str"; expected "float"
4 nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
5 incompatible type "None"; expected "float"
```

如您所见,根据 [Coordinate](#) 的定义,Mypy 知道创建实例的 2 个参数都必须是 [float](#) 类型,但构建 [trash](#) 时却使用了 [str](#) 和 [None](#)³。

现在,我们来介绍 [类型提示 \(Type Hints\)](#) 的语法和含义。

5.5.2 变量注解语法

[typing.NamedTuple](#) 与 [@dataclass](#) 均使用“[PEP 526 - Syntax for Variable Annotations](#)”中定义的 [变量注解 \(Variable Annotation\)](#) 语法。本节简要介绍在 [class](#) 语句中定义属性的 [变量注解 \(Variable Annotation\)](#) 语法。

[变量注解 \(Variable Annotation\)](#) 的基本语法如下所示:

```
1 var_name: some_type
```

³在 [类型提示 \(Type Hints\)](#) 的上下文中, [None](#) 并不是 [NoneType](#) 的 [单例 \(Singleton\)](#),而是 [NoneType](#) 本身的别名。仔细想想,这很奇怪,但却符合直觉。而且,对于返回 [None](#) 的函数,使用 [None](#) 来注解函数的返回值,可读性更强。

PEP 484 中的“可接受类型提示”一节,介绍了哪些是变量注解可接受的类型。但在定义数据类时,如下这些类型可能更有用:

- 一个 **具体类** (Concrete Class),如 str 或 FrenchDeck。
- 一个参数化的容器 (collection) 类型,如 list[int]、tuple[str, float] 等。
- typing.Optional,例如,Optional[str]——用于声明一个可以是 str 或 None 的字段。

你也可以为变量指定初始值。在 typing.NamedTuple 或 @dataclass 声明中,如果在构造函数被调用时省略了对应的参数值,该初始值将成为该属性的默认值:

```
1 var_name: some_type = a_value
```

5.5.3 变量注解意义

在“5.5.1 运行时无效”中,我们提过 **类型提示** (Type Hints) 在程序运行时没有效果。但是,Python 在 import (加载) 模块时,会读取 **类型提示** (Type Hints) 来构建 `__annotations__` 字典。然后,typing.NamedTuple 与 @dataclass 使用此字典来增强正在构建的数据类。

我们将从示例 5.12 中的简单类开始探索,以便稍后可以看到通过 typing.NamedTuple 与 @dataclass 增加了哪些额外的功能。

</> 示例 5.12: eaning/demo_plain.py: 带有类型提示的简单类

```
1 class DemoPlainClass:  
2     a: int          ❶ # a 不会成为类属性, 因为没有绑定值  
3     b: float = 1.1 ❷ # b 和 c 将被存储为类属性, 因为它们都绑定了值  
4     c = 'spam'      ❸ # c 是一个普通的类属性, 而不是注解
```

- ❶ a 会成为 `__annotations__` 字典中的一个条目,但会被丢弃:类中不会创建名为 a 的属性。
- ❷ b 被保存为注解,同时也成为一个类属性,属性值为 1.1。
- ❸ c 只是一个普通的(旧)类属性,而不是注释。

我们可以在控制台中验证这一点,首先读取 DemoPlainClass 的类属性 `__annotations__`,然后尝试获取名为 a、b 和 c 的属性:

```
1 >>> from demo_plain import DemoPlainClass  
2 >>> DemoPlainClass.__annotations__  
3 {'a': <class 'int'>, 'b': <class 'float'>}  
4 >>> DemoPlainClass.a  
5 Traceback (most recent call last):  
6 File "<stdin>", line 1, in <module>  
7 AttributeError: type object 'DemoPlainClass' has no attribute 'a'  
8 >>> DemoPlainClass.b  
9 1.1  
10 >>> DemoPlainClass.c  
11 'spam'
```

请注意，`__annotations__` 这个特殊的类属性是由解释器创建的，用于记录源代码中出现的类型提示（Type Hints）——即使是在普通类中。

`a` 只作为注解存在，但它不会成为类属性，因为没有为它绑定属性值⁴。`b` 和 `c` 将被存储为类属性（Class Attribute），因为它们都绑定了值。

这 3 个属性（`a`、`b`、`c`）都不会出现在 `DemoPlainClass` 的新实例中。若您创建一个对象 `o=DemoPlainClass()`，则 `o.a` 将引发 `AttributeError` 异常。而 `o.b` 和 `o.c` 将检索类属性，值分别为 1.1 和“spam”——行为与常规的 Python 对象相同。

5.5.3.1 研究 `typing.NamedTuple` 数据类

现在，让我们来研究一个用 `typing.NamedTuple` 构建的数据类（如示例 5.13 所示）。该类使用的属性和类型提示（Type Hints）与“5.5.3 变量注解意义”中的示例 5.12 一致。

</> 示例 5.13: meaning/demo_nt.py: 用 `typing.NamedTuple` 构建的数据类

```

1 import typing
2
3 class DemoNTClass(typing.NamedTuple):
4     a: int          ❶ # a 是一个实例属性，也是一个注解，没有默认值
5     b: float = 1.1 ❷ # b 是一个实例属性，也是一个注解，默认值 1.1
6     c = 'spam'      ❸ # c 是一个普通的类属性，而不是注解

```

- ❶ `a` 是一个实例属性，它有一个类型注解 `int`，但没有默认值。创建 `DemoNTClass` 实例时，必须为属性 `a` 提供初始值。每个 `DemoNTClass` 实例都会有一个名为 `a` 的属性。
- ❷ `b` 也是一个实例属性，它有一个类型注解 `float`，并且默认值为 1.1。如果创建 `DemoNTClass` 实例时没有为 `b` 提供初始值，它将使用默认值 1.1。每个 `DemoNTClass` 实例都会有一个名为 `b` 的属性。
- ❸ `c` 只是一个普通的（旧）类属性。因为它在类体中被直接赋值，而不是作为一个 `NamedTuple` 字段定义。这意味着 `c` 是属于 `DemoNTClass` 类本身的属性，而不属于 `DemoNTClass` 类的任何实例。

研究一下 `DemoNTClass`，首先读取类属性 `DemoNTClass.__annotations__`，然后获取名为 `a`、`b` 和 `c` 的属性，结果如下：

```

1 >>> from demo_nt import DemoNTClass
2 >>> DemoNTClass.__annotations__
3 {'a': <class 'int'>, 'b': <class 'float'>}
4 >>> DemoNTClass.a
5 <_collections._tuplegetter object at 0x101f0f940>
6 >>> DemoNTClass.b
7 <_collections._tuplegetter object at 0x101f0f8b0>
8 >>> DemoNTClass.c
9 'spam'

```

示例 5.13 中，我们为 `a` 和 `b` 添加了与示例 5.12 相同的类型提示（Type Hints）。但是，`typing.NamedTuple` 创建了类属性 `a` 与 `b`。属性 `c` 只是一个普通的类属性（Class Attribute），属性值为“spam”。

类属性 `a` 与 `b` 都是描述符（Descriptor）——“二十三 属性描述符”将介绍这一高级特性。现在，我们可

⁴Python 没有 `undefined` 的概念。`undefined` 这是 JavaScript 设计中最愚蠢的错误之一。

以将描述符理解为“属性读取器（Property Getters）”——即是一种不需要显式调用运算符（），就能获取实例属性的方法。实际上，这意味着 a 和 b 将被用作只读的实例属性。这一点不难理解，因为 DemoNTClass 实例只是一种高级 tuple，而 tuple 是不可变的。

DemoNTClass 还会获得一个自定义的 docstring：

```
1 >>> DemoNTClass.__doc__
2 'DemoNTClass(a, b)'
```

让我们来研究一下 DemoNTClass 的对象（实例）。

```
1 >>> nt = DemoNTClass(8) # 实例属性 a 无初始值，实例化时需为其提供参数值
2 >>> nt.a
3 8
4 >>> nt.b # 实例属性 b 有初始值，可不提供参数值
5 1.1
6 >>> nt.c # 对象 nt 从 DemoNTClass 类中检索普通的类属性 c，并读取属性
7 值
8 'spam'
```

在构造 DemoNTClass 类的实例对象 nt 时，因为参数 a 没有默认值，所以必须为参数 a 提供参数值。而参数 b 有默认值 1.1，所以可以不为其提供参数值。对象 nt 中拥有预期的实例属性 a 与 b，但没有实例属性 c，Python 会像往常一样，从 DemoNTClass 类中检索类属性（Class Attribute）c。

如果尝试为 nt.a、nt.b、nt.c 甚至 nt.z 赋值，将会引发 Attribute Error 异常，相应的错误消息会略有不同。请您自己尝试一下，并分析错误消息的内容。

5.5.3.2 研究用装饰器 dataclass 构建的数据类

现在，我们研究一下 [示例 5.14](#)。

</> [示例 5.14](#): meaning/demo_dc.py: dataclass 装饰器构建的类

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class DemoDataClass:
5     a: int          ❶
6     b: float = 1.1 ❷
7     c = 'spam'      ❸
```

- ❶ a 成为一个注释，也是一个由 [描述符（Descriptor）](#) 控制的实例属性，每个实例都有一个名为 a 的属性，类型为 int。
- ❷ b 成为一个注解，也是一个由 [描述符（Descriptor）](#) 控制的实例属性，并且默认值为 1.1。每个实例都会有一个名为 b 的属性，如果创建实例时没有为 b 提供值，它将使用默认值 1.1。
- ❸ c 只是一个普通的（旧）[类属性（Class Attribute）](#)（而不是注解），所有实例共享这个类属性值。

现在，让我们查看 DemoDataClass⁵ 上的 __annotations__、__doc__ 和属性 a、b、c：

⁵ 因为 a 和 b 被声明为 dataclass 的字段，并且带有类型提示，这使得它们成为实例属性，每个实例都会拥有这些属性的一个副本。而 c 直接在类体中被赋值，没有使用类型提示，因此它是一个类属性，所有的实例将共享这个属性的单个副本。

```

1  >>> from demo_dc import DemoDataClass
2  >>> DemoDataClass.__annotations__
3  {'a': <class 'int'>, 'b': <class 'float'>}
4  >>> DemoDataClass.__doc__
5  'DemoDataClass(a: int, b: float = 1.1)'
6  >>> DemoDataClass.a          # DemoDataClass 中没有类属性 a
7  Traceback (most recent call last):
8  File "<stdin>", line 1, in <module>
9  AttributeError: type object 'DemoDataClass' has no attribute 'a'
10 >>> DemoDataClass.b         # b 是类属性, 用于保存实例属性 b 的默认值
11 1.1
12 >>> DemoDataClass.c         # c 只是一个普通的类属性(Class Attribute)
13     , 不会被绑定到实例上
14     'spam'

```

__annotations__ 与 __doc__ 并不令人意外。但是,与 DemoNTClass (示例 5.13) 相比, DemoDataClass (示例 5.14) 类中没有属性 a。而 DemoNTClass (示例 5.13) 具有一个 **描述符 (Descriptor)**, 可以从实例中获取只读属性 a (即那个神秘的 <_collections._tuplegetter>)。这是因为属性 a 只存在于 DemoDataClass 的实例中。除非 DemoDataClass 类被冻结 (frozen), 否则属性 a 将是一个我们可以获取和设置的公共属性。但 b 和 c 是作为类属性存在的。其中, b 存储实例属性 b 的默认值, 而 c 只是一个普通的类属性 (Class Attribute), 不会被绑定到实例上。

现在,来看看 DemoDataClass 实例的情况。

```

1  >>> dc = DemoDataClass(9)
2  >>> dc.a      # a 为实例属性
3  9
4  >>> dc.b      # b 为实例属性
5  1.1
6  >>> dc.c      # 通过实例获得的类属性
7  'spam'

```

同样, a 和 b 是实例属性, c 是我们通过实例获得的类属性。如前所述, DemoDataClass 实例是可变的, 并且在运行时不进行类型检查 (如下所示):

```

1  >>> dc.a = 10
2  >>> dc.b = 'oops'

```

甚至,还可以以为不存在的属性复制 (如下所示):

```

1  >>> dc.c = 'whatever'      # 实例属性 c, 不会影响类属性 c
2  >>> dc.z = 'secret stash'

```

现在,实例 dc 有了一个属性 c,但这对类属性 c 没有影响。我们可以新增一个属性 z。这是 Python 的正常行为:常规实例可以拥有自己的属性,但这些属性不会出现在类中⁶。

⁶执行 __init__ 之后设置属性,有悖于“3.9 dict 实现方式对实践的影响”讲过的字典键共享内存 (即共享哈希表) 优化措施。

5.6 @dataclass 详解

到目前为止,我们只看到了使用 `@dataclass` 的简单示例。`@dataclass` 装饰器可接受多个关键字参数,其完整签名如下:

```
1 @dataclass(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
```

第一个参数位置上的 * 表示后面都是关键字参数。表 5.2 介绍了这些参数。

表 5.2: `@dataclass` 装饰器可接受的关键字参数

参数	作用	默认值	备注
<code>init</code>	生成 <code>__init__</code>	<code>True</code>	若用户自己实现了 <code>__init__</code> ,则忽略该参数。
<code>repr</code>	生成 <code>__repr__</code>	<code>True</code>	若用户自己实现了 <code>__repr__</code> ,则忽略该参数。
<code>eq</code>	生成 <code>__eq__</code>	<code>True</code>	若用户自己实现了 <code>__eq__</code> ,则忽略该参数。
<code>order</code>	生成 <code>__lt__</code> 、 <code>__le__</code> 、 <code>__gt__</code> 、 <code>__ge__</code>	<code>False</code>	设为 <code>True</code> 时,若 <code>eq=False</code> ,或者自行定义或继承其他用于比较的方法,则抛出异常。
<code>unsafe_hash</code>	生成 <code>__hash__</code>	<code>False</code>	语义复杂,有多个问题需要注意,详见 dataclass 附录 。
<code>frozen</code>	让实例不可变	<code>False</code>	实例可以避免意外更改,但并非真正不可变 ⁷ 。

以上参数的默认值设置,适用于大多数情况。不过,您可能会更改以下参数的值(不使用默认值)。

- `frozen=True`
防止意外更改类的实例。
- `order=True`
允许排序数据类的实例。

鉴于 Python 对象的动态特性,多事的程序员要绕过 `frozen=True` 提供的保护并不难。不过,在代码审查阶段很容易发现这种小伎俩。

若参数 `eq` 和 `frozen` 都为 `True`,则 `@dataclass` 会生成合适的 `__hash__` 方法,因此实例将是 [可哈希 \(hashable\)](#) 的。生成的 `__hash__` 将使用所有字段的数据,通过字段选项(见“[5.6.1 字段选项](#)”)也不能排除。对于 `frozen=False`(默认值),`@dataclass` 会将 `__hash__` 设置为 `None`,表示实例是不可哈希 (hashable) 的,从而覆盖超类中的 `__hash__`。

对于 `unsafe_hash`,“[PEP 557 -Data Classes](#)”是这样说的:

可以设置 `unsafe_hash=True`,强制数据类创建 `__hash__` 方法,但是不建议这么做。若一个类在逻辑上是不可变的,但事实上是可变的,则可以这么做。但是,这是一种特殊的用例,应小心谨慎。

我暂时不对“`unsafe_hash`”做进一步讨论。若你确实需要用此选项,请查阅 [dataclasses.dataclass 文档](#)。可以在字段级别,对生成的数据类做进一步地定制(见“[5.6.1 字段选项](#)”)。

⁷ `@dataclass` 通过生成 `__setattr__` 和 `__delattr__` 来模拟不变性,当用户尝试设置或删除字段时,会引发 `dataclass.FrozenInstanceError`—`AttributeError` 的子类。

5.6.1 字段选项

至此,我们已经见到了一些最基本的字段选项:通过 [类型提示 \(Type Hints\)](#) 提供(或不提供)字段的默认值。声明的实例字段将成为生成的 `__init__` 中的参数。Python 不允许在有缺省值的参数之后出现没有缺省值的参数。因此,在声明具有默认值的字段后,所有其余字段也必须具有默认值。

对于 Python 初学者来说,可变缺省值(如 `list` 或 `dict`)是一个常见的 bug 来源。在函数定义中,当对函数的一个调用直接修改了默认值时,可变默认值很容易被破坏,从而改变了后续调用的行为⁸(此问题将在[“6.5.1 不要使用可变类型作为参数的默认值”](#)中详谈)。类属性通常被用作实例的属性默认值,数据类也是如此。`@dataclass` 使用 [类型提示 \(Type Hints\)](#) 中的默认值,为 `__init__` 生成带默认值的参数。为了防止 bug,`@dataclass` 拒绝像[示例 5.15](#)那样定义数据类。

</> [示例 5.15](#): `dataclass/club_wrong.py`:此数据类将引发 `ValueError`

```
1 @dataclass
2 class ClubMember:
3     name: str
4     guests: list = [] # 不能将可变对象指定为字段的默认值
```

加载 `ClubMember` 类所在的模块,会得到如下结果:

```
1 $ python3 club_wrong.py
2 Traceback (most recent call last):
3   File "club_wrong.py", line 4, in <module>
4     class ClubMember:
5     ...several lines omitted...
6   ValueError: mutable default <class 'list'> for field guests is not allowed:
7   use default_factory
```

`ValueError` 消息指出了问题所在,并提出了解决方案:使用 `default_factory`。[示例 5.16](#)展示了如何更正 `ClubMember`。

</> [示例 5.16](#): `dataclass/club.py`:更正[示例 5.15](#)中的 `ClubMember` 类

```
1 from dataclasses import dataclass, field
2
3 @dataclass
4 class ClubMember:
5     name: str
6     guests: list = field(default_factory=list)
```

在[示例 5.16](#)中,字段 `guests` 未使用 `list` 字面量作为其默认值,而是通过调用 `dataclasses.field` 函数,并设置 `default_factory=list` 来设置默认值。

参数 `default_factory` 的值可以是一个函数、类或者任何其他可调用对象。每次创建数据类的实例时,都会调用(不带参数)这个可调用对象,以为字段创建默认值。这样, `ClubMember` 的每个实例都将拥有自己的 `list`,而不是所有实例都共享 `ClubMember` 类中的同一个 `list`。我们很少需要共享同一个 `list`,因为它经常会产

⁸译者注:当定义函数时,如果默认值是可变对象(如 `list`、`dict`、`set`),它只会在函数定义时被创建一次,并在后续的调用中重复使用(保存指向可变对象的引用)。这意味着,如果函数体内对可变默认值进行了修改,这些修改将会在后续的函数调用中持续存在。

生 bug。



@dataclass 在定义数据类时,会拒绝使用 list 作为字段的默认值。但是,这种方案只对 list、dict、set 有效。除此之外,其他可变对象不会引起 @dataclass 注意。您需要理解这个问题,并记住 使用 default_factory 为数据类的字段设置可变默认值。

浏览 `dataclasses` 模块文档,你会看到一个使用新语法定义的列表字段,如 [示例 5.17](#) 所示。

</> [示例 5.17](#): `ataclass/club_generic.py`: 这个 ClubMember 的定义更精确

```
1  from dataclasses import dataclass, field
2
3  @dataclass
4  class ClubMember:
5      name: str
6      guests: list[str] = field(default_factory=list) ❶
```

❶ `list[str]` 的意思是由 str(字符串)构成的 list。

新语法 `list[str]` 是一种 [参数化泛型 \(Parameterized Generic Type\)](#): 从 Python 3.9 开始,内置 list 可以使用方括号([])表示法指定 list 中项的类型。



在 Python 3.9 之前,内置容器类型(如 list、dict 等)不支持泛型表示法。作为临时解决方案,在 typing 模块中有相应的集合类型。若在 Python 3.8 或更早版本中需要参数化的 list [类型提示 \(Type Hints\)](#),则必须从 typing 中导入 List 类型并使用它(例如 `List[str]`)。有关此问题的更多信息,请参阅“[8.5.4 泛化容器](#)”中的附注栏“[向后兼容与弃用的容器类型](#)”。

我们将在“[八 函数中的类型提示](#)”中介绍 [泛型 \(Generic Type\)](#)。现在,请注意 [示例 5.16](#) 与 [示例 5.17](#) 都是正确的,Mypy 程序在检查这两个类的定义时,均不会报错。

[示例 5.16](#) 与 [示例 5.17](#) 的区别在于 `guests: list` 表示 `guests` 列表中的每一项可以是任意类型的对象,而 `guests: list[str]` 则表示 `guests` 列表中的每一项都必须是 str 类型。因此,若在列表中存储无效的项,或从列表中读取到无效的项时,类型检查工具(如 MyPy)将报错。

`default_factory` 可能是 `field` 函数中最常用的选项,但还有其他几个选项,如 [表 5.3](#) 所示。

表 5.3: `field` 函数可接受的关键字参数

参数	作用	默认值
<code>default</code>	字段的默认值	<code>_MISSING_TYPE</code> ⁸
<code>default_factory</code>	用于为字段生成默认值的函数(无参数)	<code>_MISSING_TYPE</code>
<code>init</code>	将该字段包含在 <code>__init__</code> 的参数中	<code>True</code>

接下页

续表 5.3

参数	作用	默认值
repr	在 <code>__repr__</code> 中包含该字段	True
compare	在比较方法 (<code>__eq__</code> 、 <code>__lt__</code> 等) 中包含该字段	True
hash	在 <code>__hash__</code> 中使用该字段计算哈希码	None ⁹
metadata	用户自定义数据的映射; <code>@dataclass</code> 忽略此参数	None

之所以存在 `default` 参数, 是因为在 [字段注解 \(Field Annotation\)](#) 中, `field` 函数调用占据了为字段设置默认值的位置。假设我们想要创建一个 `athlete` 字段, 默认值为 `False` 并且不提供给 `__repr__` 方法使用。则需要像下面这样编写数据类。

</> 示例 5.18: 构建数据类 ClubMember

```

1 @dataclass
2 class ClubMember:
3     name: str
4     guests: list = field(default_factory=list)
5     athlete: bool = field(default=False, repr=False)

```

- ❶ 每次创建 `ClubMember` 实例, 都会调用 `list` 类 (由参数 `default_factory` 指定), 为字段 `guests` 生成默认值——即一个新的空列表 `[]`。
- ❷ 通过参数 `default`, 将 `False` 设置为字段 `athlete` 的默认值。通过 `repr` 参数, 将字段 `athlete` 从 `__repr__` 方法中排除。

5.6.2 初始化后处理: `__post_init__`

由 `@dataclass` 生成的 `__init__` 方法仅做一件事: 将传入的参数及其默认值 (若未提供参数值) 赋值给作为实例字段的实例属性。但是, 有时候初始化实例, 可能还需要做更多的工作。为此, 可以提供一个 `__post_init__` 方法。当 `__post_init__` 方法存在时, 这个方法会在生成的 `__init__` 方法中作为最后一步被调用。`__post_init__` 方法经常用于执行验证, 以及根据其他字段计算一个字段的值。下面举例说明这两种用途。

我们将定义一个 `HackerClubMember` 类, 该类继承 `ClubMember` 类 (示例 5.18)。示例 5.19 中 doctests 描述了 `HackerClubMember` 类的预期行为。

</> 示例 5.19: dataclass/hackerclub.py;HackerClubMember 的 doctest

```

1 """
2 ``HackerClubMember`` 对象接受一个可选参数 ``handle``::
3
4     >>> anna = HackerClubMember('Anna Ravenscroft', handle='AnnaRaven')
5     >>> anna
6     HackerClubMember(name='Anna Ravenscroft', guests=[], handle='AnnaRaven')
7

```

⁸`dataclass._MISSING_TYPE` 是一个 [哨兵值 \(Sentinel Value\)](#), 表示没有提供该选项。有了它, 我们就可以将 `None` 设置为实际的默认值, 这是一种常见的用例。

⁹选项 `hash=None` 表示只有在 `compare=True` 时, `__hash__` 才会使用该字段。

```
8  若省略参数 ``handle``，则将其设置为会员姓名的第一部分::
9  >>> leo = HackerClubMember('Leo Rochael')>>> leo
10 HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')
11
12 会员必须拥有唯一的 handle。下面的对象 ``leo2`` 不会被创建，因为它的 ``handle`` 值是
13  Leo，已被对象 ``leo`` 占用::
14
15 >>> leo2 = HackerClubMember('Leo DaVinci')
16 Traceback (most recent call last):
17 ...
18 ValueError: handle 'Leo' already exists.
19
20 因此，创建对象 ``leo2`` 时，必须显式指定 ``handle``::
21
22 >>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
23 >>> leo2
24 HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
    """
```

请注意，必须以关键字参数的形式提供 handle 字段。因为 HackerClubMember 从 ClubMember（[示例 5.18](#)）继承了字段 name 与 guests，并额外添加了字段 handle。HackerClubMember 生成的 docstring，说明了构造函数调用中各字段的顺序：

```
1 >>> HackerClubMember.__doc__
2 "HackerClubMember(name: str, guests: list = <factory>, handle: str = '')"
```

这里的 <factory> 是一种简略表示，表示 guests 字段的默认值由某个可调用对象产生（本例是 list 类）。上述 docstring 的重点是：如果想为 HackerClubMember 提供字段 handle，但不提供字段 guests，那么必须利用关键字参数传入 handle。

[dataclasses 模块文档](#)中的“[继承](#)”部分解释了在有多级继承时，如何计算字段的顺序。



在“[十四 继承的利与弊](#)”中，我们将讨论滥用继承的问题，特别是当超类不是抽象的时候。创建具有层次结构的数据类通常是个坏主意。但在 [示例 5.20](#) 中这么做，是为了减少 HackerClubMember 类的代码，将精力集中在字段 handle 的声明与 [__post_init__](#) 方法中的验证逻辑上，

HackerClubMember 类的实现如 [示例 5.20](#) 所示。

</> [示例 5.20](#): dataclass/hackerclub.py: 构建 HackerClubMember 类

```
1 from dataclasses import dataclass
2 from club import ClubMember
3
4 @dataclass
5 class HackerClubMember(ClubMember):
6     all_handles = set()
7     handle: str = ''
```

①
②
③

```

9  def __post_init__(self):
10    cls = self.__class__
11    if self.handle == '':
12      self.handle = self.name.split()[0]
13    if self.handle in cls.all_handles: ④
14      msg = f'handle {self.handle!r} already exists.'
15      raise ValueError(msg)
16    cls.all_handles.add(self.handle) ⑤

```

- ① HackerClubMember 类继承 ClubMember。
- ② all_handles 是一个类属性。
- ③ handle 是一个 str 类型的实例字段,默认值为空字符串;因此,它是可选的字段。
- ④ self.__class__ 获取实例所属的类。
- ⑤ 若 self.handle 是空字符串,则将其设置为 name 的第一部分
- ⑥ 若 self.handle 位于 cls.all_handles 中,则引发 ValueError。
- ⑦ 将新 handle 添加到 cls.all_handles 中。

示例 5.20 将会按我们的预期工作,但不能满足静态类型检查器的要求。“5.6.3 带类型的类属性”将说明原因,以及如何解决它。

5.6.3 带类型的类属性

如果用 Mypy 检查 [示例 5.20](#) 的代码,将会得到如下报错信息。

```

$ mypy hackerclub.py
hackerclub.py:37: error: Need type annotation for "all_handles"
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)

```

可惜的是,Mypy(我用的 0.910 版)提供的提示,对使用 @dataclass 构建数据类没什么用处。Mypy 建议使用 `typing.Set`,而我使用的是 Python 3.9,可以使用 `set`——免得再从 `typing` 模块中导入 `Set`。更重要的是,如果为 `all_handles` 添加类似 `set[...]` 这样的 [类型提示 \(Type Hints\)](#),`@dataclass` 将找到该注释,并使 `all_handles` 成为实例字段。我们在“[5.5.3 变量注解意义](#)”中,已看到过此种情况的发生。

“[PEP 526 -Syntax for Variable Annotations](#)”中定义的变通方法不太优雅。若想为类变量提供 [类型提示 \(Type Hints\)](#),需要使用一个名为 `typing.ClassVar` 的伪类型。它可借助 [泛型 \(Generic Type\)](#) 符号 `[]`,来设置变量的类型,并将变量声明为类属性。

为了让类型检查工具(如 Mypy)与 `@dataclass` 都满意,在 [示例 5.20](#)(5.6.2 节)中应当像下面这样声明类属性 `all_handles`。

```
1 all_handles: ClassVar[set[str]] = set()
```

此处的 [类型提示 \(Type Hints\)](#) 的意思是:

`all_handles` 是一个类属性,其类型为由字符串构成的 `set`(集合),默认值为空集。

要编写这个 [类型提示 \(Type Hints\)](#),必须先从 `typing` 模块中导入 `ClassVar`。

通常,`@dataclass` 装饰器不会关心注解中的类型,但是有两种例外情况。其中之一是:若属性类型为

ClassVar，则 @dataclass 不会为该属性生成实例字段。另一种情况是：声明“仅初始化（Init-only）变量”时（见 5.6.4 节）。

5.6.4 不用作实例字段的 Init-Only 变量

有时，您可能需要将不作为实例字段的参数传递给 `__init__` 方法。这种参数在“[dataclasses 模块文档](#)”中被称为“仅初始化（Init-only）”变量。为了声明此种类型的参数，`dataclasses` 模块提供了伪类型 `InitVar`，它使用与 `typing.ClassVar` 相同的语法。文档中给出的例子是一个数据类，该数据类拥有一个从数据库初始化的字段，因此必须将数据库对象传递给构造函数。

示例 5.21 展示了 `dataclasses` 文档“[Init-only variables](#)”一节的示例代码。

</> 示例 5.21: `dataclasses` 模块文档中的示例

```
1  from typing import InitVar
2
3  @dataclass
4  class C:
5      i: int
6      j: int | None = None
7      database: InitVar[DatabaseType | None] = None
8
9      def __post_init__(self, database):
10         if self.j is None and database is not None:
11             self.j = database.lookup('j')
12
13 c = C(10, database=my_database)
```

请注意属性 `database` 的声明方式。`InitVar` 会阻止 `@dataclass` 将 `database` 视为常规字段。`database` 不会被设置为实例属性，并且也不会出现在 `dataclasses.fields` 函数返回的列表中。但是，属性 `database` 是生成的 `__init__` 方法将接受的参数之一，并且它也会被传递给 `__post_init__` 方法。若您想编写自己的 `__post_init__` 方法，必须在方法签名中增加相应的参数（如 [示例 5.21](#) 所示）。

本书对 `@dataclass` 的讲解占据了较长的篇幅，涵盖了多数比较有用的功能。有些功能在前面的章节中已提到过，如 [表 5.1](#)（5.2.1 节）将 3 种数据类构建器放在一起做了比较。若想深入了解，请阅读 `dataclasses` 文档以及“[PEP 526 -Syntax for Variable Annotations](#)”。

在下一节中，将用 `@dataclass` 构建一个更长的数据类。

5.6.5 @dataclass 示例：都柏林核心模式

前面章节所见的示例中，字段数量都不多。在实际使用中，数据类通常需要更多的字段。本节根据“都柏林核心模式（Dublin Core Schema）”，用 `@dataclass` 装饰器构建一个更复杂的类。

都柏林核心模式 (Dublin Core Schema) 是一组术语, 可用于描述数字资源 (如视频、图像、网页等)、实物资源 (如书籍、CD 等) 以及艺术品等对象。

——Dublin Core on Wikipedia

都柏林核心模式 (Dublin Core Schema) 定义了 15 个可选字段; [示例 5.22](#) 中的 Resource 类使用了其中的 8 个。

</> [示例 5.22](#): dataclass/resource.py:Resource 类的代码, 一个基于 Dublin Core 术语的类

```

1  from dataclasses import dataclass, field
2  from typing import Optional
3  from enum import Enum, auto
4  from datetime import date
5
6  class ResourceType(Enum):
7      BOOK = auto()
8      EBOOK = auto()
9      VIDEO = auto()
10
11 @dataclass
12 class Resource:
13     """媒体资源描述"""
14     identifier: str
15     title: str = '<untitled>' ②
16     creators: list[str] = field(default_factory=list) ③
17     date: Optional[date] = None ④
18     type: ResourceType = ResourceType.BOOK ⑤
19     description: str = '' ⑥
20     language: str = ''
21     subjects: list[str] = field(default_factory=list) ⑥

```

- ① Enum 将为 Resource.type 字段提供类型安全的值。
- ② identifier 是唯一的必需字段。
- ③ title 是首个拥有默认值的字段。因此, 后续字段都要提供默认值。
- ④ date 字段的值可以是一个 datetime.date 实例或 None。
- ⑤ type 字段的默认值是 ResourceType.BOOK。
- ⑥ subjects 字段的默认值是由 list 类新生成的空列表 []。

[示例 5.23](#) 中的 doctest 演示了如何在代码中使用 Resource 记录。

</> [示例 5.23](#): dataclass/resource.py: 使用 Resource 类

```

1  >>> description = 'Improving the design of existing code'
2  >>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd Edition',
3  ... ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
4  ... ResourceType.BOOK, description, 'EN',
5  ... ['computer programming', 'OOP'])
6  >>> book # doctest: +NORMALIZE_WHITESPACE

```

```

7 |     Resource(identifier='978-0-13-475759-9', title='Refactoring, 2nd Edition',
8 |     creators=['Martin Fowler', 'Kent Beck'], date=datetime.date(2018, 11, 19), type=<
9 |     ResourceType.BOOK: 1>, description='Improving the design of existing code',
9 |     language='EN', subjects=['computer programming', 'OOP'])

```

@dataclass 生成的 `__repr__` 方法的效果勉强可以。但是,我们还可以进一步定制 `__repr__`,使其输出的内容更具可读性。例如,我们希望 `repr(book)` 可返回如下格式:

</> 示例 5.24: `repr(book)` 输出格式

```

1 | >>> book # doctest: +NORMALIZE_WHITESPACE
2 | Resource(
3 |     identifier = '978-0-13-475759-9',
4 |     title = 'Refactoring, 2nd Edition',
5 |     creators = ['Martin Fowler', 'Kent Beck'],
6 |     date = datetime.date(2018, 11, 19),
7 |     type = <ResourceType.BOOK: 1>,
8 |     description = 'Improving the design of existing code',
9 |     language = 'EN',
10 |    subjects = ['computer programming', 'OOP'],
11 | )

```

示例 5.25 给出的 `__repr__` 代码,可输出 示例 5.24 中的格式。此示例使用 `dataclass.fields(cls)` 来获取数据类 `Resource` 的字段名称。

</> 示例 5.25: `dataclass/resource_repr.py`:在 `Resource` 类中定制 `__repr__` 方法

```

1 | def __repr__(self):
2 |     cls = self.__class__
3 |     cls_name = cls.__name__
4 |     indent = ' ' * 4
5 |     res = [f'{cls_name}(']
6 |     for f in fields(cls):
7 |         value = getattr(self, f.name)
8 |         res.append(f'{indent}{f.name} = {value!r},')
9 |
10 |     res.append(')')
11 |     return '\n'.join(res)

```

- ❶ `self.__class__` 获取当前实例所属的类。`self` 被绑定到当前的实例对象,实例对象的 `__class__` 属性指当前实例所属的类。
- ❷ `cls.__name__` 获取类名,本例为 `Resource`。
- ❸ 声明输出字符串的列表 `res`。列表 `res` 的第一项为类名和左圆括号,本例为 “`Resource()`”。
- ❹ 通过 `fields(cls)` 遍历 `Resource` 类的各个字段,并赋值给 `f`。
- ❺ 通过 `getattr()` 方法,从实例上获取字段名称。
- ❻ 为列表 `res` 追加带缩进的行,以显示字段名称与字段值。
- ❼ 为列表 `res` 追加右圆括号,即 “`)`”。
- ❽ 根据列表 `res`,构建并返回字符串。

本节示例的灵感来自于 Dublin, Ohio (俄亥俄州都柏林) 举办的一个会议。我们对 Python 数据类构建器的探讨,到此结束。

数据类的确很方便,但如果过度使用数据类,可能会为项目带来不好的影响。下一节,将详谈过度使用数据类所带来的影响。

5.7 数据类导致的代码异味

无论是自己编写所有代码实现数据类,还是利用本章介绍的类构建器来实现数据类,都要注意一点:这可能表示您的设计存在问题。

在《重构:改善既有代码的设计》第2版一书中,Martin Fowler 和 Kent Beck 提出了“代码异味”的目录——列出了代码中可能需要进行重构的模式(Pattern)。其中,题为“Data Class”一条的内容如下:

“数据类”是指那些只包含字段、读写字段的方法,除此之外没有其他功能与逻辑。这样的类被称为“愚蠢的数据持有者”,只用于存储数据,并被其他的类以过于详细的方式进行操作。

在 Fowler 的个人网站上,有一篇题为“[代码异味 \(Code Smell\)](#)”的文章很有启发性。这篇文章与我们的讨论非常相关,因为他将数据类作为“[代码异味 \(Code Smell\)](#)”的一个示例,并提出了如何处理它的建议。以下是此篇文章的全文转载⁹:

代码异味 (Code Smell)

Martin Fowler

[代码异味 \(Code Smell\)](#) 只是一种表面迹象,通常与系统中更深层次的问题相对应。Kent Beck 在帮助我撰写《重构:改善既有代码的设计》第2版一书时首次提出了这个术语。

上面的定义比较简单,但隐含了几个微妙的点。首先,“异味”是指那些很容易被发现的问题,就像我最近所说的“异味能被嗅到”。一个内容较长的方法,就是一种异味——只要看看代码,如果发现超过十几行的 Java 代码,直觉告诉我,这段代码可能存在问题。

其次,有“[代码异味 \(Code Smell\)](#)”并不代表代码一定有问题。有些方法的内容本就应该很长。你必须更深入地研究,判断是否存在潜在的问题——[代码异味 \(Code Smell\)](#) 本身并不一定是问题,它只是问题代码的表征。

我们都喜欢容易被察觉的“[代码异味 \(Code Smell\)](#)”,因为多数时候它可以引导我们找到问题的根源。只有数据而没有行为的数据类,就是一个很好的例子。当你看到这种数据类时,请问自己一个问题:这个数据类需要什么样的行为?然后,开始重构,为其加入需要的行为。通常,简单的思考和基本的重构,是将空洞的对象转变为真正具有功能的类的关键一步。

“[代码异味 \(Code Smell\)](#)”的好处之一是:没有经验的程序员也很容易就能发现它们,即使他们还没有足够的知识来评估是否真正存在问题或纠正问题。我听说,一些研发总监会推行“一周异味之星”,要求团队人员寻找“[代码异味 \(Code Smell\)](#)”问题,并将找到的问题交给高级开发人员解决。每次解决一个“[代码异味 \(Code Smell\)](#)”问题,是个不错的做法,这可以逐步引导团队成员,使他们成为更优秀的程序员。

面向对象编程的主要思想是将行为和数据放在同一个代码单元(一个类)中。若一个类被广泛使用,但是

⁹很荣幸,Martin Fowler 是我在 Thoughtworks 的同事。因此,只用了 20 分钟就得到了他的授权。

它自身却没有什么重要的行为（即方法），那么处理其实例的代码就可能分散（有些甚至是重复的）在整个系统的方法和函数中——这样的系统对维护来说，简直就是噩梦。这就是为什么 Fowler 在重构数据类时，会将责任¹⁰放回到数据类中。

尽管如此，仍然有一些场景（如 5.7.1 节与 5.7.2 节）适合使用几乎没有行为的数据类。

5.7.1 将数据类用作脚手架

在此情况下，为了快速启动一个新项目或模块，先实现一个初始的、简单的数据类。随着时间的推移，这个数据类应该拥有自己的方法，而不是依靠其他类的方法来操作它的实例。脚手架（Scaffolding）只是暂时性的，起到了快速启动和试验的作用；最终，你自定义的数据类可能会完全独立于你用来启动它的构建器——脚手架。

脚手架示例

假设你正在开发一个社交媒体应用程序，你创建了一个名为“User”的数据类来表示用户。初始的数据类可能只有几个基本属性，比如用户名和年龄。

在初始阶段，像发送消息、添加好友等功能，可暂时使用其他类的方法来实现，这就是脚手架（Scaffolding）的作用。随着项目的发展，你应该逐渐完善用户类，使其能够独立地实现这些功能，而不依赖其他类的帮助。

脚手架（Scaffolding）只是暂时性地，以便快速启动项目。当完善了自定义的数据类之后，使其可以独立运行之后，即可移除脚手架。

在 Python 中，你可以使用一些现有的库或框架来快速搭建项目的基本结构，以加速问题的解决和实验的进行。在解决问题或完成实验后，你可以选择保留脚手架并继续使用它，也可以进一步改进代码以删除脚手架。

5.7.2 将数据类用作中间表述

数据类可用于构建要导出为 JSON 或其他交换格式的记录，或者用来保存刚刚导入的、跨越系统边界的数据。Python 的数据类构建器都提供了一种方法或函数，用于将一个数据类实例构建为一个普通的 dict。可以将 dict（用 ** 扩展）作为关键字参数，来调用数据类构建器的构造函数。这样的 dict 非常接近 JSON 记录。

在此场景下，数据类实例应作为“不可变对象”处理——即使字段是可变的，也不应在它们处于中间形式时，对其进行更改。倘若更改，就会失去数据和行为紧密结合的关键优势。当导入/导出需要更改值时，你应该实现自己的构建器方法，而不是直接使用构建器给定的“as dict”方法或标准构造函数。

现在，我们换个话题。Python 中的 模式匹配（Pattern Match）不仅可以匹配序列（见 2.6 节）与映射（见 3.3 节），还可以匹配任意类的实例。下面来看看如何编写这样的类 模式（Pattern）（见 5.8）。

¹⁰译者注：即指数据类的行为或功能逻辑

5.8 模式匹配类实例

类模式 (Class Pattern) 的设计目的是根据类型和属性 (可选) 匹配类的实例。类模式 (Class Pattern) 的 匹配对象 (Subject) 可以是任何类实例,而不仅仅是数据类的实例¹¹。

类模式 (Class Pattern) 有 3 种:简单类模式、关键字类模式与位置类模式。我们将按需,依次研究这几种类模式。

5.8.1 简单类模式

2.6节的 ?? 已经用到了类模式 (Class Pattern)。那时,是作为子模式使用的。

```
1 case [ str(name), _, _, (float(lat), float(lon)) ]:
```

此模式 (Pattern) 将匹配项数为 4 的序列,第一项必须是 str 实例,最后一项必须是具有 2 个 float 实例的二元组 (2-tuple)。

类模式 (Class Pattern) 的语法看起来像构造函数调用。下面是一个无需绑定变量,就能匹配浮点数的类模式 (Class Pattern) (如果需要,case 主体中可以直接引用 x):

```
1 match x:
2     case float():           # 注意此处的(), 这里将匹配一个 float 实例。
3         do_something_with(x)
```

但是,像下面这样写,很可能导致 bug。

```
1 match x:
2     case float:             # 危险！！！
3         do_something_with(x)
```

在上述的例子中,‘case float:’ 将匹配任何 匹配对象 (Subject),因为 Python 将 float 视为一个变量,然后将其绑定到 匹配对象 (Subject)。

float(x) 的简单 模式 (Pattern) 语法是一种特例,仅适用于 9 种内置类型 (在PEP 634 -Structural Pattern Matching: Specification 的 Class Patterns 一节的末尾列出):

bytes dict float frozenset int list set str tuple

对这些类型来说,看上去像构造函数参数的那个变量 (例如 float(x) 中的 x),将绑定整个匹配的实例。若是子模式,则将绑定匹配对象的一部分 (如示例 5.26 中的 str(name))

</> 示例 5.26: 简单类模式

```
1 case [ str(name), _, _, (float(lat), float(lon)) ]:
```

除了这 9 种内置类型之外,看上去像参数的那个变量,将表示 模式匹配 (Pattern Match) 的类实例的属性 (见 5.8.2 节的 ??)。

¹¹我之所以将这部分内容放在本章,是因为这一章是本书中最早关注用户定义类的章节。而且我认为类的 模式匹配 (Pattern Match) 非常重要,不能放到第二部分。我的理念是:知道如何使用类,比知道如何定义类更重要。

5.8.2 关键字类模式

要了解如何使用关键字 **类模式** (Class Pattern), 请看下面 [示例 5.27](#) 中的城市类和 5 个实例。

</> [示例 5.27: 关键字类模式:City 类与 5 个实例](#)

```
1 import typing
2
3 class City(typing.NamedTuple):
4     continent: str
5     name: str
6     country: str
7
8 cities = [
9     City('Asia', 'Tokyo', 'JP'),
10    City('Asia', 'Delhi', 'IN'),
11    City('North America', 'Mexico City', 'MX'),
12    City('North America', 'New York', 'US'),
13    City('South America', 'São Paulo', 'BR'),
14 ]
```

根据 [示例 5.27](#) 中的定义, 以下函数 `match_asian_cities` 将返回一个亚洲城市列表:

```
1 def match_asian_cities():
2     results = []
3     for city in cities:
4         match city:
5             case City(continent='Asia'): # 用关键字类模式, 匹配亚洲城市
6                 results.append(city)
7     return results
```

模式 (Pattern) `City(Continent='Asia')` 可以匹配任何 `Continent` 属性值为 'Asia' 的 `City` 实例, 无论其他属性值式什么。

如果你想收集 `country` 属性的值, 可以这样写 (如 [示例 5.28](#) 所示):

</> [示例 5.28: 类模式下, 捕获实例属性](#)

```
1 def match_asian_countries():
2     results = []
3     for city in cities:
4         match city:
5             case City(continent='Asia', country=cc): # 收集亚洲城市实例的 country
6                 results.append(cc)
7     return results
```

与前面一样, 模式 `City(continent='Asia', country=cc)` 也匹配亚洲城市。但是, 现在变量 `cc` (称为“模式变量”) 被绑定到了 `City` 实例的 `country` 属性上。这里“模式变量”的命名, 没有任何限制, 甚至可以与 `City` 实例的属性 `country` 同名 (如 [示例 5.29](#) 所示)。

</> 示例 5.29: 模式变量的命名,无任何限制

```

1  match city:
2      case City(continent='Asia', country=country):      # 等号左侧的 country 为实例
3          属性名, 右侧为 '模式变量'
4          results.append(country)

```

关键字类模式可读性很高,并且适用于任何具有公开 (public) 实例属性的类,不过有些繁琐。

有时候,使用位置类模式更方便。但是,要求 [匹配对象 \(Subject\)](#) 所属的类药显式支持位置类模式 (详见 5.8.3 节)。

5.8.3 位置类模式

根据 [示例 5.27 \(5.8.2\)](#) 中 City 类的定义,以下函数将使用位置类模式返回一个亚洲城市列表:

```

1  def match_asian_cities_pos():
2      results = []
3      for city in cities:
4          match city:
5              case City('Asia'):
6                  results.append(city)
7      return results

```

模式 (Pattern) City('Asia') 可以匹配第一个属性值为'Asia' 的 City 实例,无论其他属性值式什么。

如果你想收集 country 属性的值,可以这样写 (如 [示例 5.30](#)所示):

</> 示例 5.30: 位置模式下,捕获实例属性

```

1  def match_asian_countries_pos():
2      results = []
3      for city in cities:
4          match city:
5              case City('Asia', _, country):      # 收集亚洲城市实例的 country 属性值
6                  results.append(country)
7      return results

```

与前面一样,模式 City('Asia', _, Country) 也匹配亚洲城市。但是,现在 country 变量¹²被绑定到 City 实例的第 3 个属性 (即 country) 上。

我提到了“第 1 个属性”、“第 3 个属性”,但是此处的“第几个属性”表示什么意思呢?

City 类或任何类能够使用位置模式的原因是:存在一个名为 `__match_args__` 的特殊类属性。本章中的类构建器会自动创建该属性。如下是 City 类中 `__match_args__` 属性的值:

```

1  >>> City.__match_args__
2  ('continent', 'name', 'country')

```

如您所见,位置模式中属性的顺序就是 `__match_args__` 中声明的顺序。

[11.8](#) 将介绍如何为未使用类构建器创建的类,定义特殊的类属性 `__match_args__`。

¹²模式变量的命名没有任何限制,本例模式变量与实例属性使用相同的名字,即 country。



可以在一个 模式 (Pattern) 中同时使用关键字模式或者位置模式^a。
`_match_args_` 中列出的只是可供匹配的实例属性, 而不是全部属性。因此, 有时候可能需要在 模式 (Pattern) 中同时使用关键字模式和位置模式。

^a译者注: 原文此处为“关键字参数”和“位置模式”, 为了与本章节标题内容保持一致, 此处将“关键字参数”和“位置模式”分别翻译为“关键字模式”和“位置模式”。

又到了本章小结的时候了。

5.9 本章小结

本章的主要内容是数据类构建器 `collections.namedtuple`、`typing.NamedTuple` 和 `@dataclasses.dataclass`。我们看到, 每个构建器都可以根据传给工厂函数的参数, 生成数据类。`typing.NamedTuple` 与 `@dataclasses.dataclass` 还可以通过 `class` 语句为数据类提供 `类型提示 (Type Hints)`。`collections.namedtuple` 与 `typing.NamedTuple` 生成的数据类是 `tuple` 的子类; 与普通的 `tuple` 相比, 增加了通过字段名访问数据类字段的功能, 并提供了类属性 `_fields`, 以字符串元组的形式列出字段名称。

接下来, 将 3 个数据类放在一起, 研究了它们的主要功能, 包括如何以 `dict` 形式提取实例数据、如何获取字段名称与默认值, 以及如何根据现有实例创建新实例。

借此机会, 我们开始研究 `类型提示 (Type Hints)`, 尤其是那些用于在 `class` 语法中, 为属性增加 `变量注解 (Variable Annotation)` 的表示法 (Python 3.6 引入的 “PEP 526 -Syntax for Variable Annotations”)。一般来说, `类型提示 (Type Hints)` 最令人惊讶的地方可能是它们在运行时没有任何作用。Python 毕竟是一个动态语言。需要借助像 Mypy 这样的外部工具来利用 `类型提示 (Type Hints)` 信息, 通过对源代码的静态分析来检测错误。在对 “PEP 526” 中的语法进行基本概述后, 我们研究了 `类型提示 (Type Hints)` 在普通类中, 以及在由 `typing.NamedTuple` 和 `@dataclass` 构建的类中的效果。

接下来, 介绍了 `@dataclass` 的常用功能, 以及 `dataclasses.field` 函数的 `default_factory` 选项。我们还研究了在数据类的 `类型提示 (Type Hints)` 中, 非常重要的两个伪类型: `typing.ClassVar` 与 `dataclasses.InitVar`。随后, 以一个 `都柏林核心模式 (Dublin Core Schema)` 示例, 说明了如何在自定义 `_repr_` 中使用 `dataclasses.field` 函数 遍历 `Resource` 实例的属性。

然后, 告诫大家不要滥用数据类, 以免违背面向对象编程的基本原则——[数据及处理数据的函数应放在同一个类中](#)。不含业务逻辑的数据类, 可能表示您将处理逻辑放错了位置。

节 5.8 节中, 讲解了如何使用 `模式匹配 (Pattern Match)` 来匹配任意的类实例, 而不仅限于本章介绍的用类构建器构建的类。

5.10 延伸阅读

Python 标准库文档对数据类构建器的讲解很全面, 并且还有很多示例。特别是 PEP 557 -Data Classes 中关于 `@dataclass` 的大多数内容, 都被复制到了 `dataclasses` 模块文档 中。但是, PEP 557 中有几处内容较丰富的章节没有被复制, 包括 “[为何不直接使用 namedtuple](#)”、“[为何不直接使用 typing.NamedTuple](#)”, 以及 “[Rationale \(基本原理\)](#)” 3 个部分。其中, `Rationale` 一节还提出了一个问题, 并给出了解答。

什么时候不适合使用数据类?

- 需要兼容 tuple 或 dict 的 API 时。
- 需要超出 PEP 484 和 526 提供的类型验证时, 或者需要值验证或转换时。

——Eric V. Smith, PEP 557 “Rationale”

Geir Arne Hjelle 在 [RealPython.com](https://realpython.com) 上,写了一篇非常完整的文章,题目为 “Data Classes in Python 3.7+ (Guide)¹³”。

在 PyCon US 2018 上, Raymond Hettinger 发表了题为 “Dataclasses: The code generator to end all code generators¹⁴” (视频) 的演讲。

若想获得更多特性和高级功能 (如验证), 则可以研究一下比 dataclasses 早几年出现的 attrs 项目 (由 Hynek Schlawack 主导)。 attrs 项目提供了更多的功能, 并承诺 “通过将你从实现对象协议 (又称 dunder 方法¹⁵) 的繁重工作中解脱出来, 让你重拾编写类的乐趣”。Eric V. Smith 在 PEP 557 中感谢了 attrs 项目 对 @dataclass 的影响。Smith 所指的影响可能包括最重要的 API 决策——使用装饰器 (而不是基类和/或元类) 来完成这项工作¹⁶。

Glyph (Twisted 项目的创始人) 在 “The One Python Library Everyone Needs” 中, 对 attrs 项目 做了精彩介绍。 attrs 文档也包含了对替代方案的讨论。

本书作者、讲师和狂热的计算机科学家 Dave Beazley 编写了另一个数据类生成器——cluegen。如果你看过 Dave 的演讲, 你就会知道他是 Python 元编程大师。因此, 我从 cluegen 的 README.md 文件中, 了解到促使他编写 @dataclass 替代品的具体用例, 以及他的理念: 提供一种解决问题的方案, 而不是提供一种工具。出解决问题的 Dave Beazley 还是要实现一种替代方案。工具一开始是使用方便, 但是解决方案更加灵活, 并适用于各种场景。

将数据类视为一种 代码异味 (Code Smell), 我找到的最好的资料来源是 Martin Fowler 的《重构: 改善既有代码的设计》第 2 版。这一版去掉了本章开头引用的那句话, 即 “数据类就像孩子...”, 但仍不失为该书最好的版本。对于 Python 爱好者来说尤其如此, 因为书中的示例用的是现代的 JavaScript (更接近于 Python), 而不像第 1 版用的是 Java。

网站 Refactoring Guru 中, 也有关于 “数据类代码异味” 的描述。

杂谈

The Jargon File 中的词条 Guido, 讲的是 Guido van Rossum (Python 的创建者)。其中, 有一部分内容如下:

传说中, 除了 Python 本身之外最重要的特点是他的时光机。据说这是因为用户对新功能的请求经常会得到回应:“我昨晚刚实现了这个...”。

一直以来, Python 缺少在类中快速声明实例属性的标准语法。而许多面向对象的语言中都有这种语

¹³《Data Classes in Python 3.7+ (Guide)》:《Python 3.7 数据类终极指南》

¹⁴《Dataclasses: The code generator to end all code generators》:《Dataclasses: 代码生成器的终结者》

¹⁵dunder 方法: 即带双下划线的特殊方法, 如 `__init__`、`repr__`、`__eq__` 等。

¹⁶译者注: 此处的这项工作, 指的是通过装饰器来构建数据类。

法。下面是 Smalltalk 语言中定义 Point 类的一部分：

```
1 Object subclass: #Point
2   instanceVariableNames: 'x y'
3   classVariableNames: ''
4   package: 'Kernel-BasicObjects'
```

第 2 行列出了实例属性名称(即 x 与 y)。若有类属性的话，则会将它们放在第 3 行。

如果类属性有初始值, Python 总是提供一种简单的方法来声明它们。但实例属性更常用, Python 程序员们不得不在 `__init__` 方法中查找实例属性。而且, 总是担心在类的其他地方(甚至是其他类的外部函数或方法中)会创建实例属性。

现在好了, 我们有了 `@dataclass`。但是, 问题也随之而来。

首先, 使用 `@dataclass` 时不能省略 [类型提示 \(Type Hints\)](#)。过去 7 年间, “[PEP 484 –Type Hints](#)”给我们的承诺是: [类型提示 \(Type Hints\)](#) 将始终是可选的。而现在, `@dataclass` 这个重要的功能却要求必须提供 [类型提示 \(Type Hints\)](#)。若如果你对静态类型的趋势不感兴趣, 可以考虑使用 `attrs` 库来替代 `@dataclass` 装饰器。

其次, 用于注解实例和类属性的 [PEP 526 语法](#), 颠覆了 `class` 语句的既定约定: [以前, 在 class 块顶层声明的所有内容, 都是类属性 \(方法也是类属性\)。而有了 PEP 526 与 @dataclass 之后, 在顶层声明的带有类型提示 \(Type Hints\) 的属性都变成了实例属性。](#)

```
1 @dataclass
2 class Spam:
3   repeat: int      # 实例属性
```

下面的 `repeat` 也是一个实例属性:

```
1 @dataclass
2 class Spam:
3   repeat: int = 99 # 实例属性
```

但是, 如果没有 [类型提示 \(Type Hints\)](#), 一下就回到了从前——在类顶层声明的属性只属于类自身(即类属性), 如下所示:

```
1 @dataclass
2 class Spam:
3   repeat = 99      # 类属性
```

最后, 若要为类属性添加 [类型提示 \(Type Hints\)](#), 则不能使用常规类型。因为, 这样它将成为实例属性。必须使用 [伪类型 ClassVar](#)。

```
1 @dataclass
2 class Spam:
3   repeat: ClassVar[int] = 99 # 真乱！！！
```

这是例外中的例外(格式混乱), 我觉得这不太符合 Python 风格。

我未参与 “[PEP 526 –Syntax for Variable Annotations](#)” 或 “[PEP 557 –Data Classes](#)” 的讨论, 我希望

实现的是下面这种语法。

```
1 @dataclass
2 class HackerClubMember:
3     .name: str          ❶
4     .guests: list = field(default_factory=list)
5     .handle: str = ''
6
7     all_handles = set() ❷
```

- ❶ 声明实例属性时, 必须在前面加上符号.。
- ❷ 任何不带. 前缀的属性, 都是类属性(像以前一样)。

为此, 语法必须做出改变。我觉得这种语法的可读性很高, 而且看起来不那么乱。

真希望我能将 Guido 的时间机器借来一用, 回到 2017 年, 让 Python 核心团队采纳我的想法。

对象引用、可变性与垃圾回收

“你很伤心，”白骑士用忧虑的语气说：“让我唱首歌来安慰你吧。 [...] 这首歌的名字叫《黑线鳕的眼睛》。”“哦，这就是这首歌的名字，是吗？”爱丽丝说道，视图表现出感兴趣的样子。“不，你不懂，”白骑士说，看起来有些恼火。“这只是别人这么叫的。真正的名字是《年迈的老人》”

——Lewis Carroll,《爱丽丝镜中奇遇记》

爱丽丝与白骑士为本章要讨论的内容定下了基调。本章主题是对象与对象名称之间的区别。名称不是对象。名称是一个独立的东西。

本章开始，我们用一个比喻来解释 Python 中的变量：变量是标签，而不是盒子¹。若引用变量对你来说不是什么新鲜事，则这个比喻仍然有用，尤其是需要向其他人解释别名问题的时候。

然后，我们将讨论对象标识 (id)、对象值和别名的概念。随后，我们发现了 tuple 的一个令人惊讶的特征：虽然 tuple 是不可变的，但是其中的值却可能可变的。这引出了对浅拷贝与深拷贝的讨论。引用与函数参数是我们的下一个主题：可变参数默认值的问题，以及如何安全地处理函数调用者传入的可变参数。

本章的最后几节介绍了垃圾回收、del 命令，以及 Python 处理不可变对象的一些技巧。

本章内容相当枯燥，但这些内容却是解决 Python 程序中许多不易察觉的 bug 的关键。

6.1 本章新增内容

本章涵盖的内容非常基础，且非常稳定。第 2 版中没有什么大变化。

在 6.3.1 节的末尾，增加了一个用 is 来测试哨兵对象的示例，以及一个关于误用 is 运算符的警告。

本章原本在 四 的开头，第 2 版将其移到了前面。因为，我觉得这一章更适合作为 一 的结尾。

¹ 变量实际上并不存储值，而是指向内存中存储值的位置（相当于贴在内存位置上的标签）。因此，当你将一个变量赋值给另一个变量时，实际上是将另一个标签贴在相同的内存位置上。



本书第1版中有关弱引用(Weak Reference)的部分,现已成为 fluentpython.com 网站上的一篇文章。

首先,让我们先忘记“变量就像一个存储数据的盒子”这一概念,开始本章的学习。

6.2 变量不是盒子

1997年,我在MIT参加了Java暑期课程。Lynn Stein²教授指出,人们经常使用“变量是盒子”这样的比喻,实际上阻碍了对面向对象语言中引用变量的理解。Python变量就像Java中的引用变量;更好的比喻是将Python变量视为附加到对象上的名称标签。下一个示例与图,将帮助您理解为什么。

示例6.1是一个简单的交互,无法使用“变量是盒子”来解释。图6.1说明了为何Python中不能将变量视作盒子,而便利贴才是变量的真正用途。

</>示例6.1: 变量a和b保存对同一列表的引用,而不是列表的副本

```

1  >>> a = [1, 2, 3] ❶
2  >>> b = a ❷
3  >>> a.append(4) ❸
4  >>> b ❹
5  [1, 2, 3, 4]

```

- ❶ 创建列表[1, 2, 3],并将其绑定到变量a。
- ❷ 将变量b绑定到变量a引用的相同值。
- ❸ 通过附加一项,修改变量a引用的列表。
- ❹ 可通过变量b来观察效果。如果将变量b想象成一个盒子,其中存储了盒子a中的[1,2,3]的副本,那么此处的行为就说不通了。

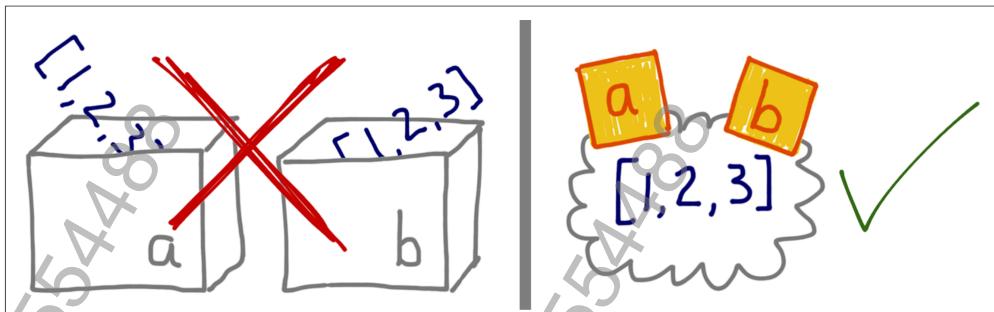


图6.1: 将变量视为盒子,无法解释Python中的赋值。
将变量视作便利贴,更易解释示例6.1的行为。

因此,b=a语句不会将盒子a中的内容复制到盒子b中。而是在已经贴了标签a的对象上,再贴一个标签b。

²Lynn Andrea Stein是一位屡获殊荣的计算机科学教育家,目前在Olin College of Engineering(奥林工程学院)任教。

Stein 教授还很谨慎地谈论了赋值的问题。例如,当谈到模拟中的 seesaw 对象时,她会说:“将变量 s 分配给 seesaw”,而绝不会说“seesaw 被分配给变量 s”。对于引用变量,说“变量被分配给对象”更合理,反过来说“对象被分配给变量”则会有问题。毕竟,对象是在赋值之前创建的。[示例 6.2](#) 证明了赋值语句的右侧先执行。

由于动词“分配 (assign)”在使用上存在矛盾,经常使用“绑定 (bind)”代替:Python 的赋值语句 x=... 将名称 x 绑定到右侧已创建的(或引用的)对象。并且在绑定名称之前,对象必须已存在,示例 6.2 证明了这一点。

</> [示例 6.2:](#) 创建对象之后,才能绑定变量

```

1  >>> class Gizmo:
2      ...     def __init__(self):
3      ...         print(f'Gizmo id: {id(self)}')
4      ...
5  >>> x = Gizmo()
6  Gizmo id: 4301489152      ❶
7  >>> y = Gizmo() * 10      ❷
8  Gizmo id: 4301489432      ❸
9  Traceback (most recent call last):
10     File "<stdin>", line 1, in <module>
11     TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
12  >>>
13  >>> dir()                ❹
14  ['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
15  '__package__', '__spec__', 'x']
```

- ❶ 输出的 Gizmo id: 4301489432 是创建 Gizmo 实例的副作用。
- ❷ 对 Gizmo 实例进行乘法运算,将引发异常。
- ❸ 这里表明,在尝试乘法运算之前,会先创建一个新的 Gizmo 实例 (id 为 4301489432)。
- ❹ 但是,变量 y 从未被创建。因为在对赋值语句的右侧进行求解时,引发了异常。



要理解 Python 中的赋值,首先要阅读赋值语句右侧的内容:这是创建或检索对象的地方。然后,左侧的变量与右侧的对象绑定,就像为对象贴了一个标签一样。忘掉盒子吧。

因为变量只是标签,所以没有什么可以阻止为一个对象绑定多个标签。多个标签,将成为“别名”,这是我们下一节的主题。

6.3 同一性、相等性与别名

Lewis Carroll 是 Charles Lutwidge Dodgson 教授的笔名。Carroll 先生不仅等同于 Dodgson 教授,他们还是同一人。[示例 6.3](#) 用 Python 表达了这一思想。

</> [示例 6.3:](#) charles 与 lewis 引用同一对象

```

1  >>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
2  >>> lewis = charles      ❶
```

```

3  >>> lewis is charles
4  True>>> id(charles), id(lewis) ②
5  (4300473992, 4300473992)
6  >>> lewis['balance'] = 950 ③
7  >>> charles
8  {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}

```

- ① lewis 是 charles 的别名。
- ② 通过 id() 函数的结果, 可以确认这一点。
- ③ 向 lewis 添加一项, 相当于向 charles 也添加了一项。

然而, 假设有一个冒名顶替者——姑且称他为 Alexander Pedachenko 博士——声称自己是出生于 1832 年的 Charles L. Dodgson。此冒充者的证件可能是一样的, 但是 Pedachenko 博士不是 Dodgson 教授。此种情况, 如图 6.2 所示。

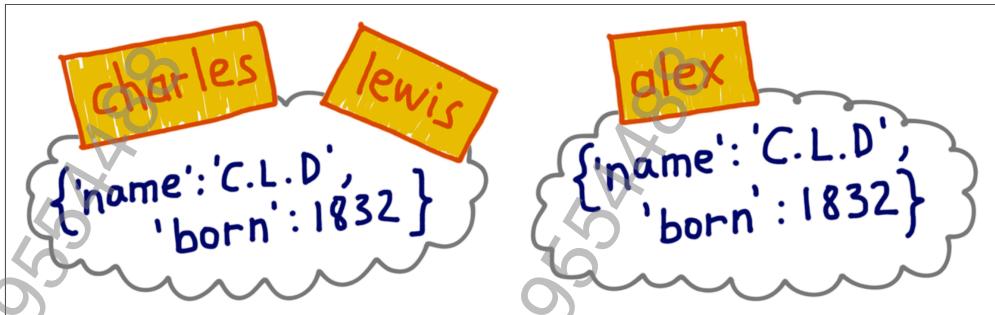


图 6.2: charles 和 lewis 被绑定到同一个对象; alex 被绑定到一个等值的不同对象。

示例 6.4 实现并测试了 图 6.2 所示的 alex 对象。

</> 示例 6.4: 经比较, alex 与 charles 相等, 但 alex 不是 charles。

```

1  >>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950} ①
2  >>> alex == charles ②
3  True
4  >>> alex is not charles ③
5  True

```

- ① alex 引用的对象是分配给 charles 的对象的副本。
- ② 由于 dict 类的 __eq__ 实现, 对象的比较结果是相等的。
- ③ 但它们是不同的对象。在 Python 中, 用 a is not b 来判断 a 与 b 两个对象是否不同。

示例 6.3 用到了别名。在该代码中, lewis 和 charles 就是别名——即两个变量被绑定到同一对象。但 alex 不是 charles 的别名。因为, 二者被绑定到了不同的对象。绑定到变量 alex 与 charles 的这两个对象, 拥有相同的值 (== 比较的就是对象的值), 但是它们的标识 (id) 不同。

《Python 语言参考中》中的“3.1 节对象、值和类型”指出:

对象一旦被创建, 其标识 (ID) 将始终不变。可将标识视为对象在内存中的地址。is 运算符比较两个对象的标识。id() 函数返回表示对象标识的整数。

对象 ID 的真正含义取决于具体的 Python 实现。在 CPython 中, `id()` 返回对象的内存地址。但在另一个 Python 解释器中, 它可能是其他内容。关键的一点是, ID 保证是一个唯一的整数标签, 并且在对象的生命周期内不会改变。

在实际编程中, 很少使用 `id()` 函数。对象身份的检查通常用 `is` 运算符来完成, 它会直接比较对象的 ID, 而无需显式调用 `id()` 函数。接下来, 我们将讨论 `is` 与 `==`。



本书的技术审校 Leonardo Rochael 指出, `id()` 函数最常用于调试。当两个对象的 `repr()` 看起来相似, 但您需要了解两个引用是别名, 还是指向不同的对象。若引用处于不同的上下文中(如不同的堆栈帧), 则不能用 `is` 运算符比较两个引用。

6.3.1 在 `==` 和 `is` 之间选择

`==` 运算符比较对象的值(它们保存的数据), 而 `is` 则比较对象的身份(ID)。

在编程期间, 我们通常更关系对象的值, 而不是对象的身份。因此, 运算符 `==` 在 Python 代码中出现的频率更高。

但是, 若要将变量与 [单例 \(Singleton\)](#) 进行比较, 则应该使用 `is` 运算符。到目前为止, 最常见的情况是用 `is` 检查变量绑定的值是不是 `None`。推荐的写法如下:

```
1 x is None
```

其否定形式的正确写法如下:

```
1 x is not None
```

`None` 是我们用 `is` 测试时, 最常见的 [单例 \(Singleton\)](#)。哨兵 ([Sentinel](#)) 对象是我们使用 `is` 测试的另一个单例。下面是创建和测试 [哨兵 \(Sentinel\)](#) 对象的一种方法:

```
1 END_OF_DATA = object()
2 # ... 省略很多行
3 def traverse(...):
4     # ... 省略很多行
5     if node is END_OF_DATA:
6         return
7     # 等等...
```

`is` 运算符比 `==` 更快, 因为它不能被重载。所以, Python 不必查找和调用特殊方法来计算它, 而是直接比较两个整数 ID。相反, `a==b` 其实是 `a.__eq__(b)` 的语法糖 ([Syntactic Sugar](#))。从 `object` 继承而来的 `__eq__` 方法比较两个对象的 ID, 所以其结果与 `is` 一样。但是, 大多数内置类型都会使用更有意义的实现, 来覆盖默认的 `__eq__` 方法, 将对象的属性值纳入比较的范围。相等测试可能涉及大量处理工作, 例如, 在比较大型集合或嵌套层级较深的结构时。



通常, 我们更关心的是对象的相等性, 而不是同一性。通常, `is` 运算符只用于检查 `None`。我们在审查代码时, 看到的大多数其他 `is` 用法都是错误的。如果您不确定, 可以使用 `==`。它通常可以满足你的需要, 而且也适用于 `None` (尽管速度没那么快)。

为了结束对同一性与相等性的讨论, 我们将会发现不可变的 `tuple` (元组), 并不像你想象的那样一成不变。

6.3.2 元组的相对不可变性

`tuple` (元组) 与大多数 Python 容器类型 (如 `list`、`dict`、`set` 等) 一样: 它们持有对象的引用³, 而不是对象本身。如果引用的项是可变的, 则即便 `tuple` (元组) 本身未发生变化, 它们也可能会发生变化。换句话说, `tuple` (元组) 的不可变性其实是指 `tuple` 数据结构的物理内容 (即它所持有的引用) 不可变, 而不包括所引用的对象。

示例 6.5 说明了一个 `tuple` (元组) 的值由于元组中引用的可变对象的更改, 而发生变化的情况。`tuple` (元组) 中永远不会改变的是它所包含的项的标识 (ID)。

</> 示例 6.5. 起初, `t1` 与 `t2` 相等。修改 `t1` 中一个可变项后, 二者不等了。

```

1  >>> t1 = (1, 2, [30, 40]) ❶
2  >>> t2 = (1, 2, [30, 40]) ❷
3  >>> t1 == t2            ❸
4  True
5  >>> id(t1[-1])        ❹
6  4302515784
7  >>> t1[-1].append(99) ❺
8  >>> t1
9  (1, 2, [30, 40, 99])
10 >>> id(t1[-1])        ❻
11 4302515784
12 >>> t1 == t2          ❼
13 False

```

- ❶ `t1` 是不可变的, 但是 `t1[-1]` (即 `[30, 40]`) 是可变的。
- ❷ 构建元组 `t2`, 其所持有的项与 `t1` 相同。
- ❸ 虽然 `t1` 与 `t2` 引用了不同的对象, 但是二者相等——符合预期。
- ❹ 检查 `t1[-1]` 处列表的 ID 标识。
- ❺ 就地更改 `t1[-1]` 处的列表。
- ❻ `t1[-1]` 的 ID 标识未改变, 只是它的值变了。
- ❼ 现在, `t1` 与 `t2` 不相等。

`tuple` (元组) 的这种相对不可变性, 解释了“[2.8.3 一个 `+=` 运算符赋值谜题](#)”节的谜题。这也是有些 `tuple` (元组) 不可哈希 (`hashable`) (见[3.4.1](#)节) 的原因。

³相比之下, `str`、`bytes` 与 `array.array` 等扁平序列不包含引用, 而是直接将其内容 (字符、字节、数字) 保存在连续的内存中。

当需要复制一个对象时,相等性与同一性之间的区别会产生更进一步的影响。副本是具有不同 ID 的相等对象(即对象 ID 不同,但内容相等)。但是,对象中包含其他对象,那么副本也应该复制内部对象么?可以共享内部对象吗?这些问题没有唯一的答案。我们将在 6.4 节,进行进一步讨论。

6.4 默认做浅拷贝

复制 list(或多数的内置可变容器类型)的最简单方法是使用类型本身的内置构造函数。例如:

```
1 >>> l1 = [3, [55, 44], (7, 8, 9)]
2 >>> l2 = list(l1)      ❶
3 >>> l2
4 [3, [55, 44], (7, 8, 9)]
5 >>> l2 == l1          ❷
6 True
7 >>> l2 is l1          ❸
8 False
```

- ❶ list(l1) 会创建 l1 的副本。
- ❷ 副本与源列表相等。
- ❸ 但是,二者引用了不同的对象。

对于列表与其他可变序列,快捷方式 l2=l1[:] 也会生成一个副本。

但是,使用构造函数或 [:] 会生成浅拷贝(即,最外层的容器被复制,但副本中填充了对原始容器所保存的相同项的引用)。若所有项都是不可变的,则可以节省内存并且不会导致任何问题。但是,如果存在可变项,则可能会导致令人不悦的意外。

在 [示例 6.6](#) 中,我们为一个 list(其中包含另一个 list 和一个 tuple)创建浅拷贝。然后,对浅拷贝进行更改,以查看更改对引用对象的影响。



强烈建议将 [示例 6.6](#) 的代码复制粘贴到 [Online Python Tutor](#) 网站中。并在网站上观看 [示例 6.6](#) 的交互式动画。当我写这篇文章时,无法直接提供 [pythontutor.com](#) 网站中已准备好的示例链接。但这个工具非常棒,值得花点时间复制粘贴代码。

</> [示例 6.6](#): 对包含另一个 list 的 list 做浅拷贝。建议将代码复制到 [Online Python Tutor](#) 网站,查看效果。

```
1 l1 = [3, [66, 55, 44], (7, 8, 9)]
2 l2 = list(l1)      ❶
3 l1.append(100)    ❷
4 l1[1].remove(55)  ❸
5 print('l1:', l1)
6 print('l2:', l2)
7 l2[1] += [33, 22]  ❹
8 l2[2] += (10, 11)  ❺
9 print('l1:', l1)
10 print('l2:', l2)
```

- ❶ `l2` 是 `l1` 的浅层拷贝。这种状态如图所示。
 - ❷ 将 100 附加到 `l1`, 对 `l2` 没有影响。
 - ❸ 在这里, 我们从内层列表 `l1[1]` 中删除 55。这会影响到 `l2`, 因为 `l2[1]` 与 `l1[1]` 绑定在同一个 list 中。
 - ❹ 对于可变对象来说 (如由 `l2[1]` 引用的 list, 即 `[66,55,44]`)。运算符 `+=` 会就地更改 list。此更改在 `l1[1]` 中也有体现, 因为, `l1[1]` 是 `l2[1]` 的别名。
 - ❺ tuple 上的 `+=` 运算符, 会创建一个新的 tuple, 并将变量 `l2[2]` 重新绑定到新创建的 tuple。这等同于执行了 `l2[2] = l2[2] + (10,11)`。现在, `l1` 与 `l2` 最后一个位置上的 tuple 不再是同一个对象 (如图 6.3 所示)。

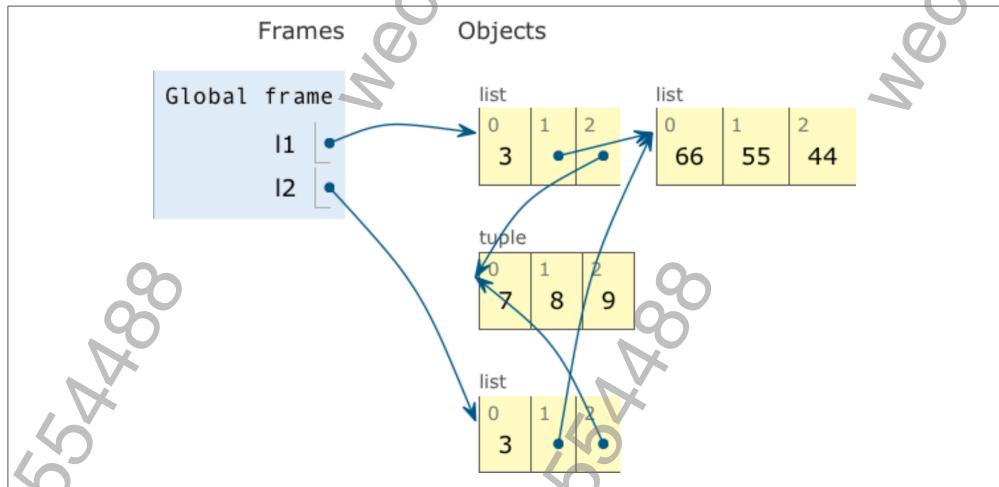


图 6.3: 示例 6.6 执行 `l2=list(l1)` 赋值后的程序状态⁴。

示例 6.6 的输出如示例 6.7 所示, 对象的最终状态如图 6.4 所示。

</> 示例 6.7：示例 6.6 的输出

```
1 l1: [3, [66, 44], (7, 8, 9), 100]
2 l2: [3, [66, 44], (7, 8, 9)]
3 l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
4 l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

现在应该清楚了，浅拷贝（shadow copy）操作简单。但是，得到的结果可能并不是你想要的。接下来，说明如何创建深拷贝（deep copy）。

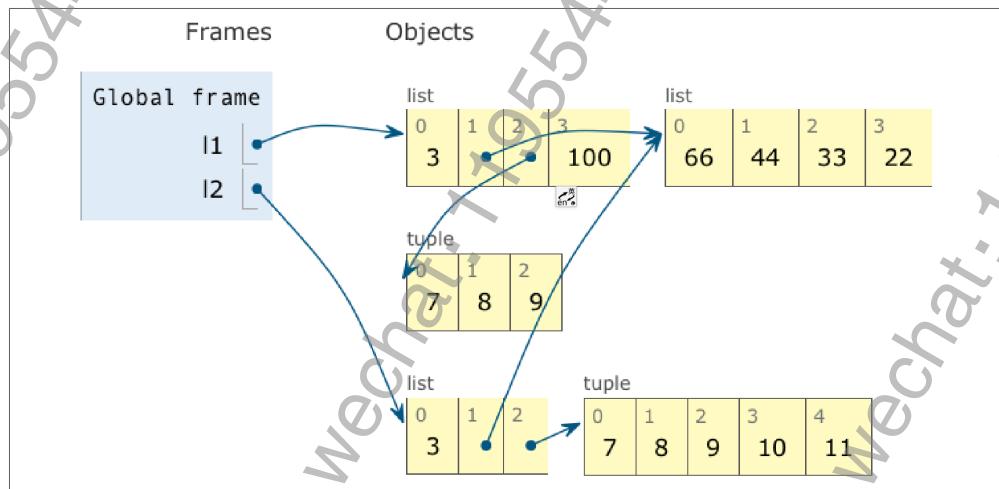
6.4.1 任意对象的深拷贝与浅拷贝

一般来说，浅拷贝也没什么问题，但有时我们需要的是深拷贝（即副本不共享内部对象的引用）。[模块 copy](#) 提供的函数 `copy` 与 `deepcopy`，可以分别对任意对象执行浅拷贝与深拷贝。

为了演示 `copy()` 与 `deepcopy()` 的用法, [示例 6.8](#) 定义了一个简单类 `Bus`, 用于表示运载乘客的校车, 乘客在途中上有上下。

41) 与 l2 引用不同的 list,但是 l1[1] 与 l2[1] 引用的相同对象,即列表 [66,55,44];l1[2] 与 l2[2] 引用的相同对象,即元组 (7,8,9)。图表由 [Online Python Tutor](#) 网站生成。

⁵|1 与 |2 仍然共享对同一个列表对象的引用,现在包含 [66,44,33,22]。但是,操作 |2[2]+=(10,11) 创建了一个内容为 (7,8,9,10,11) 的新元组,与 |1[2] 引用的元组 (7,8,9) 无关。图表由 [Online Python Tutor](#) 网站生成。

图 6.4: l1 和 l2 的最终状态⁵

</> 示例 6.8: 校车乘客在途中有上有下

```

1  class Bus:
2
3      def __init__(self, passengers=None):
4          if passengers is None:
5              self.passengers = []
6          else:
7              self.passengers = list(passengers)
8
9      def pick(self, name):
10         self.passengers.append(name)
11
12     def drop(self, name):
13         self.passengers.remove(name)

```

接下来,在 [示例 6.9](#) 中的交互式控制台中,我们将创建一个 Bus 对象 (bus1) 与两个副本。一个是浅拷贝副本 (bus2),另一个是深拷贝副本 (bus3)。看看 bus1 有学生下车后,会发生什么?

</> 示例 6.9: copy 与 deepcopy 产生的不同效果

```

1  >>> import copy
2  >>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
3  >>> bus2 = copy.copy(bus1)
4  >>> bus3 = copy.deepcopy(bus1)
5  >>> id(bus1), id(bus2), id(bus3)
6  (4301498296, 4301499416, 4301499752) ❶
7  >>> bus1.drop('Bill')
8  >>> bus2.passengers
9  ['Alice', 'Claire', 'David'] ❷
10 >>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
11 (4302658568, 4302658568, 4302657800) ❸
12 >>> bus3.passengers
13 ['Alice', 'Bill', 'Claire', 'David'] ❹

```

- ❶ 用 `copy` 与 `deepcopy`, 创建 3 个不同的 Bus 实例。
- ❷ `bus1` 中的 'Bill' 下车后, 他也从 `bus2` 上消失了。
- ❸ 检查 `passengers` 属性后发现, `bus1` 与 `bus2` 共享同一个列表对象。因为, `bus2` 是 `bus1` 的浅拷贝副本。
- ❹ `bus3` 是 `bus1` 的深拷贝副本。因此, 它的 `passengers` 属性引用另一个列表。

请注意, 一般情况下, 深拷贝并不是一件简单的事。因为对象可能存在循环引用, 简单的算法可能会进入无限循环。为解决此问题, 函数 `deepcopy` 会记住已经拷贝的对象, 来处理循环引用, 以使拷贝过程更加健壮。如示例 6.10 所示。

</> 示例 6.10: 循环引用, b 引用 a, 又将 b 追加到 a 中, `deepcopy` 会想办法复制 a。

```

1  >>> a = [10, 20]
2  >>> b = [a, 30]
3  >>> a.append(b)
4  >>> a
5  [10, 20, [[...], 30]]
6  >>> from copy import deepcopy
7  >>> c = deepcopy(a)
8  >>> c
9  [10, 20, [[...], 30]]

```

此外, 某些情况下, 深拷贝可能太深。例如, 对象可能引用了不应复制的外部资源或单例 (Singleton)。通过实现特殊方法 `__copy__` 与 `__deepcopy__`, 可以控制 `copy` 与 `deepcopy` 的行为, 详见 “[copy 模块文档](#)”。

通过别名共享对象, 还能解释 Python 中参数传递的原理, 以及用可变类型作为参数默认值的问题。接下来, 将讨论这些问题。

6.5 作为引用的函数参数

Python 唯一支持的参数传递模式是 [共享调用 \(Call by Sharing\)](#)。这与大多数面向对象语言 (包括 JavaScript、Ruby 与 Java⁶) 中使用的模式相同。[共享调用 \(Call by Sharing\)](#) 意味着函数的每个 **形参 (Parameter)** 都会获得 [实参 \(Argument\)](#) 中每个引用的副本。换句话说, **函数内部的形参 (Parameter)** 是 [实参 \(Argument\)](#) 的别名。

此种传参模式的结果是, 函数可以更改作为参数传入的可变对象, 但不能更改这些可变对象的标识 (即不能将一个对象彻底替换为另一个对象)。示例 6.11 展示了一个简单函数, 它在形参上调用 `+=` 运算符。分别将数字、列表、元组传给该函数, 传入的 [实参 \(Argument\)](#) 会以不同的方式, 受到影响。

</> 示例 6.11: 函数可以更改它接收的可变对象

```

1  >>> def f(a, b):
2  ...     a += b
3  ...     return a
4  ...
5  >>> x = 1
6  >>> y = 2
7  >>> f(x, y)

```

⁶Java 中的引用类型, 按 [共享调用 \(Call by Sharing\)](#) 传参; 而原始类型按值传参。

```
8 3
9 >>> x, y(1, 2)
10 >>> a = [1, 2]
11 >>> b = [3, 4]
12 >>> f(a, b)
13 [1, 2, 3, 4]
14 >>> a, b
15 ([1, 2, 3, 4], [3, 4])
16 >>> t = (10, 20)
17 >>> u = (30, 40)
18 >>> f(t, u)
19 (10, 20, 30, 40)
20 >>> t, u
21 ((10, 20), (30, 40))
```

① 数字 x 未更改。

② 列表 a 已更改。

③ 元组 t 未更改。

与函数参数相关的另一个问题是使用可变值（如 list 等）作为默认值。[6.5.1](#) 节将讨论这一点。

6.5.1 不要使用可变类型作为参数的默认值

可以为可选参数指定默认值，是 Python 函数定义的一个特性。此特性可以使我们的 API 在演进的同时，还能保证向后兼容。但是，应避免将可变对象用作参数的默认值。

为说明这一点，在 [示例 6.12](#) 中，我们在 [示例 6.8](#)（[6.4.1](#) 节）中 Bug 类的基础之上，更改其 `__init__` 方法，来定义 `HauntedBus` 类。此处，我们试图耍个“小聪明”：`passengers` 的默认值不是 `None`，而是 `[]`，从而避免了之前 `__init__` 中的 `if` 语句。但是，这种“小聪明”却给我们带来了麻烦。

</> [示例 6.12](#): 用一个简单的类，来说明可变默认值的危险

```
1 class HauntedBus:
2     """被幽灵乘客困扰的校车模型"""
3
4     def __init__(self, passengers=[]): ❶
5         self.passengers = passengers ❷
6
7     def pick(self, name):
8         self.passengers.append(name) ❸
9
10    def drop(self, name):
11        self.passengers.remove(name)
```

❶ 当未传入参数 `passengers` 时，则为 `passengers` 绑定默认的列表对象（最初为 `[]` 空列表）。

❷ 该赋值使 `self.passengers` 成为 `passengers` 的别名。而当未提供参数 `passengers` 时，`passengers` 本身又是默认列表的别名。

❸ 当在 `self.passengers` 上调用 `.remove()` 与 `.append()` 方法时，修改的其实是默认列表。而默认列表是函

数对象的一个属性。

示例 6.13 展示了 HauntedBus 的诡异行为。

</> 示例 6.13: 备受“幽灵乘客”折磨的校车

```

1  >>> bus1 = HauntedBus(['Alice', 'Bill']) ①
2  >>> bus1.passengers
3  ['Alice', 'Bill']
4  >>> bus1.pick('Charlie')
5  >>> bus1.drop('Alice')
6  >>> bus1.passengers ②
7  ['Bill', 'Charlie']
8  >>> bus2 = HauntedBus() ③
9  >>> bus2.pick('Carrie')
10 >>> bus2.passengers
11 ['Carrie']
12 >>> bus3 = HauntedBus() ④
13 >>> bus3.passengers ⑤
14 ['Carrie']
15 >>> bus3.pick('Dave')
16 >>> bus2.passengers ⑥
17 ['Carrie', 'Dave']
18 >>> bus2.passengers is bus3.passengers ⑦
19 True
20 >>> bus1.passengers ⑧
21 ['Bill', 'Charlie']

```

出版社

① 起初, bus1 有 2 位乘客。

② 目前为止, 一切顺利: bus1 未出现意外。

③ 起初, bus2 是空的。因此, 默认的空列表 [] 被分配给 self.passengers。

④ bus3 起初也是空的, 同样也分配了默认的空列表 []。

⑤ 但是, 默认的列表却不为空。

⑥ 现在, 登上 bus3 的 Dave, 出现在了 bus2 中。

⑦ 问题: bus2.passengers 与 bus3.passengers 引用同一个列表。

⑧ 但是, bus1.passengers 是不同的列表。

问题在于, 未指定初始乘客 (即未提供 passengers 参数值) 的 HauntedBus 实例, 共享了同一个乘客列表。

此类问题很难发现。如 [示例 6.13](#) 所示, 实例化 HauntedBus 时, 若传入了 passengers 参数, 则一切正常。只有不为 HauntedBus 指定 passengers 参数, 才会发生奇怪的事情。因为, 此时 self.passengers 变成了 passengers 参数默认值 (即 []) 的别名。出现此问题的根源是: 每个默认值都会在函数定义时 (通常在加载模块时) 进行求解, 并且默认值会成为函数对象的属性。因此, 若默认值是可变对象, 并且您对它进行了更改, 则此更改将影响函数后续的每次调用。

运行 [示例 6.12](#) 中的代码之后, 可以检查 HauntedBusBus.__init__ 对象, 并查看“幽灵学生”在其 defaults__ 属性中出现的情况 (如下所示)。

```

1 >>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
2 ['__annotations__', '__call__', ..., '__defaults__', ...]>>> HauntedBus.__init__
3 __defaults__
4 (['Carrie', 'Dave'],)

```

最后,我们可以验证 bus2.passengers 是绑定到 HauntedBus.__init__.__defaults__ 属性的第一个元素的别名

```

1 >>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
2 True

```

可变默认值的问题,解释了为什么 None 通常被用作可能接收可变值的参数的默认值。在 [示例 6.8 \(6.4.1\)](#) 中, __init__ 方法检查参数 passengers 是否为 None。若 passengers 是 None, 则 self.passengers 将绑定到一个新的空列表(即 [])。若 passengers 不是 None, 则正确的实现方式是将参数 passengers 的副本绑定到 self.passengers。下一节将解释为什么复制 [实参 \(Argument\)](#) 才是最佳实践。

6.5.2 可变参数的防御性编程

在编写接收可变参数的函数时,应充分考虑调用者是否希望修改传入的参数。

例如,如果您的函数接收到一个 dict,并需要在处理过程中对其进行修改,那么这个副作用是否应该在函数外部可见?事实上,这取决于上下文。这实际上是一个协调函数编码者和调用者的期望的问题。

下面是本章最后一个有关校车的示例。在此示例中, TwilightBus 实例与客户端共享乘客列表,这会产生意料之外的结果。在研究其具体实现之前,先从客户端角度看 TwilightBus 类是如何工作的(如 [示例 6.14](#) 所示)。

</> [示例 6.14](#): 乘客从 TwilightBus 下车后,就消失了

```

1 >>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ❶
2 >>> bus = TwilightBus(basketball_team) ❷
3 >>> bus.drop('Tina') ❸
4 >>> bus.drop('Pat')
5 >>> basketball_team ❹
6 ['Sue', 'Maya', 'Diana']

```

- ❶ basketball_team 中有 5 个学生名字。
- ❷ 使用列表 basketball_team 实例化了 TwilightBus。
- ❸ 学生 Tina 先下了车,随后 Pat 也下了车。
- ❹ 下车的乘客,从 basketball_team 中消失了。

TwilightBus 违反了接口设计的最佳实践“[最少惊讶原则 \(Principle of Least Astonishment\)](#)”⁷。令人惊讶的是,当学生离开校车后,其名字就会从 basketball_team 中消失。

[示例 6.15](#) 是 TwilightBus 的实现,随后解释了出现此问题的原因。

</> [示例 6.15](#): 一个简单的类,说明接收可变参数的风险

⁷见维基百科“[Principle of least astonishment](#)”。

```

1 class TwilightBus:
2     """让乘客消失的校车"""
3
4     def __init__(self, passengers=None):
5         if passengers is None:
6             self.passengers = [] ❶
7         else:
8             self.passengers = passengers ❷
9
10    def pick(self, name):
11        self.passengers.append(name)
12
13    def drop(self, name):
14        self.passengers.remove(name) ❸

```

- ❶ 此处谨慎处理。当 passengers 为 None 时, 创建一个新的空列表 []。
- ❷ 然而, 这个赋值语句使 self.passengers 成为 passengers 的别名。而 passengers 本身又是传给 __init__ 的 **实参 (Argument)** 的别名 (即 [示例 6.14](#) 中的 basketball_team)。
- ❸ 当在 self.passengers 上调用.remove() 与.append() 方法时, 实际上是在修改传给构造方法的原始列表 (即 [示例 6.14](#) 中的 basketball_team)。

这里的问题是, 校车为传给构造方法的列表创建了别名。正确的做法是: 校车应自己维护乘客列表。修正方法很简单: 像 [示例 6.8 \(6.4.1 节\)](#) 那样, 在 __init__ 中, 当提供参数 passengers 时, 应使用它的副本来自始化 self.passengers (如下所示)。

```

1 def __init__(self, passengers=None):
2     if passengers is None:
3         self.passengers = []
4     else:
5         self.passengers = list(passengers) ❶

```

- ❶ 创建 passengers 列表的副本。若传入的不是列表, 则将其转换为列表。

现在, 我们对乘客列表 (self.passengers) 的内部处理, 不会影响初始化校车时传入的参数。另外, 此种处理方式还更加灵活: 现在, 传给参数 passengers 的值可以是元组, 或任何其他可迭代对象。例如, 一个 set, 甚至数据库的查询结果。因为, 列表构造函数 list() 可接受任何可迭代对象。当我们自己创建并管理列表时, 可以确保.pick() 与.drop() 方法内的.remove() 和.append() 操作都能正常执行。



除非某个方法确实要更改通过参数传入的对象, 否则在将 **实参 (Argument)** 对象赋值给类中的实例变量之前, 一定要三思。因为, 这样会为 **实参 (Argument)** 对象创建别名。如果不确定, 那就创建一个副本, 免得客户端麻烦。当然, 创建副本会消耗一定的 CPU 与内存。但是, 在与速度和资源相比, 在 API 中埋下难以察觉的 bug, 显然是更严重的问题。

接下来, 谈一谈 Python 中最容易被误解的一个语句: del。

6.6 del 与垃圾回收

对象绝不会被显式销毁；但是，当它们变得不可达时，可能会被垃圾回收。

——《Python 语言参考》，第 3 章“数据模型”

第一个令人惊讶的事实是：del 不是函数，而是语句。写作 `del x`，而不是 `del(x)`。虽然，后一种写法也起作用，但这仅仅是因为在 Python，`x` 与 `(x)` 这两个表达式，往往是同一个意思。

第二个令人惊讶的事实是：del 删除的是引用，而不是对象。若删除的变量是对象的最后一个引用，那么 Python 的垃圾回收器可能会因为 del 的间接结果，而从内存中删掉一个对象。重新绑定变量也可能导致对象的引用数归零，从而导致对象被销毁。

```

1  >>> a = [1, 2] ❶
2  >>> b = a ❷
3  >>> del a ❸
4  >>> b ❹
5  [1, 2]
6  >>> b = [3] ❺

```

- ❶ 创建对象 `[1,2]`，为其绑定变量 `a`。
- ❷ 变量 `b` 也被绑定到对象 `[1,2]`。
- ❸ 删除引用 `a`。
- ❹ 对象 `[1,2]` 不受影响。因为还有引用它的变量 `b`。
- ❺ 将变量 `b` 重新绑定到另一个对象。对象 `[1,2]` 的最后一个引用也随之删除。现在，垃圾回收程序可以销毁对象 `[1,2]` 了。



虽然有一个特殊方法 `__del__`，但它不负责销毁实例，而且不应在代码中调用它。当实例即将被销毁时，Python 解释器会调用 `__del__`，来给实例释放外部资源的机会。自己编写的代码中，很少需要实现 `__del__` 方法。有些 Python 程序员会花时间实现 `__del__`，但吃力不讨好。因为正确使用 `__del__` 是一个棘手的问题。详见《Python 语言参考》第 3 章“数据模型”中对特殊方法 `__del__` 的说明。

在 CPython 中，垃圾回收的主要算法是“引用计数 (Refcount)”。本质上，每个对象都会记录指向它的引用数量。当引用计数归零时，对象即可立即被销毁：CPython 在对象上调用 `__del__` 方法（如已定义），然后释放分配给对象的内存。CPython 2.0 增加了“分代垃圾回收 (generational garbage collection)”算法，用于检测引用循环中涉及的对象组——若一组对象之间全都是相互引用，即使再出色的引用方式也会导致该组中的对象是不可达的。其他的一些 Python 实现，垃圾回收更复杂。它们不依赖于引用计数，这意味着即使对象的引用计数为零，可能也不会立即调用 `__del__` 方法。有关 `__del__` 的不当与正确用法，请参阅“[PyPy, Garbage Collection, And A Deadlock](#)”（作者：A. Jesse Jiryu Davis）。

为了掩饰对象生命周期终结时的情形，[示例 6.16](#) 用 `weakref.finalize`⁸ 注册了一个在对象销毁时，被调用

⁸`weakref.finalize(obj, func, /, *args, **kwargs)`：用于为 `obj` 注册回调函数 `func`。当 `obj` 引用的对象被垃圾回收时，会自动调用回调函数 `func`。该函数返回一个可调用的终结器 (finalizer) 对象，通过终结器对象的 `alive` 属性可查看对象是否已被销毁。

的回调函数 (Callback Function)。

</> 示例 6.16: 当对象的引用计数归零时, 监控对象生命周期终结的情形

```

1  >>> import weakref
2  >>> s1 = {1, 2, 3}
3  >>> s2 = s1      ❶
4  >>> def bye():    ❷
5      ...     print('...like tears in the rain.')
6      ...
7  >>> ender = weakref.finalize(s1, bye)  ❸
8  >>> ender.alive      ❹
9  True
10 >>> del s1
11 >>> ender.alive      ❺
12 True
13 >>> s2 = 'spam'      ❻
14 ...like tears in the rain.
15 >>> ender.alive
16 False

```

- ❶ s1 与 s2 是引用同一对象 1, 2, 3 的别名。
- ❷ 需要确保回调函数 (此处为 bye) 不依赖于即将被销毁的对象, 也不能持有对即将被销毁对象的引用。
- ❸ 在 s1 引用的对象上, 注册回调函数 (bye)。
- ❹ 在调用 finalize 对象之前,.alive 属性为 True。
- ❺ 如前所述, del 并未删除对象, 只是删除了对象的引用 s1。
- ❻ 重新绑定最后一个引用 s2, 会导致对象 1,2,3 不可达。对象被销毁了, 调用了回调函数 bye, ender.alive 的值变为 False。

示例 6.16 的目的是明确指出: del 不会删除对象, 但对象可能会因为使用 del 后, 导致无法访问而被删除。

您可能会好奇, 为何 示例 6.16 中的对象 1,2,3 被销毁了? 毕竟, s1 引用已被传递给 finalize 函数, 而 finalize 函数必须保留该引用, 才能监控对象并调用 回调函数 (Callback Function)。这是因为 finalize 只是持有对 1,2,3 的 “弱引用 (Weak Reference)”。对象的 弱引用 (Weak Reference) 不会增加对象的引用计数。因此, 弱引用 (Weak Reference) 不会阻止目标对象被垃圾回收。弱引用在缓存应用程序中非常有用, 因为我们不希望由于存在对缓存对象的引用, 而导致该对象无法被删除。



弱引用 (Weak Reference) 是个非常专业的话题。这就是为何我选择在第 2 版中跳过它的原因。但是, 我在 fluentpython.com 上发布了关于弱引用的文章, 详见: “Weak References”。

6.7 Python 对不可变类型施加的把戏



本部分讨论了一些对于 Python 用户而言并不重要的 Python 细节, 这些细节可能不适用于其他 Python 实现, 甚至可能不适用于 CPython 的未来版本。尽管如此, 我还是看到过一些人偶然遇到这些极端情况时, 错误地使用了 `is` 运算符。所以, 我觉得有必要讲一下这些细节。

我很惊讶地发现, 对于元组 `t` 来说, 执行 `t[:]` 后并不会创建副本, 而是返回对同一个对象的引用。此外, `tuple(t)` 获得的也是对同一元组的引用⁹。示例 6.17 证明了这一点。

</> 示例 6.17: 用一个元组构建另一个元组, 其实得到的是同一个元组

```
1 >>> t1 = (1, 2, 3)
2 >>> t2 = tuple(t1)
3 >>> t2 is t1 ❶
4 True
5 >>> t3 = t1[:]
6 >>> t3 is t1 ❷
7 True
```

- ❶ `t1` 与 `t2` 被绑定到同一个对象。
- ❷ `t3` 也是。

对于 `str`、`bytes` 与 `frozenset` 实例, 也可以观察到同样的行为。请注意, `frozenset` 并不是序列。因此, 若 `fs` 是一个 `frozenset`, 则 `fs[:]` 是不起作用的。但是, `fs.copy()` 也会产生同样的效果¹⁰: 如示例 6.18 所示, 它会欺骗你, 返回对同一个对象的引用, 而不是创建副本。

</> 示例 6.18: 字符串字面量可能会创建共享的对象

```
1 >>> t1 = (1, 2, 3)
2 >>> t3 = (1, 2, 3) ❶
3 >>> t3 is t1 ❷
4 False
5 >>> s1 = 'ABC'
6 >>> s2 = 'ABC' ❸
7 >>> s2 is s1 ❹
8 True
```

- ❶ 从头新建一个元组。
- ❷ `t1` 与 `t3` 相等, 但二者不是同一个对象。
- ❸ 再新建一个字符串。
- ❹ 奇怪的事情发生了, `s1` 与 `s2` 引用同一个 `str` 对象。

⁹ 文档中明确指出了这个行为。在 Python 控制台中输入 `help(tuple)` 后, 可以看到这样一段话: “If the argument is a tuple, the return value is the same object (若参数是一个元组, 则返回值是同一个对象)。”。在编写本书之前, 我还以为我对元组了如指掌。

¹⁰ `copy` 方法不复制对象, 这是一个善意的谎言, 为的是接口的兼容性: 它使 `frozenset` 与 `set` 更加兼容。无论如何, 对于最终用户来说, 两个相同的不可变对象是相同的, 还是复制的副本, 并没有什么区别。

共享字符串字面量是一种优化技术,称为“interning(共享)”。CPython对程序中经常出现的小整数(如0、1、-1等)也使用了类似的技术,以避免不必要的复制(duplication)。需要注意的是,Cpython并不对所有的字符串或整数都执行interning优化,并且它的实现细节也没有相关的文档记录。



切勿依赖str或整数的“interning(共享)”!比较字符串或整数是否相等,应使用`==`,而不是`is`。“interning(共享)”是Python解释器内部使用的一种优化措施。

本节讨论的技巧,包括`frozenset.copy()`的行为,都是无害的“谎言”。它们可以节省内存,并使解释器速度更快。不用担心,它们不会给您带来麻烦。因为,它们只适用于不可变类型。或许这些细枝末节的最佳用途,就是赢得与其他Python爱好者的赌注¹¹。

6.8 本章小结

每个Python对象都有标识(ID)、类型和值。只有对象的值可能随着时间而改变¹²。

如果2个变量引用了具有相同值的不可变对象(即`a == b`为True),则在实践中它们是引用副本,还是引用同一对象的别名并不重要。因为不可变对象的值不会改变,但有一个例外。这个例外就是不可变容器(如元组):若一个不可变容器持有对可变项的引用,那么当可变项的值发生变化时,这个不可变容器的值实际上也会发生变化。实践中,这种情况不常见。在不可变容器中,永远不变的是其中对象的标识(ID)。`frozenset`类不存在这个问题,因为它只能保存可哈希(hashable)元素。并且根据定义,可哈希(hashable)对象的值永远不会改变。

变量中保存的是引用,这一点对Python编程有很多实质影响:

- 简单赋值不会创建副本。
- 若左侧变量绑定的是不可变对象,则使用`+=`或`*=`的增强复制,会创建新对象。但可能会就地更改可变对象。
- 为现有变量分配新值,并不会改变先前与之绑定的对象。这就是所谓的“重新绑定”:变量现在被绑定到了一个不同的对象。若该变量是对先前对象的最后一个引用,则先前对象将被垃圾回收。
- 函数参数是作为别名传递的(即共享调用(Call by Sharing)),这意味着函数可以更改作为参数接收的可变对象。除了制作本地副本或使用不可变对象(如传一个元组而不是列表)之外,没有其他方法可以阻止这种情况发生。
- 使用可变对象作为函数参数的默认值,这种做法是很危险的。因为若参数被就地更改,那么默认值也会随之更改,从而影响后续依赖默认值的每次函数调用。

在CPython中,一旦对象的引用数为零,对象就会被销毁。如果多个对象形成具有循环引用(但不具有外部引用)的组,则它们也可能被销毁。

在某些情况下,持有一个对象的引用可能是有用的,但这个引用本身并不能使对象保持活跃状态。其中一个例子是一个类希望跟踪其所有当前实例。这可以通过弱引用(Weak Reference)来实现,弱引用是一种底层机制,是`weakref`模块中`WeakValueDictionary`、`WeakKeyDictionary`、`WeakSet`等有用的容器类,以及

¹¹若在面试或认证考试中,提出这方面的考题,那就太可怕了。千万不要这么做,因为有更多更重要的Python知识点可以考察。

¹²实际上,只需给对象的`|__class__|`属性指定一个不同的类,就可以改变对象的类型。但这是在作恶,我后悔写了这个脚注。

`finalize()` 函数的底层基础。更多信息详见 fluentpython.com 的“Weak References”。

6.9 延伸阅读

《Python 语言参考》的“Data model”开头处,清晰地解释了对象的标识与对象的值。

Wesley Chun (《Core Python 系列丛书》的作者),他在 EuroPython 2011 大会上发表了题为“Understanding Python’s Memory Model, Mutability, and Methods (视频)”的演讲,演讲内容不仅涵盖了本章内容,还涵盖了特殊方法的使用。

Doug Hellmann 撰写了文章“copy — Duplicate Objects”与“weakref —Impermanent References to Objects”。其中,涵盖了了我们刚刚讨论的一些主题。

有关 CPython 分代垃圾回收器的更多信息,详见 [gc 模块文档](#),该文档的第一句话是“该模块为可选的垃圾回收器提供了一个接口”¹³。此处的限定词“可选的”可能会令人惊讶,但是,《Python 语言参考》的“Data model”也说了¹⁴:

可以推迟垃圾回收或完全省略它——垃圾回收的实现(implementation)方式是一个实现质量的问题。只要没有回收仍然可达的对象,垃圾回收的实现(implementation)可以采用不同的方式。

Pablo Galindo 在《Python 开发者指南》中的“垃圾回收器设计”一文,对 Python 的 GC 进行了更深入的阐述,该指南面向 CPython 实现的新贡献者与经验丰富的贡献者。

CPython 3.4 垃圾收集器使用 `__del__` 方法改进了对对象的处理,详见“[PEP 442 -Safe object finalization](#)”。

维基百科有一篇关于“[字符串共享 \(String interning\)](#)”的文章,提到了这种技术在多种语言(包括 Python)中的使用。

维基百科还有一篇关于“[Haddocks’ Eyes](#)”的文章,即我在本章顶部引用的 Lewis Carroll 的歌曲。维基百科的编辑写道,这些歌词被用在逻辑与哲学作品中,“详细说明名称概念的象征地位:作为识别标记的名称可以分配给任何事物,包括另一个名称,从而引入不同层次的符号化 (Symbolization.)”

杂谈

平等对待所有对象

在发现 Python 之前,我已经学习了 Java。我一直都觉得 Java 中的 `==` 运算符是错误的。对于程序员来说,相等性比同一性更常见。但 Java 的 `==` 运算符,比较的是对象(不含原始类型)的引用,而不是对象值。即使是比较字符串这样的基本操作,Java 也强制使用 `equals` 方法。`equals` 方法还存在另一个问题:对于表达式 `a.equals(b)`,若 `a` 是 `null`,你会得到一个空指针异常。Java 设计人员认为有必要为字符串重载 `+` 运算符,那么为什么不将 `==` 也重载了呢?

Python 在这方面做得很好。运算符 `==` 比较的是对象值,而 `is` 比较的是对象的引用。因为 Python 具有运算符重载,所以 `==` 可以正确处理标准库中的所有对象,包括 `None`——这是一个正常的对象,与 Java 的 `null` 不同。

¹³ 英文原文为:This module provides an interface to the optional garbage collector.

¹⁴ 原文:An implementation is allowed to postpone garbage collection or omit it altogether—it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

当然,你也可以在自己的类中定义 `__eq__` 方法,来决定如何比较类的实例。如果不覆盖 `__eq__` 方法,则从 `objects` 继承的 `__eq__` 方法会比较对象的 ID。因此,用户定义类的所有实例都被认为是不同的。

1998 年 9 月的一个下午,当我读完《The Python Tutorial》(Python 教程)后,考虑到上述这些因素,我立即从 Java 转向了 Python。

可变性

如果所有 Python 对象都是不可变的,本章就没有存在的必要了。在处理不变对象时,变量中保存的是实际对象,还是保存对实际对象的引用,并没有什么区别。若 `a==b` 为 `True`,并且两个对象都不可变,则它们可能是相同的对象。这就是字符串可以使用“字符串共享 (String interning)”的原因。只有当对象可变(如 `list`)时,对象标识才重要。

在“纯”函数式编程中,所有数据都是不可变的:为容器进行追加项操作,实际上会创建一个新的容器。Elixir 是一种易学、实用的函数式编程,其中所有内置类型都是不可变的,包括列表。

然而,Python 并不是函数式语言,更不可能是纯函数式语言。在 Python 中,用户定义类的实例默认是可变的(大多数面向对象语言均是如此)。在创建自己的对象时,如果需要用到不可变对象,就需要格外小心。此时,对象的每个属性也必须是不可变的,否则就会出现类似于元组的情况:就对象 ID 而言,是不可变的,但如果元组中包含一个可变对象,则元组的值可能会改变。

可变对象也是使用线程编程难以成功的主要原因:某个线程更改对象后,如果没有正确同步,就会产生错误的数据,但是过度同步又会导致死锁。Erlang 语言与平台(包括 Elixir)旨在最大限度地延长电信交换机等高并发式应用的正常运行时间。很自然地,他们默认选择了不可变数据。

对象销毁与垃圾回收

Python 中没有直接销毁对象的机制,而这么做实际上是一个很好的特性:如果可以随时销毁对象,那么指向它的现有引用会怎样呢?

Cython 中的垃圾回收主要是通过引用计数来完成的,这种方法很容易实现,但在存在引用循环时容易造成内存泄漏。因此在 2.0 版本(2000 年 10 月)中实现了分代垃圾收集器,它能够处理因引用循环而存活的不可用的对象。

但是,引用计数仍然是一个基准。一旦引用计数归零,就立即销毁对象。这意味着,在 Cython 中(至少目前),如下的编码是安全的。

```
1 open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

该代码是安全的,因为 `write` 方法返回后,文件对象的引用计数将为零。Python 将立即关闭文件,然后再销毁内存中代表文件的对象。但是,这段代码在 Jython 或 IronPython 中是不安全的,因为它们使用的是宿主运行时(Java VM 与 .NET CLR)中的垃圾回收程序。这些回收程序更复杂,而且不依赖引用计数,可能需要更长的时间来销毁对象并关闭文件。在所有情况下(包括 Cython),最好的做法是显式关闭文件,而最可靠的方法是使用 `with` 语句。`with` 语句可保证即使在打开文件时引发异常,文件也会被正常关闭。使用 `with` 语句后,前面的代码片段将变成:

```
1 with open('test.txt', 'wt', encoding='utf-8') as fp:
```

```
2 fp.write('1, 2, 3')
```

如果您对垃圾回收器感兴趣，不妨阅读一下 Thomas Perl 的论文“Python 垃圾回收器实现：CPython,PyPy 与 GaS”。从中我了解到了 CPython 中有关 open().write() 的安全性知识。

参数传递：共享调用 (Call by Sharing)

解释 Python 中的参数传递方式时，人们常说的一句话：“参数按值传递，但是这里的值是引用。”。这么说没错，但是会引起误解。因为在早期的语言中，最常用的参数传递模式是“按值传递”与“按引用传递^a”。在 Python 中，函数得到的是实参 (Argument) 的副本，但是实参 (Argument) 始终都是引用。因此，若引用的是可变对象，那么对象的值就可能被修改，但对象标识不变。此外，由于函数得到的是实参 (Argument) 引用的副本，所以函数体内重新绑定引用，对函数外没有任何影响。在阅读了 Michael L. Scott (Morgan Kaufmann 出版社) 所著的《Programming Language Pragmatics (第三版)》中的“8.3.1: Parameter Modes”部分后，我决定采用“共享调用 (Call by Sharing)”这一术语：

^a (1) 按值传递：函数得到实参 (Argument) 的副本。(2) 按引用传递：函数得到实参 (Argument) 的指针。

wechat: 119554488

PART II

第二部分

函数即对象

- 第 7 章 函数是一等对象
- 第 8 章 函数中的类型提示
- 第 9 章 装饰器与闭包
- 第 10 章 用一等函数实现设计模式

wechat: 119554488

函数是一等对象

无论别人怎么说或怎么想,我从未觉得 Python 受到函数式语言太多的影响。我非常熟悉像 C 和 Algol 68 这样的命令式语言。虽然我将函数视为一等对象,但我并不认为 Python 是一种函数式语言。

——Guido van Rossum, Python BDFL^a。

^a摘自 Guido 的 The History of Python 博客,标题为 “Origins of Python’s ‘Functional’ Features”

Python 中的函数是 **一等对象** (First-class Object)。编程语言研究人员将“一等对象”定义为程序实体,可以是:

- 在运行时创建。
- 能被复制给数据结构中的变量或元素。
- 能作为参数,传给函数。
- 能作为函数的返回结果。

整数、字符串和字典是 Python 中一等对象的其他示例——这里没什么特别的。将函数视为“一等对象”是函数式语言(如 Clojure、Elixir 和 Haskell)的基本特征。然而,一等函数是如此有用,以至于 JavaScript、Go 和 Java(自 JDK 8 起)等流行语言也采用了这种设计,但这些语言都是“函数式语言”。

本章和第 3 部分的大部分内容,都探讨了将函数视为对象的实际应用。



人们经常将“把函数视为一等对象”简称为“一等函数”。此种说法并不完美,似乎表明这是函数中的“精英”。在 Python 中,所有函数都是一等函数。

7.1 本章新增内容

本书第 1 版中的“5.4 节 7 种可调用对象”,在第 2 版中变成了“9 种可调用对象”。新增的 2 种可调用对象是原生协程(Python 3.5 引入)与异步生成器(Python 3.6 引入)。这 2 种可调用对象将在第二十一章中介

绍。但为了内容完整,本章将它们与其他可调用对象一起提及。

新增了“[7.7.1 仅限位置参数](#)”一节,涵盖了 Python 3.8 中新增的一个功能。

我将运行时访问函数注解的讨论,移到了“[15.5.1 运行时的注解问题](#)”中。当我编写第 1 版时,“[PEP 484 -Type Hints](#)”还在讨论中,没有统一的注解语法。从 Python 3.5 开始,注解应该符合“[PEP 484](#)”。因此,最好在讨论 [类型提示 \(Type Hints\)](#) 时,再介绍注解。



本书第 1 版中,有一些关于函数对象的章节,这些章节过于低级,偏离了本章的主题。我将这部分内容合并到了 [fluentpython.com](#) 上一篇名为“[Introspection of Function Parameters](#)”的文章中。

现在,让我们来看看 Python 函数为什么是成熟的对象。

7.2 将函数视为对象

[示例 7.1](#) 中的控制台会话表明了 Python 函数就是对象。示例中,我们创建一个函数,调用它,读取它的 `__doc__` 属性,并检查函数对象本身是否是 `function` 类的实例。

</> 示例 7.1: 创建一个函数,并检查函数类型。

```

1  >>> def factorial(n): ❶
2  ...     """returns n!"""
3  ...     return 1 if n < 2 else n * factorial(n - 1)
4  ...
5  >>> factorial(42)
6  1405006117752879898543142606244511569936384000000000
7  >>> factorial.__doc__ ❷
8  'returns n!'
9  >>> type(factorial) ❸
10 <class 'function'>

```

❶ 这是一个控制台会话,所以我们要在“运行时”创建一个函数。

❷ `__doc__` 是函数对象的几个属性之一。

❸ `factorial` 是 `function` 类的实例。

`__doc__` 属性用于生成对象的帮助文本。在 Python 控制台中,命令 `help(factorial)` 将显示如 [图 7.1](#) 所示的屏幕。

[示例 7.2](#) 说明了函数对象的“一等 (First-Class)”本性。可以将函数 `factorial` 赋值给变量 `fact`,并通过变量 `fact` 调用函数 `factorial`。还可以将函数 `factorial` 作为参数,传给 `map` 函数。调用 `map(function, iterable)` 会返回一个可迭代对象。其所含的每个项都是将第一个参数 (function) 应用到第二个参数 (可迭代对象,本例为 `range(10)`) 中各个元素上得到的结果。

</> 示例 7.2: 通过其他名称调用 `factorial` 函数,并将 `factorial` 作为参数传递

```

1  >>> fact = factorial
2  >>> fact

```

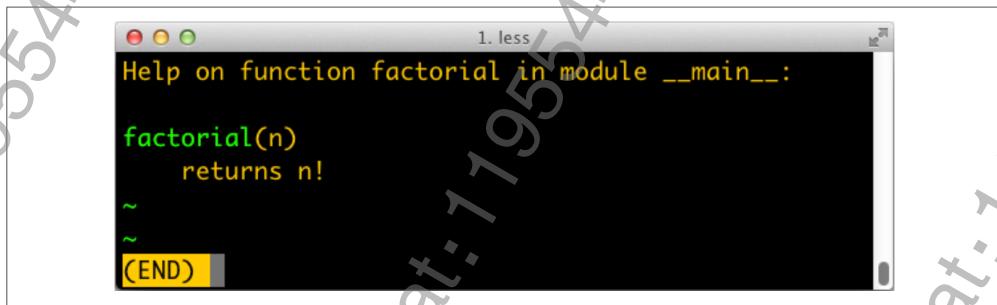


图 7.1: factorial 的帮助屏幕;文本来自函数的 `__doc__` 属性。

```
3 <function factorial at 0x...>
4 >>> fact(5)
5 120>>> map(factorial, range(11))
6 <map object at 0x...>
7 >>> list(map(factorial, range(11)))
8 [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

有了“一等函数”，我们就可以用函数式风格进行编程。函数式编程的特色之一是使用 高阶函数，这也是我们的下一个主题。

7.3 高阶函数

接收函数为参数，或将函数作为结果返回的函数，就是“高阶函数（Higher-Order Function）”。示例 7.2（7.2节）中的函数 `map` 就是一个例子。此外，内置函数 `sorted` 也是：通过可选的参数 `key` 提供一个函数，将函数应用到每一项上进行排序（参见“2.9 `list.sort` 与内置函数 `sorted`”节）。若想根据单词长度排序，只需将函数 `len` 传给参数 `key`，如示例 7.3 所示。

</> 示例 7.3：根据单词长度排序列表

```
1 >>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
2 >>> sorted(fruits, key=len)
3 ['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
4 >>>
```

任何单参数函数都可以作为参数 `key` 的值。例如，在创建韵律词典时，可能需要对每个单词的反向拼写进行排序。示例 7.4 中，请注意列表中的单词没有任何变化，只是将它们的反向拼写作为排序标准，以便后缀为 `berry` 的单词都排在一起。

</> 示例 7.4：按反向拼写对单词列表进行排序

```
1 >>> def reverse(word):
2     ...
3     return word[::-1]
4 >>> reverse('testing')
5 'gnitset'
6 >>> sorted(fruits, key=reverse)
7 ['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

在函数式编程范例中,最著名的高阶函数有 `map`、`filter`、`reduce` 和 `apply`。`apply` 函数在 Python 2.3 中已被弃用,并在 Python 3 中被删除,因为不再需要它。如果需要调用带有动态参数集的函数,可以写 `fn(*args, **kwargs)` 代替 `apply(fn, args, kwargs)`。

`map`、`filter`、`reduce` 这些高阶函数仍然存在,但在大多数场景中都会有更好的替代方案,如下一节所示。

7.3.1 map、filter 与 reduce 的替代品

函数式编程语言中,通常都会提供 `map`、`filter` 与 `reduce` 这些高阶函数(有时名称不同)。在 Python3 中,函数 `map` 与 `filter` 仍然是内置函数。但自从引入了 **列表推导式 (Listcomp)** 与 **生成器表达式 (Genexp)** 后,这些高阶函数就不那么重要了。**列表推导式 (Listcomp)** 与 **生成器表达式 (Genexp)** 可以替代使用 `map` 和 `filter` 函数,它们可以以更简洁的方式实现相同的功能,并且更容易理解和阅读。如 [示例 7.5](#) 所示。

</> [示例 7.5: 列表推导式与高阶函数比较](#)

```

1  >>> list(map(factorial, range(6))) ①
2  [1, 1, 2, 6, 24, 120]
3  >>> [factorial(n) for n in range(6)] ②
4  [1, 1, 2, 6, 24, 120]
5  >>> list(map(factorial, filter(lambda n: n % 2, range(6)))) ③
6  [1, 6, 120]
7  >>> [factorial(n) for n in range(6) if n % 2] ④
8  [1, 6, 120]
9  >>>

```

- ① 用 $0! \sim 5!$, 构建一个阶乘列表。
- ② 用 **列表推导式 (Listcomp)** 执行与①相同的操作。
- ③ 用 `map`、`filter` 计算 5 以内奇数的阶乘列表。
- ④ 用 **列表推导式 (Listcomp)** 执行与③相同的操作。换掉了 `map` 与 `filter`, 也无需使用 `lambda` 表达式。

在 Python 3 中,函数 `map` 与 `filter` 会返回生成器(一种迭代器),因此它们的直接替代品是 **生成器表达式 (Genexp)**。而在 Python 2 中,函数 `map` 和 `filter` 会返回列表,因此它们最接近的替代品是 **列表推导式 (Listcomp)**。

在 Python 3 中,函数 `reduce` 从 Python 2 的内置函数降级到了 `functools` 模块。函数 `reduce` 最常用于求和运算。但自 2003 年 Python 2.3 发布以来,内置函数 `sum` 在执行求和运算时,效果更好。在可读性与性能方面,函数 `sum` 也是一项重大改善(如 [示例 7.6](#) 所示)。

</> [示例 7.6: 分别用 reduce 与 sum 计算 0-99 之间的整数和](#)

```

1  >>> from functools import reduce ①
2  >>> from operator import add ②
3  >>> reduce(add, range(100)) ③
4  4950
5  >>> sum(range(100)) ④
6  4950
7  >>>

```

- ① 从 Python 3.0 开始,reduce 不再是内置函数。
- ② 导入 add,以避免创建一个只求两数之和的函数。
- ③ 计算 0~99 的整数和。
- ④ 用函数 sum 执行与 ③ 相同的操作。无需导入并调用函数 reduce 与 add。



函数 sum 与 reduce 的整理运作方式是一样的——即将某个操作连续应用到序列中的每个项上;然后,累计前一个结果;最后,再将一系列值规约 (Reducing) 成一个值。

内置的规约 (Reducing) 函数还有 all 与 any:

- **all(iterable)**

如果可迭代对象 iterable 中的所有元素都为 true (或 iterable 为空,如 []),则返回 True。

- **any(iterable)**

如果可迭代对象 iterable 中的任一元素为 true,则返回 True。如果可迭代对象 iterable 为空 (如 []),则返回 False。

“12.7 Vector 类第 4 版: 哈希与快速等值测试”中将详细介绍函数 reduce,届时我们会不断改进一个示例,为函数 reduce 的使用提供有意义的上下文。本书将在“17.10 可迭代的规约 (Reducing) 函数”中,重点讨论可迭代对象,届时会总结各个规约函数。

为了使用高阶函数,有时创建一次性的小型函数会更便利。这便是匿名函数存在的原因 (详见 7.4 节)。

7.4 匿名函数

在 Python 中可以用 lambda 关键字创建匿名函数,所以又称“lambda 函数”。但是,受 Python 简单语法的限制,lambda 函数的主体必须是纯表达式。换句话说,lambda 函数主体不能包含其他 Python 语句 (如 while、try 等)。带有 = 的赋值也是一种语句,因此也不能出现在 lambda 中。虽然可以在 lambda 使用新型赋值表达式 (:=),但是这种表达式会使 lambda 过于复杂,可读性差,应该使用 def 将其重构为常规函数。

匿名函数最适合被用在高阶函数的参数列表中。例如,示例 7.7 中用 lambda 表达式对示例 7.4 (7.4 节) 的单词排序进行了改写。

</> 示例 7.7: 用 lambda 表达式改写示例 7.4 (7.4 节)

```
1  >>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
2  >>> sorted(fruits, key=lambda word: word[::-1])
3  ['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
4  >>>
```

匿名函数除了被用作高阶函数的参数之外,Python 中很少使用匿名函数。由于 Python 对 lambda 函数的语法限制,使某些复杂逻辑无法在 lambda 函数中表达或实现。若 lambda 函数难以阅读,则强烈建议遵循标注栏中 Fredrik Lundh 的建议对其进行重构。

若因使用 lambda 表达式导致一段代码难以理解, Fredrik Lundh 建议按如下步骤对 lambda 表达式进行重构。

- ① 编写注释, 说明 lambda 表达式的作用。
- ② 研究注释, 然后找出一个名称来概括 ① 的注释。
- ③ 将 lambda 表达式转换为 def 语句, 用 ② 的名称定义函数。
- ④ 移除 ① 中的注释。

以上步骤引用自《Functional Programming HOWTO》。这篇文章不可错过。

在 Python 中, lambda 表达式只是一种 语法糖 (Syntactic Sugar), 它实际上创建了一个函数对象, 就像使用 def 语句定义函数一样。lambda 表达式只是 Python 中几种可调用对象的一种。下一节, 会介绍所有可调用对象。

7.5 9 种可调用对象

除了函数之外, 调用运算符 () 还可应用于其他对象。要确定对象是否可调用, 可使用内置的 `callable()` 内置函数。从 Python 3.9 开始, `Data model` 列出了 9 种可调用类型:

- **用户定义的函数**

用 `def` 语句或 `lambda` 表达式创建的函数。

- **内置函数**

用 C 语言 (CPython) 实现的函数, 如 `len` 或 `time.strftime`。

- **内置方法**

用 C 语言实现的方法, 如 `dict.get`。

- **方法**

在类主体中定义的函数。

- **类**

当类被调用时, 它会运行其 `__new__` 方法来创建一个实例; 然后, 运行 `__init__` 方法来初始化实例; 最后, 将实例返回给调用者。因为, Python 中没有 `new` 运算符, 所以调用类就像调用函数一样¹。

- **类的实例**

如果一个类定义了 `__call__` 方法, 那么它的实例就可以像函数一样被调用。详见“[7.6 用户定义的可调用类型](#)”。

- **生成器函数**

函数体中包含 `yield` 关键字的函数或方法。生成器函数被调用时, 会返回一个生成器对象。

- **原生协程函数 (≥ Python 3.5)**

使用 `async def` 定义的函数或方法。被调用时, 会返回一个协程 (Coroutine) 对象。Python 3.5 中添加。

- **异步生成器函数 (≥ Python 3.6)**

¹通常, 调用类时会创建类的实例。不过, 若覆盖 `__new__` 方法, 则可能出现其他行为。“[22.2.3 用 `__new__` 灵活创建对象](#)”提供了一个示例。

使用 `async def` 定义的函数或方法, 其主体中包含 `yield`。调用这些函数或方法时, 会返回一个异步生成器, 以与 `async for` 一起使用。

生成器函数、原生协程函数和异步生成器函数, 与其他可调用类型不同。它们的返回值不是应用数据, 而是需要进一步处理以产生应用数据, 或是执行某种操作的对象。生成器函数将返回迭代器 (详见“[十七 迭代器、生成器和经典协程](#)”)。原生协程函数与异步生成器函数返回的对象, 只能在异步编程框架 (如 `asyncio`) 的帮助下工作 (详见“[二十一 异步编程](#)”)。



鉴于 Python 中现有的可调用类型种类繁多, 确定对象是否可调用的最安全方法是使用内置函数 `callable()`:

```
1  >>> abs, str, 'Ni!'
2  (<built-in function abs>, <class 'str'>, 'Ni!')
3  >>> [callable(obj) for obj in (abs, str, 'Ni!')]
4  [True, True, False]
```

接下来, 讲述如何将类的实例变成可调用对象。

7.6 用户定义的可调用类型

不仅 Python 函数是实实在在的对象, 任意的 Python 对象也可以被定义为类似于函数的行为。只需实现一个 `__call__` 实例方法, 就可以使一个对象具备函数的可调用特性。

[示例 7.8](#) 实现了一个 `BingoCage` 类。该类可根据一个可迭代对象, 来构建类实例。并且, 在实例内部会存储一个乱序的元素列表。调用实例时, 会从该列表中弹出一个元素²。

</> [示例 7.8: bingocall.py: 从乱序的列表中取出一个元素](#)

```
1  import random
2
3  class BingoCage:
4
5      def __init__(self, items):
6          self._items = list(items) ❶
7          random.shuffle(self._items) ❷
8
9      def pick(self): ❸
10         try:
11             return self._items.pop()
12         except IndexError:
13             raise LookupError('pick from empty BingoCage') ❹
14
15     def __call__(self): ❺
16         return self.pick()
```

²既然有现成的 `random.choice` 可用, 为何还要建立一个 `BingoCage`? 函数 `random.choice` 可能会多次返回同一个元素。因为选中的元素, 不会被从给定的容器中删除。只要实例中的值都是唯一的, 调用 `BingoCage` 就永远不会返回重复的结果。

1. `__init__` 接受可迭代对象；构建本地副本可防止对作为参数传递的列表产生意外的副作用。
2. 因为 `self._items` 是个列表，所以 `random.shuffle` 一定可以正常工作（即打乱列表 `self._items` 的顺序）。
3. 起主要作用的方法。
4. 若 `self._items` 为空，则引发异常，并显示自定义信息。
5. 使 `bingo()` 成为 `bingo.pick()` 的快捷方式。

下面是 [示例 7.8](#) 的一个简单演示。请注意，可将实例 `bingo` 当作函数来调用，而内置的 `callable()` 会将 `bingo` 实例识别为可调用对象。

```

1  >>> bingo = BingoCage(range(3))
2  >>> bingo.pick()
3  1
4  >>> bingo()
5  0
6  >>> callable(bingo)
7  True

```

实现 `__call__` 的类是一种创建类似函数对象的简便方式，这些函数对象具有一些必须在调用之间保持的内部状态，就像 `BingoCage` 中剩余的项目一样。`__call__` 的另一个很好的用例是实现装饰器。装饰器必须是可调用的，有时在装饰器的多次调用之间“记住”某些东西很方便（例如，对于记忆化——将昂贵计算的结果缓存起来以供以后使用）或者将复杂的实现拆分为独立的方法。

`__call__` 的另一个用处是实现装饰器。装饰器必须是可调用的，而且有时在装饰器的调用之间“记住”某些东西会很方便（例如，用于[记忆化（Memoization）](#)——将昂贵计算的结果缓存起来以供以后使用）或者将复杂的实现拆分为多个独立的方法。

在函数式编程中，要创建保持内部状态的函数，需要使用“闭包”。闭包与装饰器，将在“[九 装饰器与闭包](#)”中讨论。

下面探讨 Python 为声明函数[形参（Parameter）](#)与传入[实参（Argument）](#)所提供的强大语法。

7.7 从位置参数到仅限关键字参数

Python 函数的最佳特性之一是极其灵活的参数处理机制。与此密切相关的是，当我们调用函数时，可以使用 * 和 ** 将可迭代对象和映射解包为单独的参数。要了解这些功能的实际效果，请参阅 [示例 7.9](#) 中的代码，以及[示例 7.10](#)中显示其使用效果的测试。

</> [示例 7.9](#): 函数 `tag` 用于生成 HTML 标签³

```

1  def tag(name, *content, class_=None, **attrs):
2      """生成一个或多个 HTML 标签"""
3      if class_ is not None:
4          attrs['class'] = class_
5      attr_pairs = (f' {attr}="{value}"' for attr, value in sorted(attrs.items()))
6      attr_str = '\n'.join(attr_pairs)
7      if content:
8          elements = (f'<{name}{attr_str}>{c}</{name}>' for c in content)
9          return '\n'.join(elements)
10 else:

```

```
11  return f'<{name}{attr_str} />'
```

可以通过多种方式调用 tag 函数,如 [示例 7.10](#) 所示。

</> [示例 7.10](#): tag 函数([示例 7.9](#))的多种调用方式

```

1  >>> tag('br')
2  '<br />'
3  >>> tag('p', 'hello')
4  '<p>hello</p>'
5  >>> print(tag('p', 'hello', 'world'))
6  <p>hello</p>
7  <p>world</p>
8  >>> tag('p', 'hello', id=33) ❶
9  '<p id="33">hello</p>'
10 >>> print(tag('p', 'hello', 'world', class_='sidebar')) ❷
11 <p class="sidebar">hello</p>
12 <p class="sidebar">world</p>
13 >>> tag(content='testing', name='img') ❸
14  '<img content="testing" />'
15 >>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
16  ...           'src': 'sunset.jpg', 'class': 'framed'}
17 >>> tag(**my_tag) ❹
18  

```

❶ 单个位置参数会产生一个带有该名称的空标签。

❷ 第一个参数之后的任意数量的参数,都会被 *content 捕获,并存入一个 tuple。

❸ tag 函数签名中未明确指定名称的关键字参数,都会被 **attr 捕获,并存入一个 dict。

❹ class_ 参数只能作为关键字参数传入。

❺ 第一个位置参数也可以作为关键字传入。

❻ 以 ** 为前缀的字典 my_tag,可将 my_tag 的所有项目作为独立的参数依次传入 tag 函数。字典 my_tag 中,与 [形参 \(Parameter\)](#) 同名的键 (key),将被绑定到 tag 函数的同名 [形参 \(Parameter\)](#) 上;字典中余下的项,则被 **attrs 捕获。在此字典中,可以用 'class' 作为键,因为它是字符串,与保留字 class 不冲突。

[仅限关键字参数 \(Keyword-only Argument\)](#)⁴ 是 Python 3 新增的功能。在 [示例 7.9](#) 中,参数 class_ 只能通过关键字参数指定——关键字参数绝不会捕获无名称的位置参数。定义函数时,若要将参数指定为 [仅限关键字参数 \(Keyword-only Argument\)](#),请在参数列表中第一个 [仅限关键字参数 \(Keyword-only Argument\)](#) 之前放置一个 * (如 [示例 7.11](#) 中的参数 b)。若不想支持数量不定的位置参数,但仍需要仅限关键字参数,可在函数签名中单独添加一个符号 *。

</> [示例 7.11](#): * 后面的参数 b 是 [仅限关键字参数 \(Keyword-only Argument\)](#)

```

1  >>> def f(a, *, b):
2  ...     return a, b
3  ...

```

³可以用名为 class_ 的仅限关键字参数,传入 "class" 属性,这是一种变通方法,因为 "class" 是 Python 中的关键字。

⁴仅限关键字参数 (Keyword-only Argument):表示函数 [实参 \(Argument\)](#) 必须以关键字参数的形式传入。

```

4  >>> f(1, b=2)
5  (1, 2)>>> f(1, 2)
6  Traceback (most recent call last):
7    File "<stdin>", line 1, in <module>
8  TypeError: f() takes 1 positional argument but 2 were given

```

请注意,仅限关键字参数 (Keyword-only Argument) 不需要有默认值:它们可以像 [示例 7.11](#) 中的 b 那样,强制要求传入实参 (Argument)。

7.7.1 仅限位置参数

从 Python 3.8 开始,用户定义的函数签名可以指定 仅限位置参数 (Positional-only Parameter)。此特性一直存在于内置函数中,例如 `divmod(a,b)` 只能用位置参数调用,而不能写成 `divmod(a=10,b=4)`。

若要定义一个需要 仅限位置参数 (Positional-only Parameter) 的函数,请在 仅限位置参数 (Positional-only Parameter) 列表后增加一个符号 “/”。

[示例 7.13](#) 摘自 “[Python 3.8 新功能](#)”,演示了如何模拟 `divmod` 内置函数的参数行为:

</> [示例 7.12](#): 模拟 `divmod` 内置函数

```

1  def divmod(a, b, /):
2      return (a // b, a % b)

```

仅限位置参数 (Positional-only Parameter) 将放置在 “/” 之前。在 “/” 之后,可以指定其他参数,这些参数的处理方式一同往常。“/” 用于在逻辑上将 仅限位置参数 (Positional-only Parameter) 与其余参数分开。如果函数定义中没有 “/”,则说明没有 仅限位置参数 (Positional-only Parameter)。



在 Python 3.7 或更早版本中,参数列表中的 “/” 是语法错误。

以 [示例 7.9](#) (7.7节) 中的函数 `tag` 为例。若希望参数 `name` 是 仅限位置参数 (Positional-only Parameter),可以在函数签名中添加一个符号 “/” (如下所示)。

</> [示例 7.13](#): 模拟 `divmod` 内置函数

```

1  def tag(name, /, *content, class_=None, **attrs):
2      ...

```

在 “[What's New In Python 3.8](#)” 与 “[PEP 570](#)” 中,包含一些关于 仅限位置参数 (Positional-only Parameter) 的其他示例。

在深入了解 Python 灵活的参数声明特性之后,本章的其余部分将介绍标准库中为函数式编程提供支持的常用包。

7.8 支持函数式编程的包

尽管 Guido 明确表示, 他并未将 Python 设计为函数式编程语言。但是, 得益于一等函数、模式匹配 (Pattern Match), 以及 `operator` 与 `functools` 等包的支持, 函数式编码风格也可以在 Python 中得到很好的应用。接下来的章节, 将介绍 `operator` 与 `functools`。

7.8.1 operator 模块

在函数式编程中, 将算术运算符用作函数会很方便。例如, 假设你想在不使用递归的情况下, 计算阶乘。对于求和, 可以使用函数 `sum`。而对于阶乘这样的连续求积, 却没有对应的函数。此时, 可以使用 `reduce` 函数来解决这个问题 (详见“7.3.1 `map`、`filter` 与 `reduce` 的替代品”一节), 但是需要提供一个计算序列中两项之积的函数。[示例 7.14](#) 展示了如何用 `lambda` 表达式解决此问题。

</> [示例 7.14](#): 用 `reduce` 与 `lambda` 实现阶乘

```
1 from functools import reduce
2
3 def factorial(n):
4     return reduce(lambda a, b: a*b, range(1, n+1))
```

`operator` 模块 为多种算术运算符提供了等效函数, 因此无需再手动编写诸如 “`lambda a,b:a*b`” 之类的匿名函数。用算术运算符函数, 可将[示例 7.14](#) 改写为 [示例 7.15](#)。

</> [示例 7.15](#): 用 `reduce` 与 `mul` 函数实现阶乘

```
1 from functools import reduce
2 from operator import mul
3
4 def factorial(n):
5     return reduce(mul, range(1, n+1))
```

`operator` 模块 还提供了一组工厂函数:`itemgetter` 与 `attrgetter`。二者可替代从序列中提取项或读取对象属性的 `lambda` 表达式。

[示例 7.16](#) 展示了 `itemgetter` 的一种常见用法: 根据元组中的某个字段, 对元组列表进行排序。在此示例中, 我们按“国家代码(元组中第 2 个字段)”的顺序, 打印各个城市的信息。本质上, `itemgetter()` 会创建一个接受容器的函数, 并返回索引 1 处的项。这比作用相同的 `lambda` `fields[1]` 表达式更容易编写和阅读。

</> [示例 7.16](#): 用 `itemgetter` 排序一个元组列表⁵

```
1 >>> metro_data = [
2 ...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
3 ...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
4 ...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
5 ...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
6 ...     ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
7 ... ]
8 >>>
```

```

9  >>> from operator import itemgetter
10 >>> for city in sorted(metro_data, key=itemgetter(1)):...     print(city)
11 ...
12 ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833))
13 ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
14 ('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
15 ('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
16 ('New York-Newark', 'US', 20.104, (40.808611, -74.020386))

```

如果向 `itemgetter` 传递多个索引参数, `itemgetter` 构建的函数将返回包含提取值的元组, 以方便根据多个键进行排序:

```

1  >>> cc_name = itemgetter(1, 0)
2  >>> for city in metro_data:
3      ...     print(cc_name(city))
4 ...
5 ('JP', 'Tokyo')
6 ('IN', 'Delhi NCR')
7 ('MX', 'Mexico City')
8 ('US', 'New York-Newark')
9 ('BR', 'São Paulo')
>>>

```

由于 `itemgetter` 使用 `[]` 运算符, 因此它不仅支持序列, 还支持映射以及任何实现了 `__getitem__` 的类。

与 `itemgetter` 作用类似, `attrgetter` 会创建按名称提取对象属性的函数。若为 `attrgetter` 传递多个属性名, `attrgetter` 构建的函数将返回由提取值构成的元组。此外, 若参数名中包含符号.(点), `attrgetter` 将浏览嵌套对象以提取属性。这些行为如⁵ 所示。此控制台会话很长, 因为我们需要建立一个嵌套结构来展示 `attrgetter` 如何处理包含.(点)的属性名。

</> 示例 7.17: 用 `attrgetter` 处理 示例 7.16 定义的元组 `metro_data`

```

1  >>> from collections import namedtuple
2  >>> LatLon = namedtuple('LatLon', 'lat lon')    ❶
3  >>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')  ❷
4  >>> metro_areas = [Metropolis(name, cc, pop, LatLon(lat, lon))  ❸
5  ...     for name, cc, pop, (lat, lon) in metro_data]
6  >>> metro_areas[0]
7 Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLon(lat=35.689722,
8 lon=139.691667))
9  >>> metro_areas[0].coord.lat    ❹
10 35.689722
11  >>> from operator import attrgetter
12  >>> name_lat = attrgetter('name', 'coord.lat')    ❺
13  >>>
14  >>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):    ❻
15  ...     print(name_lat(city))    ❼
16  ...
17 ('São Paulo', -23.547778)

```

⁵示例 7.16 的数据源自示例 2.8 (“2.5.3 嵌套解包”)

```

18 ('Mexico City', 19.433333)
19 ('Delhi NCR', 28.613889)('Tokyo', 35.689722)
20 ('New York-Newark', 40.808611)

```

- ① 用 `namedtuple` 定义元组 `LatLon`。
- ② 用 `namedtuple` 定义元组 `Metropolis`。
- ③ 用 `Metropolis` 实例构建列表 `metro_areas`。注意, 此处会用 嵌套元组解包提取 `(lat,lon)`。然后, 再用 `(lat,lon)` 构建 `LatLon`, 作为 `Metropolis` 的 `coord` 属性。
- ④ 进入元素 `metro_areas[0]`, 获得其纬度。
- ⑤ 定义一个 `attrgetter`, 获取 `name` 属性与嵌套的 `coord.lat` 属性。
- ⑥ 再次用 `attrgetter`, 按纬度(即 `coord.lat`)对城市列表进行排序。
- ⑦ 使用 ⑤ 中定义的 `attrgetter`, 仅显示城市名称与纬度。

如下是 `operator 模块` 中定义的部分函数列表(省略了以 `_` 开头的名称, 因为它们大多是实现细节):

```

1 >>> [name for name in dir(operator) if not name.startswith('_')]
2 ['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
3 'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
4 'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imatmul',
5 'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior',
6 'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter',
7 'itruediv', 'ixor', 'le', 'length_hint', 'lshift', 'lt', 'matmul',
8 'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not_', 'or_', 'pos',
9 'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']

```

在列出的 54 个函数名称中, 大多数都是不言而喻的。以 `i` 为前缀, 后面是另一个运算符的那些函数名(如 `iadd`、`iand` 等), 对应的是增量赋值运算符(如 `+=`、`&=` 等)。增量赋值运算符的第一个参数如果是可变的, 那么这些函数会就地更改第一个参数; 否则, 作用与不带 `i` 的函数一样, 直接返回运算结果。

`operator 模块` 余下的函数中, 最后要介绍的是 `methodcaller`。它的作用与 `itemgetter`、`attrgetter` 类似, 即动态创建函数。`methodcaller` 创建的函数, 会在对象上调用由参数指定的方法(如 [示例 7.18](#) 所示)。

</> [示例 7.18](#): `methodcaller` 示例

```

1 >>> from operator import methodcaller
2 >>> s = 'The time has come'
3 >>> upcase = methodcaller('upper')
4 >>> upcase(s)
5 'THE TIME HAS COME'
6 >>> hyphenate = methodcaller('replace', ' ', '-')
7 >>> hyphenate(s)
8 'The-time-has-come'

```

[示例 7.18](#) 中的第一个测试只是为了展示 `methodcaller` 的用法。若需要将 `str.upper` 作为函数使用, 只需在 `str` 类上调用它, 并将字符串作为参数传入即可, 如下所示:

```

1 >>> str.upper(s)
2 'THE TIME HAS COME'

```

示例 7.18 中的第二个测试表明, `methodcaller` 也可以像 `functools.partial` 函数那样, 进行部分应用以冻结某些参数。这就是我们的下一个主题。

7.8.2 用 `functools.partial` 冻结参数

`functools` 模块 提供了多个高阶函数, 比如 “7.3.1 `map`、`filter` 与 `reduce` 的替代品” 中用过的 `reduce` 函数。另一个值得关注的函数是 `partial`, 它可以根据提供的可调用对象产生一个新的可调用对象。新的可调用对象为原可调用对象的某些参数绑定了预定的值。用函数 `partial` 可将接受一个或多个参数的函数, 改造成需要更少参数的回调 API。示例 7.19 是一个简单的演示。

</> 示例 7.19: 用 `partial` 进行函数改造

```

1  >>> from operator import mul
2  >>> from functools import partial
3  >>> triple = partial(mul, 3)      ❶
4  >>> triple(7)                  ❷
5  21
6  >>> list(map(triple, range(1, 10))) ❸
7  [3, 6, 9, 12, 15, 18, 21, 24, 27]
```

- ❶ 用 `mul` 创建 `triple` 函数, 将第一个位置参数绑定为 3。
- ❷ 测试 `triple` 函数。
- ❸ 在 `map` 中使用 `triple` 函数; 在此示例中, `mul` 不能与 `map` 一起使用。

`functools.partial` 更有用的示例是 `unicodedata.normalize` 函数 (见 “4.7 Unicode 字符规范化”)。在处理包含多国语言的文本时, 您可能想在比较或排序字符串 `s` 之前, 对其使用 `unicodedata.normalize('NFC', s)` 函数进行规范化。若需要经常这样做, 则可以创建一个 `nfc` 函数 (如 示例 7.20 所示)。

</> 示例 7.20: 用 `partial` 构建一个便利的 Unicode 规范化函数

```

1  >>> import unicodedata, functools
2  >>> nfc = functools.partial(unicodedata.normalize, 'NFC')
3  >>> s1 = 'café'
4  >>> s2 = 'cafe\u0301'
5  >>> s1, s2
6  ('café', 'cafe\u0301')
7  >>> s1 == s2
8  False
9  >>> nfc(s1) == nfc(s2)
10 True
```

`partial` 函数 的第一个参数是一个可调用参数, 后面是任意数量的要绑定的位置参数和关键字参数。

示例 7.21 展示了如何将 `partial` 与 示例 7.9 (7.7) 中的 `tag` 函数一起使用, 以冻结一个位置参数和一个关键字参数。

</> 示例 7.21: 将 `partial` 应用到 示例 7.9 (7.7) 节的 `tag` 函数上

```

1  >>> from tagger import tag
```

```

2  >>> tag
3  <function tag at 0x10206d1e0> ❶
4  >>> from functools import partial
5  >>> picture = partial(tag, 'img', class_='pic-frame') ❷
6  >>> picture(src='wumpus.jpeg')
7  '' ❸
8  >>> picture
9  functools.partial(<function tag at 0x10206d1e0>, 'img', class_='pic-frame') ❹
10 >>> picture.func
11 <function tag at 0x10206d1e0> ❺
12 >>> picture.args
13 ('img',)
14 >>> picture.keywords
15 {'class_': 'pic-frame'}

```

- ❶ 从 [示例 7.9 \(7.7 节\)](#) 导入 tag 函数，并显示其 ID。
- ❷ 用 tag 函数创建函数 picture。并将函数 picture 的第一个位置参数固定为 “img”，将 class_ 关键字参数固定为 “pic-frame”。
- ❸ 函数 picture 的行为符合预期。
- ❹ partial() 返回一个 [functools.partial 对象](#)⁶。
- ❺ [functools.partial 对象](#)，提供了可访问原始函数和固定参数的属性。

[functools.partialmethod 函数](#)与 partial 函数功能一样，但旨在与方法一起使用。

[functools 模块](#)还包括一些高阶函数，如 [cache](#) 和 [singledispatch](#) 等，这些函数可用作函数装饰器。这些函数将在“[九 装饰器与闭包](#)”中介绍，该章还将解释如何实现自定义装饰器。

7.9 本章小结

本章的目标是探索 Python 中函数的“一等 (First-Class)”本性。其主要思想是，你可以将函数赋值给变量，将它们传递给其他函数，将它们存储在数据结构中，并访问函数属性，从而允许框架和工具对这些信息进行操作。

高阶函数是函数式编程的主要内容，在 Python 中很常见。[sorted](#)、[min](#) 和 [max](#) 内置函数，以及 [functools.partial](#) 都是 Python 语言中常用的高阶函数。由于 [列表推导式 \(Listcomp\)](#)（以及 [生成器表达式 \(Gen-exp\)](#) 等类似结构）以及 [sum](#)、[all](#) 和 [any](#) 等内置归约函数的加入，[map](#)、[filter](#) 和 [reduce](#) 的使用不再像以前那样常见。

自 Python 3.6 起，Python 中有 9 种可调用对象，从用 [lambda 表达式](#)创建的简单函数，到实现 [__call__](#) 方法的类实例。生成器与协程也是可调用对象，不过其行为与其它可调用对象差异很大。所有可调用对象都可以通过内置的 [callable\(\)](#) 进行检测。可调用对象支持丰富的 [形参 \(Parameter\)](#) 声明语法，包括 [仅限关键字参数 \(Keyword-only Argument\)](#)、[仅限位置参数 \(Positional-only Parameter\)](#)、[变量注解 \(Variable Annotation\)](#)。

最后，本章介绍了 [operator 模块](#) 中的一些函数，以及 [functools.partial 函数](#)。借助这些函数，最大限度地减少了实现函数式编程的复杂度，实现函数式编程，减少了对复杂 [lambda 语法](#) 的需求。

⁶[functools.partial 对象](#)：从 [functools.py 源代码](#) 中可以看出，[functools.partial](#) 是用 C 语言实现的（默认使用此实现）。若此实现不可用，则自 Python3.4 起可以使用 [partial](#) 的纯 Python。

7.10 延伸阅读

接下来的 3 章将继续探讨如何使用函数对象进行编程。“八 函数中的类型提示”专门讨论函数参数与返回值中的 **类型提示 (Type Hints)**；“九 装饰器与闭包”深入探讨函数装饰器（一种特殊的高阶函数），以及使其发挥作用的闭包机制；“十 用一等函数实现设计模式”展示了一等函数如何简化某些经典的面向对象设计模式。

在《Python 语言参考》中，“3.2. The standard type hierarchy”介绍了 9 种可调用类型以及所有其他内置类型。

《Python Cookbook, 3rd ed》(David Beazley、Brian K. Jones 著) 中的第 7 章，是对本书第 7 章与第 9 章的很好补充，以不同方式概述了大部分相关概念。

若对 **仅限关键字参数 (Keyword-only Argument)** 的功能与用例感兴趣，请参阅“[PEP 3102 -Keyword-Only Arguments](#)”。

A. M. Kuchling 的《Python 函数编程 HOWTO》是用 Python 进行函数式编程的最佳入门读物。不过，该书的重点是迭代器和生成器的使用，这也是本书“[十七 迭代器、生成器和经典协程](#)”的主题。

StackOverflow 的问题“[Python: Why is functools.partial necessary?](#)”中，有个详实而有趣的回答，答主是《Python in a Nutshell》(O'Reilly)一书的作者——Alex Martelli。

为了思考“Python 是函数式语言吗？”这个问题，我创作了我最喜欢的演讲之一“[Beyond Paradigms](#)”，并在 PyCaribbean、PyBay 和 PyConDE 上进行了演讲。请参阅柏林演讲的[幻灯片⁷](#)与[视频⁸](#)，在那次演讲我遇到了本书的两位技术审校 Miroslav Šedivý 和 Jürgen Gmach。

杂谈

Python 是一门函数式语言吗？

2000 年的时候，我参加了在美国 Zope 公司举办的 Zope 研讨会，当时 Guido van Rossum 来到了教室（他不是讲师）。在随后的问答中，有人问他 Python 的哪些特性是从其他语言借鉴过来的。Guido 回答说“Python 中的所有优点都是从其他语言中窃取的。”

布朗大学(Brown University)计算机科学教授 Shriram Krishnamurthi 在其“[Teaching Programming Languages in a Post-Linnaean Age^a](#)”论文的开头这样写道：

编程语言“范式(Paradigms)”已近末日，是旧时代的遗留产物，令人厌烦。既然现代语言的设计者对范式不屑一顾，那我们的课程为何还要盲目地遵循它呢？

在该论文中，下面这段话点名提到了 Python：

对于 Python、Ruby 或 Perl 这样的语言，我们还能做些什么呢？这些语言的设计者对这些 **林奈层次结构 (Linnaean Hierarchies)** 的细枝末节毫无耐心；设计者随心所欲地按自己的意愿借用各种特性，创造出完全无法定性的混合体。

Krishnamurthi 认为，与其试图按照某种分类法对语言进行分类，不如把它们看作是各种特征的集合。

⁷演讲幻灯片：<https://speakerdeck.com/ramalho/beyond-paradigms-berlin-edition>

⁸演讲视频：<https://www.youtube.com/watch?v=bF3a2VYXxa0>

本节前文提到的演讲“超越范式”就是受到他这一观点的启发。

即使函数式编程不是 Guido 的目标, 但 Python 的一等函数特性, 为 Python 的函数式编程打开了大门。他在“[Python 功能特性的起源](#)”一文中表示, 最初将 lambda 添加到 Python 的目的, 是为了支持 map、filter 和 reduce。根据 CPython 源代码中的 [Misc/HISTORY](#) 部分, 所有这些功能都是 Amrit Prem 在 1994 年贡献给 Python 1.0 的。

像 map、filter 和 reduce 这样的函数最早出现在最初的函数式语言 Lisp 中。然而, Lisp 并不限制 lambda 内部可以执行的操作, 因为 Lisp 中的所有内容都是表达式。Python 使用面向语句的语法, 其中表达式不能包含语句, 并且许多语言构造都是语句, 包括 try/catch, 这是我在编写 lambda 时最常忽略的。这是 Python 高度可读的语法所付出的代价^b。Lisp 也有很多优点, 但不包括可读性。

讽刺的是, 从函数式语言 Haskell 中借用 [列表推导式 \(Listcomp\)](#) 之后, 使 Python 大大减少了对 map、filter 以及 lambda 表达式的需求。

除了匿名函数语法上的限制, Python 中广泛采用函数式编程的最大障碍是缺乏 [尾部调用消除 \(Tail-call Elimination\)](#)。这是一项优化技术, 允许对在函数体“尾部”进行递归调用的函数进行内存高效计算, 提高递归调用的内存效率。在另一篇博文“[Tail Recursion Elimination](#)”中, Guido 给出了 [尾部调用消除 \(Tail-call Elimination\)](#) 技术不适合 Python 的几个原因。此文的技术论点非常值得一读, 但更重要的是其中给出的前 3 个(也是最重要的)原因都是可用性问题。Python 作为一种易于使用、学习和教学的语言, 并非偶然, 有 Guido 在为我们把关。

综上所述, 从设计上讲, 无论函数式语言的定义如何, Python 都不是一种函数式语言。它只是从函数式语言中借鉴了一些较好的想法。

匿名函数的问题

除了 Python 特有的语法限制外, 匿名函数在任何语言中都有一个严重的缺点: 它们没有名字。

函数有了名称之后, 堆栈跟踪会更易于阅读。匿名函数是一种方便的捷径, 人们乐于用它来编码, 但有时会得意忘形——尤其是在鼓励深层嵌套匿名函数的语言和环境中, 如 JavaScript。匿名函数的嵌套层级太深, 不利于调试和处理错误。Python 中的异步编程更加结构化, 或许就是因为 lambda 语法受限, 而防止了 lambda 的滥用, 并强制采用更明确的方法。Promise、Future 和 Deferred 是现代异步 API 中使用的概念。它们与协程一起, 为所谓的“回调地狱”提供了出路。我保证, 后面会进一步讨论异步编程的内容, 但这个主题必须推迟到“[二十一 异步编程](#)”。

^a林奈层次结构 (Linnaean Hierarchies) 是指将编程语言中的各种元素(如关键字、数据类型、函数等)根据特性和层次关系进行分类和命名的层次结构体系。

^b此外, 将代码粘贴到网络论坛时还存在缩进丢失的问题, 但这是题外话。

wechat: 119554488

函数中的类型提示

仍需强调的是, Python 仍是一种动态类型语言, 作者无意强制使用类型提示, 这只是一种约定。

——Guido van Rossum, Jukka Lehtosalo, Łukasz Langa, “[PEP 484—Type Hints](#)^a”

^a[PEP 484—Type Hints](#): “Rationale and Goals” 保留了原文的粗体强调。

自 2001 年发布的 Python 2.2 中 [统一了类型和类](#) 以来, [类型提示 \(Type Hints\)](#) 是 Python 历史上最大的变革。然而, 并不是所有 Python 用户都能从 [类型提示 \(Type Hints\)](#) 中受益。这也是类型提示是一种可选功能的原因。

[PEP 484 -Type Hints](#) 引入了函数参数、返回值和变量中显式类型声明的语法和语义。其目的是帮助开发工具通过静态分析即可发现 Python 代码中的 bug, 而无需通过测试实际运行代码。

[类型提示 \(Type Hints\)](#) 的主要受益者是使用 IDE (集成开发环境) 和 CI (持续集成) 的专业软件工程师。这类人群看重的是 [类型提示 \(Type Hints\)](#) 带来的成本效益分析, 但并不是所有 Python 用户都关注这一点。

Python 的用户群体分布广泛, 包括科学家、交易员、记者、艺术家、制造商、分析师以及许多领域的学生等等。对于他们中的大多数人来说, 学习类型提示的成本可能会更高—除非他们已经掌握了一种具有静态类型、子类型和泛型的语言。考虑到这些用户与 Python 交互的方式, 以及他们的代码库和团队 (通常是一个人的“团队”) 的规模较小, 对于这些用户中的许多人来说, 学习类型提示的收益会更低。Python 的默认动态类型更加简单, 且更具表现力。因此, 特别适合在数据科学、创造性计算和学习中, 编写用于探索数据和想法的代码。

本章重点介绍 Python 函数签名中的类型提示。“[十五 类型注解进阶 1136](#)”将探讨类中的类型提示以及其他类型模块特性。

本章的主要内容包括:

- 通过 Mypy, 以实践的方式介绍渐进式类型。
- 鸭子类型与名义类型的互补作用。
- 注解中可能出现的主要类型概述——约占本章的 60%。
- [类型提示 \(Type Hints\)](#) 可变参数 (*args, **kwargs)。
- [类型提示 \(Type Hints\)](#) 和静态类型的限制和缺点。

8.1 本章新增内容

本章是全新的内容。在我完成《流畅的 Python》第 1 版之后,发布的 Python 3.5 中才出现 [类型提示 \(Type Hints\)](#)。

鉴于静态类型系统的局限性,PEP 484 的最佳想法是引入 [渐进式类型系统 \(Gradual Type System\)](#)。首先,让我们来明确一下这一概念的定义。

8.2 关于渐进式类型

PEP 484 为 Python 引入了 [渐进式类型系统 \(Gradual Type System\)](#)。其他采用渐进式类型系统的语言包括 Microsoft 的 TypeScript、Dart (谷歌开发的 Flutter SDK 的语言) 和 Hack (Facebook 的 HHVM 虚拟机支持的 PHP 方言)。Mypy 类型检查程序本身最初也是一种语言:一种带有自己的解释器的渐进式类型的 Python 语言。Guido van Rossum 说服了 MyPy 的创建者 Jukka Lehtosalo,使 MyPy 成为检查带注释的 Python 代码的工具。

[渐进式类型系统 \(Gradual Type System\)](#) 具有以下特点:

- **类型提示是可选的**

默认情况下,类型检查器不应该对没有 [类型提示 \(Type Hints\)](#) 的代码发出警告。相反,当类型检查器无法确定对象的类型时,会假定其为 Any 类型。Any 类型被认为与所有其他类型都兼容。

- **不在运行时,捕获类型错误**

静态类型检查器、linter 和 IDE 使用 [类型提示 \(Type Hints\)](#) 来发出警告。它们不会阻止在运行时将不一致的值传递给函数或分配给变量。

- **不能改善性能**

理论上,类型提示 (Type Hints) 提供的数据可以优化生成的字节码,但据我所知,截至 2021 年 7 月,还没有任何 Python 运行时实现了这种优化¹。

[渐进式类型系统 \(Gradual Type System\)](#) 的最大可用性在于,注释始终是可选的。

在静态类型系统中,大多数类型约束都很容易表达,但也有许多约束比较繁琐,有些约束很难表达,还有一些约束是不可能表达的²。你很可能编写出一段优秀的 Python 代码,具有良好的测试覆盖率并通过测试,但仍然无法添加令类型检查器满意的 [类型提示 \(Type Hints\)](#)。没关系,将有问题的类型提示都删掉,然后把它发布出去就可以了!

类型提示 (Type Hints) 在所有层面上都是可选的:你可以让整个软件包都没有 [类型提示 \(Type Hints\)](#);你也可以在将无类型提示的软件包导入到使用类型提示的模块中时,让类型检查器保持沉默;你还可以添加特殊注释,让类型检查器忽略代码中的特定行。

¹像 PyPy 这样的即时编译器拥有比类型提示更好的数据:它会在 Python 程序运行时对其进行监控,检测使用中的具体类型,并为这些具体类型生成优化的机器代码。

²例如,截至 202 年 7 月,不支持递归类型——请参阅 “typing module issue #182,Define a JSON type” 与 “Mypy issue #731,Support recursive types”。



追求 100% 的類型提示覆蓋率很可能過於激進，如未經深思熟慮添加了錯誤的注解，而這一切都是為了追求覆蓋率指標。激進的類型提示可能會阻礙團隊充分利用 Python 的強大功能和靈活性。應該坦然接受沒有 [類型提示 \(Type Hints\)](#) 的代碼，以防止注解會降低 API 的易用性，或增加 API 的實現難度。

8.3 漸進式類型實踐

讓我們看看 [漸進式類型系統 \(Gradual Type System\)](#) 在實踐中是如何工作的：從一個簡單的函數開始，在 Mypy 的指導下逐步添加 [類型提示 \(Type Hints\)](#)。



除了 PyCharm 等 IDE 中嵌入的類型檢查器之外，還有許多與 PEP 484 兼容的 Python 類型檢查器，包括 Google 的 [pytype](#)、Microsoft 的 [Pyright](#)、Facebook 的 [Pyre](#)。我之所以選擇 Mypy 作為示例，是因為它最有名。不過，對於某些項目或團體來說，其他語言可能更適合。比如 [Pytype](#) 可以處理沒有類型提示的代碼庫，並仍然可提供有用的建議。它比 Mypy 更寬鬆，還能為你的代碼生成注釋。

我們將注釋一個 `show_count` 函數，該函數返回一個字符串，其中包含一個計數和一個單數或複數單詞（取決於計數）：

```
1  >>> show_count(99, 'bird')
2  '99 birds'
3  >>> show_count(1, 'bird')
4  '1 bird'
5  >>> show_count(0, 'bird')
6  'no birds'
```

示例 8.1 顯示了 `show_count` 的源代碼，但未加注釋。

</> [示例 8.1：來自 `messages.py` 的 `show_count`，未加類型提示](#)

```
1  def show_count(count, word):
2      if count == 1:
3          return f'1 {word}'
4      count_str = str(count) if count else 'no'
5      return f'{count_str} {word}s'
```

8.3.1 Mypy 初體驗

為了開始類型檢查，我在 `messages.py` 模塊上運行了 `mypy` 命令：

```
…/no_hints/ $ pip install mypy
[lots of messages omitted...]
…/no_hints/ $ mypy messages.py
Success: no issues found in 1 source file
```

使用默认设置的 Mypy, 没有发现 [示例 8.1](#) 的问题。



我使用的是 Mypy 0.910, 这是我在 2021 年 7 月审稿时 Mypy 的最新版本。按 Mypy 文档所说, Mypy “严格来说是测试版软件。偶有打破向后兼容性的改动”。Mypy 回显的报告中至少有一份与我在 2020 年 4 月撰写本章时得到的报告不一致。当您读到本章时, 也可能会得到与此处不同的结果。

如果函数签名没有注释, Mypy 默认会忽略它, 除非另有配置。

对于 [示例 8.2](#), 我用 pytest 进行单元测试。下面是 messages_test.py 中的代码。

</> [示例 8.2: 没有类型提示的 messages_test.py](#)

```

1  from pytest import mark
2  from messages import show_count
3
4  @mark.parametrize('qty, expected', [
5      (1, '1 part'),
6      (2, '2 parts'),
7  ])
8
9  def test_show_count(qty, expected):
10     got = show_count(qty, 'part')
11     assert got == expected
12
13 def test_show_count_zero():
14     got = show_count(0, 'part')
15     assert got == 'no parts'

```

下面在 Mypy 的指导下添加 [类型提示 \(Type Hints\)](#)。

8.3.2 让 Mypy 更加严格

为 Mypy 指定命令行选项 `--disallow-untyped-defs`, 可以让 Mypy 标记没有为参数和返回值添加 [类型提示 \(Type Hints\)](#) 的函数定义。

在测试文件中使用 `--disallow-untyped-defs` 会产生 3 个错误和 1 个注释:

```

.../no_hints/ $ mypy --disallow-untyped-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
messages_test.py:10: error: Function is missing a type annotation
messages_test.py:15: error: Function is missing a return type annotation
messages_test.py:15: note: Use "> None" if function does not return a value
Found 3 errors in 2 files (checked 1 source file)

```

对于 [渐进式类型系统 \(Gradual Type System\)](#) 的第一步, 我更喜欢使用另一个选项: `--disallow-incomplete-defs`。起初, 它什么也不会告诉我:

```

.../no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
Success: no issues found in 1 source file

```

现在,我只需在 messages.py 中为函数 show_count 添加返回值类型即可:

```
1 def show_count(count, word) -> str:
```

这足以让 Mypy 检查它。使用与之前相同的命令行检查 messages_test.py, Mypy 将同时检查 messages.py:

```
…/no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
for one or more arguments
Found 1 error in 1 file (checked 1 source file)
```

现在,我可以逐步为每个函数添加 [类型提示 \(Type Hints\)](#),不让 Mypy 再报告关于函数没有注释的警告。如下是一个包含完整注释的函数签名,能满足 Mypy 的要求:

```
1 def show_count(count: int, word: str) -> str:
```



若不想每次都输入 `--disallow-incomplete-defs` 等命令行选项,可以将需要的选项保存到 Mypy 配置文件(详见“[Mypy 文档](#)”)中。设置可以针对全局生效,也可以针对单个模块生效。下面是一个简单的 `mypy.ini` 配置示例,仅供参考。

```
1 [mypy]
2 python_version = 3.9
3 warn_unused_configs = True
4 disallow_incomplete_defs = True
```

8.3.3 默认参数值

示例 8.2 中的 `show_count` 函数仅适用于常规名词。如果复数不能通过附加“s”来拼写,我们应该让用户提供复数形式(如下所示):

```
1 >>> show_count(3, 'mouse', 'mice')
2 '3 mice'
```

让我们来进行一下“测试驱动开发”。首先,我们添加一个使用第 3 个参数的测试。别忘了在测试函数中,为返回值类型添加提示,否则 Mypy 不会检查它。

```
1 def test_irregular() -> None:
2     got = show_count(2, 'child', 'children')
3     assert got == '2 children'
```

Mypy 检测到错误如下:

```
…/hints_2/ $ mypy messages_test.py
messages_test.py:22: error: Too many arguments for "show_count"
Found 1 error in 1 file (checked 1 source file)
```

现在,我编辑 `show_count`,在 [示例 8.3](#) 中添加可选的复数参数。

</> 示例 8.3. hints_2/messages.py:有一个可选参数的 show_count 函数

```

1 def show_count(count: int, singular: str, plural: str = '') -> str:
2     if count == 1:
3         return f'1 {singular}'
4     count_str = str(count) if count else 'no'
5     if not plural:
6         plural = singular + 's'
7     return f'{count_str} {plural}'

```

现在,Mypy 报告“Success (成功)”。

这是一个 Python 无法捕获的类型错误。你能发现它吗?



```

1 def hex2rgb(color=str) -> tuple[int, int, int]:

```

Mypy 的错误报告并没有什么帮助:

```

colors.py:24: error: Function is missing a type annotation
for one or more arguments

```

参数 color 的类型提示应该是 color: str。我错写成了 color=str, 这不是注解, 而是将 color 的默认值设置为 str。

根据我的经验, 这是一个很常见的错误, 很容易被忽视, 尤其是在复杂的 [类型提示](#) (Type Hints) 中。

编写 [类型提示](#) (Type Hints) 时, 建议遵守以下最佳实践:

- 参数名与符号: 之间没有空格, 而符号: 之后要加一个空格。例如, “color: str”。
- 默认参数值前面的赋值符号 = 两侧要有空格。例如, “plural: str = 'No'”。

另一方面, PEP 8 规定, 如果没有特定参数的类型提示, 则 = 的周围不应该有空格。

使用 flake8 和 blue 检查代码样式

与其记住这些繁琐的规则, 不如使用 flake8 和 blue 等工具。flake8 会报告代码样式以及许多其他问题; blue 则会根据根据代码格式化工具 black 内置的(大多数)规则, 来重写源码。

在统一代码风格方面, black 比 blue 更好。因为 black 遵循 Python 自己的风格, 即默认使用单引号, 而将双引号作为备选。

```

1 >>> 'I prefer single quotes'
2 'I prefer single quotes'

```

从 CPython 源码中的 repr() 等处, 可以看出 Python 对单引号的偏爱。依赖 repr() 的 doctest 模块默认也使用单引号。

blue 的作者之一 Barry Warsaw, 也是 PEP 8 的共同起草人, 自 1994 年以来一直是 Python 核心开发人员, 并且从 2019 年至今(2021 年 7 月)还是 Python 指导委员会成员。默认使用单引号是有坚强后盾的。

如果必须使用 black, 请使用 black -S 选项。这样它就会保持引号的原样。

8.3.4 将“None”设为默认值

在示例 8.3 (8.3.3节) 中, 参数 plural 被注释为 str, 默认值为”, 因此不存在类型冲突。

我喜欢这种解决方案, 但在其他情况下, “None” 可能是默认值更好的选择。如果可选参数需要可变类型, 那么 None 是唯一合理的默认值 (详见 “6.5.1 不要使用可变类型作为参数的默认值”)。

要将 None 作为参数 plural 的默认值, 函数签名如下所示:

```
1 from typing import Optional
2
3 def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

下面来分析一下。

- Optional[str] 表示参数 plural 可以是 str 或 None。
- 必须显式提供默认值, 即 = None。

如果没有为 plural 分配默认值, Python 运行时会将其视为必填参数。请记住: 在运行时, 类型提示 (Type Hints) 将被忽略。

请注意, 我们需要从 typing 模块 导入 Optional。导入类型时, 最好使用 “from typing import X” 的语法, 以减少函数签名的长度。



Optional 并不是一个好名字, 因为该注解并没有使参数成为可选参数。分配了默认值的参数, 才是可选参数。Optional[str] 的意思很简单, 表明参数的类型可以是 str 或 NoneType。类似于 Haskell 和 Elm 语言中的 Maybe 类型。

现在, 我们已对 渐进式类型系统 (Gradual Type System) 有了初步的了解。接下来, 让看一下“类型”概念在实践中的具体含义。

8.4 类型由支持的操作来定义

文献中对类型概念有多种定义。这里我们假设类型是一组值和一组可以应用于这些值的函数。

——“PEP 483 –The Theory of Type Hints”

在实践中, 最好将受支持的操作集视为类型的关键特征³。

例如, 从操作的可行性来看, 以下函数中的 x 应该是什么类型?

```
1 def double(x):
2     return x * 2
```

³除了 Enum 之外, Python 不提供控制类型的可能值集的语法。例如, 使用类型提示无法将 Quantity 定义为 1 到 1000 之间的整数, 也无法将 AirportCode 定义为 3 个字母的组合。NumPy 提供了 uint8、int16 和其他面向机器的数字类型, 但在 Python 标准库中, 我们只能使用值集非常小的类型 (NoneType、bool) 或值集非常大的类型 (float、int、str、所有可能的元组等)。

参数 `x` 的类型可以是数字 (`int`、`complex`、`Fraction`、`numpy.uint32` 等)，可以是序列 (`str`、`tuple`、`list`、`array`)、 N 维 `numpy.array`，也可以是实现或继承接受 `int` 参数的 `__mul__` 方法的任何其他类型。

再来看一下下面这个带注解的 `double` 参数。请暂时忽略缺少的返回类型，让我们将精力集中在参数类型上：

```

1  from collections import abc
2
3  def double(x: abc.Sequence):
4      return x * 2

```

类型检查器将拒绝接受这段代码。如果您告诉 Mypy，`x` 是 `abc.Sequence` 类型，Mypy 会将 `x * 2` 标记为错误，因为 `Sequence` ABC 没有实现或继承 `__mul__` 方法。在运行时，该代码既能处理具体 (Concrete) 序列 (如 `str`、`tuple`、`list`、`array` 等)，也能处理数字。因为在运行时 [类型提示 \(Type Hints\)](#) 会被忽略。但类型检查器只关心显式声明的内容，而 `abc.Sequence` 没有 `__mul__`。

这就是为什么本节的标题是“类型由支持的操作来定义”。前文给出的两版 `double` 函数，都可被 Python 接受，参数 `x` 的值可以是任何对象。若参数 `x` 不支持乘法，则 `x*2` 操作会引发 `TypeError` 异常；若参数 `x` 支持乘法，则 `x*2` 操作可成功执行。但是，在分析带注解的 `double` 源码时，Mypy 会将 `x*2` 的声明视为错误。因为类型注解中的 `x: abc.Sequence` 不支持乘法操作。

在 [渐进式类型系统 \(Gradual Type System\)](#) 中，以下两种类型视图互相影响彼此：

- [鸭子类型 \(Duck Typing\)](#)

鸭子类型 (Duck Typing) 视图是 Smalltalk (面向对象语言的先驱) 以及 Python、JavaScript 和 Ruby 所采用的观点。在鸭子类型视图中，对象有类型，但变量 (包括参数) 没有类型。在实践中，对象声明的类型并不重要，重要的是它实际支持哪些操作。如果可以调用 `birdie.quack()`，那么在此上下文中 `birdie` 就是一只鸭子。根据定义，只有在运行时 (即尝试操作对象时)，才会强制执行鸭子类型的相关检查。这比 [名义类型](#)更灵活，但代价是在运行时潜在的错误更多⁴。

- [名义类型 \(Nominal Typing\)](#)

名义类型 (Nominal Typing) 视图是 C++、Java 和 C# 采用的观点，带 [类型提示 \(Type Hints\)](#) 的 Python 也采用此观点。在名义类型视图中，对象与变量都有类型。但是，对象仅存在于运行时，而类型检查器只关注源代码中用 [类型提示 \(Type Hints\)](#) 标注的变量 (包括参数)。如果 `Duck` 是 `Bird` 的子类，则可以将 `Duck` 实例赋值给注解为 `birdie: Bird` 的参数。但在函数体中，类型检查器会认为调用 `birdie.quack()` 是非法的，因为 `birdie` 名义上是 `Bird`，但该类 (即 `Bird`) 并未提供 `.quack()` 方法。运行时的 [实参 \(Argument\)](#) 是不是 `Duck` 并不重要，因为名义类型检查是静态执行的。类型检查器不会运行程序的任何部分，它只读取源码。这比 [鸭子类型](#)更严格，其优点是可以在构建流水线中，甚至是在 IDE 中输入代码的过程中，更早地捕获一些 bug。

[示例 8.4](#)是一个没有实用价值的示例，仅用于对比鸭子类型和名义类型，以及静态类型检查和运行时行为⁵。

```
</> 示例 8.4: birds.py
```

⁴Duck 类型是结构类型的一种隐式形式，Python 3.8 通过引入 `typing.Protocol` 也支持这种类型。本章稍后将对此进行介绍——“[8.5.10 静态协议](#)”。更多详细信息，参见“[十三 接口、协议与抽象基类](#)”。

⁵继承经常被过度使用，并且很难在务实而简单的示例中证明其合理性。因此，请将这个动物示例视为对子类型的一种简单说明。

```

1  class Bird:
2      pass
3  class Duck(Bird):
4      def quack(self):
5          print('Quack!')
6
7      def alert(birdie):
8          birdie.quack()
9
10     def alert_duck(birdie: Duck) -> None: ①
11         birdie.quack()
12
13     def alert_bird(birdie: Bird) -> None: ②
14         birdie.quack()

```

- ① Duck 是 Bird 的子类。
- ② 函数 alert 没有 [类型提示 \(Type Hints\)](#) ,因此类型检查器会忽略它。
- ③ 函数 alert_duck 接收一个 Duck 类型的参数。
- ④ 函数 alert_bird 接收一个 Bird 类型的参数。

使用 Mypy 对 birds.py 进行类型检查,我们发现了一个问题:

```

~/birds/ $ mypy birds.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)

```

仅仅通过分析源代码,Mypy 就发现 alert_bird 中的问题: [类型提示 \(Type Hints\)](#) 声明了 birdie 参数的类型为 Bird,但 alert_bird 函数体内调用了 birdie.quack()——而 Bird 类中没有这个方法。

如 [示例 8.5](#) 所示,在 daffy.py 中试用一下 birds 模块。

</> [示例 8.5:](#) 在 daffy.py 中试用一下 birds 模块 ([示例 8.4](#))

```

1  from birds import *
2
3  daffy = Duck()
4  alert(daffy)      ①
5  alert_duck(daffy) ②
6  alert_bird(daffy) ③

```

- ① 有效的调用,因为 alert 没有 [类型提示 \(Type Hints\)](#) 。
- ② 有效的调用,因为 alert_duck 接受的参数为 Duck 类型,而 daffy 是 Duck 对象。
- ③ 有效的调用,因为 alert_bird 接受的参数为 Bird 类型,而 daffy 也是 Bird (即 Duck 的超类) 对象。

运行 Mypy 检查 daffy.py 时,报告的错误与在 birds.py ([示例 8.4](#)) 中定义的 alert_bird 函数内调用 quack 一样。

```

~/birds/ $ mypy daffy.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)

```

但 Mypy 认为 daffy.py 本身没有问题:即 3 个函数调用都没问题。

现在,如果运行 daffy.py,就会得到以下结果:

```
…/birds/ $ python3 daffy.py
Quack!
Quack!
Quack!
```

一切正常,鸭子类型 (Duck Typing) 的优点体现的淋漓尽致。

Python 在运行时,对于声明的类型并不关心,只使用了鸭子类型 (Duck Typing)。Mypy 在 alert_bird 中标记了一个错误,但在运行时用 daffy 调用 alert_bird 却很正常。这可能会让许多 Python 爱好者一开始感到惊讶:静态类型检查器有时会在我们明确知道可以成功执行的代码中发现错误。

不过,如果几个月后你需要扩展这个 Bird 示例 (示例 8.4),您可能会感谢 Mypy。请看示例 8.6 中的 woody.py 模块,它也使用了 Birds 类。

</> 示例 8.6: woody.py

```
1 from birds import *
2
3 woody = Bird()
4 alert(woody)
5 alert_duck(woody)
6 alert_bird(woody)
```

Mypy 在检查 woody.py 时,发现了 2 处错误:

```
…/birds/ $ mypy woody.py
birds.py:16: error: "Bird" has no attribute "quack"
woody.py:5: error: Argument 1 to "alert_duck" has incompatible type "Bird";
expected "Duck"
Found 2 errors in 2 files (checked 1 source file)
```

第 1 处错误,出现在 birds.py 中:alert_bird 中的 birdie.quack() 调用,我们之前已经见过。第 2 处错误,出现在 woody.py:woody 是 Bird 的实例,因此调用 alert_duck(woody) 无效,因为该函数需要 Duck。每只鸭子都是鸟,但并非每只鸟都是鸭子。

在运行时,woody.py 中的所有调用都不会成功。示例 8.7 中的调用控制台会话最能说明故障的连续性。

</> 示例 8.7: 运行时的错误以及 Mypy 可以提供的帮助

```
1 >>> from birds import *
2 >>> woody = Bird()
3 >>> alert(woody)      ❶
4 Traceback (most recent call last):
5 ...
6 AttributeError: 'Bird' object has no attribute 'quack'
7 >>>
8 >>> alert_duck(woody) ❷
9 Traceback (most recent call last):
10 ...
```

```
11  AttributeError: 'Bird' object has no attribute 'quack'  
12  >>>>> alert_bird(woody) ❸  
13  Traceback (most recent call last):  
14  ...  
15  AttributeError: 'Bird' object has no attribute 'quack'
```

- ❶ Mypy 无法检测到此错误,因为函数 alert 中没有 [类型提示 \(Type Hints\)](#)。
- ❷ Mypy 报告了此问题:Argument 1 to "alert_duck" has incompatible type "Bird"; expected "Duck"
- ❸ 从 [示例 8.4](#) 开始,Mypy 就一直报告,函数 alert_bird 的主体有错误:"Bird" has no attribute "quack"。

这个小实验表明,鸭子类型 (Duck Typing) 更容易上手并且更灵活。但是它允许不受支持的操作在运行时导致错误。而 [名义类型 \(Nominal Typing\)](#) 会在运行前检测错误,但有时候会拒绝实际可正常运行的代码,例如 [示例 8.4](#) 中的 alert_bird(daffy) 调用。即使函数 alert_bird() 有时可以正常执行,但是该函数名称也不是很恰当:函数主体中的对象需要支持.quack() 方法,而 Bird 类没有这个方法。

该示例没有什么实际意义,函数主体只包含一行内容。现实中的函数主体会很长,可能会将参数 birdie 传给更多的函数,而且参数 birdie 的原始位置可能在许多函数调用之外,从而很难查明运行时出错的原因。类型检查器可以防止许多此类运行时错误的发生。



[类型提示 \(Type Hints\)](#) 的价值很难通过这种小示例体现出来。Python 代码的体量越大, [类型提示 \(Type Hints\)](#) 所带来的价值也就越明显。这也是为何拥有数百万行 Python 代码的公司 (如 Dropbox、Google 与 Facebook) 愿意投资团队和工具,以便在全公司范围内采用 [类型提示 \(Type Hints\)](#), 并且在其 CI 流水线中对 Python 代码库中的大量代码进行类型检查。

在本节中,我们从简单的 double() 函数开始,探讨了 [鸭子类型 \(Duck Typing\)](#) 和 [名义类型 \(Nominal Typing\)](#) 中类型和操作的关系。我们还没有为函数 double() 添加完整的 [类型提示 \(Type Hints\)](#)。8.5 节,将介绍可用于注解函数的重要类型。8.5.10 节将介绍为函数 double() 添加类型提示的一种更好的方式。但是,在此之前,我们还需要了解一些基础类型。

8.5 注解中可用的类型

几乎所有 Python 类型都可以在 [类型提示 \(Type Hints\)](#) 中使用,但也有一些限制和建议。此外, [typing 模块](#) 还引入了一些特殊的结构,其语义有时会令人惊讶。

本节涵盖了可用于 [类型提示 \(Type Hints\)](#) 的所有主要类型,包括:

- `typing.Any`;
- 简单的类型和类;
- `typing.Optional` 与 `typing.Union`;
- 泛化容器 (Generic Collections),包括元组与映射;
- 抽象基类 (ABCs);
- 泛化可迭代对象;
- 参数化泛型 (Generic Type) 与 `TypeVar`;

- `typing.Protocol`——静态鸭子类型 (Duck Typing) 的关键;
- `typing.Callable`;
- `typing.NoReturn`;

我们将依次介绍其中的每一种类型。先从一种看起来奇怪,似乎用处不大,但又非常重要的 `Any` 类型开始。

8.5.1 Any 类型

`Any` 类型,也称为“**动态类型**”,是 **渐进式类型系统** (Gradual Type System) 的基础。下面是一个没有类型信息的函数。

```
1 def double(x):
2     return x * 2
```

当类型检查器看到此函数时,会假定该函数拥有以下类型信息:

```
1 def double(x: Any) -> Any:
2     return x * 2
```

也就是说,参数 `x` 与函数返回值都可以是任何类型,二者甚至可以是不同的类型。`Any` 类型支持所有可能的操作。

以下述函数签名为例,对比一下 `Any` 与 `object`。

```
def double(x: object) -> object:
```

该函数也可接受所有类型的参数,因为任何类型都是 `object` 的子类型。

但是,类型检查器会拒绝以下函数。

```
1 def double(x: object) -> object:
2     return x * 2
```

这是因为 `object` 不支持 `__mul__` 操作。Mypy 报告的错误信息如下所示。

```
…/birds/ $ mypy double_object.py
double_object.py:2: error: Unsupported operand types for * ("object" and "int")
Found 1 error in 1 file (checked 1 source file)
```

越是通用的类型,具有的接口就越窄,即支持的操作越少。`object` 类实现的操作比 `abc.Sequence` 少;`abc.Sequence` 实现的操作比 `abc.MutableSequence` 少;`abc.MutableSequence` 实现的操作比 `list` 少。

但 `Any` 是一种神奇的类型,它位于类型层次结构的顶部和底部。它同时是最通用的类型(因此参数 `n: Any` 可接受所有类型的值),同时也是最特殊的类型,支持各种可能的操作。至少,类型检查器是这样理解 `Any` 的。

当然,任何类型都不可能支持所有可能的操作。因此,`Any` 类型不利于类型检查器完成其核心任务——检测潜在的非法操作,防止运行时异常导致程序崩溃。

8.5.1.1 子类型与相容

传统的面向对象的 [名义类型](#) (Nominal Typing) 系统依赖于 [子类型](#) (Subtype-of) 关系。给定一个类 T1 和一个子类 T2,那么 T2 就是 T1 的 [子类型](#) (Subtype-of)。

以下述代码为例:

```
1 class T1:  
2     ...  
3  
4 class T2(T1):  
5     ...  
6  
7 def f1(p: T1) -> None:  
8     ...  
9  
10 o2 = T2()  
11  
12 f1(o2) # 有效
```

f1(o2) 调用应用了 [里氏替换原则](#) (Liskov Substitution Principle)。Barbara Liskov⁶ 实际是从受支持的操作的角度,来定义子类型的:如果用类型 T2 的对象替换类型 T1 的对象后,程序仍可正常运行,则 T2 是 T1 的子类型。

继续前面的代码,像如下这种操作则违背了 [里氏替换原则](#) (Liskov Substitution Principle)。

```
1 def f2(p: T2) -> None:  
2     ...  
3  
4 o1 = T1()  
5  
6 f2(o1) # 类型错误
```

从受支持的操作的角度来看,这种行为非常合理:作为 T1 的子类,T2 继承并必须支持 T1 的所有操作。因此,在任何需要 T1 实例的地方都可以使用 T2 实例。但是,反过来就不一定了:T2 可能实现了其他方法。因此,在需要 T2 实例的地方,不一定都能使用 T1 实例。[行为子类型](#) (Behavioral Subtyping) 这一术语更能体现子类型的关注重点是受支持的操作。[行为子类型](#) (Behavioral Subtyping) 也用于指代 [里氏替换原则](#) (Liskov Substitution Principle)。

在 [渐进式类型系统](#) (Gradual Type System) 中,还有另一种关系:[相容](#) (Consistent-with)。能满足 [子类型](#) (Subtype-of) 关系的对象,一定是 [相容](#) (Consistent-with) 的。不过,对 Any 类型有一些特殊的规则(如下所示)。

1. 给定 T1 和子类型 T2,则 T2 与 T1 [相容](#) (Consistent-with) (里氏替换)。
2. 任何类型都与 Any 相容:声明为 Any 类型的参数,可接受任何类型的对象。
3. Any 与任何类型都相容:声明为任何类型的参数,都可接受 Any 类型的对象。

下面使用先前定义的对象 o1 与 o2 来说明规则 2 与规则 3。此段代码中的所有调用都是有效的。

⁶Barbara Liskov:麻省理工学院教授、编程语言设计师、图灵奖获得者。

```

1  def f3(p: Any) -> None:
2      ...
3  o0 = object()
4  o1 = T1()
5  o2 = T2()
6
7  f3(o0) #
8  f3(o1) # 都有效: 规则 2
9  f3(o2) #
10
11 def f4(): # 隐式返回类型为 `Any`
12     ...
13
14 o4 = f4() # 推断出类型为 `Any`
15
16 f1(o4) #
17 f2(o4) # 都有效: 规则 3
18 f3(o4) #

```

每个 [渐进式类型系统 \(Gradual Type System\)](#) 都需要有一个像 Any 这样的通配符类型。



动词“推断”是“猜测”的同义词,用于类型分析。Python 和其他语言中的现代类型检查器不需要到处都有类型注释,因为它们可以推断出许多表达式的类型。例如,如果我写 `x = len(s) * 10`,类型检查器不需要显式注解就能知道 `x` 是一个 `int`。因为,类型检查器可根据内置 `len` 中的类型提示猜测出 `x` 的类型。

接下来,探讨可在注释中使用的其他类型。

8.5.2 简单的类型与类

像 `int`、`float`、`str` 和 `bytes` 这样的简单类型可以直接在[类型提示 \(Type Hints\)](#) 中使用。标准库、外部包中的[具体类 \(Concrete Class\)](#),以及用户定义的[具体类 \(Concrete Class\)](#) (`French Deck`、`Vector2d` 和 `Duck`),也可以在[类型提示 \(Type Hints\)](#) 中使用。

[抽象基类 \(ABCs\)](#) 在[类型提示 \(Type Hints\)](#) 中也能用到。[“8.5.7 抽象基类”](#) 中将会讨论[抽象基类 \(ABCs\)](#)。

对于类来说,相容 (Consistent-with) 的定义与[子类型 \(Subtype-of\)](#) 类似: 即子类与其所有超类都相容 (Consistent-with)。

然而,“实用胜过纯粹 (practicality beats purity)⁷”,凡事都有例外,详见下面的提示栏。

⁷ practicality beats purity: 纯粹性 (purity) 指严格遵循原则、规范和标准; 实用性 (practicality) 指更注重寻找简洁、高效的解决方案,以满足实际需求。因此,“practicality beats purity”表示在软件开发中,要优先考虑实际需求,寻求简洁、高效、可行的解决方案,同时尽量保证代码的质量和可维护性。



`int` (整数) 与 `complex` (复数) 相容

内置类型 `int`、`float` 和 `complex` 之间没有名义上的子类型 (Subtype-of) 关系：它们都是 `object` 的直接子类。但 PEP 484 中却声明, `int` 与 `float` 相容 (Consistent-with), `float` 与 `complex` 相容 (Consistent-with)。从实用性角度看, 这一声明是合理的：`int` 实现了 `float` 的所有运算符, 并且 `int` 还额外实现了其他运算符, 如与 (&)、或 (|)、左移 (<<) 等。最终结果是, `int` 与 `complex` 也相容 (Consistent-with)。例如, 对于变量 `i=3` 的情况, `i.real` (实部) 返回的值为 3, 而 `i.imag` (虚部) 返回的值为 0。

8.5.3 Optional 与 Union 类型

“8.3.4 将 “None” 设为默认值” 中提到过特殊类型 `Optional`。该章节的一个示例用 `Optional` 解决了将 `None` 用作默认值的问题 (如下所示)。

```
1 from typing import Optional
2
3 def show_count(count: int, singular: str, plural: Optional[str] = None) -> str:
```

`Optional[str]` 实际上是 `Union[str, None]` 的缩写, 这意味着参数 `plural` 的类型可以是 `str` 或 `None`。



Python 3.10 为 `Optional` 与 `Union` 提供了更好的语法

从 Python 3.10 开始, `Union[str, bytes]` 可以写成 `str | bytes`。这种写法输入的内容更少, 并且不用从 `typing` 模块中导入 `Optional` 或 `Union`。对比一下函数 `show_count` 中参数 `plural` 的类型提示的新旧语法:

```
1 plural: Optional[str] = None # 旧语法
2 plural: str | None = None # 新语法 (Python >= 3.10)
```

或 (|) 运算符还可以用于构建 `isinstance` 或 `issubclass` 的第 2 个参数, 例如 `isinstance(x, int | str)`。详见 “PEP 604 –Allow writing union types as X | Y”。

内置函数 `ord` 的签名是 `Union` 的一个简单示例 - 它接受 `str` 或 `bytes` 类型的参数⁸, 并返回 `int`。

```
1 def ord(c: Union[str, bytes]) -> int: ...
```

下面的示例函数, 接受一个 `str`, 但返回 `str` 或 `float`。

```
1 from typing import Union
2
3 def parse_token(token: str) -> Union[str, float]:
4     try:
5         return float(token)
6     except ValueError:
7         return token
```

⁸严格来说, 函数 `ord` 接受的 `str` 或 `bytes` 必须满足 `len(s)==1`。但是, 类型系统目前还不能表达这种约束。

应尽可能避免创建返回 Union 类型的函数,因为它会给用户增加额外的负担——迫使用户必须在运行时检查返回值类型,以便了解如何处理它。但是,前述代码中的 `parse_token` 在简单表达式求值器的场景中,可以返回 Union 类型。



“4.10 支持 str 和 bytes 的双模式 API”中,介绍过一些接受 str 或 bytes 参数的函数。当参数为 str 时,函数返回 str;当参数为 bytes,函数返回 bytes。在此情况下,函数返回值类型由输入的参数类型决定,因此并不适合用 Union。要为此类函数提供正确的类型提示 (Type Hints),需要用到类型变量 (参见“8.5.9 参数化泛型与 TypeVar<229页>”)或重载(参见“15.2 重载的签名<418页>”)。

Union[] 至少需要 2 种类型。嵌套的 Union 类型与扁平 Union 的效果相同。因此,如下 2 种 [类型提示 \(Type Hints\)](#) 的作用是一致的。

```
1 Union[A, B, Union[C, D, E]]
2 Union[A, B, C, D, E]
```

Union[] 中的多个类型之间应该是不 [相容 \(Consistent-with\)](#) 的。例如,Union[int, float] 是多余的,因为 int 与 float [相容 \(Consistent-with\)](#)。如果只用 float 来注解参数,该参数也会接受 int 值。

8.5.4 泛化容器

大多数 Python 容器(如 list、tuple、set 等)都是异构的,可以容纳不同类型的元素⁹。例如,一个 list 中可以混合存放不同类型的项。但是,容器类型的这种异构特性实用性不高。在实践中,当将对象放入容器中后,你可能会希望在后续操作中对它们进行一些操作(如遍历、筛选、转换等)。这意味着,容器中的这些对象至少要共享一个通用的方法或行为。

可以用 [类型参数 \(Type Parameters\)](#) 来指定 [泛型 \(Generic Type\)](#),以指定容器中可以处理的项的类型。

例如,可以对 list 进行参数化,以限制 list 中元素的类型,如 [示例 8.8](#) 所示。

</> [示例 8.8: 带类型提示的 tokenize 函数 \(Python ≥ 3.9\)](#)

```
1 def tokenize(text: str) -> list[str]:
2     return text.upper().split()
```

在 Python > 3.9 中,上述类型提示表示函数 `tokenize` 返回一个 list,并且 list 中的各项均为 str 类型。

`stuff: list` 与 `stuff: list[Any]` 这两个 [类型提示 \(Type Hints\)](#) 的意思相同,表示 `stuff` 是一个 list,并且 `stuff` 中的项可以是任何类型的对象。



对于 Python ≤ 3.8,概念是一样的,只是需要编写更多的代码,详见附注栏“[向后兼容与弃用的容器类型](#)”。

⁹在 ABC (对 Python 初始设计影响最大的语言)中,每个列表只能接受单一类型的值:即列表中所有项都与第一项的类型保持一致。

“PEP 585—Type Hinting Generics In Standard Collections”列出了标准库中可接受 [泛型 \(Generic Type \)](#) 类型提示 (Type Hints) 的容器类型。如下仅列出了可使用最简单的 [泛型 \(Generic Type \)](#) 类型提示 (Type Hints) 的容器类型,如 `container[item]`。

list	<code>collections.deque</code>	<code>abc.Sequence</code>	<code>abc.MutableSequence</code>
set	<code>abc.Container</code>	<code>abc.Set</code>	<code>abc.MutableSet</code>
frozenset	<code>abc.Collection</code>		

`tuple` 与映射类型都支持更复杂的 [类型提示 \(Type Hints \)](#),详见“8.5.5 元组类型”与“8.5.6 泛化映射”。

截至 Python 3.10,很难准确的为 `array.array` 类型添加 [类型提示 \(Type Hints \)](#)。因为 `array.array` 类型有一个 `typecode` 构造函数参数,用于确定数组中存储的是整数还是浮点数。更棘手的问题是,如何对整数范围进行类型检查,以防止在向数组添加元素时引发 `OverflowError`。例如, `typecode='B'` 的数组只能保存 0 ~ 255 的整数值。目前,Python 的静态类型系统无法应对这个挑战。

向后兼容与弃用的容器类型

(若您只使用 Python ≥ 3.9 版本,则可跳过此附注栏。)

对于 Python 3.7 与 3.8,需要从 `__future__` 模块 导入相关内容,以使内置容器可以与符号 `[]` 一起工作。如 [示例 8.9](#) 所示。

</> [示例 8.9](#): 带类型提示的 tokenize 函数 (Python ≥ 3.7)

```
1 from __future__ import annotations
2
3 def tokenize(text: str) -> list[str]:
4     return text.upper().split()
```

`__future__` 模块 导入不适用于 Python ≤ 3.6 版本。[示例 8.10](#) 展示了适用于 Python ≥ 3.5 的 [类型提示 \(Type Hints\)](#) 方法。

</> [示例 8.10](#): 带类型提示的 tokenize 函数 (Python ≥ 3.5)

```
1 from typing import List
2
3 def tokenize(text: str) -> List[str]:
4     return text.upper().split()
```

最初,为了提供对 [泛型 \(Generic Type\)](#) 类型提示 (Type Hints) 的支持,PEP 484 作者在 `typing` 模块中创建了数十种 [泛型 \(Generic Type\)](#) 类型。表 8.1 列出了其中的一部分,完整列表详见 `typing` 模块文档。

表 8.1: 部分容器类型及对应的类型提示

容器类型	对应的类型提示
list	typing.List
set	typing.Set
frozenset	typing.FrozenSet
collections.deque	typing.Deque
collections.abc.MutableSequence	typing.MutableSequence
collections.abc.Sequence	typing.Sequence
abc.Set	typing.AbstractSet
abc.MutableSet	typing.MutableSet

为了提高 **泛型 (Generic Type)** **类型提示 (Type Hints)** 的可用性, “[PEP 585 –Type Hinting Generics In Standard Collections](#)” 发起过一项历时多年的行动。整个行动分为 4 个步骤。

1. 在 Python 3.7 中引入 `from __future__ import annotations`, 以允许在 **类型提示 (Type Hints)** 中使用标准库的类(如 `list`、`tuple` 等)作为 **泛型 (Generic Type)**, 如 `list[str]`。
2. 在 Python 3.9 中, 将 `list[str]` 这种表示法设为默认行为: 即无需导入 `__future__` 即可使用 `list[str]` 这种表示法。
3. 弃用 `typing` 模块 中的冗余 **泛型 (Generic Type)**^a。Python 解释器不会对弃用的类型发出警告。因为, 类型检查器在检查 Python ≥ 3.9 的代码时, 会对弃用的类型做出标记。
4. 在 Python 3.9 发布 5 年后的第一个 Python 版本(按当前节奏, 应该是 Python 3.14, 即 Python Pi)中, 删除多余的 **泛型 (Generic Type)**。

^a在 Guido van Rossum 的指导下, 我将 `typing` 模块文档 “[Module contents](#)” 下面的条目重新组织成一个小节, 并添加了数十条弃用警告。这是我对我 `typing` 模块文档 的贡献之一。

接下来, 看看如何为泛化元组添加 **类型提示 (Type Hints)**。

8.5.5 元组类型

注解元组类型有 3 种方式:

- 用作记录的元组;
- 用作记录的具名字段元组;
- 用作不可变序列的元组;

8.5.5.1 用作记录的元组

若将元组用作记录, 则使用 `tuple[各字段类型]` 来注解元组, 并在 `[]` 声明记录中各个字段的类型。例如, 一个接受 “城市名称、人口、所属国家” 的元组: `('上海', 24.28, '中国')`, 其 **类型提示 (Type Hints)** 为 `tuple[str, float, str]`。

已知函数 `geohash()` 接受的参数是一对经纬度地理坐标, 返回值是一个 `Geohash`。其用法如下所示:

```
1 >>> shanghai = 31.2304, 121.4737
2 >>> geohash(shanghai)
```

```
3 'wtw3sjq6q'
```

示例 8.11 展示了如何用 PyPI 中的 `geolib` 包 定义函数 `geohash`。

</> 示例 8.11: coordinates.py:geohash 函数

```
1 from geolib import geohash as gh # type: ignore ❶
2
3 PRECISION = 9
4
5 def geohash(lat_lon: tuple[float, float]) -> str: ❷
6     return gh.encode(*lat_lon, PRECISION)
```

❶ 此注释会阻止 Mypy 报告 `geolib` 包 没有 [类型提示 \(Type Hints\)](#)。

❷ “`lat_lon: tuple[float, float]`” 将参数 `lat_lon` 注解为包含 2 个 `float` 字段的元组。



对于 Python < 3.9, 注解用作记录的元组需要导入并使用 `typing.Tuple`。自 Python 3.9 开始, `typing.Tuple` 已被弃用, 不过在 2024 年之前仍会被保留在标准库中。

8.5.5.2 用作记录的具名字段元组

要注解一个包含多个字段的元组, 或者要注解代码中多次用到的特定类型元组, 则强烈推荐使用 `typing.NamedTuple` (参见 “[五 数据类构建器](#)”)。示例 8.12 用 `NamedTuple` 注解了 [示例 8.11](#) 中的函数 `geohash`。

</> 示例 8.12: coordinates_named.py:具名元组 Coordinates 与 geohash 函数

```
1 from typing import NamedTuple
2 from geolib import geohash as gh # type: ignore
3
4 PRECISION = 9
5
6 class Coordinate(NamedTuple):
7     lat: float
8     lon: float
9
10 def geohash(lat_lon: Coordinate) -> str:
11     return gh.encode(*lat_lon, PRECISION)
```

如 “[5.2 数据类构建器概述](#)” 所描述, `typing.NamedTuple` 是 `tuple` 子类的 [工厂函数 \(Factory Function\)](#)¹⁰。因此, `Coordinate` 与 `tuple[float, float]` 相容 ([Consistent-with](#))。但反过来则不成立, 因为 `NamedTuple` 为 `Coordinate` 额外添加了方法 (例如 `__asdict`)。另外, 用户也可以定义方法。

在实践中, `Coordinate` 实例可以安全地传给如下定义的 `display` 函数, 而不会引发类型错误或异常。

```
1 def display(lat_lon: tuple[float, float]) -> str:
```

¹⁰译者注: 表示 `typing.NamedTuple` 可使用指定的名称与字段, 构建 `tuple` 的子类。

```

2     lat, lon = lat_lon
3     ns = 'N' if lat >= 0 else 'S'     ew = 'E' if lon >= 0 else 'W'
4     return f'{abs(lat):0.1f}°{ns}, {abs(lon):0.1f}°{ew}'

```

8.5.5.3 用作不可变序列的元组

若要为长度不定、用作不可变序列的元组增加 [类型提示 \(Type Hints\)](#)，则需要使用格式为“tuple[类型, ...]”的语法¹¹。例如，tuple[int, ...] 表示所有项都为 int 类型的元组。

省略号 (...) 表示元组的元素数量 ≥ 1 。可变长度元组的 [类型提示 \(Type Hints\)](#) 中，不能为字段指定不同的类型，即只能指定一个类型。

“stuff: tuple[Any, ...]” 与 “stuff: tuple” 这 2 个 [类型提示 \(Type Hints\)](#) 的意思相同，都表示 stuff 是一个长度不定的元组，其中的元素可以是任意类型。

下面的代码用函数 columnize 将一个序列转换成了元组列表（类似于表中的行与单元格），列表中的每个元组长度不定。最后，按列显示元组中的各项，如下所示。

```

1  >>> animals = 'drake fawn heron ibex koala lynx tahr xerus yak zapus'.split()
2  >>> table = columnize(animals)
3  >>> table
4  [('drake', 'koala', 'yak'), ('fawn', 'lynx', 'zapus'), ('heron', 'tahr'), ('ibex', 'xerus')]
5  >>> for row in table:
6      print(''.join(f'{word:10}' for word in row))
7
8  drake    koala    yak
9  fawn     lynx     zapus
10 heron    tahr
11 ibex     xerus

```

函数 columnize 的实现如 [示例 8.13](#) 所示。请注意 [类型提示 \(Type Hints\)](#) 中的返回值类型。

```

1  list[tuple[str, ...]]      # 返回一个元组列表，且每个元组的长度不定

```

</> [示例 8.13: columnize.py: 返回一个列表，列表中的项是字符串元组](#)

```

1  from collections.abc import Sequence
2
3  def columnize(
4      sequence: Sequence[str], num_columns: int = 0
5  ) -> list[tuple[str, ...]]:      # 返回一个元组列表，且每个元组的长度不定
6      if num_columns == 0:
7          num_columns = round(len(sequence) ** 0.5)
8          num_rows, remainder = divmod(len(sequence), num_columns)      # len() 需要一个预先
9         构建的序列
10         num_rows += bool(remainder)
11         return [tuple(sequence[i::num_rows]) for i in range(num_rows)]

```

¹¹tuple[int, ...]:这里的 int 表示元组中的各项都是 int 类型；... 是 3 个点，而不是 Unicode 字符 U+2026 (即 HORIZONTAL ELLIPSIS)，表示元组的长度不定；

8.5.6 泛化映射

泛化映射类型的 [类型提示 \(Type Hints\)](#) 语法为 `MappingType[KeyType,ValueType]`。在 Python ≥ 3.9 中, 内置 `dict`、`collections` 与 `collections.abc` 中的映射类型均可使用此种语法添加 [类型提示 \(Type Hints\)](#)。对于早期版本的 Python, 为泛化映射添加 [类型提示 \(Type Hints\)](#) 必须使用 `typing.Dict` 以及 `typing` 模块中的映射类型, 详见 “[8.5.4 泛化容器](#)” 中的附注栏“向后兼容与弃用的容器类型”。

示例 8.14 中定义了一个反回倒排索引的实用函数 `name_index`。该函数用于按名称搜索 Unicode 字符，是对“示例 4.21<122页>”的改造，更适合在服务器端运行（详见“[二十一 异步编程](#)”）。

函数 `name_index` 根据给定起始、终止的 Unicode 字符编码，返回一个类型为 `dict[str, set[str]]` 的映射。该映射是一个倒排序索引，将每个单词映射到一组名称中包含该单词的字符。例如，对于从 32 到 64 的 ASCII 字符进行索引后，以下是与单词 ‘SIGN’ 和 ‘DIGIT’ 相关的字符集合。这里还展示了如何搜索名为 ‘DIGIT EIGHT’ 的字符。

```
1 >>> index = name_index(32, 65)
2 >>> index['SIGN']
3 {'$', '>', '=', '+', '<', '%', '#'}
4 >>> index['DIGIT']
5 {'8', '5', '6', '2', '3', '0', '1', '4', '7', '9'}
6 >>> index['DIGIT'] & index['EIGHT']
7 {'8'}
```

示例 8.14 展示了函数 `name_index` 所在的 `charindex.py` 源码。除了 `dict[]` 类型提示之外，该示例还包含了本书中首次出现的 3 个功能。

</> 示例 8.14: charindex.py

```
1 import sys
2 import re
3 import unicodedata
4 from collections.abc import Iterator
5
6 RE_WORD = re.compile(r'\w+')
7 STOP_CODE = sys.maxunicode + 1
8
9 def tokenize(text: str) -> Iterator[str]:          ❶
10    """返回由大写单词构成的可迭代对象"""
11    for match in RE_WORD.finditer(text):
12        yield match.group().upper()
13
14 def name_index(start: int = 32, end: int = STOP_CODE) -> dict[str, set[str]]: ❷
15    index: dict[str, set[str]] = {}
16    for char in (chr(i) for i in range(start, end)):
17        if name := unicodedata.name(char, ''): ❸
18            for word in tokenize(name):
19                index.setdefault(word, set()).add(char)
20
21 return index
```

- ① tokenize 是一个生成器函数,返回值类型为 `Iterator[str]`。将在“十七 迭代器、生成器和经典协程”介绍生成器。
- ② 为局部变量 `index` 添加 [类型提示 \(Type Hints\)](#)。若不加此类型提示, Mypy 将发出提示:“Need type annotation for 'index' (hint: "index: dict[<type>, <type>] = ...").”。
- ③ `if` 条件中使用了 [海象运算符 \(:=\)](#)¹²。该运算符将 `unicodedata.name()` 的结果赋值给 `name`, 并将该结果作为 `if` 语句的判断条件。若该结果是表示 `False` 的”, 则不更新 `index`。



当将 `dict` 用作记录时, 通常所有键(Key)使用 `str` 类型, 而值(Value)则根据键(Key)的含义, 采用不同的类型。详见“[15.3 TypedDict<423页>](#)”。

8.5.7 抽象基类

发送信息时要保守, 接受信息时要大方。

——波斯特尔定律, 又称“稳健性原则”

“[表 8.1<221页>](#)”列出了 `collections.abc` 中的几个抽象类。理想情况下, 函数参数应该接受这些抽象类(或者 Python 3.9 之前, `typing` 模块中对应的等效类型), 而不是具体类(Concrete Class)。这样, 可以为调用方提供更大的灵活性。

以下函数签名为例:

```
1 from collections.abc import Mapping
2 def name2hex(name: str, color_map: Mapping[str, int]) -> str:
```

由于参数 `color_map` 的注解类型是 `abc.Mapping`, 因此调用者可以为该参数提供 `dict`、`defaultdict`、`ChainMap`、`UserDict` 子类的实例, 或者 `Mapping` 子类型(`Subtype-of`)的实例。

相比之下, 请看如下的函数签名:

```
1 def name2hex(name: str, color_map: dict[str, int]) -> str:
```

由于参数 `color_map` 的注解类型是 `dict`, 因此调用者必须为其提供 `dict` 或 `dict` 的子类型(如 `defaultDict` 或 `OrderedDict`)。尽管 Python 推荐用 `UserDict` 子类创建用户定义的映射(), 但 `UserDict` 子类会被 MyPy 类型检查器拒绝。因为, `UserDict` 与 `dict` 不是子类型(`Subtype-of`)关系, 而是兄弟关系。即二者都是 `abc.MutableMapping` 的子类¹³。

因此, 一般来说, 在参数类型提示中最好使用 `abc.Mapping` 或 `abc.MutableMapping`, 而不是 `dict`(或 `typing.Dict`)。若函数 `name2hex` 不需要更改给定的 `color_map`, 则 `color_map` 最准确的类型提示应该是 `abc.Mapping`。这样, 调用者就无需提供实现了 `setdefault`、`pop` 和 `update` 等方法的对象。因为, 这些方法属于 `MutableMapping` 接口, 但不属于 `Mapping` 接口。此种做法体现了“波斯特尔定律”的后半部分:“接受信

¹²本书有些示例使用了海象运算符(`:=`), 但本书并未对该运算符做过多介绍。有关海象运算符(`:=`)的详细信息, 参见“[PEP 572 - Assignment Expressions](#)”。

¹³实际上, `dict` 是 `abc.MutableMapping` 的虚拟子类(`Virtual Subclass`) (有关虚拟子类的介绍, 详见“[十三 接口、协议与抽象基类](#)”)。现在, 只需知道 `issubclass(dict, abc.MutableMapping)` 为 `True`, 尽管 `dict` 是用 C 语言实现的, 并且未从 `abc.MutableMapping` 继承任何东西, 而只是继承自 `object`。

息时要大方”。

“波斯特尔定律”还告诉我们“发送信息时要保守”。函数的返回值应该是 **具体类 (Concrete Class)** 的对象，即返回值的 **类型提示 (Type Hints)** 应该是一个 **具体类型 (Concrete Type)**，如“示例 8.8<220页>”的返回值的类型提示为“list[str]”。

```
1 def tokenize(text: str) -> list[str]:  
2     return text.upper().split()
```

在 `typing.List` 文档中，有这样一段话：

泛化 `list` 通常用于注解函数返回值；若要注解函数参数，则推荐使用抽象容器类型（如 `abc.Sequence` 或 `abc.Iterable`）。

`typing.Dict` 与 `typing.Set` 文档中，也有类似的说明。

请记住，从 Python 3.9 开始，`collections.abc` 中的大多数 **抽象基类 (ABCs)**、`collections` 中的 **具体类 (Concrete Class)**，以及内置的容器类型（如 `list`、`tuple` 等），全都支持 **泛型 (Generic Type)** 类型提示（`Type Hints`），例如 `collections.deque[str]`。只有在 Python ≤ 3.8 中编写 `Type Hints` 代码时，才需要使用 `typing` 模块中对应的容器类型。关于 **泛型 (Generic Type)** 类的完整列表，详见“[PEP 585 - Type Hinting Generics In Standard Collections](#)”的“[Implementation](#)”一节。

在结束讨论 `Type Hints` 中的 **抽象基类 (ABCs)** 之前，有必要讲一下 `numbers` 包中的 **抽象基类 (ABCs)**。

8.5.7.1 数字塔的崩塌

`numbers` 包 定义了在“[PEP 3141 - A Type Hierarchy for Numbers](#)”中提出的所谓 **数字塔 (Numeric Tower)**。该塔是一种由 **抽象基类 (ABCs)** 构成的线性层次结构，`Number` 位于最顶层。

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

这些 **抽象基类 (ABCs)** 非常适合运行时类型检查，但不适合静态类型检查。“[PEP 484 - Type Hints](#)”中的“[The numeric tower](#)”一节拒绝使用 `numbers` 包中的 **抽象基类 (ABCs)**，并规定内置类型 `complex`、`float`、`int` 应被视为特例（详见“[8.5.2 简单的类型与类<218页>](#)”的提示栏“`int`（整数）与 `complex`（复数）相容”）。

我们将在“[13.6.8 number 模块中的抽象基类与 Numeric 协议<385页>](#)”比较协议与抽象基类（`ABCs`）时，再来讨论这个问题。

在实践中，若要注解数字参数以便进行静态类型检查，可有以下几种选择。

1. 按“[PEP 484 - Type Hints](#)¹⁴”的建议，使用 `int`、`float` 或 `complex` 中的某个 **具体类型 (Concrete Type)**。
2. 声明 `Union` 类型，如 `Union[float, Decimal, Fraction]`。

¹⁴译者注：原书此处为“PEP 488”。但根据上下文猜测可能是笔误。因为，“PEP 488”是关于 PYO 文件的提案，与 `Type Hints` 不相关。

3. 若想避免使用 [具体类型 \(Concrete Type\)](#) 的硬编码 (Hardcoded), 可以使用 `SupportsFloat` 等数字协议 (详见 “[13.6.2 运行时可检查的静态协议<377页>](#)”)。要了解数字协议, 必须先阅读 “[8.5.10 静态协议<233页>](#)” 的内容。

接下来, 介绍一下在 [类型提示 \(Type Hints\)](#) 中经常用到的一种 [抽象基类 \(ABCs\)](#), 即 `abc.Iterable`。

8.5.8 Iterable

前文中引用的 `typing.List` 文档中, 推荐使用 `abc.Sequence` 或 `abc.Iterable` 来注解函数参数。

标准库中的 `math.fsum` 函数就是一个使用 `Iterable` 注解函数参数的例子:

```
1 def fsum(__seq: Iterable[float]) -> float:
```



存根 (Stub) 文件与 Typeshed 项目

截至 Python 3.10, 标准库中本身不含 [类型提示 \(Type Hints\)](#)。但是像 Mypy、PyCharm 等工具可以在 [Typeshed 项目](#) 中找到所需的 [类型提示 \(Type Hints\)](#), 这些类型提示位于一种 [存根文件 \(stub file\)](#) 中。存根文件是一种具有 `.pyi` 扩展名的特殊源码文件, 其中保存了带有 [类型提示 \(Type Hints\)](#) 的函数签名和方法签名, 但是没有函数或方法的具体实现, 类似于 C 语言的头文件。

`math.fsum` 函数的签名位于 “`/stdlib/2and3/math.pyi`” 文件中。参数 `__seq` 中的双下划线(`__`)是 “[PEP 484 – Type Hints](#)” 对 [仅限位置参数 \(Positional-only Parameter\)](#) 的约定, 详见 “[8.6 注解仅限位置参数与变长参数<240页>](#)”。

[示例 8.15](#) 中的函数 `zip_replace` 是另一个使用 `Iterable` 注解函数参数的例子。参数 `changes` 产生的项为 `tuple[str, str]` 类型。函数 `zip_replace` 的用法如下:

```
1 >>> l33t = [('a', '4'), ('e', '3'), ('i', '1'), ('o', '0')]
2 >>> text = 'mad skilled noob pownd leet'
3 >>> from replacer import zip_replace
4 >>> zip_replace(text, l33t)
5 'm4d sk1ll3d n00b p0wn3d l33t'
```

[示例 8.15](#) 展示了函数 `zip_replace` 的具体实现。

</> [示例 8.15: replacer.py](#)

```
1 from collections.abc import Iterable
2
3 FromTo = tuple[str, str] ❶
4
5 def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷
6     for from_, to in changes:
7         text = text.replace(from_, to)
8
9     return text
```

❶ `FromTo` 是一个类型别名。此处, `FromTo` 是 `tuple[str, str]` 类型的别名, 这样可以提高 `zip_replace` 函数

签名的可读性。

- ② 参数 changes 的类型为 Iterable[FromTo]。Iterable[FromTo] 与 Iterable[tuple[str, str]] 的效果相同, 但前者可使 zip_replace 的函数签名更简洁, 可读性更高。



Python 3.10 中的显式类型别名

“PEP 613 –Explicit Type Aliases” 引入了一种特殊类型——TypeAlias, 可使创建类型别名的赋值操作更显而易见, 也更易于类型检查。从 Python 3.10 开始, 创建类型别名的首选方法如下所示:

```
1 from typing import TypeAlias
2 FromTo: TypeAlias = tuple[str, str]
```

8.5.8.1 abc.Iterable 与 abc.Sequence

math.fsum 和 replacer.zip_replace 都必须遍历整个 Iterable 参数才能返回结果。若传入一个无限迭代对象 (如 `itertools.cycle` 生成器), 这两个函数将会耗尽内存, 并导致 Python 进程崩溃。尽管存在潜在的风险, 但在现代 Python 中, 提供接受 Iterable 输入并等待 Iterable 参数被完全处理完毕才返回结果的函数是很常见的。这样, 调用者就可以选择以生成器的形式提供输入数据, 而不是提供预先构建的序列。若输入项的数据量很大, 生成器可以节省大量内存。

另外, “示例 8.13<224页>” 中的函数 columnize 需要的参数类型是 Sequence, 而不是 Iterable。因为函数中的 `len()` 需要一个预先构建好的序列, 才能计算出序列中的项数。

与 Sequence 一样, Iterable 也适合用于注解函数参数, 而不适合注解函数返回值。因为函数返回值的类型需要更加明确和具体, 以方便调用者进行下一步地处理。

与 Iterable 密切相关的是 Iterator 类型。如 “示例 8.14<225页>” 所示, 将 Iterator 类型用作函数 `tokenize()` 的返回值类型。“十七 迭代器、生成器和经典协程” 在讲解生成器与迭代器时, 将详细介绍 Iterator 类型。

8.5.9 参数化泛型与 TypeVar

参数化泛型 (Parameterized Generic Type) 也是一种 泛型 (Generic Type), 用 `list[T]` 表示。其中, `T` 是一个类型变量, 在每次使用时都会将 `T` 绑定到一个特定的类型。这样可以将参数类型 (`T`) 反映在结果类型上。

示例 8.16 中定义的函数 `sample`, 将接受 2 个参数: 一个是元素类型为 `T` 的 Sequence, 另一个是 `int`。该函数会返回一个 `list`, `list` 中的元素类型为 `T` (具体类型由函数第一个参数 `Sequence(T)` 决定)。

```
</> 示例 8.16: sample.py
1 from collections.abc import Sequence
2 from random import shuffle
3 from typing import TypeVar
4
5 T = TypeVar('T')
6
7 def sample(population: Sequence[T], size: int) -> list[T]:
8     if size < 1:
9         raise ValueError('size must be >= 1')
```

```

10     result = list(population)
11     shuffle(result)      return result[:size]

```

如下两个示例说明为何要在函数 sample 中使用类型变量：

- 调用者若传入 tuple[int, ...] (与 Sequence[int] 相容) 类型的元组。类型参数 T 为 int, 则返回值类型为 list[int]。
- 调用者若传入一个 str (与 Sequence[str] 相容)。类型参数 T 为 str, 则返回值类型为 list[str]。



为何需要 TypeVar

“PEP 484”的作者希望通过添加 `typing` 模块来引入 **类型提示**(Type Hints), 而不用更改语言中的其他任何内容。通过巧妙地元编程, 可以使运算符 [] 在 Sequence[T] 这样的类上工作。但方括号内类型变量 T 的名称必须在某处定义, 否则需要大规模改动 Python 解释器才能让 **泛型** (Generic Type) 支持特殊的 [] 表示法。正因如此, 我们增加了构造函数 `typing.TypeVar`: 将类型变量的名称引入到当前名称空间。由于 Java、C# 和 TypeScript 等语言不需要预先声明类型变量的名称, 因此没有与 Python 的 `TypeVar` 类相对应的数据结构。

标准库中的 `statistics.mode` 函数 是 **参数化泛型** (Parameterized Generic Type) 的另一个示例, 该函数可以返回一系列数值中最常出现的数据点。以下是摘自 `statistics.mode` 文档 的一个使用示例：

```

1  >>> mode([1, 1, 2, 3, 3, 3, 3, 4])
2  3

```

若不使用 `typing.TypeVar`, 函数 mode 的签名可能如 [示例 8.17](#) 所示。

</> [示例 8.17: mode_float.py](#): 可操作 float 及其子类型的 mode 函数¹⁵

```

1  from collections import Counter
2  from collections.abc import Iterable
3
4  def mode(data: Iterable[float]) -> float:
5      pairs = Counter(data).most_common(1)
6      if len(pairs) == 0:
7          raise ValueError('no mode for empty data')
8      return pairs[0][0]

```

函数 mode 可用于处理 int 或 float 值, 但是 Python 中还支持其他数字类型 (如 decimal 等)。为了提高函数 mode 的通用性与灵活性, 我们希望函数 mode 的返回值类型, 能够根据输入参数 Iterable 的元素类型进行自动推断。例如, 传入的参数为 Iterable[decimal], 则返回值类型为 decimal。为了实现此目标, 可使用 `typing.TypeVar` 改进函数 mode 的签名。先来看一个简单, 但不正确的参数化函数签名。

```

1  from collections.abc import Iterable
2  from typing import TypeVar
3
4  T = TypeVar('T')

```

¹⁵这里的实现比 Python 标准库 `statistics` 模块中的实现要更简单。

```
5  
6 def mode(data: Iterable[T]) -> T:
```

在这里, `TypeVar` 是一种 `泛型 (Generic Type)` 类型变量, 它可以表示任意类型。当第一次出现在函数签名中时, `TypeVar` 可以代表任何类型。当第二次出现时, 它表示和第一次相同的类型。

因此, 任何可迭代对象 (如列表、元组、集合), 只要它包含的元素类型相同, 即可视为与 `Iterable[T]` `相容 (Consistent-with)`, 包括 `collections.Counter` 无法处理的不可哈希 (`hashable`) 的可迭代类型。在某些情况下, 需要限制可分配给 `泛型 (Generic Type)` 类型变量 `T` 的类型, 以使 `泛型 (Generic Type)` 能够适应特定的数据类型或范围。限制 `泛型 (Generic Type)` 类型变量 `T` 的类型, 可以提高代码的类型安全性、可读性与健壮性。

下面介绍可限制 `泛型 (Generic Type)` 类型变量的两种方式: 受限的 `TypeVar`、有届的 `TypeVar`。

8.5.9.1 受限的 `TypeVar`

`TypeVar` 接受额外的位置参数来限制类型参数。我们可以改进函数 `mode` 的签名, 以使其仅接受特定的数字类型, 如下所示。

```
1 from collections.abc import Iterable  
2 from decimal import Decimal  
3 from fractions import Fraction  
4 from typing import TypeVar  
5  
6 NumberT = TypeVar('NumberT', float, Decimal, Fraction)  
7  
8 def mode(data: Iterable[NumberT]) -> NumberT:
```

这样就好多了, 通过 `TypeVar` 的位置参数, 限制了 `Iterable` 的元素类型只能是 `float`、`Decimal` 或 `Fraction`。这也是 2020 年 5 月 25 日 `typeshed` 上 `statistics.pyi` 存根文件中函数 `mode` 的签名。

然而, `statistics.mode` 文档中还有以下示例。

```
1 >>> mode(["red", "blue", "blue", "red", "green", "red", "red"])  
2 'red'
```

看到此示例, 你的第一反应可能是在 `NumberT` 的定义中再添加一个 `str`。

```
1 NumberT = TypeVar('NumberT', float, Decimal, Fraction, str)
```

这么做也不可以, 但是 `NumberT` 的名称就文不对题了。更重要的是, 我们不能每发现一个可以处理的类型, 就为 `NumberT` 添加一个类型。更好的方法是使用 `TypeVar` 的另一项功能: 有届的 `TypeVar`。

8.5.9.2 有届的 `TypeVar`

示例 8.17 中的 `mode` 函数使用了 `Counter` 类来处理排名。`Counter` 类是基于 `dict` 实现的, 因此可迭代对象 `data` 中的元素类型必须是 `可哈希 (hashable)` 的。

乍一看, 这个函数签名似乎有效:

```
1 from collections.abc import Iterable, Hashable  
2 def mode(data: Iterable[Hashable]) -> Hashable:
```

现在的问题是,函数的返回值类型是 `Hashable`——一个仅实现了 `__hash__` 方法的 [抽象基类 \(ABCs\)](#)。类型检查器将限制我们对 `Hashable` 返回值的操作:除了可以调用其中的 `hash()` 方法以外,不能做其他任何操作。所以,返回 `Hashable` 并没有什么实际意义。

解决方案是使用 `TypeVar` 另一个可选的关键字参数 `bound`。此参数会为 `TypeVar` 可接受的类型设定一个上边界。[示例 8.18](#) 用 `bound=Hashable` 指明,参数化泛型 (Parameterized Generic Type) 的类型变量可以是 `Hashable` 或 `Hashable` 的子类型 (Subtype-of)¹⁶。

</> [示例 8.18: mode_hashable.py](#): 与 [示例 8.17](#)类似,但签名更灵活

```

1  from collections import Counter
2  from collections.abc import Iterable, Hashable
3  from typing import TypeVar
4
5  HashableT = TypeVar('HashableT', bound=Hashable)
6
7  def mode(data: Iterable[HashableT]) -> HashableT:
8      pairs = Counter(data).most_common(1)
9      if len(pairs) == 0:
10          raise ValueError('no mode for empty data')
11      return pairs[0][0]
```

总结一下:

- 受限的类型变量,会将类型设置为 `TypeVar` 声明中指定的类型之一。例如,`NumberT = TypeVar('NumberT', float, Decimal, Fraction)` 表示 `NumberT` 的类型只能为 `float`、`Decimal` 或 `Fraction`。
- 有界的类型变量,会将类型设置为根据表达式推断出来的类型,但前提是推断出来的类型需要与 `TypeVar` 的 `bound` 关键字参数声明的边界相容。例如,`HashableT = TypeVar('HashableT', bound=Hashable)` 表示 `HashableT` 的类型只能为 `Hashable` 或 `Hashable` 的子类型 (Subtype-of)。因为 `Hashable` 及其子类型,都与 `Hashable` 相容。



为 `TypeVar` 声明边界的关键字参数名为“`bound`”,这个命名有时会令人困惑。因为在 Python 的引用语义中,“`bound`”通常用于表示将一个名称与某个值相关联,是一个动词。而在声明有界 `TypeVar` 时,参数“`bound`”却表示指定边界(即类型的上限)。若将这个指定边界的参数命名为“`boundary`”,可能会更清晰明了,不会让人误解。

`typing.TypeVar` 构造函数其他两个可选参数:`covariant` 与 `contravariant`。将在“[15.7 型变<438页>](#)”中介绍这两个参数。

最后,让我们用 `typing.AnyStr` 来结束对 `typing.TypeVar` 的介绍。

8.5.9.3 预定义的类型变量 `AnyStr`

`typing` 模块包含一个预定义的名为 `AnyStr` 的 `TypeVar`,其定义如下:

¹⁶我将此解决方案贡献给了 `typeshed` 项目。自 2020 年 5 月 26 日起, `statistics.pyi` 就是这样注解 `mode` 函数的。

```
1 AnyStr = TypeVar('AnyStr', bytes, str)
```

许多接受 bytes 或 str 参数的函数都会使用 AnyStr, 返回值的类型也是 bytes 或 str。

接下来换个话题, 讲一下 typing.Protocol。这是 Python 3.8 的一个新特性, 旨在以更符合 Python 风格的方式编写 [类型提示 \(Type Hints\)](#)。

8.5.10 静态协议



在面向对象编程中, 将“[协议 \(protocol\)](#)”作为一种非正式接口的概念可以追溯到历史久远的 Smalltalk, 并且从一开始就是 Python 的重要组成部分。而在 [类型提示 \(Type Hints\)](#) 的上下文中, “[协议 \(protocol\)](#)”是一个 typing.Protocol 的子类, 用于定义一个类型检查器可以验证的接口。这两种类型的协议在“[十三 接口、协议与抽象基类](#)”中都有介绍, 本节是在 [类型提示 \(Type Hints\)](#) 上下文中对其的简要介绍。

“[PEP 544 –Protocols: Structural subtyping \(static duck-typing\)](#)”中介绍的 Protocol 类型, 类似于 Go 语言中的接口: [协议 \(protocol\)](#) 类型是通过指定一个或多个方法来定义的, 类型检查器会在需要协议类型的地方验证这些方法是否已被实现。

在 Python 中, 通过继承 typing.Protocol 来定义“[协议 \(protocol\)](#)”。

在 Python 中, “[协议 \(protocol\)](#)”定义被编写成 typing.Protocol 的子类。但是, 实现协议的类无需与定义协议的类建立任何关联 (如无需继承, 也无需注册)。类型检查器负责查找可用的协议类型, 并强制检查协议的使用。

下面是一个可以借助 Protocol 和 TypeVar 解决的问题。假设要创建一个函数 top(it,n), 返回可迭代对象 it 中最大的 n 个元素:

```
1 >>> top([4, 1, 5, 2, 6, 7, 3], 3)
2 [7, 6, 5]
3 >>> l = 'mango pear apple kiwi banana'.split()
4 >>> top(l, 3)
5 ['pear', 'mango', 'kiwi']
6 >>>
7 >>> l2 = [(len(s), s) for s in l]
8 >>> l2
9 [(5, 'mango'), (4, 'pear'), (5, 'apple'), (4, 'kiwi'), (6, 'banana')]
10 >>> top(l2, 3)
11 [(6, 'banana'), (5, 'mango'), (5, 'apple')]
```

使用 [参数化泛型 \(Parameterized Generic Type\)](#) 定义的 top 函数, 如 [示例 8.19](#) 所示。

</> [示例 8.19](#): top 函数, 有一个未定义的类型参数 T

```
1 def top(series: Iterable[T], length: int) -> list[T]:
2     ordered = sorted(series, reverse=True)
3     return ordered[:length]
```

那么,如何约束 T 的取值呢? T 不能是 Any 与 object,因为参数 series 必须支持用函数 sorted 进行排序。内置函数 sorted 实际上是支持 Iterable[Any] 的,但前提是可选参数 key 传入的函数能为每个元素计算出一个排序键。若将一个普通对象列表传给函数 sorted,但不提供参数 key,会发生什么呢?试试看就知道了:

```

1  >>> l = [object() for _ in range(4)] # 构建一个列表, 列表包含 4 个对象。
2  >>> l
3  [<object object at 0x10fc2fc0a0>, <object object at 0x10fc2fb0b0>,
4  <object object at 0x10fc2fb0c0>, <object object at 0x10fc2fb0d0>]
5  >>> sorted(l)
6  Traceback (most recent call last):
7  File "<stdin>", line 1, in <module>
8  TypeError: '<' not supported between instances of 'object' and 'object'
```

错误信息表明,函数 sorted 在可迭代对象的元素上应用了 < 运算符。但只应用 < 运算符就足够了吗?让我们再看一个实验¹⁷:

```

1  >>> class Spam:
2  ...     def __init__(self, n): self.n = n
3  ...     def __lt__(self, other): return self.n < other.n
4  ...     def __repr__(self): return f'Spam({self.n})'
5  ...
6  >>> l = [Spam(n) for n in range(5, 0, -1)]
7  >>> l
8  [Spam(5), Spam(4), Spam(3), Spam(2), Spam(1)]
9  >>> sorted(l)
10 [Spam(1), Spam(2), Spam(3), Spam(4), Spam(5)]
```

这证实了一点:因为 Spam 类实现了支持 < 运算符的特殊方法 __lt__,所以我可以对由 Spam 对象构成的列表进行排序,

因此,示例 8.19 中的类型参数 T 应被限制为实现了 __lt__ 的类型。“示例 8.18<232页>”需要实现了 __hash__ 的类型参数,因此可将类型参数的上边界(即 TypeVar 的 bound 参数)设置为 Hashable。但是,对于目前遇到的问题是:typing 或 collections.abc 中没有合适的(即仅实现了 __lt__ 方法的)类型可用,所以需要自己创建这个类型。

示例 8.20 中通过继承 typing.Protocol 类,定义了一个 协议 (protocol) 类型:SupportsLessThan。

</> 示例 8.20: comparable.py: 定义协议类型 SupportsLessThan

```

1  from typing import Protocol, Any
2
3  class SupportsLessThan(Protocol):
4      def __lt__(self, other: Any) -> bool: ...
```

① 协议 (protocol) 类型需要定义为 typing.Protocol 的子类,即需要继承 typing.Protocol。

② 协议 (protocol) 主体中定义一个或多个方法,方法的主体为...。

若类型 T 实现了协议 P 中定义的所有方法,并且具有匹配的类型签名,则类型 T 与协议 P 相容。

¹⁷ 打开交互式控制台,像刚才那样依靠 鸭子类型 (Duck Typing) 来探索语言功能,是多么美妙的事情。当我使用不支持这种功能的语言时,我非常怀念这种探索方式。

若类型 T 实现了协议 P 中定义的所有方法,并且具有匹配的类型签名,则类型 T 与协议 P 相容。

有了协议 `SupportsLessThan`,即可将“示例 8.19<233页>”中的函数 `top` 改写为可正常使用的版本(如示例 8.21 所示)。

</> 示例 8.21: `top.py`:用上边界为 `SupportsLessThan` 的 `TypeVar` 定义函数 `top`

```
1  from collections.abc import Iterable
2  from typing import TypeVar
3
4  from comparable import SupportsLessThan
5
6  LT = TypeVar('LT', bound=SupportsLessThan)
7
8  def top(series: Iterable[LT], length: int) -> list[LT]:
9      ordered = sorted(series, reverse=True)
10     return ordered[:length]
```

下面用 `pytest` 测试一下 `top` 函数。示例 8.22 展示了测试套件的一部分。先尝试用生成 `“tuple[int, str]”` 的生成器表达式调用函数 `top`;然后,再尝试用一个 `object` 列表调用函数 `top`。在使用 `object` 列表调用 `top` 时,我们期望得到 `TypeError` 异常。

</> 示例 8.22: `top_test.py`:`top` 函数测试套件的一部分

```
1  from collections.abc import Iterator
2  from typing import TYPE_CHECKING
3
4  import pytest
5
6  from top import top
7
8  # 省略若干行
9
10 def test_top_tuples() -> None:
11     fruit = 'mango pear apple kiwi banana'.split()
12     series: Iterator[tuple[int, str]] = ( (len(s), s) for s in fruit ) ❶
13     length = 3
14     expected = [(6, 'banana'), (5, 'mango'), (5, 'apple')]
15     result = top(series, length)
16     if TYPE_CHECKING: ❷
17         reveal_type(series)
18         reveal_type(expected)
19         reveal_type(result)
20     assert result == expected
21
22     # intentional type error
23 def test_top_objects_error() -> None:
24     series = [object() for _ in range(4)]
25     if TYPE_CHECKING:
26         reveal_type(series)
27     with pytest.raises(TypeError) as excinfo:
```

```

28     top(series, 3)           ❸
29     assert "<' not supported" in str(excinfo.value)

```

- ❶ typing.TYPE_CHECKING 常量 在运行时始终为 False，但类型检查器在进行类型检查时会假定其为 True。
- ❷ 显式声明变量 series 的类型为 “Iterator[tuple[int, str]]”，以使 Mypy 的输出更易读¹⁸。
- ❸ 这个 if 语句会在运行此测试脚本时，禁止执行后面的 3 行代码。而在用 Mypy 对此脚本执行静态检查时，会执行后面的 3 行代码。
- ❹ 此脚本在运行时，不会调用函数 reveal_type()，因为它不是常规函数，而是 Mypy 的调试工具，因此无需通过 import 导入此函数。Mypy 每遇到一个 reveal_type() 伪函数调用，就会输出一个调试信息，以显示推断出的参数类型。
- ❺ 此行将被 Mypy 标记为错误。

上述测试可以顺利通过——无论 top.py 中是否含有 [类型提示 \(Type Hints\)](#)，上述测试都会顺利通过。更重要的是，若我用 Mypy 检查该测试文件，会发现 TypeVar 正在按预期工作。请参见示例 8.23 中 Mypy 命令的输出。



从 Mypy 0.910 (2021 年 7 月) 开始，在某些情况下 reveal_type 的输出并不精确显示我声明的类型，而是显示兼容类型。例如，我使用了 `abc.Iterator`，但输出可能会显示兼容的 `typing.Iterator`。请忽略此细节。Mypy 的输出仍然有用。在讨论输出时，我会假定 Mypy 的这个问题已解决。

</> 示例 8.23: mypy top_test.py 的输出 (为便于阅读，将行数进行了拆分)

```

1  …/comparable/ $ mypy top_test.py
2  top_test.py:32: note:
3  Revealed type is "typing.Iterator[Tuple[builtins.int, builtins.str]]"
4  top_test.py:33: note:
5  Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]"
6  top_test.py:34: note:
7  Revealed type is "builtins.list[Tuple[builtins.int, builtins.str]]" ❷
8  top_test.py:41: note:
9  Revealed type is "builtins.list[builtins.object*]" ❸
10 top_test.py:43: error:
11 Value of type variable "LT" of "top" cannot be "object" ❹
12 Found 1 error in 1 file (checked 1 source file)

```

1. `reveal_type(series)` 显示在 `test_top_tuples` 中，`series` 是一个 “`Iterator[tuple[int, str]]`” 类型——这是我显式声明的类型。
2. `reveal_type(result)` 确认了 `top` 调用返回的类型正是我想要的：根据为 `series` 声明的类型，`result` 的类型是 “`list[tuple[int, str]]`”。
3. `reveal_type(series)` 显示 `test_top_objects.error` 中的 `series` 是 “`list[object*]`” 类型。Mypy 会在推断出

¹⁸若不显式声明此[类型提示 \(Type Hints\)](#)，Mypy 会将参数 `series` 的类型推断为 “`Generator[Tuple[builtins.int, builtins.str*], None, None]`”。这个推断出的类型虽然冗长，但是与 “`Iterator[tuple[int, str]]`” 相容，详见 “[17.12 泛化可迭代类型 <516页>](#)”。

的类型后面加上`*`:在此测试中,我没有为`series`添加类型提示(Type Hints),因此`series`的类型是由Mypy推断出来的。

4. Mypy标记了此测试中有意触发的错误:可迭代对象`series`中的元素类型不能是`object`(必须是`SupportsLessThan`类型)。

与抽象基类(ABCs)相比,协议(protocol)类型的主要优势是,类型无需做特殊声明即可与协议(protocol)类型相容。这样就可以利用现有的类型,或在不受我们控制的代码中实现的类型,来创建协议。我们不需要用`SupportsLessThan`协议,派生或注册`str`、`tuple`、`float`、`set`等类型,即可在需要`SupportsLessThan`参数的地方使用这些类型。这些类型仅需实现`__lt__`方法即可。而且类型检查器仍然可以发挥作用,因为`SupportsLessThan`是显式定义的协议,这与鸭子类型(Duck Typing)中常见的隐式协议不同,后者对类型检查器是不可见的。

“PEP 544 –Protocols: Structural subtyping (static duck typing)”引入了特殊的Protocol类。“示例 8.21<235页>”演示了为何这个新特性被称为“静态鸭子类型(Static Duck Typing)”:函数`top`中参数`series`的类型提示(Type Hints)表达的意思是“`series`的名义类型(Nominal Typing)不重要,只需实现`__lt__`方法即可。”鸭子类型(Duck Typing)隐式地传达了这一要求,而静态类型检查器却无法理解鸭子类型(Duck Typing)的这种隐式要求。类型检查器无法读取CPython的C源代码,也无法通过执行控制台实验,来发现`sorted`函数只要求元素支持运算符`<`。

`typing.Protocol`实现的是静态鸭子类型(Static Duck Typing)¹⁹。因为通过使用Protocol类,可以描述对象必须具备的方法与属性,并利用静态类型检查器对代码进行类型检查。

关于`typing.Protocol`,值得讲的内容还有很多。我们将在“三延伸阅读”中再次讨论`typing.Protocol`,其中“十三接口、协议与抽象基类”中对比了结构类型、鸭子类型(Duck Typing)与抽象基类(ABCs)——形式化协议的另一种方法。此外,“15.2 重载的签名<418页>”解释了如何用`@typing.overload`声明重载的函数签名,其中包含一个大量使用`typing.Protocol`与有届TypeVar的示例。



使用`typing.Protocol`可以注解“8.4 类型由支持的操作来定义”中定义的`double`函数,而且不会损失功能。关键是要定义一个含有`__mul__`方法的协议类。建议您尝试一下,实现方式见“13.6.1 为函数`double`添加类型提示<376页>”。

8.5.11 Callable

`collections.abc`模块提供的`Callable`类型(对于Python<3.9的用户,可以用`typing.Callable`)可用于注解回调参数或由高阶函数返回的可调用对象。

`Callable`类型可像如下这样进行参数化:

```
1 Callable[[ParamType1, ParamType2], ReturnType]
```

参数列表`“[ParamType1, ParamType2]”`中可以有零个或多个类型。

下面以“18.3 案例分析:lis.py 中的模式匹配”实现的简单交互式解释器中的`repl`函数为例²⁰。

```
1 def repl(input_fn: Callable[[Any], str] = input) -> None:
```

¹⁹我不知道是谁发明了“静态鸭子类型”这一术语,但它随着Go语言的出现,变得更加流行。Go语言中的接口在语义上,与Python中的协议更像,与Java的名义接口不太一样。

²⁰REPL是Read-Eval-Print-Loop的简称,这是交互式解释器的基本行为。

在常规使用时, repl 函数使用 Python 的内置函数 input 来读取用户的表达式。然而, 对于自动化测试或其他输入源集成, 则函数 repl 会接受一个可选参数 input_fn——这是一个 Callable 类型, 参数类型和返回值类型都与函数 input 相同。

内置函数 input 在 Typeshed 项目 上具有以下签名:

```
1 def input(__prompt: Any = ...) -> str: ...
```

函数 input 的签名与下面 Callable 的 类型提示 (Type Hints) 相容 (Consistent-with)。

```
1 Callable[[Any], str]
```

可选参数类型或关键字参数类型, 都没有专门的类型提示 (Type Hints) 语法。正如 typing.Callable 文档所述:“此类函数类型很少用作回调类型”。若想让类型提示 (Type Hints) 可匹配具有灵活签名的函数, 可将整个参数列表替换为 ..., 如下所示:

```
1 Callable[..., ReturnType]
```

泛化类型参数与类型层次结构的交互, 引入了一个新的类型概念:型变 (variance)。

8.5.11.1 Callable 类型的型变

假设一个温控系统中定义了如 示例 8.24 所示的函数 update。该函数调用 probe 函数获取当前温度, 再调用 display 函数向用户显示当前温度。出于教学目的, probe 与 display 都会作为参数传给函数 update。此示例的目的是比较 Callable 的 2 种注解方式:一种有返回值类型, 另一种有参数类型。

</> 示例 8.24: 说明型变 (variance)

```
1 from collections.abc import Callable
2
3 def update(
4     probe: Callable[[], float], ❶
5     display: Callable[[float], None] ❷
6 ) -> None:
7     temperature = probe()
8     # 假设此处有大量控制代码
9     display(temperature)
10
11 def probe_ok() -> int: ❸
12     return 42
13
14 def display_wrong(temperature: int) -> None: ❹
15     print(hex(temperature))
16
17 update(probe_ok, display_wrong) # 类型错误 ❺
18
19 def display_ok(temperature: complex) -> None: ❻
20     print(temperature)
21
```

```
22 | update(probe_ok, display_ok) # OK
```

- ❶ 函数 `update` 将 2 个可调用参数作为参数。
- ❷ `probe` 必须是不接受参数, 并返回 `float` 值的可调用对象。
- ❸ `display` 必须是接受一个 `float` 参数, 并返回 `None` 的可调用对象。
- ❹ `probe_ok` 与 “`Callable[[], float]`” [相容 \(Consistent-with\)](#)。因为, 返回 `int` 值对预期 `float` 值的代码无任何影响。
- ❺ `display_wrong` 与 “`Callable[[float], None]`” [不相容 \(Consistent-with\)](#)。因为预期 `int` 值的函数 `display_wrong`, 可能处理不了 `float` 值。例如, Python 的 `hex` 函数可接受 `int` 值, 但拒绝 `float` 值。
- ❻ Mypy 标记此行的原因是: `display_wrong` 与 `update` 函数的 `display` 参数中的 [类型提示 \(Type Hints\)](#) 不兼容。
- ❼ `display_ok` 与 “`Callable[[float], None]`” [相容 \(Consistent-with\)](#)。因为接受 `complex` 值的函数, 可以处理 `float` 参数。
- ❽ 通过了 Mypy 检查。

综上所述, 当代码需要一个返回 `float` 值的回调时, 为其提供一个返回 `int` 值的回调也是可以的。因为在需要 `float` 值的地方, 也可以使用 `int` 值。

从形式上, 我们可以说 “`Callable[[], int]`” 是 “`Callable[[], float]`” 的子类型——就像 `int` 是 `float` 的子类型一样。这意味着, `Callable` 的返回值类型是 [协变 \(covariance\)](#) 的。因为 `int` 与 `float` 之间的子类型关系, 与使用它们作为返回类型的 `Callable` 类型的子类型关系是同方向的。

反过来, 当需要一个可处理 `float` 值的回调时, 却提供了一个可处理 `int` 值的回调, 则会导致类型错误。因为, 能处理 `int` 值的函数, 可能处理不了 `float` 值。

从形式上看, “`Callable[[int], None]`” 并不是 “`Callable[[float], None]`” 的子类型。虽然 `int` 是 `float` 的子类型, 但在参数化 `Callable` 类型中, 二者的关系却是相反的: “`Callable[[float], None]`” 是 “`Callable[[int], None]`” 的子类型。因此, 我们说 `Callable` 在声明的参数类型上是 [逆变 \(contravariance\)](#) 的。

“[15.7 型变](#)” 将进一步说明 [型变 \(variance\)](#), 并给出了一些关于 “[不变 \(invariant\) 类型](#)”、“[协变 \(covariance\) 类型](#)” 与 “[逆变 \(contravariance\) 类型](#)” 的示例。



目前, 可以放心地认为大多数的参数化泛型 (Parameterized Generic Type) 都是不变 (invariant) 的, 因此更加简单。例如, 如果我声明了 “`scores: list[float]`”, 就只能将 `list[float]` 赋值给 `scores`, 而不能将声明为 `list[int]` 或 `list[complex]` 的对象赋值给 `scores`。

- 不能接受 `list[int]` 对象的原因: `list[int]` 中无法存放 `float` 值, 但代码中需要处理 `float` 值。
- 不能接受 `list[complex]` 对象的原因: 代码可能需要对分数进行排序以找出中位数, 但 `complex` 类型未提供 `__lt__` 方法, 因此 `list[complex]` 不能排序。

现在, 我们来讨论本章要介绍的最后一种特殊类型。

8.5.12 NoReturn

此种特殊类型仅用于注解无返回值函数的返回类型。通常,这些函数的存在是为了引发异常。标准库中有许多这类函数。

例如,sys.exit() 通过引发 SystemExit,来终止 Python 进程。它在 [Typeshed 项目](#) 的 [函数签名](#) 如下:

```
1 def exit(_status: object = ...) -> NoReturn: ...
```

_status 是 [仅限位置参数 \(Positional-only Parameter\)](#),而且有默认值。存根 (stub) 文件不会列出默认值,而是用... 代替。_status 的类型是 object,因此也可能是 None。所以没必要将其注解为 Optional[object]。

“[示例 24.6<729页>](#)”在 __flag_unknown_attrs 方法中使用了 NoReturn。该方法旨在生成用户友好且全面的错误消息,然后引发 AttributeError。

下一节是关于为 [仅限位置参数 \(Positional-only Parameter\)](#) 与变长参数添加 [类型提示 \(Type Hints\)](#) 的内容。

8.6 注解仅限位置参数与变长参数

回顾一下 [示例 7.9](#) 中的 tag 函数。上次我们看到它的函数签名是在 “[7.7.1 仅限位置参数<196页>](#)” 中:

```
1 def tag(name, /, *content, class_=None, **attrs):  
2     ...
```

这是带有完整 [类型提示 \(Type Hints\)](#) 的 tag 函数,以多行形式编写——这是长函数签名的常见约定,并按照 [blue](#) 格式化工具的方式进行换行:

```
1 from typing import Optional  
2  
3 def tag(  
4     name: str,  
5     /,  
6     *content: str,  
7     class_: Optional[str] = None,  
8     **attrs: str,  
9 ) -> str:
```

注意在本例中,任意个位置参数的类型提示是 “*content: str”,这表明所有这些位置参数必须是 str 类型。在函数主体中,局部变量 content 的类型将是 “tuple[str, ...]”。

在本例中,任意个关键字参数的类型提示是 “**attrs: str”。因此,函数主体内 attrs 的类型将为 “dict[str, str]”。若类型提示是 “**attrs: float”,则函数主体内 attrs 的类型将是 “dict[str, float]”。

如果任意关键字参数 attrs 必须接受不同类型的值,则需要使用 Union[] 或 Any,类型提示为 “**attrs: Any”。

针对 [仅限位置参数 \(Positional-only Parameter\)](#) 的 / 表示法,只可在 Python ≥ 3.8 中使用。在 Python ≤ 3.7 中,这种表示法会导致语法错误。“[PEP 484 - Type Hints](#)”的约定是,在每个 [仅限位置参数 \(Positional-only Parameter\)](#) 的名称前加两个下划线 (__)。下面使用 “[PEP 484](#)” 约定的方式,为函数 tag 的签名增加类型提示(写成两行)。

```
1 from typing import Optional
2
3 def tag(__name: str, *content: str, class_: Optional[str] = None, **attrs: str) ->
4     str:
```

上述这两种 仅限位置参数 (Positional-only Parameter) 的声明方式,均可被 Mypy 识别并检查。

最后,简单介绍一下 [类型提示 \(Type Hints\)](#) 及静态类型系统的局限性。

大型企业代码库的维护者报告说,静态类型检查程序发现了许多错误,而且修复的成本比代码在生产中运行后才发现的错误更低。不过,必须指出的是,在我所知道的公司中,早在引入静态类型之前,自动化测试就已经成为标准做法并被广泛采用。

8.7 类型不完美,测试需全面

大型企业代码库的维护者反映,静态类型检查器可发现许多 bug。而且此阶段的修复成本,比代码上线运行后才发现 bug 的修复成本更低。然而,有必要指出,早在引入静态类型之前,自动化测试就已被纳入行业标准,并且已被许多公司广泛采用。

尽管静态类型的优势诸多,但也不保证绝对正确。静态类型很难发现以下问题。

- **误报**

代码中正确的类型,被类型检查工具误报为类型错误。

- **漏报**

代码中不正确的类型,未被类型检查工具发现。

此外,若对所有代码都做类型检查,我们将会失去 Python 的一些表现力。

- 有些方便的功能,无法进行类型检查。例如,像 config(**settings) 这样的参数解包。
- 通常,类型检查工具对特性 (property)、描述符、元类和元编程等高级功能的支持较差,或者根本无法理解。
- 类型检查工具会滞后于 Python 版本的发布 (有时滞后不止一年),有些新的语言特性会被类型检查工具拒绝。

[类型提示 \(Type Hints\)](#) 无法表达常见的数据约束——即使是简单的约束。例如,[类型提示 \(Type Hints\)](#) 无法确保 “quantity 必须是大于 0 的整数” 或 “label 必须是包含 6~12 个 ASCII 字母的字符串”。通常,[类型提示 \(Type Hints\)](#) 无法捕获业务逻辑中存在的错误。

鉴于此,[类型提示 \(Type Hints\)](#) 不能作为软件质量的保障支柱,而且盲目使用 [类型提示 \(Type Hints\)](#) 也会适得其反。

建议将类型检查工具纳入现代 CI 流水线,与测试运行程序、linter 程序等结合在一起使用。CI 流水线的目的是减少软件故障,而自动化测试可以捕获许多超出 [类型提示 \(Type Hints\)](#) 能力范围的 bug。任何用 Python 编写的代码都可以用 Python 进行测试,无论有没有 [类型提示 \(Type Hints\)](#)。



本节的标题与结论的灵感均来自 Bruce Eckel 的文章 “Strong Typing vs. Strong Testing”。该文章被收录在由 Joel Spolsky 所著的《THE BEST SOFTWARE WRITING I》一书中。Bruce 是 Python 的粉丝, 也是 C++、Java、Scala 与 Kotlin 相关书籍的作者。在那篇文章中, 他讲述了自己在学习 Python 之前, 一直是静态类型的拥护者, 并得出结论: “如果一个 Python 程序有足够的单元测试, 那么它就能像 C++、Java 或 C# 程序一样健壮(不过 Python 测试编写起来更快)。”

至此, 我们对 Python 类型提示的介绍就暂时告一段落。“十五 类型注解进阶 1136” 将会继续重点介绍这一话题, 其中涵盖泛化类 (Generic Classes)、型变 (variance)、签名重载、类型转换等。同时, 本书的多个示例中都会出现 **类型提示 (Type Hints)** 的身影。

8.8 本章小结

本章首先简要介绍了 **渐进式类型系统 (Gradual Type System)** 的概念, 然后开始实践。不借助可读取 **类型提示 (Type Hints)** 的工具, 很难理解 **渐进式类型系统 (Gradual Type System)**。因此, 我们在 Mypy 的指引下, 逐步为一个函数添加了 **类型提示 (Type Hints)**。

接着, 我们又回到 **渐进式类型系统 (Gradual Type System)** 的概念上, 指出这其实是一种混合概念, 混合了 Python 传统的 **鸭子类型 (Duck Typing)** 与 Java、C++ 等静态类型语言的 **名义类型 (Nominal Typing)**。

本章用大量的篇幅分门别类地介绍了在 **类型提示 (Type Hints)** 中可使用的主要类型。本章涉及的许多类型都与我们熟悉的 Python 对象类型 (如 set、tuple 及可调用对象) 有关。这些类型也被扩展为支持泛型表示法 (如 Sequence[float])。在 Python 3.9 中将标准类型修改为支持 **泛型 (Generic Type)** 之前, 这些类型中的许多都是在 typing 模块中临时实现的。

有些类型是特殊的实体。Any、Optional、Union、NoReturn 与内存中的实际对象无关, 它们仅存在于类型系统的抽象域中。

我们研究了参数化泛型与类型变量。它们在不牺牲类型安全性的情况下, 为 **类型提示 (Type Hints)** 带来了更大的灵活性。

通过使用 Protocol, **参数化泛型 (Parameterized Generic Type)** 变得更具表现力。由于 Protocol 在 Python 3.8 中才出现, 所以它还未被广泛使用, 但它却非常重要。Protocol 实现了 **静态鸭子类型 (Static Duck Typing)**: 它是 **鸭子类型 (Duck Typing)** 与 **名义类型 (Nominal Typing)** 之间的重要桥梁, 令静态类型检查工具可捕获更多的 bug。

在介绍其中一些类型时, 我们用 Mypy 进行了实验, 借助 Mypy 神奇的 reveal_type() 函数查看类型检查错误和推断出的类型。

最后介绍了如何为 **仅限位置参数 (Positional-only Parameter)** 与可变参数添加 **类型提示 (Type Hints)**。

类型提示 (Type Hints) 是一个复杂且不断变化的话题。幸运的是, 它是一个可选特性。因此, Python 广泛的用户群体都不受影响。不要听信 **类型提示 (Type Hints)** 布道者的言论, 认为所有 Python 代码都需要 **类型提示 (Type Hints)**。

我们的 BDFL²¹ 名誉主席推动了在 Python 中使用 [类型提示 \(Type Hints\)](#) 的进程,因此本章的开头与结尾都引用了他的话:

我不喜欢道德绑架用户必须为代码一直添加 [类型提示 \(Type Hints\)](#) 的 Python 版本。类型提示确实有它们的用处,但很多时候也会得不偿失。用与不用,都由您自己选择,这是多么美妙的事情! ^a

——Guido van Rossum

^a摘自 YouTube 视频 “[Type Hints by Guido van Rossum \(March 2015\)](#)”。引用开始于 “13'40””。为了清晰起见,我做了一些简单的剪辑。

8.9 延伸阅读

Bernát Gábor 在他的精彩文章 “[The state of type hints in Python](#)” 中写道:

需要编写单元测试的地方,都应该添加 [类型提示 \(Type Hints\)](#)。

我非常喜欢测试,但也经常进行探索性编码。当我进行探索性编码时,测试和类型提示对我毫无帮助。它们只会拖后腿。

Gábor 的文章是我找到的关于 Python 类型提示的最佳介绍之一,此外还有 Geir Arne Hjelle 的 “[Python Type Checking \(Guide\)](#)”。Claudio Jolowicz 的 “[Hypermodern Python Chapter 4: Typing](#)” 是一个简短的介绍,还涵盖了运行时类型检查验证。

要深入了解静态类型检查, [Mypy 文档](#) 是最好的资料。无论您使用哪种类型检查器,它都很有参考价值,因为 [Mypy 文档](#) 不仅解读 Mypy 工具自身,还有关于 Python 类型的一般教程和参考页面。[Mypy 文档](#) 还提供了一分便利的速查表,并对常见问题给出了解决方案。

[typing 模块文档](#) 是一个很好的快速参考,但并不详细。“[PEP 483—The Theory of Type Hints](#)” 包含关于 [型变 \(variance\)](#) 的深入解释,使用 Callable 来说明了 [逆变 \(contravariance\)](#)。最终的参考资料是与类型相关的 20 多篇 PEP 文档。PEP 文档的目标受众是 Python 核心开发人员和 Python 指导委员会。如果没有大量的预备知识,则读起来会很吃力。

如前所述,“[十五 类型注解进阶 1136](#)”将继续探讨类型话题,“[15.10 延伸阅读](#)”还会提供更多的参考资料,其中 [表 15.1](#) 列出了截至 2021 年底已批准或正在讨论的与类型有关的 PEP。

“[Awesome Python Typing](#)” 仓库收集了相关工具和资料的链接,具有一定的参考价值。

杂谈

行动起来

²¹“Benevolent Dictator For Life” 请参阅 Guido van Rossum 关于 “[Origin of BDFL](#)” 的文章。

忘掉那些超轻、不舒服的单车,忘掉华丽的运动衫,忘掉夹在小踏板上的笨重鞋子,忘掉无尽里程的磨砺。像小时候那样骑行——跨上单车,感受骑行的真正乐趣。

——Grant Petersen

《骑行手册:放在口袋里的骑行实用指南》(Workman 出版社)

若编程不是你的本职工作,而只是你工作中的协助工具;或者只是你用来学习、捣鼓小项目,又或者只是个人兴趣,那么你可能不需要 [类型提示\(Type Hints\)](#)。就像大多数骑行的人,并不需要专业的骑行装备一样。只需编程即可。

类型的认知效应

我担心 [类型提示\(Type Hints\)](#) 会影响 Python 的编码风格。

我认同大多数 API 用户都能从 Python 的 [类型提示\(Type Hints\)](#) 中受益。但是,Python 最吸引我的原因之一是,它提供了非常强大的函数,完全可以取代整个 API。并且我们也可以自己编写同样强大的函数。以内置的函数 `max()` 为例,它既功能强大,又易于理解。然而,若要在代码中正确地对 `max()` 函数进行 [类型提示\(Type Hints\)](#),需要编写 14 行代码,并且还需要使用 `typing.Protocol` 与一些 `TypeVar` 定义来支持这些 [类型提示\(Type Hints\)](#)。

我担心的是,倘若代码库对 [类型提示\(Type Hints\)](#) 的要求过于严格,导致编写强大函数变得复杂繁琐,可能会使程序员们不再考虑编写这样的函数。

根据维基百科,“[Linguistic relativity \(语言相对论\)](#)”,又称“萨丕尔-沃尔夫假说 (Sapir-Whorf hypothesis)”,是一个“声称语言结构会影响语言使用者的世界观或认知的理论”。维基百科进一步解释道:

- 强势版本认为,语言决定思维,语言类别限制并决定认知类别。
- 弱势版本认为,语言类别和用法只会影响思维和决策。

大部分学者认为语言并不完全决定思维,但语言的结构和使用方式确实会对思维产生一定的影响。

我不知道是否有针对编程语言方面的相关研究,但根据我的经验,编程语言对我处理问题的方式有很大影响。我从事编程工作使用的第一门编程语言是 8 位计算机时代的 Applesoft BASIC。当时的 BASIC 并不直接支持递归,必须自己手动实现递归的调用栈。因此,我从未考虑过使用递归算法或递归数据结构。我知道这些东西的存在,但我绝不会用它们来解决问题。

几十年后,当我接触到 Elixir 时,我开始喜欢用递归来解决问题,并逐渐过度依赖它。直到我发现,如果使用 Elixir `Enum` 和 `Stream` 模块中的现有函数,我的许多解决方案会变得更加简单。我了解到,地道的 Elixir 应用级代码很少显式使用递归调用,而是使用底层已实现了递归的 Elixir `Enum` 和 `Stream`。

“[Linguistic relativity \(语言相对论\)](#)”可以解释一个普遍的观点(也是未经证实的):学习不同的编程语言(尤其是支持不同编程范式的语言),可以让你成为更好的程序员。有了使用 Elixir 的经验,让我在编写 Python 或 Go 代码时更有可能使用函数式编程模式。

现在,来看具体的例子。

如果 Kenneth Reitz 决定对 `requests` 包的所有函数都进行注解,那么 `requests` 包的 API 势必会截然不同。他的初衷是编写出易用、灵活且功能强大的 API。事实证明, `requests` 包非常受欢迎,他成功了。

在 2020 年 5 月, requests 包在 PyPI Stats 中排名第 4, 每天下载量达 260 万次。排名第 1 的是 urllib3, 它是 requests 的一个依赖项。

2017 年, requests 包的维护人员决定不再投入时间编写 [类型提示 \(Type Hints\)](#)。其中, 一位维护人员 Cory Benfield 在 E-Mail 中写道:

我觉得, 符合 Python 风格的 API, 最不需要采用这种类型系统, 因为其得到的回报最少。

在该邮件中, Benfield 给出了一个极端的例子: 如果为 `requests.request()` 的 `files` 关键字参数添加 [类型提示 \(Type Hints\)](#), 将是下面这样:

```

1  Optional[
2      Union[
3          Mapping[
4              basestring,
5              Union[
6                  Tuple[basestring, Optional[Union[basestring, file]]],
7                  Tuple[basestring, Optional[Union[basestring, file]]],
8                  Optional[basestring]],
9                  Tuple[basestring, Optional[Union[basestring, file]]],
10                 Optional[basestring], Optional[Headers]]]
11             ]
12         ],
13     Iterable[
14         Tuple[
15             basestring,
16             Union[
17                 Tuple[basestring, Optional[Union[basestring, file]]],
18                 Tuple[basestring, Optional[Union[basestring, file]]],
19                 Optional[basestring]],
20                 Tuple[basestring, Optional[Union[basestring, file]]],
21                 Optional[basestring], Optional[Headers]]]
22             ]
23         ]
24     ]
25 ]
26 ]
```

而且, 还要先定义以下类型:

```

1  Headers = Union[
2      Mapping[basestring, basestring],
3      Iterable[Tuple[basestring, basestring]],
4  ]
```

如果 requests 包的维护者, 坚持 100% 的 [类型提示 \(Type Hints\)](#) 覆盖率, 你认为 requests 包还会像现在这么受欢迎么? 另一个太注重使用 [类型提示 \(Type Hints\)](#) 的重要软件包是 SQLAlchemy。

这些软件包之所以优秀, 是因为它们坚守了 Python 的动态特性。

虽然 [类型提示 \(Type Hints\)](#) 会带来不少好处, 但是也要付出代价。

首先, 需要投入大量精力来了解类型系统的工作原理, 这是一次性成本。除此之外, 还有永无终止的经常性成本。

如果坚持对所有内容都进行类型检查, 则势必会失去 Python 的一些表现力。诸如参数拆包 (例如 `config(**settings)`) 这种好用的功能, 就超出了类型检查器的能力范围。

如果要对 `config(**settings)` 这样的调用进行类型检查, 就必须将每个参数拆解出来。这让我想起了 35 年前写的 Turbo Pascal 代码。

对于使用元编程的库来说, 很难为其添加 [类型提示 \(Type Hints\)](#), 有时也无法添加。当然, 元编程可能会被滥用, 不过很多 Python 包这么好用, 还是得益于元编程。

在大公司中, 如果自上而下无一例外地强制使用 [类型提示 \(Type Hints\)](#), 我敢打赌, 我们很快就会看到有人使用代码生成工具, 来自动化生成 [类型提示 \(Type Hints\)](#), 从而减少编写 [类型提示 \(Type Hints\)](#) 的工作量和重复性代码的数量——这是动态程度较低的语言的常见做法。

对于某些项目和上下文, 类型提示根本没有意义。即便能起到一定作用, 作用也不大。因此, 关于使用 [类型提示 \(Type Hints\)](#) 的任何合理政策都应该包含例外情况。

开创了面向对象编程的图灵奖获得者 Alan Kay 曾经说过:

有些人对类型系统非常虔诚, 作为一名数学家, 我也很喜欢类型系统的想法。但是从来没有人创造出一个范围合理的类型系统。

感谢 Guido 将 [类型提示 \(Type Hints\)](#) 定为可选功能。可以让我们根据实际需求, 有选择地使用 [类型提示 \(Type Hints\)](#)。严格遵守 Java 1.5 的那种编程风格, 为所有内容都提供 [类型提示 \(Type Hints\)](#) 是不可取的。

鸭子类型的优势显著

[鸭子类型 \(Duck Typing\)](#) 适合我的思维方式。[静态鸭子类型 \(Static Duck Typing\)](#) 是一个很好的折衷方案, 既能进行静态类型检查, 又不会失去很多灵活性。而一些 [名义类型 \(Nominal Typing\)](#) 系统可能只能通过引入大量复杂性, 来提供相同的灵活性。

在 “[PEP 544](#)” 之前, [类型提示 \(Type Hints\)](#) 的整个概念似乎与 Pythonic 格格不入。我很高兴看到 `Typing.Protocol` 在 Python 中落地, 因为它为 Python 带来了平衡, 使得静态类型检查能够与 Python 的灵活性相结合。

泛化还是特化

从 Python 的角度来看, 在 [类型提示 \(Type Hints\)](#) 上下文中使用 “泛化(generic)” 这个术语, 似乎有些落后了。“泛化(generic)” 一般包含两个意思: “通用的”、“无商标的”。

以 `list` 与 `list[str]` 为例, 前者属于 “泛化(generic)”, 可接受任何类型的对象; 后者属于 “特化(specific)”, 仅接受 `str`。

然而, 在 Java 中, “泛化(generic)” 这个术语是有意义的。在 Java 1.5 之前, 所有 Java 容器(除了神奇的数组)都是 “特化(specific)” 的: 它们仅能持有 `Object` 引用。从容器中取出的项, 都要进行强制转换才能使用。而从 Java 1.5 开始, 容器可接受类型参数, 从而变成了 “泛化(generic)”。

装饰器与闭包

有很多人抱怨,将这个功能命名为“装饰器”不合适。主要原因是,这个名称与《设计模式》一书中
的用法不一致。“装饰器”这个名称可能更适合在编译器领域使用——因为它会遍历与注解语法树。

——“PEP 318 –Decorators for Functions and Methods”

函数装饰器让我们可以在源代码中“标记”函数,以某种方式增强函数的行为,这是一个强大的功能。若
想掌握装饰器的功能,则需要先理解闭包——当函数捕获在函数主体之外定义的变量时,我们就会得到 [闭包](#)
([closures](#))。

Python 中最晦涩难懂的保留关键字是在 Python 3.0 中引入的 `nonlocal`。如果您严格遵守以类为中心的
面向对象原则,那么作为一名 Python 程序员,那么即便不知道这个关键字的存在也不会受影响。但是,如果想
实现自己的函数装饰器,就必须了解 [闭包](#) ([closures](#)) 的方方面面,也包括关键字 `nonlocal`。

[闭包](#) ([closures](#)) 除了可用在装饰器中,它还是回调式编程风格与函数式编程风格的重要基础。

本章的最终目标是准确解释函数装饰器的工作原理,从最简单的注册装饰器开始,一直到较为复杂的参
数化装饰器。然而,在实现这一目标之前,我们需要先介绍以下内容:

- Python 如何求解装饰器语法。
- Python 如何判断一个变量是否是局部变量。
- 为什么存在闭包,以及闭包如何工作。
- `nonlocal` 解决了什么问题。

有了上述这些基础,我们就可以处理更多的装饰器主题了。如下所示:

- 实现一个行为良好的装饰器。
- 标准库中强大的装饰器:`@cache`、`@lru_cache` 和 `@singledispatch`。
- 实现一个参数化装饰器。

9.1 本章新增内容

Python 3.9 中新增的缓存装饰器 `functools.cache` 比传统的 `functools.lru_cache` 更简单, 所以本章先介绍 `functools.cache`。`functools.lru_cache` 与 Python 3.8 新增的简化版, 将在“[9.9.2 使用 `lru_cache`](#)”中介绍。

对“[9.9.3 单分派泛化函数](#)”的内容做了扩充, 增加了 [类型提示 \(Type Hints\)](#), 这是自 Python 3.7 以来, 使用 `functools.singledispatch` 的首选方式。

“[9.10 参数化装饰器](#)”增加了一个基于类的示例, 即“[示例 9.27 <272页>](#)”。

将第 1 版的第 6 章移到了第二部分末尾(即第 10 章), 以改善本书的流畅性。第 1 版中的 7.3 节也移到了“[十 用一等函数实现设计模式](#)”, 与“使用可调用对象实现的其他策略设计模式”放在一起。

下面, 先简要介绍装饰器的基础知识。然后, 再讨论本章开篇列出的其余主题。

9.2 装饰器基础知识

装饰器是一个可调用对象, 它接受另一个函数(被装饰的函数)作为参数。

装饰器可能会对被装饰的函数进行一些处理。然后, 将其返回, 或将其替换为另一个函数或可调用对象¹。

换句话说, 假设已有一个名为 `decorate` 的装饰器。那么下述两种写法将具有一样的效果。

```

1 @decorate
2 def target():
3     print('running target()')

```

```

1 def target():
2     print('running target()')
3
4 target = decorate(target)

```

在这两个代码片段执行完毕后, `target` 都被绑定到了 `decorate(target)` 返回的函数上——这个返回的函数可能是最初命名为 `target` 的函数, 也可能是另一个函数。

为了确认被装饰的函数已被替换, 请看示例 9.1 中的控制台会话。

</> [示例 9.1: 装饰器通常将传入的参数, 替换为另一个函数](#)

```

1 >>> def deco(func):
2 ...     def inner():
3 ...         print('running inner()')
4 ...     return inner ❶
5 ...
6 >>> @deco
7 ... def target():
8 ...     print('running target()')
9 ...
10 >>> target() ❸
11 running inner()
12 >>> target ❹
13 <function deco.<locals>.inner at 0x10063b598>

```

¹若将这句话中的“函数”替换为“类”, 差不多就是类装饰器的作用。类装饰器将在“[二十四 类元编程 \(Class Metaprogramming\)](#)”中介绍。

- ❶ deco 返回其内部函数对象 inner。
- ❷ 函数 target 被 deco 装饰。
- ❸ 调用被装饰的函数 target, 实际运行的是函数 inner。
- ❹ 通过检查, 发现 target 现在是 inner 的引用。

严格来讲, 装饰器只是一种 [语法糖 \(Syntactic Sugar\)](#)。正如刚才所见, 你可以像调用普通的可调用对象一样调用装饰器, 将另一个函数作为参数传递给装饰器。有时, 这种方式还更加方便, 特别是在进行元编程 (即在运行时改变程序的行为) 时,

综上所述, 装饰器包含 3 个基本性质:

- 装饰器是一个函数或一个可调用对象。
- 装饰器可以用另一个不同的函数替换被装饰的函数。
- 在模块加载时, 会立即执行装饰器。

下面重点讲解第三点: 在模块加载时, 会立即执行装饰器。

9.3 Python 何时执行装饰器

装饰器的一个主要特点是, 它们会在被装饰函数定义后立即运行。这通常是在导入时 (即 Python 加载模块时)。请看 [示例 9.2](#) 中的 registration.py。

```
</>示例 9.2: registration.py 模块
1 registry = []
2
3 def register(func):
4     print(f'running register({func})')
5     registry.append(func)
6     return func
7
8 @register
9 def f1():
10     print('running f1()')
11
12 @register
13 def f2():
14     print('running f2()')
15
16 def f3():
17     print('running f3()')
18
19 def main():
20     print('running main()')
21     print('registry ->', registry)
22     f1()
23     f2()
24     f3()
25
```

```

26 if __name__ == '__main__':
27     main() ❹

```

- ❶ registry 列表将保存被 @register 装饰后的函数的引用。
- ❷ register 的参数是一个函数。
- ❸ 为了演示,显示正在被装饰的函数。
- ❹ 将 func 存入 registry 列表。
- ❺ 返回 func; 必须返回一个函数,这里返回的函数与通过参数传入的函数一样。
- ❻ f1 与 f2 被 @register 装饰。
- ❼ f3 未被装饰。
- ❽ main 首先显示 registry,然后调用 f1()、f2()、f3()。
- ❾ 仅当 registration.py 被作为脚本运行时,才会调用 main()。

以脚本形式运行 registration.py 的输出结果如下:

```

1 $ python3 registration.py
2 running register(<function f1 at 0x100631bf8>)
3 running register(<function f2 at 0x100631c80>)
4 running main()
5 registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
6 running f1()
7 running f2()
8 running f3()

```

请注意,register 在模块中的任何其他函数之前运行(两次)。在调用 register 时,它会接收被装饰的函数对象作为参数,例如 <function f1 at 0x100631bf8>。

模块加载后,register 列表会保存两个被装饰函数的引用:f1 和 f2。这些函数以及 f3 仅在由 main 显式调用时才会被执行。

如果导入 registration.py(而不是作为脚本运行),输出结果如下:

```

1 >>> import registration
2 running register(<function f1 at 0x10063b1e0>)
3 running register(<function f2 at 0x10063b268>)

```

此时,若要检查 register 的值,则会得到如下内容:

```

1 >>> registration.registry
2 [<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]

```

示例 9.2 的主要目的是强调:函数装饰器在导入模块时,会立即执行。而被装饰的函数,仅在显式调用时,才会执行。由此,可看出 Python 程序员所说的导入时与运行时之间有什么区别。

9.4 注册装饰器

考虑到装饰器在实际代码中的常见用法,示例 9.2 有两处不同寻常的地方:

- 装饰器函数与被装饰函数,被定义在同一模块中。在实践中,装饰器通常定义在单独的模块中,并应用

于其他模块中的函数。

- register 装饰器返回的函数与通过参数传入的函数相同。在实践中,大多数装饰器都会定义一个内部函数,并返回这个内部函数。

尽管 [示例 9.2](#) 中的 register 装饰器原封不动地返回了被装饰函数,但是这种技术并非毫无用处。许多 Python 框架中都使用类似的装饰器,将函数添加到中央注册中心。例如,将 URL 模式映射到生成 HTTP 响应的函数的注册中心。这种注册装饰器可能会,也可能不会更改被装饰的函数。

我们将在“[10.3 用装饰器改进策略模式](#)”中看到这种注册装饰器的应用(“[示例 10.9<287页>](#)”)。

但是,大多数装饰器都会更改被装饰的函数。通常的做法是,在装饰器内部定义一个函数,用此内部函数替换被装饰的函数,并返回这个内部函数。使用内部函数的代码几乎总是依赖闭包,才能继续运行。要理解闭包,我们需要先回顾一下 Python 中的变量作用域规则。

9.5 变量作用域规则

在 [示例 9.3](#) 中,定义并测试了一个函数。该函数会读取两个变量:一个是定义为函数参数的局部变量 a;一个是未在函数中定义的变量 b。

</> [示例 9.3: 读取局部变量和全局变量的函数](#)

```

1  >>> def f1(a):
2      ...     print(a)
3      ...     print(b)
4      ...
5  >>> f1(3)
6  3
7  Traceback (most recent call last):
8      File "<stdin>", line 1, in <module>
9      File "<stdin>", line 3, in f1
10     NameError: global name 'b' is not defined

```

出现这样的错误并不奇怪。继续 [示例 9.3](#),如果我们为全局变量 b 赋值,然后调用 f1,f1 就可以正常工作:

```

1  >>> b = 6
2  >>> f1(3)
3  3
4  6

```

现在,让我们看一个可能会让您感到惊讶的示例。

看一下 [示例 9.4](#) 中的函数 f2。它的前 2 行与 [示例 9.3](#) 中的函数 f1 相同,然后为变量 b 赋值。但赋值前的“print(b)”语句失败了。

</> [示例 9.4: b 是局部变量,因为在函数内为其赋值了](#)

```

1  >>> b = 6
2  >>> def f2(a):
3      ...     print(a)
4      ...     print(b)
5      ...     b = 9

```

```

6 ...
7 >>> f2(3)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10  File "<stdin>", line 3, in f2
11 UnboundLocalError: local variable 'b' referenced before assignment

```

注意,首先输出的是3,这表明print(a)语句已执行成功。但是,print(b)语句却从未执行。当我第一次看到这个结果时,我很惊讶,认为应该打印6,因为有一个全局变量b,并且对局部变量b的赋值是在print(b)之后进行的。

但事实是,当Python编译函数体时,它认为b是一个局部变量,因为它是在函数内赋值的。生成的字节码反映了这一判断,并尝试从本地作用域获取b。随后,当调用f2(3)时,f2的主体获取并打印了局部变量a的值,但当试图获取局部变量b的值时,它发现b是未绑定的(unbound)。

这不是bug,而是一种设计选择:Python不要求声明变量,但会假定在函数主体中分配的变量是局部变量。这比JavaScript的行为要好得多,JavaScript也不要求声明变量,但如果忘记声明变量是局部变量(使用var),则可能会在不知情的情况下破坏一个全局变量。

在函数中赋值时,若希望解释器将b视为全局变量,并在函数中为其分配新值,就需要用global关键字来声明变量b:

```

1 >>> b = 6
2 >>> def f3(a):
3 ...     global b
4 ...     print(a)
5 ...     print(b)
6 ...     b = 9
7 ...
8 >>> f3(3)
9 3
10 6
11 >>> b
12 9

```

在前面的示例中,我们可以发现两个作用域:

- 模块全局作用域

模块全局作用域(Module Global Scope)是由在类或函数块之外分配了值的变量名称所组成的。

- f3的函数局部作用域

函数局部作用域(Function Local Scope)是由以形参(Parameter)方式分配了值的变量名称,或直接在函数体中分配了值的变量名称所组成的。

变量还可以来自另一个作用域,我们称之为“非局部作用域(nonlocal)”,它是闭包的基础,我们稍后详述。

在仔细了解了Python中变量作用域的工作原理之后,可以在下一节(“9.6 闭包”)中讨论闭包(closures)。如果对示例9.3与示例9.4中函数字节码差异感到好奇,请参阅下面的附注栏:

比较字节码

dis 模块为反汇编 Python 函数字节码提供了一种简单方法。示例 9.5 与示例 9.6 分别是示例 9.3 中 f1 与示例 9.4 中 f2 的字节码。

</> 示例 9.5: 反汇编示例 9.3 中的 f1 函数

```

1  >>> from dis import dis
2  >>> dis(f1)
3  2      0 LOAD_GLOBAL      0 (print)    ❶
4      3 LOAD_FAST         0 (a)        ❷
5      6 CALL_FUNCTION    1 (1 positional, 0 keyword pair)
6      9 POP_TOP
7
8  3      10 LOAD_GLOBAL     0 (print)
9      13 LOAD_GLOBAL     1 (b)        ❸
10     16 CALL_FUNCTION   1 (1 positional, 0 keyword pair)
11     19 POP_TOP
12     20 LOAD_CONST      0 (None)
13     23 RETURN_VALUE

```

❶ 加载全局名称 print。

❷ 加载局部名称 a。

❸ 加载全局名称 b。

</> 示例 9.6: 反汇编示例 9.4 中的 f2 函数

```

1  >>> dis(f2)
2  2      0 LOAD_GLOBAL      0 (print)
3      3 LOAD_FAST         0 (a)
4      6 CALL_FUNCTION    1 (1 positional, 0 keyword pair)
5      9 POP_TOP
6
7  3      10 LOAD_GLOBAL     0 (print)
8      13 LOAD_FAST         1 (b)        ❶
9      16 CALL_FUNCTION    1 (1 positional, 0 keyword pair)
10     19 POP_TOP
11  4      20 LOAD_CONST      1 (9)
12     23 STORE_FAST        1 (b)
13     26 LOAD_CONST      0 (None)
14     29 RETURN_VALUE

```

❶ 加载局部名称 b。这表明, 虽然在后面才为 b 赋值, 但编译器会将 b 视为局部变量。这是因为变量的性质(是不是局部变量)在函数体内不能改变。

运行字节码的 CPython 虚拟机(VM)是一个堆栈(stack)机器, 因此 LOAD 与 POP 操作都是指向堆栈(stack)的。深入描述 Python 操作码超出了本书的范围, 但它们与 dis 模块一起被记录在“[dis 模块文档](#)”中。

9.6 闭包

在博客圈中,人们有时会将闭包与匿名函数相混淆,这是有历史原因的。在匿名函数出现之前,在函数内部定义函数(即嵌套函数)并不常见,也不方便。而闭包只有在嵌套函数中才会出现。因此,很多人会同时学习这两个概念。

实际上,闭包(closures)是一个具有扩展作用域的函数(称它为f),它包含了f主体中引用的非全局变量和局部变量。这些变量必须来自包含f的外部函数的局部作用域。

函数是否匿名并不重要,重要的是它可以访问在其主体之外定义的非全局变量。

这是一个难以掌握的概念,通过示例可以更好地理解。

考虑使用avg函数来计算不断增长的一系列数值的平均值,例如某种商品在整个历史上的平均收盘价。每天都会增加一个新价格,平均值的计算会考虑到迄今为止的所有价格。

先来看avg函数的用法:

```
1 >>> avg(10)
2 10.0
3 >>> avg(11)
4 10.5
5 >>> avg(12)
6 11.0
```

avg从何而来,它又在哪里保存以前值的历史记录呢?

对于初学者来说,可能会像示例9.7那样,使用一种基于类的实现。

</>示例9.7: average_oo.py:一个计算累计平均值的类

```
1 class Averager():
2
3     def __init__(self):
4         self.series = []
5
6     def __call__(self, new_value):
7         self.series.append(new_value)
8         total = sum(self.series)
9         return total / len(self.series)
```

Averager类的实例是可调用对象。

```
1 >>> avg = Averager()
2 >>> avg(10)
3 10.0
4 >>> avg(11)
5 10.5
6 >>> avg(12)
7 11.0
```

示例9.8是使用了高阶函数make_averager的函数式实现。

</>示例9.8: average.py:一个计算累计平均值的高阶函数

```
1 def make_averager():
2     series = []
3
4     def averager(new_value):
5         series.append(new_value)
6         total = sum(series)
7         return total / len(series)
8
9     return averager
```

调用 make_averager 时,会返回一个 averager 函数对象。每次调用 averager 时,它都会将传入的参数追加到 series 中,并计算当前的平均值(如示例 9.9 所示)。

</> 示例 9.9: 测试示例 9.8

```
1 >>> avg = make_averager()
2 >>> avg(10)
3 10.0
4 >>> avg(11)
5 10.5
6 >>> avg(15)
7 12.0
```

请注意这两个示例的相似之处:调用 Averaver() 或 make_averager() 来获取可调用对象 avg,该对象将更新历史 series 并计算当前平均值。在示例 9.7 中,avg 是 Averaver 的实例。而在示例 9.8 中,avg 是内部函数 averager。无论采用哪种方法,只需调用 avg(n) 就能将 n 纳入 series 中,并得到更新后的平均值。

作为 Averaver 类的实例,avg 在哪里存储历史值很明显——实例属性 self.series。但是,第二个示例中的 avg 函数在哪里寻找 series 呢?

请注意,series 是 make_averager 的局部变量,因为赋值 series = [] 是在 make_averager 的函数体中进行的。但是当调用 avg(10) 时,make_averager 已经返回,并且它的本地作用域早已消失。

在 averager 中,series 是一个 **自由变量 (Free Variable)**。自由变量 (Free Variable) 是一个技术术语,指未在局部作用域中绑定的变量。如图 9.1 所示。

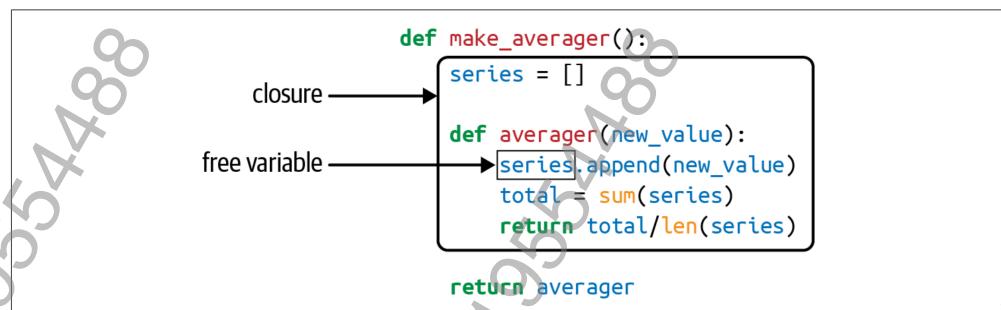


图 9.1: averager 函数的闭包扩展到了该函数自身的作用域之外,包含自由变量 series 的绑定。

查看返回的 averager 对象,可以看到 Python 如何将局部变量和自由变量的名称保存在 `__code__` 属性(表示编译后的函数主体)中,如示例 9.10 所示。

</> 示例 9.10: 检查由 [示例 9.8](#) 中 make_averager 创建的函数

```

1  >>> avg.__code__.co_varnames
2  ('new_value', 'total')
3  >>> avg.__code__.co_freevars
4  ('series',)

```

series 的值保存在返回的 avg 函数的 `__closure__` 属性中。`avg.__closure__` 中的每个项都对应 `avg.__code__.co_freevars` 中的一个名称。这些项都是 `cell` 对象, 它们有一个名为 `cell_contents` 的属性, 该属性中保存了真正的值。[示例 9.11](#) 显示了这些属性。

</> 示例 9.11: 继续 [示例 9.9](#)

```

1  >>> avg.__code__.co_freevars
2  ('series',)
3  >>> avg.__closure__
4  (<cell at 0x107a44f78: list object at 0x107a91a48>,)
5  >>> avg.__closure__[0].cell_contents
6  [10, 11, 12]

```

综上所述, [闭包 \(closures\)](#) 是一种函数, 它保留了定义函数时存在的自由变量的绑定。以便在调用函数时, 虽然定义的作用域不再可用, 仍然可以使用这些绑定。

请注意, 只有嵌套在其他函数中的函数才可能需要处理不在全局作用域中的外部变量。这些外部变量是位于外层函数的局部作用域内。

9.7 nonlocal 声明

之前实现的 `make_averager` 效率并不高。在 [示例 9.8](#) 中, 我们将所有历史值都存储在 `series` 中, 并在每次调用 `averager` 时都计算它们的总和。更高效的实现方法是只存储迄今为止的总数与项数, 然后根据这两个数字计算平均值。

[示例 9.12](#) 是一个只为说明问题的、有缺陷的函数实现。您能看出缺陷在哪里么?

</> 示例 9.12: 一个有缺陷的计算累计平均值的高阶函数, 不保存所有历史值

```

1  def make_averager():
2      count = 0
3      total = 0
4
5      def averager(new_value):
6          count += 1
7          total += new_value
8          return total / count
9
10     return averager

```

如果尝试执行 [示例 9.12](#), 将会得到如下输出:

```

1  >>> avg = make_averager()

```

```
2  >>> avg(10)
3  Traceback (most recent call last): ...
4  UnboundLocalError: local variable 'count' referenced before assignment
5  >>>
```

缺陷在于,当 count 是一个数字或不可变类型时,语句 `count+=1` 实际上与 `count = count + 1` 的含义相同。因此,我们实际上是在 `averager` 的主体中给 `count` 赋值。这就使 `count` 成为了一个局部变量。同样的问题,也会影响到变量 `total`。

示例 9.8 中,没有遇到这个问题,因为我们从未给 `series` 赋值;只是调用了 `series.append`,并对其调用了 `sum` 与 `len`。也就是说,我们利用了“列表是可变对象”这一事实。

但是对于数字、`str`、`tuple` 等不可变类型,只能读取,不能更新。若试图重新绑定(更新)它们,如 `count=count+1`,则会隐式创建一个局部变量 `count`。如此一来,`count` 就不是一个 **自由变量**(Free Variable)了,因此它不会保存在闭包中。

为了解决这个问题,Python 3 中引入了 `nonlocal` 关键字。它允许您将变量声明为 **自由变量**(Free Variable),即使变量是在函数内赋值(绑定)的。如果为 `nonlocal` 声明的变量赋予新值,则闭包中保存的绑定也会随之更新。最新版 `make_averager` 的正确实现如示例 9.13 所示。

</> 示例 9.13: 计算累计平均值,不保存所有历史(用 `nonlocal` 修正)

```
1  def make_averager():
2      count = 0
3      total = 0
4
5      def averager(new_value):
6          nonlocal count, total  # nonlocal 关键字, 声明自由变量
7          count += 1
8          total += new_value
9          return total / count
10
11  return averager
```

在学会使用 `nonlocal` 关键字之后,我们来总结一下 Python 查找变量的逻辑。

9.7.1 变量查找逻辑

当定义一个函数时,Python 字节码编译器会按如下规则,决定如何获取函数中出现的变量 `x`²:

- 若存在 `global x` 声明,则 `x` 来自模块全局作用域³,并赋予模块全局作用域中 `x` 的值。
- 若存在 `nonlocal x` 声明,则 `x` 来自最近一个定义它的外层函数,并赋予那个外层函数中局部变量 `x` 的值。
- 若 `x` 是函数的 **形参**(Parameter),或在函数主体内赋值,那么 `x` 就是局部变量。
- 若 `x` 被引用,但未被赋值,也不是参数。则遵循如下规则:
 - `x` 将在外层函数主体的局部作用域中查找。即在 `nonlocal` 作用域中查找。

²感谢技术审核 Leonardo Rochael 建议做出此处的总结。

³Python 没有程序全局作用域,只有模块全局作用域。

- 若在外层函数作用域内找不到, 将从模块全局作用域中查找。
- 若在模块全局作用域内也找不到, 将从 `__builtins__` 中查找。

掌握了 Python 闭包 (closures) 之后, 就可以有效地使用嵌套函数实现装饰器了。

9.8 实现一个简单的装饰器

示例 9.14 是一个装饰器, 它能对被装饰的函数的每次调用进行计时, 并输出函数执行时长、传入的参数以及函数调用的结果。

```
</> 示例 9.14: lockdeco0.py: 一个显示函数运行时间的简单装饰器
1 import time
2
3 def clock(func):
4     def clocked(*args):          ❶
5         t0 = time.perf_counter()
6         result = func(*args)     ❷
7         elapsed = time.perf_counter() - t0
8         name = func.__name__
9         arg_str = ', '.join(repr(arg) for arg in args)
10        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
11        return result
12    return clocked             ❸
```

- ❶ 定义内部函数 `clocked`, 以接受任意数量的位置参数。
- ❷ 此行之所以有效, 是因为 `clocked` 的闭包包含了自由变量 `func`。
- ❸ 返回内部函数, 取代被装饰的函数。

示例 9.15 展示了装饰器 `clock` 的用法。

```
</> 示例 9.15: 使用装饰器 clock
1 import time
2 from clockdeco0 import clock
3
4 @clock
5 def snooze(seconds):
6     time.sleep(seconds)
7
8 @clock
9 def factorial(n):
10    return 1 if n < 2 else n*factorial(n-1)
11
12 if __name__ == '__main__':
13    print('*' * 40, 'Calling snooze(.123)')
14    snooze(.123)
15    print('*' * 40, 'Calling factorial(6)')
16    print('6! =', factorial(6))
```

运行 [示例 9.15](#) 的输出如下所示：

```

1 $ python3 clockdeco_demo.py
2 **** Calling snooze(.123)
3 [0.12363791s] snooze(0.123) -> None
4 **** Calling factorial(6)
5 [0.0000095s] factorial(1) -> 1
6 [0.00002408s] factorial(2) -> 2
7 [0.00003934s] factorial(3) -> 6
8 [0.00005221s] factorial(4) -> 24
9 [0.00006390s] factorial(5) -> 120
10 [0.00008297s] factorial(6) -> 720
11 6! = 720

```

9.8.1 装饰器工作原理

请记住，如下的左侧代码，实际等价于右侧的代码。

```

1
2 @clock      # 应用装饰器的语法糖
3 def factorial(n):
4     return 1 if n < 2 else n*factorial(n
-1)

```

```

1 def factorial(n):
2     return 1 if n < 2 else n*factorial(n
-1)
3
4 factorial = clock(factorial)

```

因此，在这两个示例中，函数 `clock` 将函数 `factorial` 作为参数，传递给 `func`（见“[示例 9.14<258页>](#)”）。然后，创建并返回 `clocked` 函数。Python 解释器将函数 `clocked` 赋值给 `factorial`（`@clock` 语法糖是在幕后赋值）。事实上，如果导入 `clockdeco_demo` 模块并检查 `factorial` 的 `__name__`，将得到以下结果：

```

1 >>> import clockdeco_demo
2 >>> clockdeco_demo.factorial.__name__
3 'clocked'
4 >>>

```

因此，`factorial` 现在实际上保存了对函数 `clocked` 的引用。从现在起，每次调用 `factorial(n)` 时，都会执行 `clocked(n)`。本质上，函数 `clocked` 执行以下操作：

1. 记录初始时间 t_0 。
2. 调用原来的 `factorial` 函数，并保存结果。
3. 计算函数 `factorial` 的运行时间。
4. 格式化并显示收集到的数据。
5. 返回步骤 2 中保存的结果。

这是装饰器的典型行为：它用一个接受相同参数的新函数替换被装饰函数，并（通常）返回被装饰函数本应返回的内容，同时还进行一些额外的处理。



在 Gamma 等人所著的《设计模式》一书中,有一段针对装饰器模式的简短描述:“动态为对象附加额外的职责。”函数装饰器符合该描述。但在实现层面上,Python 装饰器与该著作中描述的经典装饰器几乎没有相似之处。本章最后的“杂谈”部分会进一步探讨这一话题。

示例 9.14 实现的 clock 装饰器有几个缺点:不支持关键字参数,并且屏蔽了被装饰函数的 `_name_` 与 `doc_` 属性。**示例 9.16** 用 `functools.wraps` 装饰器将相关属性从 `func` 身上复制到 `clocked` 中。此外,这个新版装饰器还可以正确处理关键字参数。

</> **示例 9.16:** clockdeco.py: 经过改进的 clock 装饰器

```

1  import time
2  import functools
3
4  def clock(func):
5      @functools.wraps(func)
6      def clocked(*args, **kwargs):
7          t0 = time.perf_counter()
8          result = func(*args, **kwargs)
9          elapsed = time.perf_counter() - t0
10         name = func.__name__
11         arg_lst = [repr(arg) for arg in args]
12         arg_lst.extend(f'{k}={v!r}' for k, v in kwargs.items())
13         arg_str = ', '.join(arg_lst)
14         print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
15         return result
16     return clocked

```

`functools.wraps` 只是 Python 标准库中可开箱即用的装饰器之一。下一节,将介绍 `functools` 模块中最让人深刻的装饰器,即 `cache`。

9.9 标准库中的装饰器

Python 有 3 个用作装饰器的内置函数:`property`、`classmethod`、`staticmethod`。将在“[22.4 用特性验证属性](#)”中讨论 `property`,在“[11.5 classmethod 与 staticmethod](#)”中讨论 `classmethod` 与 `staticmethod`。

“[示例 9.16<260页>](#)”中用到了一个重要的装饰器,即 `functools.wraps`,是一个用于构建行为良好的装饰器的辅助工具。标准库中最吸引人的几个装饰器(`cache`、`lru_cache`、`singledispatch`),均来自 `functools` 模块。下面将分别介绍这几个装饰器。

9.9.1 用 `functools.cache` 进行记忆化

`cache` 装饰器实现了 [记忆化 \(Memoization\)](#)⁴ 这种优化技术,该技术可将先前耗时的函数结果保存起来,避免在为函数传入相同参数时的重复计算。

⁴注意,此处的拼写没有错误。`memoization` 是一个计算机专业术语,与“`memorization`”有一点关系,但不是同一个概念。



`functools.cache` 是 Python 3.9 新增的装饰器。若想在 Python 3.8 中运行本节示例, 请将代码中的 `@cache` 替换为 `@lru_cache`。对于更早期版本的 Python, 必须调用装饰器, 写成 `@lru_cache()`(详见“[9.9.2 使用 lru_cache<263页>](#)”),

`@cache` 适用于慢速的递归函数中, 如生成 Fibonacci(斐波那契) 序列中的第 n 个数字(如[示例 9.17](#)所示)。

</> [示例 9.17](#): 生成第 n 个斐波那契数, 递归非常耗时

```
1  from clockdeco import clock
2
3  @clock
4  def fibonacci(n):
5      if n < 2:
6          return n
7      return fibonacci(n - 2) + fibonacci(n - 1)
8
9  if __name__ == '__main__':
10     print(fibonacci(6))
```

下面是运行 `fibo_demo.py` 的结果。除最后一行外, 所有输出都是由 `clock` 装饰器([示例 9.16](#))生成的:

```
1  $ python3 fibo_demo.py
2  [0.00000042s] fibonacci(0) -> 0
3  [0.00000049s] fibonacci(1) -> 1
4  [0.00006115s] fibonacci(2) -> 1
5  [0.00000031s] fibonacci(1) -> 1
6  [0.00000035s] fibonacci(0) -> 0
7  [0.00000030s] fibonacci(1) -> 1
8  [0.00001084s] fibonacci(2) -> 1
9  [0.00002074s] fibonacci(3) -> 2
10 [0.00009189s] fibonacci(4) -> 3
11 [0.00000029s] fibonacci(1) -> 1
12 [0.00000027s] fibonacci(0) -> 0
13 [0.00000029s] fibonacci(1) -> 1
14 [0.00000959s] fibonacci(2) -> 1
15 [0.00001905s] fibonacci(3) -> 2
16 [0.00000026s] fibonacci(0) -> 0
17 [0.00000029s] fibonacci(1) -> 1
18 [0.00000997s] fibonacci(2) -> 1
19 [0.00000028s] fibonacci(1) -> 1
20 [0.00000030s] fibonacci(0) -> 0
21 [0.00000031s] fibonacci(1) -> 1
22 [0.00001019s] fibonacci(2) -> 1
23 [0.00001967s] fibonacci(3) -> 2
24 [0.00003876s] fibonacci(4) -> 3
25 [0.00006670s] fibonacci(5) -> 5
26 [0.00016852s] fibonacci(6) -> 8
27 8
```

耗费时间是显而易见的:fibonacci(1)被调用8次,fibonacci(2)被调用5次,等等。但只需增加2行代码,使用@cache,性能就会大大提高。如示例9.18所示。

</>示例9.18:用@cache实现示例9.17,速度更快

```

1 import functools
2 from clockdeco import clock
3
4 @functools.cache ❶
5 @clock ❷
6 def fibonacci(n):
7     if n < 2:
8         return n
9     return fibonacci(n - 2) + fibonacci(n - 1)
10
11 if __name__ == '__main__':
12     print(fibonacci(6))

```

❶此行适用于Python ≥ 3.9 版本。有关早期版本的替代方案,参见“9.9.2 使用lru_cache”。

❷这是堆叠装饰器的示例:@cache应用到@clock返回的函数上。

堆叠(Stacked)装饰器



为了理解堆叠装饰器,请记住@是语法糖,用于将装饰器函数应用到下面的函数。如果有多个装饰器,它们的行为就像嵌套的函数调用。即如下两段代码等价。

```

1 @alpha
2 @beta
3 def my_fn():
4     ...

```

```

1 my_fn = alpha(beta(my_fn))
2

```

上述两段代码表示:首先,应用beta装饰器;然后,将其返回的函数传递给alpha装饰器。

示例9.18中使用了@cache装饰器后,对于每个n值,仅调用一次fibonacci函数(如下所示)。

```

1 $ python3 fibo_demo_lru.py
2 [0.00000043s] fibonacci(0) -> 0
3 [0.00000054s] fibonacci(1) -> 1
4 [0.000006179s] fibonacci(2) -> 1
5 [0.00000070s] fibonacci(3) -> 2
6 [0.000007366s] fibonacci(4) -> 3
7 [0.00000057s] fibonacci(5) -> 5
8 [0.000008479s] fibonacci(6) -> 8
9 8

```

为了计算fibonacci(30),示例9.18在0.00017秒(总时间)内完成了所需的31次调用。而未用@cache的版本(“示例9.17<261页>”)在配有Intel Core i7处理器的笔记本电脑上花费了12.09秒。因为仅fibonacci(1)就调用了832,040次,总计2,692,537次。

被 `@cache` 装饰的函数,所接受的参数必须是 可哈希 (hashable),因为底层 `lru_cache` 使用 `dict` 来存储结果,并且 `dict` 的键是由调用中使用的位置参数与关键字参数构成的。

除了可用于优化递归算法, `@cache` 还非常适合用在从远程 API 中获取信息的应用程序中。在这些场景中, `@cache` 都可发挥巨大作用。



如果缓存条目非常多, `functools.cache` 可能会耗尽所有可用内存。我认为它更适合用在耗时较短的命令行脚本中。在长时间运行的程序中,我建议使用 `functools.lru_cache`,并设置一个合适的 `maxsize` 参数,这将在下一节中介绍。

9.9.2 使用 `lru_cache`

`functools.cache` 装饰器实际上是对旧版 `functools.lru_cache` 函数的简单封装。`functools.lru_cache` 更加灵活,并可与 Python 3.8 及更早版本兼容。

`@lru_cache` 的主要优势在于其内存使用量受 `maxsize` 参数限制,该参数的默认值(比较保守)为 128。表示缓存中最多只能容纳 128 个条目。

LRU (Last Recently Used,) 是“最近很少使用”的意思,表示一段时间不用的旧条目将被丢弃,以为新条目腾出空间。

自 Python 3.8 起, `lru_cache` 可以通过两种方式使用。下面是最简单的方式:

```
1 @lru_cache
2 def costly_function(a, b):
3     ...
```

另一种方法(适用于 Python ≥ 3.2)是使用(),将其作为函数调用(如下所示):

```
1 @lru_cache()
2 def costly_function(a, b):
3     ...
```

这两种使用方式,都采用以下默认参数:

- `maxsize=128`

设置要缓存的最大条目数。缓存满后,最近使用次数最少的条目将被丢弃,以便为新条目腾出空间。为了获得最佳性能, `maxsize` 应该是 2 的幂。如果传递 `maxsize=None`, LRU 逻辑将被禁用。因此,缓存速度更快,但条目永远不会被丢弃,这可能会消耗太多内存。这也是 `@functools.cache` 的作用。

- `typed=False`

确定不同参数类型的结果是否单独存储。例如,在默认设置中,被视为相等的浮点和整数参数仅存储一次,因此调用 `f(1)` 和 `f(1.0)` 只对应一个缓存条目。如果 `typed=True`, `f(1)` 和 `f(1.0)` 将产生不同的条目,可能存储不同的结果。

以下是使用非默认参数调用 `@lru_cache` 的示例:

```
1 @lru_cache(maxsize=2**20, typed=True)
2 def costly_function(a, b):
3     ...
```

现在,让我们来学习另一个功能强大的装饰器:`functools.singledispatch`。

9.9.3 单分派泛化函数

假设我们正在开发一个调试 Web 应用程序的工具,需要为不同类型的 Python 对象生成 HTML 展示代码。

为此,可能会编写如下函数:

```
1 import html
2
3 def htmlize(obj):
4     content = html.escape(repr(obj))
5     return f'<pre>{content}</pre>'
```

此函数适用于所有 Python 类型。但现在需要对其进行扩展,以便为某些 Python 类型生成自定义 HTML 展示代码。例如:

- str
用“`
\n`”替换内嵌的换行符,用 `<pre>` 标记替代 `<p>` 标记。
- int
以十进制和十六进制显示数字 (bool 除外)。
- list
输出 HTML 列表,并根据各项的类型对其进行格式化。
- float 与 Decimal
正常输出数值,但也可以以分数的形式输出。

我们想要的行为如 [示例 9.19](#) 所示:

</> [示例 9.19: `htmlize\(\)` 为不同 Python 对象类型生成 HTML](#)

```
1 >>> htmlize({1, 2, 3})          ❶
2 '<pre>{1, 2, 3}</pre>'
3 >>> htmlize(abs)
4 '<pre>&lt;built-in function abs&gt;</pre>'          ❷
5 >>> htmlize('Heimlich & Co.\n- a game')          ❸
6 '<p>Heimlich & Co.<br/>\n- a game</p>'
7 >>> htmlize(42)                  ❹
8 '<pre>42 (0x2a)</pre>'
9 >>> print(htmlize(['alpha', 66, {3, 2, 1}]))      ❺
10 <ul>
11 <li><p>alpha</p></li>
12 <li><pre>66 (0x42)</pre></li>
13 <li><pre>{1, 2, 3}</pre></li>
14 </ul>
15 >>> htmlize(True)              ❻
16 '<pre>True</pre>'          ❼
17 >>> htmlize(fractions.Fraction(2, 3))          ❽
18 '<pre>2/3</pre>'
```

```
19  >>> htmlize(2/3)          ❷
20  '<pre>0.6666666666666666 (2/3)</pre>'
21  >>> htmlize(decimal.Decimal('0.02380952'))
22  '<pre>0.02380952 (1/42)</pre>'
```

- ❶ 原始 `htmlize()` 函数是针对 `object` 类型的。当不存在为某种 Python 类型生成 HTML 展示的函数时, 会调用 `htmlize()`, 以生成默认的 HTML 代码。相当于一种兜底实现。
- ❷ `str` 对象需要进行 HTML 转义。将字符串内容放在 `<p></p>` 标签内, 并且在每个 ‘`\n`’ 之前, 插入换行标签 `
`。
- ❸ `int` 类型将被放在 `<pre></pre>` 标签内, 并以十进制或十六进制进行显示。
- ❹ `list` 列表中的各项, 根据其类型分别进行格式化。整个 `list` 会被渲染为 HTML 无序列表。
- ❺ 虽然 `bool` 是 `int` 的子类型, 但是它需要特殊处理。
- ❻ 以分数形式显示 `Fraction` (分数) 对象。
- ❼ 以近似的分数显示 `float` 值和 `Decimal` 值。

9.9.3.1 单一分派函数

因为 Python 不支持 Java 风格的方法重载, 所以不能简单地用不同的函数签名, 为各种 Python 类型创建不同的 `htmlize` 变体。在 Python 中, 一个可行的解决方案是将 `htmlize` 变成一个 “分派 (dispatch) 函数”。分派函数会通过一系列 `if/elif/...` 或 `match/case/...`, 来调用特化函数 (Specialized Function), 如 `htmlize_str`、`htmlize_int` 等。但这种方式既不易于扩展, 也不够灵活: 随着时间的推移, `htmlize` 分派函数会变得过于庞大, 而且它与特化函数 (Specialized Function) 之间的耦合非常紧密。

`functools.singledispatch` 装饰器允许不同的模块为整体解决方案提供各自的贡献, 甚至可以很容易地为第三方包中无法编辑的类型提供特化函数 (Specialized Function)。被 `@singledispatch` 装饰过的普通函数, 会成为 泛化函数 (Generic Function) 的入口点: 泛化函数 (Generic Function) 是一组函数, 这组函数会根据第一个实参 (Argument) 的类型, 以不同方式执行相同的操作, 这也是 单一分派 (Single Dispatch) 的含义。如果根据多个实参 (Argument) 选择特化函数 (Specialized Function), 则称为 多分派 (Multiple Dispatch)。示例 9.20 展示了 `functools.singledispatch` 装饰器的用法。



`functools.singledispatch` 自 Python 3.4 起就存在, 但自 Python 3.7 起才支持类型提示 (Type Hints)。示例 9.20 中, 最后两个函数使用的语法支持自 Python 3.4 起的所有版本。

</> 示例 9.20: 用 `@singledispatch` 创建 `@htmlize.register` 装饰器, 将多个函数绑定为一个泛化函数

```
1  from functools import singledispatch
2  from collections import abc
3  import fractions
4  import decimal
5  import html
6  import numbers
7
8  @singledispatch ❶
```

```

9  def htmlize(obj: object) -> str:
10     content = html.escape(repr(obj))      return f'<pre>{content}</pre>'
11
12 @htmlize.register ❷
13 def _(text: str) -> str:      ❸
14     content = html.escape(text).replace('\n', '<br>\n')
15     return f'<p>{content}</p>'
16
17 @htmlize.register ❹
18 def _(seq: abc.Sequence) -> str:
19     inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
20     return '<ul>\n<li>' + inner + '</li>\n</ul>'
21
22 @htmlize.register ❺
23 def _(n: numbers.Integral) -> str:
24     return f'<pre>{n} (0x{n:x})</pre>'
25
26 @htmlize.register ❻
27 def _(n: bool) -> str:
28     return f'<pre>{n}</pre>'
29
30 @htmlize.register(fractions.Fraction) ❻
31 def _(x) -> str:
32     frac = fractions.Fraction(x)
33     return f'<pre>{frac.numerator}/{frac.denominator}</pre>'
34
35 @htmlize.register(decimal.Decimal)
36 @htmlize.register(float)
37 def _(x) -> str:
38     frac = fractions.Fraction(x).limit_denominator()
39     return f'<pre>{x} ({frac.numerator}/{frac.denominator})</pre>'
```

- ❶ `@singledispatch` 标记了处理对象类型的基本 (base) 函数。
- ❷ 每个 **特化函数 (Specialized Function)**, 都要用 `@<base>.register` 进行装饰。
- ❸ 运行时传入的第 1 个 **实参 (Argument)** 类型决定了何时使用这个 **特化函数 (Specialized Function)**。
特化函数 (Specialized Function) 的函数名不重要, 建议用 `_`, 简单明了⁵。
- ❹ 为每个需要特殊处理的 Python 类型注册一个 **特化函数 (Specialized Function)**, 并为第 1 个参数添加相应 Python 类型的 **类型提示 (Type Hints)**。
- ❺ `numbers` 包中的 **抽象基类 (ABCs)** 可以有效地与 `@singledispatch` 装饰器一起使用⁶。
- ❻ `bool` 是 `numbers.Integral` 的子类型, 但 `singledispatch` 逻辑会寻找与指定类型最匹配的 **特化函数 (Specialized Function)**。
- ❼ 如果不想或不能为被装饰的 **特化函数 (Specialized Function)** 添加 **类型提示 (Type Hints)**, 则可将参数类型传给 `@<base>.register` 装饰器。Python ≥ 3.4 支持此种语法。
- ❽ `@<base>.register` 装饰器会返回装饰之前的函数, 因此可以堆叠多个 `register` 装饰器, 让同一个特

⁵ 不幸的是, Mypy 0.770 在看到多个同名函数时, 会报出错误。

⁶ 尽管 “8.5.7.1 数字塔的崩塌” 否定了 `numbers` 包中的 **抽象基类 (ABCs)**。但是, 这些抽象基类并没有被废弃。

化函数 (Specialized Function) 可支持 2 个或更多类型⁷。

示例 9.20 中的 `@singledispatch` 装饰器将函数 `htmlize` 变成了 泛化函数 (Generic Function) 的入口点, 而每个应用了 `@htmlize.register` 装饰器的函数, 都是泛化函数的一个实现, 也称为“特化函数 (Specialized Function)”。

在注册 特化函数 (Specialized Function) 时, 形参 (Parameter) 的类型提示应该尽量使用 抽象基类 (ABCs) (如 `numbers.Integral` 与 `abc.MutableSequence`), 而不要使用 具体类 (Concrete Class) (如 `int` 与 `list`)。这样, 注册的特化函数 (Specialized Function) 可处理更多的兼容类型。例如, Python 扩展可以通过子类化 `numbers.Integral`, 用固定位长度实现 `int` 类型的替代方案⁸。



与 `@singledispatch` 一起使用 抽象基类 (ABCs) 或 `typing.Protocol` 可以让代码支持现有或未来的类。这些类可以是 抽象基类 (ABCs) 的具体子类 (Concrete Subclass) 或 虚拟子类 (Virtual Subclass), 也可以是实现了这些协议的具体子类 (Concrete Subclass) 或 虚拟子类 (Virtual Subclass)。抽象基类 (ABCs) 的使用以及 虚拟子类 (Virtual Subclass) 的概念, 将在“[十三 接口、协议与抽象基类](#)”中进行探讨。

`singledispatch` 机制的一个显著特点是, 你可以在系统的任何地方、任何模块中注册 特化函数 (Specialized Function)。如果将来在新模块中定义了新类型, 你也可以轻松地提供新的 特化函数 (Specialized Function) 来处理这个新类型。此外, 还可以为别人编写的类或无法修改的类编写自定义函数。

`singledispatch` 是经过深思熟虑后, 才新增到 Python 标准库中的。它提供了很多功能, 此处无法一一描述。“[PEP 443 -Single-dispatch generic functions](#)”是一份很好的参考资料, 但是未提到 **类型提示 (Type Hints)** 的使用, 毕竟类型提示出现的较晚。`functools` 模块文档 有所改进, 在其 `singledispatch` 条目中有几个关于 **类型提示 (Type Hints)** 的使用示例。



`@singledispatch` 并不是为了将 Java 风格的方法重载引入 Python 而设计的。虽然在一个类中为同一个方法定义多个重载变体, 要比在一个函数中使用一长串 `if/elif/elif/elif` 语句块要好。但是, 这两种方案都存在缺陷, 因为它们会让一个代码单元 (类或函数) 承担太多的职责。`@singledispatch` 的优势在于支持模块化扩展: 每个模块可以为其支持的每种类型注册一个 特化函数 (Specialized Function)。在实践中, 不会像“[示例 9.20<265页>](#)”那样, 将 泛化函数 (Generic Function) 的实现^a都放在同一个模块中。

^a此处的实现是指用于处理每种新类型的 特化函数 (Specialized Function)。

我们已经看到了一些接受参数的装饰器, 例如“[示例 9.20<265页>](#)”中 `@singledispatch` 创建的 `@lru_cache()` 和 `htmlize.register(float)`。下一节将介绍如何创建接受参数的装饰器。

⁷也许有一天, 只需要提供一个无参数的 `register` 装饰器, 并在类型提示中使用 `Union` 类型, 就可表达这一点。但现在这么尝试, 会引发 `TypeError` 异常并提示 `Union` 不是一个类。因此, 尽管 `@singledispatch` 已支持 PEP 484 语法, 但在语义上还未实现。

⁸例如, NumPy 实现了多种面向机器的整数类型与浮点类型。

9.10 参数化装饰器

在解析源码中的装饰器时,Python 会将被装饰的函数作为第一个参数传递给装饰器函数。那么如何让装饰器接受其他参数呢?答案是:创建一个装饰器工厂函数(Factory Function)来接受那些参数,并返回一个装饰器,然后将该装饰器应用于要装饰的函数。是不是有些迷惑?让我们从一个最简单的装饰器(如示例9.21所示)开始。

</> 示例 9.21: “示例 9.2<249页>”中 registration.py 模块的删减版,再次给出以方便查看

```

1 registry = []
2
3 def register(func):
4     print(f'running register({func})')
5     registry.append(func)
6     return func
7
8 @register
9 def f1():
10    print('running f1()')
11
12 print('running main()')
13 print('registry ->', registry)
14 f1()

```

9.10.1 一个参数化的注册装饰器

为了便于启用或禁用 register 执行的函数注册,我们为 register 增加一个可选参数 active。如果该参数为 False,则跳过注册被装饰的函数。示例 9.22 展示了如何实现。从概念上讲,新的 register 函数并不是一个装饰器,而是一个装饰器工厂函数(Factory Function)。调用时此工厂函数时,它会返回将应用于目标函数的实际装饰器。

</> 示例 9.22: 为了接受参数,新的 register 装饰器必须作为函数调用

```

1 registry = set()          ①
2
3 def register(active=True): ②
4     def decorate(func):     ③ # 实际的装饰器
5         print('running register')
6         f'(active={active})->decorate({func})'
7         if active:          ④ # 在 register 函数作用域中检索
8             registry.add(func)
9         else:
10             registry.discard(func) ⑤
11
12         return func          ⑥
13     return decorate         ⑦
14

```

```
15 @register(active=False)          ❸
16 def f1():    print('running f1()')
17
18 @register()
19 def f2():
20     print('running f2()')
21
22 def f3():
23     print('running f3()')
```

- ❶ registry 是一个 set 对象,因此添加与删除的速度更快。
- ❷ registry 接受一个可选的关键字参数 active。
- ❸ 内部的 decorate 函数是实际的装饰器;注意,它的参数是一个函数对象。
- ❹ 仅当参数 active (从闭包中检索) 为 True 时,才注册 func。
- ❺ 若参数 active 不为 True,且 func 在 registry 中,则从 registry 中删除 func。
- ❻ 因为 decorate 是一个装饰器,所以必须返回一个函数。
- ❼ register 是我们的装饰器 工厂函数 (Factory Function),所以它返回 decorate 装饰器。
- ❽ @register 工厂必须作为函数来调用,并为其提供所需的参数。
- ❾ 即使不提供参数,register 也必须作为函数 (@register()) 来调用,以返回真正的 decorate 装饰器。

重点是,register() 返回 decorate 装饰器,然后将该装饰器应用到被装饰的函数上。

示例 9.22 中的代码位于 registration_param.py 模块中。如果我们导入它,会得到以下结果:

```
1 >>> import registration_param
2 running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
3 running register(active=True)->decorate(<function f2 at 0x10063c268>)
4 >>> registration_param.registry
5 [<function f2 at 0x10063c268>]
```

注意,registry 中只出现了函数 f2,而未出现函数 f1。原因是:为 f1 应用 register 装饰器工厂时,参数 active=False,所以 register 返回的 decorate 装饰器没有将 f1 注册到 registry 中。

如果不使用 @ 语法,就要像常规函数那样调用 register。若想将函数 f 注册到 registry 中,那么装饰 f 的语法是 registry()(f);若不想注册 f (或将 f 删除),则装饰 f 的语法是 registry(active=False)(f)。示例 9.23 演示了如何将函数注册到 registry 中,以及如何从 registry 中删除函数。

</> 示例 9.23: 使用示例 9.22 中的 registration_param 模块

```
1 >>> from registration_param import *
2 running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
3 running register(active=True)->decorate(<function f2 at 0x10073c268>)
4 >>> registry      ❶
5 {<function f2 at 0x10073c268>}
6 >>> register()(f3) ❷
7 running register(active=True)->decorate(<function f3 at 0x10073c158>)
8 <function f3 at 0x10073c158>
9 >>> registry      ❸
10 {<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
```

```

11  >>> register(active=False)(f2) ❸
12  running register(active=False)->decorate(<function f2 at 0x10073c268>)<function f2
13  at 0x10073c268>
14  >>> registry ❹
15  {<function f3 at 0x10073c158>}

```

- ❶ 导入 registration_param 模块后, registry 中就会出现 f2。
 ❷ register() 表达式返回 decorate 装饰器, 然后应用到 f3 上。
 ❸ 上一行(即❷处), 将 f3 注册到 registry 中。
 ❹ 此调用, 将从 registry 中删除 f2。
 ❺ 确认 registry 中只剩下 f3。

参数化装饰器的工作原理相当复杂, 而刚刚讨论的示例比大多数的参数化装饰器都要简单。参数化装饰器通常会替换被装饰的函数, 并且在结构上需要多增加一层嵌套。接下来, 将探讨这种函数金字塔的架构。

9.10.2 参数化 clock 装饰器

在本节中, 我们将重温 clock 装饰器, 并为其增加一项功能: 用户可以传入格式化字符串, 来控制 clock 函数的输出格式, 如示例 9.24 所示。



为简单起见, [示例 9.24](#) 是基于 “[示例 9.14<258页>](#)” 中最初的 clock 实现的, 而不是基于 “[示例 9.16<260页>](#)” 中那个使用 `functools.wraps` 改进的版本。因为, 那个改进的版本增加了一层函数。

</> [示例 9.24: clockdeco_param.py 模块:参数化的 clock 装饰器](#)

```

1  import time
2
3  DEFAULT_FMT = '{elapsed:0.8f}s {name}({args}) -> {result}'
4
5  def clock(fmt=DEFAULT_FMT): ❶ # 装饰器工厂
6      def decorate(func): ❷ # 实际的装饰器
7          def clocked(*_args): ❸
8              t0 = time.perf_counter()
9              _result = func(*_args) ❹
10             elapsed = time.perf_counter() - t0
11             name = func.__name__
12             args = ', '.join(repr(arg) for arg in _args) ❺
13             result = repr(_result) ❻
14             print(fmt.format(**locals())) ❼
15             return _result ❽
16             return clocked ❾
17             return decorate ❿
18
19         if __name__ == '__main__':
20

```

```
21 @clock()  
22     def snooze(seconds):      ❶  
23         time.sleep(seconds)  
24  
25     for i in range(3):  
26         snooze(.123)
```

- ❶ `clock` 是参数化的装饰器工厂函数 (Factory Function)。
- ❷ `decorate` 是实际的装饰器。
- ❸ `clocked` 包装了被装饰的函数。
- ❹ `_result` 是被修饰函数的实际结果。
- ❺ `_args` 保存了 `clocked` 的实际参数, 而 `args` 是用于显示的 `str`。
- ❻ `result` 是 `_result` 的字符串表示形式, 用于显示。
- ❼ 此处使用 `**locals()`, 可在 `fmt` 中引用 `clocked` 的所有局部变量。
- ❽ `clocked` 将取代被装饰的函数, 因此 `clocked` 应返回被装饰函数返回的所有结果。
- ❾ `decorate` 返回 `clocked` 函数对象。
- ❿ `clock` 返回 `decorate` 函数对象。
- ❾ 在当前模块中测试, 调用函数 `clock()` 时不传入参数, 因此应用的装饰器将使用默认的格式化字符串。

在 shell 中运行 [示例 9.24](#), 将得到以下结果:

```
1 $ python3 clockdeco_param.py  
2 [0.12412500s] snooze(0.123) -> None  
3 [0.12411904s] snooze(0.123) -> None  
4 [0.12410498s] snooze(0.123) -> None
```

为了练习新功能, 让我们看一下 `??` 和 [示例 9.26](#), 及它们的输出。这两个示例是使用了 `clockdeco_param` 的另外两个模块。

</> [示例 9.25: clockdeco_param_demo1.py](#)

```
1 import time  
2 from clockdeco_param import clock  
3  
4 @clock('{name}: {elapsed}s')  
5 def snooze(seconds):  
6     time.sleep(seconds)  
7  
8     for i in range(3):  
9         snooze(.123)
```

[示例 9.25](#) 的输出, 如下所示。

```
1 $ python3 clockdeco_param_demo1.py  
2 snooze: 0.12414693832397461s  
3 snooze: 0.1241159439086914s  
4 snooze: 0.12412118911743164s
```

</> [示例 9.26: clockdeco_param_demo2.py](#)

```

1  from clockdeco_param import clock
2  @clock('{name}({args}) dt={elapsed:0.3f}s')
3      time.sleep(seconds)
4
5  for i in range(3):
6      snooze(.123)

```

示例 9.26 的输出,如下所示。

```

1  $ python3 clockdeco_param_demo2.py
2  snooze(0.123) dt=0.124s
3  snooze(0.123) dt=0.124s
4  snooze(0.123) dt=0.124s

```



本书第 1 版的技术审校 Lennart Regebro 认为,最好通过定义了 `__call__` 方法的类来实现装饰器,而不是用函数来实现(如本章的示例所示)。我认同“类更适合创建重要的装饰器”这一说法。但是,为了解释装饰器的基本思想,函数更容易理解。“9.12 延伸阅读<273页>”提到了一些工业级的装饰器构建技术,尤其不要错过 Graham Dumpleton 的博客和 wrapt 模块。

下一节会列举一些例子,按 Regebro 和 Dumpleton 推荐的风格创建装饰器。

9.10.3 基于类的 clock 装饰器

作为最后一个示例,示例 9.27 用一个定义了 `__call__` 方法的类,实现了参数化装饰器 `clock`。对比“示例 9.24<270页>”与示例 9.27之后,您更喜欢哪一个?

</> 示例 9.27: colckdeco_cls.py 模块:通过类实现的参数化装饰器 clock

```

1  import time
2
3  DEFAULT_FMT = '{elapsed:0.8f}s {name}({args}) -> {result}'
4
5  class clock: ❶
6
7      def __init__(self, fmt=DEFAULT_FMT): ❷
8          self.fmt = fmt
9
10     def __call__(self, func): ❸
11         def clocked(*_args):
12             t0 = time.perf_counter()
13             _result = func(*_args) ❹
14             elapsed = time.perf_counter() - t0
15             name = func.__name__
16             args = ', '.join(repr(arg) for arg in _args)
17             result = repr(_result)
18             print(self.fmt.format(**locals()))
19             return _result

```

20

```
return clocked
```

- ❶ `clock` 类是一个参数化装饰器工厂, 无需再定义外层 `clock` 函数。类名以小写字母命名, 以表明此处的实现可以直接替代“[示例 9.24](#)”中的 `clock` 装饰器。
- ❷ `clock(my_format)` 传入的参数, 将赋值给此处的 `fmt` 参数。类构造函数会返回一个 `clock` 实例, `self.ftmt` 中存储了 `my_format`。
- ❸ `__call__` 使 `clock` 实例成为一个可调用对象。调用 `clock` 实例时, 会用函数 `clocked` 取代被装饰的函数。
- ❹ `clocked` 包装了被装饰的函数。

对函数装饰器的探索到此结束。我们将在“[二十四 类元编程 \(Class Metaprogramming\)](#)”中看到类装饰器。

9.11 本章小结

本章中介绍了一些难以理解的领域。我试图让这个过程尽可能顺利, 但我们毕竟已进入了元编程领域。

我们从一个没有内部函数的简单装饰器 `@register` 开始, 并以一个涉及两层嵌套函数的参数化装饰器 `@clock()` 结束。

注册装饰器虽然本质上很简单, 但在 Python 框架中却有实际应用。下一章将使用这种注册方式实现一个策略设计模式。

要了解装饰器的实际工作原理, 需要了解导入时与运行时的差异; 然后, 深入理解变量作用域、[闭包 \(closures\)](#)、`nonlocal` 声明。掌握 [闭包 \(closures\)](#) 与 `nonlocal` 声明不仅对构建装饰器有价值, 而且对编写面向事件的 GUI 程序或带有回调的异步 I/O 程序也很有价值, 还能在合理的情况下采用函数式风格编程。

参数化装饰器几乎都要涉及至少两层嵌套函数, 如果想用 `@functools.wraps` 生成可为高级技术提供更好支持的装饰器, 则嵌套层级可能更深, 如“[示例 9.18](#)”所示。对于更复杂的装饰器, 基于类的装饰器实现可能更易于阅读和维护。

作为标准库中参数化装饰器的示例, 我们介绍了 `functools` 模块中功能强大的 `@cache` 和 `@singledispatch`。

9.12 延伸阅读

Brett Slatkin 的《[Effective Python](#) 第 2 版 (Addison-Wesley 出版社)》的第 26 条涉及函数装饰器的最佳实践, 并建议始终使用 `functools.wraps` (如“[示例 9.16](#)”所示)⁹。

Graham Dumpleton 发表了一系列博文, 从“[How you implemented your Python decorator is wrong](#)”开始, 深入剖析了如何实现行为良好的装饰器。他在这方面的深厚专业知识充分体现在他所编写的 `wrapt` 模块中, 旨在简化装饰器与动态函数包装器的实现。这些装饰器和包装器支持[内省 \(Introspection\)](#), 并且在多层装饰、应用于方法 (methods), 以及用作属性描述符 (attribute descriptors) 时, 都表现出了正确行为。“[延伸阅读](#)”中的“[二十三 属性描述符](#)”将专门讨论描述符 (descriptors)。

⁹为了保证代码简单易懂, 本书中某些示例并未遵从 Slatkin (《Effective Python: 编写高质量 Python 代码的 90 个有效方法》第 2 版的作者) 的出色建议。

《Python Cookbook (第3版)》(David Beazley与Brian K. Jones著)第9章“元编程”包含了从基础装饰器到复杂装饰器的几个示例。其中,9.6节展示了如何定义一个接受可选参数的装饰器,该装饰器既可作为常规的装饰器来调用(如@clock),也可作为装饰器工厂函数来调用(如@clock())。

Michele Simionato编写了decorator包,根据其文档描述,decorator包旨在“简化普通程序员对装饰器的使用,并通过展示各种复杂示例来普及装饰器。”

“Python Decorator Library”维基页面,在Python刚添加装饰器功能时就创建了。该页面包含了数十个装饰器示例,但由于该页面是多年前创建的,其中一些技术已过时,但仍不失为一个极好的灵感来源。

Fredrik Lundh撰写的博文《Closures in Python》,详细解释了闭包(closures)这一术语。

“PEP 3104 -Access to Names in Outer Scopes”说明了引入nonlocal声明的原因——允许重新绑定既不是局部,也不是全局的名称。此外,还出色地概述了其他动态语言(Perl、Ruby、JavaScript等)是如何解决这一问题的,以及Python可用的设计方案的优缺点。

在理论层面上,“PEP 227 -Statically Nested Scopes”说明了Python 2.1引入的词法作用域。词法作用域在Python 2.1中是一个可选方案,而在Python 2.2中变成了标准方案。此外,该PEP还解释了在Python中实现闭包的基本原理和设计选择。

“PEP 443 - Single-dispatch generic functions”提供了单分派泛化函数设施的基本原理与详细描述。Guido van Rossum的一篇旧博文(2005年3月)“Five-minute Multimethods in Python”介绍了使用装饰器实现泛化函数(又称multimethods,多方法)。他的代码支持多分派(即基于多个位置参数的分派)。Guido的多方法代码很有趣,但只是一个教学示例。若想用现代的技术实现可用于生产的多分派泛化函数,请参阅Martijn Faassen编写的Reg库¹⁰。Martijn Faassen还是模型驱动REST式Web框架Morepath的作者。

杂谈

动态作用域与词法作用域

任何将函数视作一等对象的语言,其设计者都会面临这样一个问题:作为一等对象,函数是在某个作用域中定义的,但可以在其他作用域中调用。问题是:如何求解自由变量?首先出现的、最简单的处理方式是使用“动态作用域”。也就是说,根据调用函数的环境来求解自由变量。

如果Python支持动态作用域,但不支持闭包(closures)。我们可以用与“示例9.8<254页>”类似的方法改进avg,如下所示:

```

1  >>> ### 这不是真实的 Python 控制台会话! ####
2  >>> avg = make_averager()
3  >>> series = []      ❶
4  >>> avg(10)
5  10.0
6  >>> avg(11)        ❷
7  10.5
8  >>> avg(12)
9  11.0
10 >>> series = [1]    ❸
11 >>> avg(5)
12 3.0

```

¹⁰Reg是一个Python库,为Python提供通用函数支持。它可以帮助您为应用程序、库或框架构建强大的注册和配置API。

- ① 在使用 `avg` 之前，必须先自己定义 `series = []`，因此我们必须知道 `averager`（在 `make_averager` 内部）引用的是一个名为 `series` 的列表。
- ② 在幕后，`series` 会累计需要计入平均值的值。
- ③ 当执行 `series = [1]` 时，之前的列表会丢失。当同时计算 2 个独立的累计平均值时，可能会发生这种意外。

函数应该是一个黑盒，对用户隐藏其内部的实现。但在动态作用域中，如果函数使用了自由变量，程序员就必须了解其内部结构，才能建立正确运行函数所需的环境。在与 `\TeX` 文档语言斗争多年后，直到阅读《Practical `\TeX`》(George Grätzer 著)一书后才意识到，原来 `\TeX` 变量使用的是动态作用域。难怪以前觉得 `\TeX` 变量这么令人困惑。

Emacs Lisp 也使用动态作用域，至少默认是这样。相关简短说明，请参阅 Emacs Lisp 手册中的“[动态绑定](#)”。

动态作用域更容易实现，这也许就是为什么 John McCarthy 在创建 Lisp（第一门将函数视作一等对象的语言）时选择了动态作用域的原因。Paul Graham 的文章“[The Roots of Lisp](#)”通俗易懂地解读了《[Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I](#)》^a 这一论文。Paul Graham 将此论文由数学原理翻译成了英语和可运行的代码，同时还指出实现动态作用域很棘手。以下文字摘自“[The Roots of Lisp](#)”：

就连 Lisp 的第一个高阶函数示例都动态作用域而被破坏，这充分证明了动态作用域的危险性。也许 McCarthy 在 1960 年还未完全意识到动态作用域的影响。动态作用域在 Lisp 实现中保留的时间特别长，直到 Sussman 和 Steele 在 1975 年开发出 Scheme 为止。词法作用域对 eval 的定义并无太大影响，但它可能会增加编译器的编写难度。

如今，词法作用域已成为常态——根据函数定义的环境求解 [自由变量 \(Free Variable\)](#)。词法作用域因需要 [闭包 \(closures\)](#) 的支持，所以难以实现支持一等函数的语言。但是，词法作用域可以使源码更易于阅读。Algol 之后出现的大多数语言都使用词法作用域，而 JavaScript 是个例外。在 JavaScript 中，特殊变量 `this` 最让人头疼。因为根据代码的编写方式，`this` 既可以使用词法 (lexically) 作用域，也可以使用动态作用域。

多年来，Python 的 `lambda` 表达式一直不支持 [闭包 \(closures\)](#)，导致 `lambda` 表达式在博客圈的函数式编程极客中口碑不佳。这个问题已在 Python 2.2 (2001 年 12 月) 中得到了修复，但博客圈的固有印象很难改变。从那时起，`lambda` 表达式就因为语法局限而一直处于尴尬的境地。

Python 装饰器与装饰器设计模式

Python 函数装饰器符合 Gamma 等人在《设计模式》中对装饰器的描述：“动态地为对象附加额外的职责。装饰器为扩展功能提供了除子类化之外的灵活选择”。

在实现层面上，Python 装饰器与经典的装饰器设计模式并不相似，但可以进行类比。

在装饰器设计模式中，`Decorator` 与 `Component` 都是抽象类。一个具体装饰器的实例会封装一个具体组件的实例，以便为具体组件添加行为。以下内容因于《设计模式》：

装饰器的接口与被其装饰的组件接口一致,因此装饰器的存在对于组件的客户端是透明的。装饰器将请求转发给组件,并可在转发之前或之后执行附加操作(如绘制边框)。这种透明性,使得你可以递归嵌套多个装饰器,从而可以添加任意数量的附加功能。

在Python中,装饰器函数相当于Decorator的[具体子类\(Concrete Subclass\)](#),它返回的内部函数是一个装饰器实例。返回的函数包装(wrap)了被装饰的函数,这类似于设计模式中的组件。返回的函数是透明的,因为它与原(未被装饰)函数接受相同的参数,符合组件的接口。它可以转发对组件的调用,也可以在组件之前或之后执行其他附加操作。因此,前面引文中的最后一句话可改成“透明性使得你可以堆叠多个装饰器,从而可以添加无限数量的行为。”

请注意,我并不是建议在Python程序中使用函数装饰器来实现装饰器模式。虽然在特定的情况下可以这样做,但一般来说,装饰器模式最好使用类来实现,以表示装饰器及其将包装的组件。

^a由John McCarthy发表的关于Lisp语言的论文,是一篇如同贝多芬第九交响乐一样伟大的杰作。

用一等函数实现设计模式

遵守设计模式 (Pattern) 与否，并不是衡量好坏的标准^a。

——Ralph Johnson,《设计模式》作者之一

^a摘自 Ralph Johnson 于 2014 年 11 月 15 日在圣保罗大学 IME/CCSL 上发表的演讲 “Root Cause Analysis of Some Faults in Design Patterns”

在软件工程中, **设计模式 (Design Pattern)** 是解决常见设计问题的通用方法。学习本章无需事先了解设计模式, 我会解释示例中用到的设计模式。

《设计模式: 可复用面向对象软件的基础》¹一书普及了设计模式在编程中的应用。该书涵盖了 23 种模式 (Pattern), 由以 C++ 代码为例的类编排而成。不过, 这些设计模式也可用于其他面向对象语言。

尽管设计模式与语言无关, 但这并不意味着每种模式都适用于每种语言。例如, “[十七 迭代器、生成器和经典协程](#)”将说明, 在 Python 中效仿迭代器模式是没有意义的, 因为 Python 语言本身已经内置了迭代器的功能, 并且可以以生成器的形式随时可用。生成器无需使用类即可工作, 而且需要的代码也比经典迭代器模式更少。

《设计模式》的作者在引言 (如下所示²) 中承认, 所用的编程语言决定了哪些模式可用:

编程语言的选择很重要, 因为它会影响一个人的观点。我们的设计模式假定采用了 Smalltalk/C++ 级别的语言特性, 这种假定实际上决定了哪些模式可以轻松实现, 哪些无法轻松实现。如果我们假定使用的是过程式语言, 那么我们可能就会包含名为“继承”、“封装”和“多态”的设计模式。同样, 一些不常见的面向对象语言可以直接支持我们的某些设计模式。例如, CLOS 多方法这一概念, 就可以简化 Visitor 等设计模式的实现。

Peter Norvig 在演讲 “Design Patterns in Dynamic Languages” (1996 年) 中指出, 在最初的《设计模式》一书中的 23 种模式中, 有 16 种在动态语言中变得“不可见或更简单”(幻灯片 9)。他说的动态语言是指

¹此书英文原名为《Design Patterns: Elements of Reusable Object-Oriented Software》(Addison-Wesley 出版), 由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides (又称“四人帮”) 合著, 是一本具有里程碑意义的经典著作。

²摘自《设计模式》第 4 页。

Lisp 与 Dylan, 但 Python 中也有许多相关的动态特性。特别是, 在使用一等函数的语言中, Norvig 建议要重新审视策略 (Strategy)、命令 (Command)、模板方法 (Template Method) 和访问者 (Visitor) 等经典模式。

本章的目标是展示在某些情况下, 函数是如何完成与类相同的工作, 并使代码更易读、更简洁。我们将使用函数作为对象重构 Strategy 的实现, 删除大量模板代码。我们还将讨论简化 Command 模式的类似方法。

10.1 本章新增内容

我将本章移到了“[二 延伸阅读](#)”的末尾, 以便在“[10.3 用装饰器改进策略模式](#)”的“[示例 10.9<287页>](#)”中应用一个注册装饰器, 并在该示例中使用 [类型提示 \(Type Hints\)](#)。本章中使用的大多数 [类型提示 \(Type Hints\)](#) 并不复杂, 但却有助于提高代码的可读性。

10.2 案例研究: 策略模式重构

在 Python 语言中, 可将函数视为一等对象, 因此可简化 Python 中的设计模式 (Design Pattern), Strategy 就是一个很好的例子。在下一节中, 我们将描述策略 (Strategy) 模式的作用, 并使用《[设计模式](#)》中描述的“经典”结构来实现策略 (Strategy) 模式。如果您熟悉经典模式, 可以跳到“[10.2.2 用函数实现策略模式](#)”, 在该节我们用函数重构了策略模式的代码, 显著减少了代码行数。

10.2.1 经典策略模式

图 10.1 中的 UML 类图描述了策略模式对类的编排。

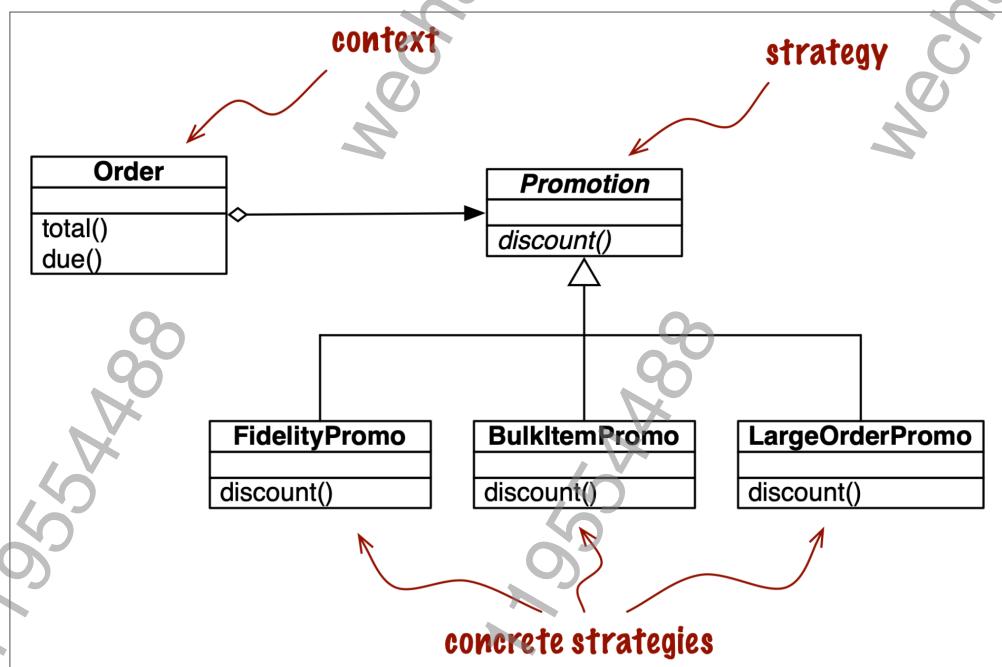


图 10.1: 用策略设计模式实现的订单折扣处理 UML 类图

《[设计模式](#)》书中对策略模式的总结如下所示:

定义一族算法^a,并将每个算法都封装在独立的类中,使得这些算法可以在运行时互相替换,而不影响客户端代码。策略模式使得算法的变化与使用算法的客户端代码相互独立,从而提高系统的灵活性与维护性。

^a一族算法 (a family of algorithms),即解决同一问题的不同方法。

在电商领域,应用策略模式的典型案例是:“根据客户属性或订单中的商品来计算订单折扣。”

假设某个网店制定了以下折扣规则:

- 积分 ≥ 1000 的顾客,每笔订单可享受订单总额 5% 的折扣。
- 同一订单中,单品数量 ≥ 20 的顾客,该种商品可享受 10% 的折扣。
- 订单中商品种类 ≥ 10 的顾客,可享受订单总额 7% 的折扣。

为简洁起见,我们假设一个订单只能享受一种折扣。策略模式的 UML 类图如图 10.1 所示,其中设计如下内容:

- 上下文 (Context)

通过将某些计算委托给执行替代算法的可互换组件来提供服务。在电商示例中,上下文是一个 Order(订单),它被配置为根据多种算法之一来应用促销折扣。

- 策略 (Strategy)

策略 (Strategy) 是实现不同算法的组件所共有的接口。在此示例中,名为 Promotion 的抽象类将扮演这一角色。

- 具体策略 (Concrete Strategy)

是策略 (Strategy) 的 **具体子类** (Concrete Subclass)。FidelityPromo、BulkPromo 与 LargeOrderPromo 是本示例实现的 3 个具体策略。

示例 10.1 中的代码遵循了图 10.1 中的方案。正如《设计模式》中所述,具体策略由上下文类的客户端来选择。在我们的示例中,在实例化订单之前,系统会以某种方式选择一种促销折扣策略,并将其传递给 Order 构造函数。而具体如何选择折扣策略,不在这个模式的职责范围内。

</> 示例 10.1: 实现 Order 类,支持可插入的折扣策略

```

1  from abc import ABC, abstractmethod
2  from collections.abc import Sequence
3  from decimal import Decimal
4  from typing import NamedTuple, Optional
5
6  class Customer(NamedTuple):
7      name: str
8      fidelity: int
9
10 class LineItem(NamedTuple):
11     product: str
12     quantity: int
13     price: Decimal
14
15     def total(self) -> Decimal:
16         return self.price * self.quantity

```

```

17
18 class Order(NamedTuple): # 上下文 (Context)
19     cart: Sequence[LineItem]
20     promotion: Optional['Promotion'] = None
21
22     def total(self) -> Decimal:
23         totals = (item.total() for item in self.cart)
24         return sum(totals, start=Decimal(0))
25
26     def due(self) -> Decimal:
27         if self.promotion is None:
28             discount = Decimal(0)
29         else:
30             discount = self.promotion.discount(self)
31         return self.total() - discount
32
33     def __repr__(self):
34         return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'
35
36 class Promotion(ABC): # 策略 (Strategy) : 一个抽象基类
37     @abstractmethod
38     def discount(self, order: Order) -> Decimal:
39         """返回折扣金额 (正值) """
40
41 class FidelityPromo(Promotion): # 第一个具体策略 (Concrete Strategy)
42     """为积分 ≥ 1000 的顾客, 提供订单总额 5% 的折扣"""
43
44     def discount(self, order: Order) -> Decimal:
45         rate = Decimal('0.05')
46         if order.customer.fidelity >= 1000:
47             return order.total() * rate
48         return Decimal(0)
49
50 class BulkItemPromo(Promotion): # 第二个具体策略 (Concrete Strategy)
51     """为单品数量 ≥ 20 的顾客, 提供该种商品 10% 的折扣"""
52
53     def discount(self, order: Order) -> Decimal:
54         discount = Decimal(0)
55         for item in order.cart:
56             if item.quantity >= 20:
57                 discount += item.total() * Decimal('0.1')
58         return discount
59
60 class LargeOrderPromo(Promotion): # 第三个具体策略 (Concrete Strategy)
61     """为订单中商品种类 ≥ 10 的顾客, 提供订单总额 7% 的折扣"""
62
63     def discount(self, order: Order) -> Decimal:
64         distinct_items = {item.product for item in order.cart}
65         if len(distinct_items) >= 10:
66             return order.total() * Decimal('0.07')

```

67 `return Decimal(0)`

注意, [示例 10.1](#) 中将 Promotion 定义为 [抽象基类 \(ABCs\)](#), 以便使用 `@abstractmethod` 装饰器。这样做, 可以更明确地表明所用的设计模式。

[示例 10.2](#) 是一些 doctest, 用于在实现了前述规则的模块中, 演示与验证相关操作。

</> [示例 10.2: 应用不同促销活动的 Order 类使用示例](#)

```

1  >>> joe = Customer('John Doe', 0)          ❶
2  >>> ann = Customer('Ann Smith', 1100)
3  >>> cart = (LineItem('banana', 4, Decimal('.5')), ❷
4  ...      LineItem('apple', 10, Decimal('1.5')),
5  ...      LineItem('watermelon', 5, Decimal(5)))
6  >>> Order(joe, cart, FidelityPromo())      ❸
7  <Order total: 42.00 due: 42.00>
8  >>> Order(ann, cart, FidelityPromo())      ❹
9  <Order total: 42.00 due: 39.90>
10 >>> banana_cart = (LineItem('banana', 30, Decimal('.5')), ❺
11 ...      LineItem('apple', 10, Decimal('1.5')))
12 >>> Order(joe, banana_cart, BulkItemPromo()) ❻
13 <Order total: 30.00 due: 28.50>
14 >>> long_cart = tuple(LineItem(str(sku), 1, Decimal(1)) ❼
15 ...      for sku in range(10))
16 >>> Order(joe, long_cart, LargeOrderPromo())    ❼
17 <Order total: 10.00 due: 9.30>
18 >>> Order(joe, cart, LargeOrderPromo())
19 <Order total: 42.00 due: 42.00>

```

- ❶ 两位顾客:joe 积分为 0, ann 积分为 1100。
- ❷ 一个购物车中包含 3 种商品。
- ❸ FidelityPromo 促销活动未给 joe 提供任何折扣。
- ❹ FidelityPromo 促销活动为 ann 提供了订单总额 5% 的折扣, 因为它的积分为 1100 (超过 1000)。
- ❺ banana_cart 有 30 串香蕉与 10 个苹果。
- ❻ BulkItemPromo 为 joe 购买的 30 串 (单品数量 ≥ 20) 香蕉提供了 1.5 美元的优惠。
- ❼ long_cart 有 10 种不同的商品, 每种售价 1 美元。
- ❽ LargeOrderPromo 为 joe 的整个订单提供了 7% 的折扣。

[示例 10.1](#) 可以运行地很好。但在 Python 中, 将函数视作对象来使用, 可以用更少的代码实现 [示例 10.1](#) 中同样的功能, 详见 “[10.2.2 用函数实现策略模式](#)”。

10.2.2 用函数实现策略模式

在 [示例 10.1](#) 中, 每个具体策略都是一个类, 并且每个类都只定义了一个方法, 即 `discount`。此外, 策略实例没有状态 (没有实例属性)。您可能会说, 这些策略实例看起来很像普通函数, 没错。[示例 10.3](#) 是对 [示例 10.1](#) 的重构, 用简单函数替换了中的具体策略, 并删除了抽象基类 `Promotion`。只需对 `Order` 类做少量修

改³。

```

</> 示例 10.3: 用函数实现折扣策略的 Order 类

1  from collections.abc import Sequence
2  from dataclasses import dataclass
3  from decimal import Decimal
4  from typing import Optional, Callable, NamedTuple
5
6  class Customer(NamedTuple):
7      name: str
8      fidelity: int
9
10 class LineItem(NamedTuple):
11     product: str
12     quantity: int
13     price: Decimal
14
15     def total(self):
16         return self.price * self.quantity
17
18 @dataclass(frozen=True)
19 class Order: # the Context
20     customer: Customer
21     cart: Sequence[LineItem]
22     promotion: Optional[Callable[['Order'], Decimal]] = None ❶
23
24     def total(self) -> Decimal:
25         totals = (item.total() for item in self.cart)
26         return sum(totals, start=Decimal(0))
27
28     def due(self) -> Decimal:
29         if self.promotion is None:
30             discount = Decimal(0)
31         else:
32             discount = self.promotion(self) ❷
33         return self.total() - discount
34
35     def __repr__(self):
36         return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'
37
38 ❸
39
40 def fidelity_promo(order: Order) -> Decimal:
41     """为积分 ≥ 1000 的顾客, 提供订单总额 5% 的折扣"""
42     if order.customer.fidelity >= 1000:

```

³由于 Mypy 的一个 bug, 我不得不用 @dataclass 重新实现 Order 类。你可以忽略这个细节, 但如果 Order 像 [示例 10.1](#) 那样继承 NamedTuple, 则 Mypy 0.910 在检查 promotion 的类型提示时将会崩溃。我尝试过为该行添加 "# type ignore", 但 Mypy 仍然崩溃。如果使用 @dataclass 构建 Order, 则 Mypy 可以正确处理同样的类型提示。截至 2021 年 7 月 19 日, [issues #9397](#) 仍未解决。希望您读到本文时, 此问题已被修复。

```

43     return order.total() * Decimal('0.05')
44     return Decimal(0)
45 def bulk_item_promo(order: Order) -> Decimal:
46     """为单品数量 ≥ 20 的顾客, 提供该种商品 10% 的折扣"""
47     discount = Decimal(0)
48     for item in order.cart:
49         if item.quantity >= 20:
50             discount += item.total() * Decimal('0.1')
51     return discount
52
53 def large_order_promo(order: Order) -> Decimal:
54     """为订单中商品种类 ≥ 10 的顾客, 提供订单总额 7% 的折扣"""
55     distinct_items = {item.product for item in order.cart}
56     if len(distinct_items) >= 10:
57         return order.total() * Decimal('0.07')
58     return Decimal(0)

```

- ❶ 此类型提示 (Type Hints) 表示: `promotion` 可以是 `None`, 也可以是接收一个 `Order` 参数并返回一个 `Decimal` 值的可调用对象。
- ❷ 调用可调用对象 `self.promotion`, 并为其传入 `self` 参数, 来计算折扣。原因参见后文的提示框“为何写成 `self.promotion(self)`”。
- ❸ 没用到 抽象基类 (ABCs), 直接用函数实现了不同的促销策略。
- ❹ 每个折扣策略都是由函数实现的。



为何写成 `self.promotion(self)`

在 `Order` 类中, `promotion` 并不是一种方法, 而是一个实例属性, 只不过它的属性值是一个可调用对象。因此, 表达式中 `self.promotion` 的作用就是获取这个可调用对象。要调用获取到的可调用对象, 必须为其提供一个 `Order` 实例, 即表达式中括号 () 内的 `self`。因此, 该表达式中出现了两个 `self`。

“[23.4 方法也是描述符](#)” 将介绍自动将方法绑定到实例的机制。不过, 该机制不适用于 `promotion`, 因为 `promotion` 不是方法。

示例 10.3 中的代码比“[示例 10.1](#)”简短。除此之外, 新的 `Order` 类使用起来也更简单, 如示例 10.4 中的 doctest 所示。

</> 示例 10.4: 以函数实现促销折扣的 `Order` 类使用示例

```

1  >>> joe = Customer('John Doe', 0)          ❶
2  >>> ann = Customer('Ann Smith', 1100)
3  >>> cart = [LineItem('banana', 4, Decimal('.5')),
4  ...           LineItem('apple', 10, Decimal('1.5')),
5  ...           LineItem('watermelon', 5, Decimal(5))]
6  >>> Order(joe, cart, fidelity_promo)        ❷
7  <Order total: 42.00 due: 42.00>
8  >>> Order(ann, cart, fidelity_promo)
9  <Order total: 42.00 due: 39.90>
10 >>> banana_cart = [LineItem('banana', 30, Decimal('.5')),

```

```

11 ... LineItem('apple', 10, Decimal('1.5'))]
12 >>> Order(joe, banana_cart, bulk_item_promo) ❶
13 <Order total: 30.00 due: 28.50>
14 >>> long_cart = [LineItem(str(item_code), 1, Decimal(1))
15 ... for item_code in range(10)]
16 >>> Order(joe, long_cart, large_order_promo)
17 <Order total: 10.00 due: 9.30>
18 >>> Order(joe, cart, large_order_promo)
19 <Order total: 42.00 due: 42.00>

```

- ❶ 同 [示例 10.2](#) 一样的测试用例。
- ❷ 要为 Order 实例应用折扣策略, 只需将促销策略函数 fidelity_promo 作为参数, 传递给 Order 类的构造函数即可。
- ❸ 这个测试使用了促销函数 bulk_item_promo, 下一个测试使用了另一个促销函数 large_order_promo。

请注意 [示例 10.4](#) 中的标号 (黑圈数字)—— 无需像 [示例 10.2](#) 那样为每个新订单都实例化一个新的促销对象, 那些用函数实现的促销函数随时可用。

值得注意的是, 《[设计模式](#)》一书指出: “策略对象通常是很好的享元 (flyweights)。”⁴ 该书对享元 (flyweights) 的定义是: “享元 (flyweights) 是一个可以同时在多个上下文中使用的共享对象。”⁵ 当对每个新上下文 (如电商案例中不同的 Order 实例) 反复应用相同的策略时, 建议将具体策略对象设计成享元对象, 以降低每次创建新的具体策略对象的开销。因此, 为了克服策略模式的缺点 (即运行时成本), 《[设计模式](#)》的作者建议应该同时应用多种设计模式, 如在应用策略模式的同时, 将具体策略对象设计成可共享的享元对象。但这样做, 代码的行数与维护成本会不断增加。

在更复杂的场景中, 当需要用具体策略维护内部状态时, 可能需要结合使用 策略 (Strategy) 模式与享元 (Flyweight) 模式的各种部分来处理。但是, 具体策略通常没有内部状态——它们只处理来自上下文的数据。若是此种情况, 一定要使用普通函数来代替单方法类实现单方法接口。函数比用户自定义类的实例更轻量, 而且不需要使用享元 (Flyweight) 模式来共享策略对象。因为, 每个策略函数只在 Python 进程加载模块时创建一次, 并且“函数也是可在多个上下文中使用的共享对象”。

现在, 我们已经用函数实现了策略模式, 因此也可能会出现其他更复杂的需求。假设您想创建一个“元策略”, 为给定的 Order 选择最佳可用折扣。接下来的几节中, 我们将继续重构, 利用函数与模块都是对象这一特点, 用不同的方式实现“元策略”这一需求。

10.2.3 选择最佳策略的简单方式

下面继续使用“[示例 10.4<283页>](#)”中的顾客与购物车, 并在此基础上增加 3 个测试, 如 [示例 10.5](#) 所示。

</> [示例 10.5](#): best_promo 函数计算所有折扣, 并返回幅度最大的那一个

```

1 >>> Order(joe, long_cart, best_promo) ❶
2 <Order total: 10.00 due: 9.30>
3 >>> Order(joe, banana_cart, best_promo) ❷
4 <Order total: 30.00 due: 28.50>

```

⁴ 见《[设计模式](#)》英文原版的第 323 页。

⁵ 见《[设计模式](#)》英文原版的第 196 页。

```

5  >>> Order(ann, cart, best_promo)
6  <Order total: 42.00 due: 39.90>

```

- ❶ best_promo 为客户 joe 选择了 larger_order_promo。
- ❷ 在这里, joe 因为订购了大量香蕉, 而从 bulk_item_promo 获得了折扣。
- ❸ 通过简单的购物车结账, best_promo 为忠实顾客 ann 提供了 fidelity_promo 的折扣。

示例 10.5 中 best_promo 的实现非常简单, 如示例 10.6 所示。

</> 示例 10.6: best_promo 在函数列表中遍历查找折扣最大的函数

```

1  promos = [fidelity_promo, bulk_item_promo, large_order_promo] ❶
2
3  def best_promo(order: Order) -> Decimal:
4      """计算可用的最佳折扣"""
5      return max(promo(order) for promo in promos) ❷❸

```

- ❶ promos 列出以函数实现的各个折扣策略, 该列表中保存的是函数对象。
- ❷ 与其他 *_promo 函数一样, 函数 best_promo 的参数也是一个 Order 实例。
- ❸ 使用生成器表达式, 将 order 传给 promos 列表中的各个折扣函数, 返回折扣幅度最大的那个函数。

示例 10.6 简洁明了, promos 是一个函数列表。一旦习惯了“函数是一等对象”这一概念, 构建容纳函数的数据结构(如示例 10.6 中的 promos)就自然而然地变得有意义了。

虽然示例 10.6 可以正常运行, 且易于阅读, 但其中的一些重复, 可能会导致不易察觉的 bug: 当添加新的促销策略时, 不仅需要编写相应的促销函数, 还要记得将其添加到 promos 列表中。否则, 新的促销函数只能显式作为参数传递给 Order, 因为 best_promo 不知道新促销函数的存在。

要解决此问题, 可有多种方案, 请继续阅读下一节。

10.2.4 找出模块中的所有策略

Python 中的模块也是一等对象, 而且标准库还提供了一些可处理模块的函数。Python 文档中对内置函数 `globals` 的描述如下:

`globals()`

返回当前模块命名空间的字典。对于函数内的代码来说, 该字典是在函数被定义时设置的。并且无论函数在何处被调用, 该字典都保持不变。

示例 10.7 利用 `globals()` 函数帮助 `best_promo` 自动找到模块中其他可用的 *_promo 函数, 方法有些粗暴。

</> 示例 10.7: 通过检查模块全局命名空间来构建的 promos 列表

```

1  from decimal import Decimal
2  from strategy import Order
3  from strategy import (
4      fidelity_promo, bulk_item_promo, large_order_promo ❶
5  )

```

```

6
7 promos = [promo for name, promo in globals().items()]
8     if name.endswith('_promo') and
9         name != 'best_promo'
10    ]
11
12 def best_promo(order: Order) -> Decimal:
13     """计算可用的最佳折扣"""
14     return max(promo(order) for promo in promos)

```

- ❶ 从 strategy 模块导入促销折扣函数,使其在全局命名空间中可用⁶。
- ❷ 遍历 globals() 返回的 dict 中的各项。
- ❸ 只选择以 _promo 结尾的值。
- ❹ 过滤掉 best_promo 本身,以避免在调用 best_promo 时出现无限递归。
- ❺ best_promo 没有变化。

收集可用促销的另一种方法是,在一个单独的模块中保存所有策略函数 (best_promo 除外)。

在 [示例 10.8](#) 中,唯一显著的变化是,策略函数列表是通过对一个名为 promotions 的独立模块进行 **内省** (Introspection) 而构建的。注意,示例 10.8 需要导入 promotions 模块,以及提供高级 **内省** (Introspection) 函数的 inspect 模块。

</> [示例 10.8:](#) 对 promotions 模块进行内省,构建 promos 列表

```

1 from decimal import Decimal
2 import inspect
3
4 from strategy import Order
5 import promotions
6
7 promos = [func for _, func in inspect.getmembers(promotions, inspect.isfunction)]
8
9 def best_promo(order: Order) -> Decimal:
10     """计算可用的最佳折扣"""
11     return max(promo(order) for promo in promos)

```

函数 inspect.getmembers 按名称排序,以“(name,value)”列表的形式返回对象(本例为 promotions 模块)中的所有成员。可选的谓词参数(一个布尔值函数),可对返回的成员进行过滤筛选。本例的谓词参数是 inspect.isfunction,即只获取 promotions 模块中的函数。

[示例 10.8](#) 的正常运行不受策略函数名称的影响。唯一的要求是, promotions 模块中只能包含根据订单计算折扣的函数。当然,这是代码的一个隐含假设。如果有人在 promotions 模块中用不同的签名来定义函数,那么 best_promo 在尝试将该函数应用于订单时,将会出错。

我们可以添加更为严格的测试来过滤函数,例如检查它们的参数。[示例 10.8](#) 的重点不是提供一个完整的解决方案,而是强调模块 **内省** (Introspection) 的一种可能用法。

动态收集促销折扣函数的一个更明确的替代方案是使用一个简单的装饰器,详见下一节。

⁶flake8 与 VS Code 都会发出警告,指出虽然导入了这些名称,但未被使用。根据定义,静态分析工具无法理解 Python 的动态特性。如果听取这些工具的所有建议,我们很快就会用 Python 语法编写出与 Java 类似的严格而冗长的代码。

10.3 用装饰器改进策略模式

回想一下,“[示例 10.6<285页>](#)”的主要问题是,策略函数的定义中有函数名称,函数 `best_promo` 用来判断哪个折扣幅度最大的 `promos` 列表中也有函数名称。这种重复是有问题的,因为新增策略函数后可能会忘记将其加入到 `promos` 列表中。这将导致 `best_promo` 会悄无声息地忽略新增的策略函数,为系统引入了不易察觉的 bug。[示例 10.9](#) 使用“[9.4 注册装饰器<250页>](#)”介绍的技术解决了这个问题。

</> [示例 10.9: 用装饰器 promotion 填充 promos 列表中的值](#)

```
1 Promotion = Callable[[Order], Decimal]          ❶
2
3 promos: list[Promotion] = []                   ❷
4
5 def promotion(promo: Promotion) -> Promotion:
6     promos.append(promo)
7     return promo
8
9 def best_promo(order: Order) -> Decimal:
10    """Compute the best discount available"""
11    return max(promo(order) for promo in promos) ❸
12
13 @promotion
14 def fidelity(order: Order) -> Decimal:
15    """5% discount for customers with 1000 or more fidelity points"""
16    if order.customer.fidelity >= 1000:
17        return order.total() * Decimal('0.05')
18    return Decimal(0)
19
20 @promotion          ❹
21 def bulk_item(order: Order) -> Decimal:
22    """10% discount for each LineItem with 20 or more units"""
23    discount = Decimal(0)
24    for item in order.cart:
25        if item.quantity >= 20:
26            discount += item.total() * Decimal('0.1')
27    return discount
28
29 @promotion
30 def large_order(order: Order) -> Decimal:
31    """7% discount for orders with 10 or more distinct items"""
32    distinct_items = {item.product for item in order.cart}
33    if len(distinct_items) >= 10:
34        return order.total() * Decimal('0.07')
35    return Decimal(0)
```

- ❶ `Promotion` 是一个 `Callable` 类型 (详见“[8.5.11 Callable<237页>](#)”), 参数为 `Order` 类型, 返回值为 `Decimal` 类型。
- ❷ `promos` 列表位于模块全局命名空间中,起初是空的。

- ③ Promotion 是一个注册装饰器：它在将 promo 添加到 promos 列表后，会原封不动地返回 promo 函数（即策略函数）。
- ④ best_promo 无需更改，因为它依赖于 promos 列表。
- ⑤ 任何由 @promotion 装饰的函数，都将被添加到 promos 列表中。

与前面几种方案对比，此种方案具有如下优点：

- 促销策略函数不必使用特殊名称，如无需以 `_promo` 为名称后缀。
- `@promotion` 装饰器突出了被装饰的函数的作用，还便于临时禁用某个促销策略：将装饰器语句注释掉即可。
- 促销折扣策略可以在其他模块中定义——系统中任何位置都可以，只要为策略函数应用了 `@promotion` 装饰器即可。

下一节，我们将讨论命令（Command）设计模式。该设计模式可通过单方法类来实现，也可通过普通函数来实现。

10.4 命令模式

命令（Command）设计模式也可以通过将函数作为参数传递来简化。图 10.2 展示了 Command 设计模式对类的编排。

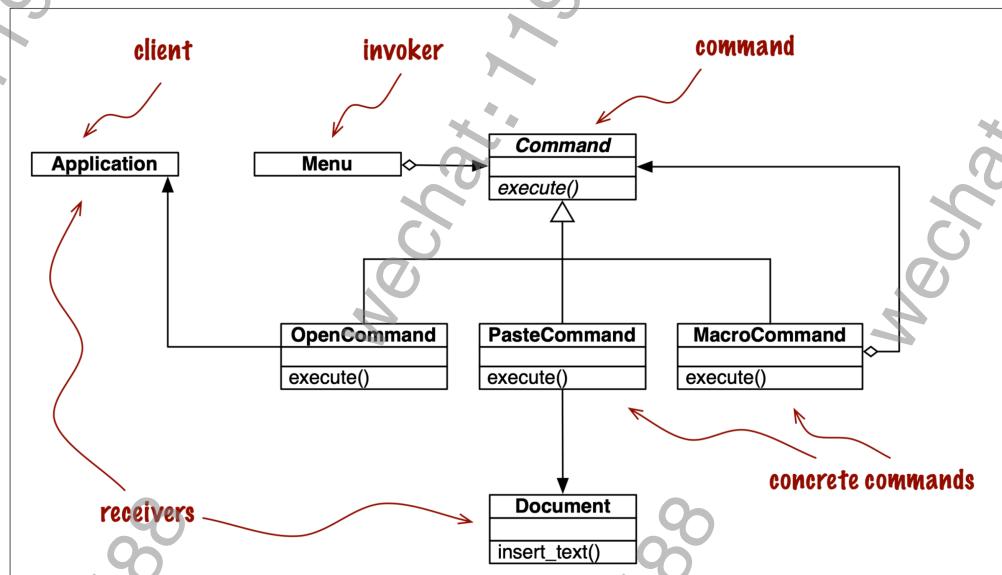


图 10.2：菜单驱动的文本编辑器的 UML 类图，用命令设计模式实现⁷

Command 设计模式的目标是将调用操作的对象（调用者）与实现该操作的提供者对象（接收者）进行解耦（即分离）。在《设计模式》一书的示例中，调用者是图形化应用程序中的菜单项，而接收者则是正在被编辑的文档或应用程序自身。

Command 设计模式的思路是在调用者与接收者之间放置一个 Command 对象，该对象实现一个只有一个方法的接口。该接口调用接收者提供的方法，执行所需的操作。这样，调用者就无需了解接收者的接

⁷ 各个命令可以有不同的接收者，即实现操作的对象。对 PasteCommand 来说，接收者是 Document；对 OpenCommand 来说，接收者是应用程序。

口，并且可以通过不同的 Command 子类适配不同的接收者。调用者被配置为一个具体的命令（concrete command），通过调用它的 execute() 方法来执行这个命令。注意，图 10.2 中的 MacroCommand 可能会存储一个命令对象的序列，MacroCommand 的 execute() 方法会依次调用序列中每个命令对象的 execute() 方法。

《设计模式》中提到“命令模式是回调机制的面向对象替代品”。问题是，我们需要回调机制的面向对象替代品么？答案是：有时需要，有时不需要。

与其为调用者提供一个 Command 实例，不如简单地给它一个函数。调用者可以直接调用 command() 函数，而无需调用 command.execute() 方法。MacroCommand 可以用一个定义了 __call__ 方法的类来实现。MacroCommand 的实例将是可调用对象，每个 MacroCommand 实例都维护着一个可供将来调用的函数列表，如 [示例 10.10](#) 所示。

</> [示例 10.10](#): 每个 MacroCommand 实例都在内部维护一个命令列表

```

1  class MacroCommand:
2      """A command that executes a list of commands"""
3
4      def __init__(self, commands):
5          self.commands = list(commands) ❶
6
7      def __call__(self):
8          for command in self.commands: ❷
9              command()

```

❶ 根据 [实参 \(Argument\)](#) commands 构建一个列表，可确保 self.commands 是一个可迭代对象，并在每个 MacroCommand 实例中保留命令引用的本地副本。

❷ 调用 MacroCommand 实例时，会按序调用 self.commands 中的每个命令。

Command 模式的高级用法（如支持撤消操作）可能需要的不仅仅是简单的回调函数。因此，可以考虑使用 Python 提供的几个替代品。

- 像 [示例 10.10](#) 中 MacroCommand 这样的可调用实例，可以保存任何所需的状态，并且除了 __call__，还可以提供其他方法。
- 可以用 [闭包 \(closures\)](#) 在调用之间保存函数的内部状态。

使用一等函数对 Command 模式重新审视到此结束。这里采用的方式与 Strategy 模式所用的方式类似：将实现单方法接口的类的实例，替换为可调用对象。毕竟，每个 Python 可调用对象都实现了一个单方法（即 __call__）接口。

10.5 本章小结

在《设计模式》出版多年后，Peter Norvig 指出：“在 23 种模式中，有 16 种模式在 Lisp 或 Dylan 中的实现比在 C++ 中的实现（至少在某些方面）要简单得多，而且还能保持同等的质量。”⁸Python 拥有 Lisp 和 Dylan 语言的一些动态特性，尤其是一等函数，这也是本书此部分的重点。

⁸ 摘自 Norvig 的演讲“Design Patterns in Dynamic Languages”（幻灯片 9）。

本章开头引用的那句话“遵守设计模式 (Pattern) 与否，并不是衡量好坏的标准”，是 Ralph Johnson 在纪念《设计模式》英文原书出版 20 周年活动上所说的，他还指出了该书的缺点之一：“过多强调设计模式的结果，而未详细说明过程。”⁹在本章中，我们以 Strategy 模式作为起点，使用一等函数简化了 Strategy 模式的实现方式。

在许多情况下，使用函数或可调用对象在 Python 中实现回调比模仿《设计模式》一书中由 Gamma、Helm、Johnson 和 Vlissides 描述的策略 (Strategy) 模式或命令 (Command) 模式更自然。本章节中对策略 (Strategy) 模式的重构和对命令 (Command) 模式的讨论是为了通过示例说明一种更为常见的做法：有时，设计模式或 API 要求组件实现一个单方法接口，而该方法具有类似“execute”、“run”或“do_it”的宽泛名称。这样的设计模式或 API，在 Python 中通常可以使用可作为一等对象的函数来实现，从而减少样板代码。

10.6 延伸阅读

《Python Cookbook (第 3 版)》的 8.21 节用一种优雅的方式实现了访问者 (Visitor) 设计模式，其中的 NodeVisitor 类将方法视为一等对象来处理。

在设计模式方面，Python 程序员的阅读选择不像其他编程语言那么多。

据我所知，《Learning Python Design Patterns》(Gennadiy Zlobin 著)是目前唯一一本专门讲解 Python 设计模式的书。不过，该书很薄 (100 页)，只涵盖了最初 23 种设计模式中的 8 种。

《Python 高级编程第 2 版》¹⁰是市场上最值得读的中级 Python 书籍之一，其最后一章“有用的设计模式”从 Python 的角度介绍了几种经典设计模式。

Alex Martelli 曾多次发表有关 Python 设计模式的演讲。在他的个人网站上有一段 2011 年的 [EuroPython 演讲视频](#) 和 [一组幻灯片](#)。这些年，我找到过不同的幻灯片与视频，长短不一。因此，值得用“Python Design Patterns”这些关键词仔细搜索他的名字。有位出版商告诉我，Martelli 正在写一本关于设计模式的书。如果出版，我一定会拜读。

有关 Java 设计模式的书籍很多，但其中我最喜欢的是《Head First 设计模式 (第 2 版)》¹¹。概述讲解了 23 种经典设计模式中的 16 种。如果您喜欢 Head First 系列丛书的古怪风格，比需要了解设计模式这一主题，那么您一定会喜欢这部作品。该书以 Java 为中心，但第 2 版进行了更新，涵盖了 Java 中新增的一等函数，从而使一些示例更接近 Python 版本。

如果想换个新角度，从支持鸭子类型与一等函数的动态语言视角重新审视设计模式，《Ruby 设计模式》¹² (Russ Olsen 著) 中有许多见解同样适用于 Python。尽管 Python 与 Ruby 在语法上有许多差异，但二者在语义层面上很接近 (比 Java 或 C++ 更接近)。

在“Design Patterns in Dynamic Languages”(幻灯片)中，Peter Norvig 展示了如何用一等函数 (与其他动态语言特性) 简化几个经典设计模式的实现。

《设计模式》一书的“引言”部分很值得阅读，甚至比书中包含的 23 种设计模式 (涵盖了从非常重要到很少使用的各种模式的设计方案) 的目录更有价值。这个“引言”部分提供了广为引用的两个设计原则：“面向接口编程，而不是面向实现编程”和“优先使用对象组合，而不是类继承”。

⁹“Root Cause Analysis of Some Faults in Design Pattern”由 Johnson 于 2014 年 11 月 15 日在 IME-USP 上发表。

¹⁰《Python 高级编程》英文原版书名为《Expert Python Programming》(Tarek Ziade 著)。

¹¹《Head First 设计模式 (第 2 版)》英文原版书名为《Head First Design Patterns》(Eric Freeman 与 Elisa-beth Robson 著)。

¹²《Ruby 设计模式》英文原版书名为《Design Patterns in Ruby》(Russ Olsen 著)。

将模式应用于设计的思路,起源于建筑师 Christopher Alexander 等人的著作《模式语言》¹³ (牛津大学出版社)。Alexander 的想法是创建一个标准的表现形式,使设计团队在设计建筑物时能够共享共同的设计决策。M. J. Dominus 在演讲 “*“Design Patterns” Aren’t*” 中认为 Alexander 最初对模式的看法更深刻、更人性化,并且也适用于软件工程领域。

杂谈

Python 拥有一等函数和一等类型, Norvig 声称这些特性影响了 23 种模式中的 10 种 (“*Design Patterns in Dynamic Languages*” 幻灯片 10)。在 “*九 装饰器与闭包*” 中, 我们看到 Python 也支持 *泛化函数(Generic Function)* (“*9.9.3 单分派泛化函数<264页>*”), 这是 CLOS 多方法的一种有限形式。CLOS 多方法是 Gamma 等人建议作为实现经典 Visitor 模式的更简单方法。另一方面, Norvig 认为多方法简化了 Builder 模式(幻灯片 10)。将设计模式与语言特性相匹配, 并不是一门精确的科学。

在世界各地的课堂上, 经常使用 Java 示例来进行设计模式的教学。我不止一次听到学生声称:“他们被误导认为:原始设计模式在任何编程语言中都是有用的。”事实证明,《*设计模式*》中的 23 种“经典”设计模式非常适用于 Java 语言——尽管其中的大多数模式是用 C++ 展示的, 而少量示例是用 Smalltalk 展示的。但这并不意味着所有这些模式都同样适用于任何编程语言。作者在书的开头就明确指出:“一些不常见的面向对象语言可以直接支持我们的某些设计模式。”(请回顾本章第一页的完整引文)。

与 Java、C++ 或 Ruby 相比, 有关设计模式的 Python 参考书都非常薄。“*10.6 延伸阅读<290页>*”中, 我提到了 Gennadiy Zlobin 所著的《*Learning Python Design Patterns*》, 该书于 2013 年 11 月出版。相比之下, Russ Olsen 的《*Ruby 设计模式*》于 2007 年出版, 英文版有 384 页, 比 Zlobin 的作品多出了 284 页。

如今, Python 在学术界越来越流行, 希望以后会有更多以 Python 语言讲解设计模式的书籍。此外, Java 8 引入了方法引用和匿名函数, 这些备受期待的特性很可能为 Java 中的设计模式催生出新的实现方式——我们要认识到, 随着语言的发展, 运用经典设计模式的方式也必定随之进化。

夺命连环 call

在为本书做最后润色时, 本书技术审校 Leonardo Rochael 提出一个有趣的问题:

如果函数有 `__call__` 方法, 并且方法也是可调用对象。那么, `__call__` 方法是否也有 `__call__` 方法?

我不知道他的发现是否有用, 但这是一个有趣的事:

```

1  >>> def turtle():
2      ...     return 'eggs'
3
4  ...
5  >>> turtle()
6  'eggs'
7  >>> turtle.__call__()
8  'eggs'
9  >>> turtle.__call__.__call__()

```

¹³《模式语言》英文原版书名为《*A Pattern Language*》。

```
9 'eggs'  
10 >>> turtle.__call__.__call__.__call__()  
11 'eggs'  
12 >>> turtle.__call__.__call__.__call__.__call__()  
13 'eggs'  
14 >>> turtle.__call__.__call__.__call__.__call__.__call__()  
15 'eggs'  
16 >>> turtle.__call__.__call__.__call__.__call__.__call__.__call__()  
17 'eggs'  
18 >>> turtle.__call__.__call__.__call__.__call__.__call__.__call__.__call__()  
19 'eggs'
```

子子孙孙，无穷匮也。

PART III

第三部分

类与协议

- 第 11 章 [Pythonic 对象](#)
- 第 12 章 [序列的特殊方法](#)
- 第 13 章 [接口、协议与抽象基类](#)
- 第 14 章 [继承的利与弊](#)
- 第 15 章 [类型注解进阶 1136](#)
- 第 16 章 [运算符重载](#)

wechat: 119554488

一个库或框架是否符合 Python 风格,要看它能否让 Python 程序员以一种轻松而自然的方式执行任务。^a

——Martijn Faassen, Python 与 JavaScript 框架创建者

^a摘自 Faassen 题为“What is Pythonic?”的博客文章。

得益于 Python 数据模型,用户定义类型也可以像 Python 内置类型一样自然地运行。本着 鸭子类型 (Duck Typing) 的精神,自定义类型的这种丝滑可以在不依赖继承的情况下实现——只需自定义类型实现对象按预期运行所需的方法即可。

在前面几章中,我们学习了许多内置对象的行为。现在,我们将构建行为与真正的 Python 对象一样的用户定义类。您的应用程序中可能不需要,也不应该实现本章示例中这么多的特殊方法。但是,如果您正在编写一个库或框架,那么用户可能会期望它们的行为与 Python 提供的类一样。满足了这种预期就是 Pythonic 的一种方式。

本章接续“[一 Python 数据模型](#)”,展示如何实现不同类型 Python 对象中常见的特殊方法。

在本章中,我们将了解到:

- 支持将对象转换为其他类型的内置函数(如 `repr()`、`bytes()`、`complex()` 等)。
- 通过类方法实现备选的构造函数。
- 扩展由 `f-strings`、内置 `format()` 函数、`str.format()` 方法使用的格式化微语言。
- 提供对属性的只读访问权限。
- 使对象 [可哈希 \(hashable\)](#),以便在 `set` 中使用或在 `dict` 的键中使用。
- 利用 `__slots__` 节省内存。

我们将在开发 `Vector2d`(一种简单的二维欧几里得向量类型)类时,完成所有这些工作。`Vector2d` 类将成为“[十二 序列的特殊方法](#)”中 N 维向量类的基础。

在实现 `Vector2d` 类的过程中,将会讨论如下两个概念:

- 如何以及何时使用 `@classmethod` 和 `@staticmethod` 装饰器。
- Python 中的私有属性与受保护属性:用法、约定与限制。

11.1 本章新增内容

本章的第二段增加了新的引言,还增加了一些词汇以阐述 [Pythonic](#) 的概念——在第 1 版中,此概念在本章最后才讨论。

更新了“[11.6 格式化显示](#)”一节,增加了 Python 3.6 引入的 f-strings。此处改动不大,因为 f-strings 支持的格式化微语言与内置函数 `format()`、`str.format()` 方法相同。因此,以前实现的 `__format__` 方法也适用于 f-strings。

本章的其余部分几乎没有变化——毕竟自 Python 3.0 以来,特殊方法大多都没什么变化,而且核心理念在 Python 2.2 中就出现了。

现在,让我们从对象表示方法开始。

11.2 对象表示形式

每种面向对象语言都至少有一种从对象获取字符串表示形式的标准方法。而 Python 提供了两种方法:

- `repr()`

以“便于开发人员理解的方式”,返回对象的字符串表示形式。Python 控制台或调试器在显示对象时,会采用这种形式。

- `str()`

以“便于用户理解的方式”,返回对象的字符串表示形式。通过 `print()` 函数输出对象时,会采用这种形式。

正如“[一 Python 数据模型](#)”所述,背后支持 `repr()` 与 `str()` 的特殊方法分别是 `__repr__` 与 `__str__`。

除此之外,还有两个特殊方法(`__bytes__` 和 `__format__`)可为对象提供其他表示形式。`__bytes__` 方法类似于 `__str__`,它由 `bytes()` 调用以获取对象的字节序列表示形式。而 `__format__` 供 f-strings、内置 `format()` 函数、`str.format()` 方法使用,通过调用 `obj.__format__(format_spec)` 以特定的格式化代码显示对象的字符串表示形式。本章将先讨论 `__bytes__` 方法,随后再讨论 `__format__` 方法。



如果您是 Python 2 用户,请记住在 Python 3 中, `__repr__`、`__str__` 与 `__format__` 必须始终返回 Unicode 字符串(`str` 类型)。只有 `__bytes__` 应该返回字节序列(`bytes` 类型)。

11.3 再谈 Vector 类

为了演示用于生成对象表示形式的多种方法,我们将使用一个与“[一 Python 数据模型](#)”中类似的 `Vector2d` 类。我们将在本节及后续的几节中逐步构建这个类。我们预期 `Vector2d` 类具有的行为如 [示例 11.1](#) 所示。

</> [示例 11.1](#): `Vector2d` 实例的多种字符串表示形式

```
1  >>> v1 = Vector2d(3, 4)
2  >>> print(v1.x, v1.y)
```

```

3 3.0 4.0
4 >>> x, y = v1
5 >>> x, y
6 (3.0, 4.0)
7 >>> v1
8 Vector2d(3.0, 4.0)
9 >>> v1_clone = eval(repr(v1))
10 >>> v1 == v1_clone
11 True
12 >>> print(v1)
13 (3.0, 4.0)
14 >>> octets = bytes(v1)
15 >>> octets
16 b'd\x00\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x00\x10@'
17 >>> abs(v1)
18 5.0
19 >>> bool(v1), bool(Vector2d(0, 0))
20 (True, False)

```

- ① Vector2d 实例的分量可以通过属性直接访问（无需调用 getter 方法）。
- ② Vector2d 可以解包为一个变量元组。
- ③ Vector2d 的 repr 模拟了构建实例的源代码。
- ④ 此处使用 eval¹ 表明 Vector2d 的 repr 表示形式是对其构造函数调用的准确表示。
- ⑤ Vector2d 支持使用 == 进行比较；这对测试非常有用。
- ⑥ print() 函数调用 str() 函数，对于 Vector2d，str 会生成“有序对”。
- ⑦ bytes() 使用 __bytes__ 方法生成实例的二进制表示形式。
- ⑧ abs 使用 __abs__ 方法返回 Vector2d 的模（大小）。
- ⑨ bool 使用 __bool__ 方法。对于模为零的 Vector2d 实例，bool() 会返回 False，否则返回 True。

示例 11.1 中的 Vector2d 在 vector2d_v0.py（示例 11.2）文件中实现。这段代码基于“示例 1.2<10页>”，支持 + 运算与 * 运算的特殊方法将在“[十六 运算符重载](#)”中实现。为了便于测试，示例 11.2 增加了支持 == 运算符的方法。现在，Vector2d 使用了几种特殊方法，以提供 Python 用户在设计良好的对象中所期望的操作。

</> 示例 11.2: vector2d_v0.py: 目前定义的都是特殊方法

```

1 from array import array
2 import math
3
4 class Vector2d:
5     typecode = 'd'
6
7     def __init__(self, x, y):
8         self.x = float(x)
9         self.y = float(y)
10
11     def __iter__(self):

```

¹ 在这里使用 eval 克隆对象只是为了说明 repr 的问题；要克隆一个实例，使用 copy.copy 函数更安全、更快捷。

```

12     return (i for i in (self.x, self.y))      ③
13 def __repr__(self):           class_name = type(self).__name__
14     return '{}({!r}, {!r})'.format(class_name, *self)  ④
15
16 def __str__(self):
17     return str(tuple(self))      ⑤
18
19 def __bytes__(self):
20     return (bytes([ord(self.typecode)]) +      ⑥
21             bytes(array(self.typecode, self)))  ⑦
22
23 def __eq__(self, other):
24     return tuple(self) == tuple(other)      ⑧
25
26 def __abs__(self):
27     return math.hypot(self.x, self.y)      ⑨
28
29 def __bool__(self):
30     return bool(abs(self))      ⑩

```

- ❶ `typecode` 是类属性, 在 `Vector2d` 实例与字节序列之间转换时使用。
- ❷ 在 `__init__` 方法中将 `x` 和 `y` 转换为 `float`, 可以尽早捕获错误, 这在使用不合适的参数调用 `Vector2d` 时很有帮助。
- ❸ `__iter__` 方法使 `Vector2d` 实例变成可迭代对象, 这样 `Vector2d` 实例才能解包 (如 `x, y = my_vector`)。该方法的实现方式很简单, 直接调用生成器表达式, 逐个生成分量即可²。
- ❹ `__repr__` 通过使用 `{!r}` 插入分量来构建字符串以获得对象的 `repr` 字符串表示形式; 因为 `Vector2d` 实例是可迭代对象, 所以 `*self` 会将 `x` 分量与 `y` 分量提供给 `format` 方法。
- ❺ 从可迭代的 `Vector2d` 实例中, 可以轻松构建一个元组, 显示为一个“有序对”。
- ❻ 为了生成字节序列, 我们需要将 `typecode` 转换为字节序列。
- ❼ 迭代 `Vector2d` 实例, 得到一个数组, 再将数组转换成字节序列。最后, 拼接 ❻ 与 ❼ 生成的字节序列。
- ❽ 为了快速比较所有分量, 将目标对象转换为元组。对于 `Vector2d` 实例来说, 虽然可以这么做, 但仍有问题 (参见后文的警告框)。
- ❾ 模是以分量 `x` 与 `y` 为直角边, 构成的直角三角形的斜边长 (模 = $\sqrt{self.x^2 + self.y^2}$)。
- ❿ `__bool__` 使用 `abs(self)` 计算 `Vector2d` 实例的模; 然后, 将模转换为 `bool`。因此, 0.0 为 `False`, 非零值为 `True`。



示例 11.2 中的 `__eq__` 方法, 适用于操作数都是 `Vector2d` 实例。但是, 当将 `Vector2d` 实例与具有相同值的其他可迭代对象进行比较时 (如, `Vector(3,4)==[3,4]`), 比较结果也是 `True`。此行为既可被视为是特性, 也可被视为是 bug。“[十六 运算符重载](#)”在讲到运算符重载时, 将进一步讨论。

我们已经定义了一套相对完整的基本方法, 但仍缺少一个操作: 用 `bytes()` 生成的二进制表示形式重建

²此行也可以写成 `yield self.x; yield self.y`。关于 `__iter__` 特殊方法、生成器表达式和 `yield` 关键字, 我们在“[十七 迭代器、生成器和经典协程](#)”中还有更多内容要讲。

Vector2d 实例。

11.4 后备的构造函数

既然可以以字节序列的方式导出 Vector2d 实例，自然就需要一个从二进制序列导入 Vector2d 实例的方法。从标准库中寻找灵感，我们发现 `array.array` 中有一个 `.frombytes` 类方法 可以满足需求——在“表 2.3<51页>”中见过。我们为 `vector2d_v1.py` 中的 `Vector2d` 类也定义了一个同名的类方法 `frombytes`，如示例 11.3 所示。

</> 示例 11.3: `vector2d_v1.py` 的一部分: 将被添加到 `vector2d_v0.py` (示例 11.2) 的 `frombytes` 类方法

```

1  @classmethod
2  def frombytes(cls, octets):
3      typecode = chr(octets[0])
4      memv = memoryview(octets[1:]).cast(typecode)
5      return cls(*memv)

```

- ❶ 被 `@classmethod` 装饰的方法，可在类中直接调用。
- ❷ 第一个参数不是 `self`，而是类自身（习惯命名为 `cls`）。
- ❸ 从第一个字节读取 `typecode`。
- ❹ 根据传入的 `octets` 字节序列，创建一个 `memoryview`；然后，使用 `typecode` 对 `memoryview` 进行转换³。
- ❺ 将转换产生的 `momoryview`，解包为构造函数所需的参数对。

刚刚用到的 `@classmethod` 装饰器 是 Python 特有的，接下来对其进行讲解。

11.5 classmethod 与 staticmethod

Python 教程中没有提到 `@classmethod` 装饰器，也没有提到 `@staticmethod`。学过 Java 面向对象编程的人可能觉得奇怪，为何 Python 同时提供这两个装饰器，而不是只提供其中一个？

先来看看 `@classmethod` 装饰器。示例 11.3 展示了它的用法——定义一个对类（而不是对实例）进行操作的方法。`@classmethod` 装饰器 改变了方法的调用方式，因此接收的第一个参数是类本身（习惯写为 `cls`），而不是实例。`@classmethod` 最常见的用途是定义一个备选的构造函数，如示例 11.3 中的 `frombytes`。请注意，`frombytes` 的最后一行，实际上是通过调用 `cls` 参数来构建一个新实例的：`cls(*memv)`。

相比之下，`@staticmethod` 装饰器 也会改变方法的调用方式，使方法不接收特殊的第一参数（既不是类本身，也不是实例）。本质上，被 `@staticmethod` 装饰过的方法，就是一个普通函数。只不过此函数刚好定义在类主体中，而不是在模块级别内。示例 11.4 对比了 `@classmethod` 和 `@staticmethod` 的操作。

</> 示例 11.4: 比较 `classmethod` 与 `staticmethod` 的行为

```

1  >>> class Demo:
2  ...     @classmethod
3  ...     def klassmeth(*args):    # 位置参数列表中的首个参数始终是 Demo 类
4  ...         return args

```

³ 在“2.10.2 memoryview”简要介绍了 `memoryview`，并解释了其 `.cast` 方法。

```

5 ... @staticmethod
6 ... def statmeth(*args):...           return args ②
7 ...
8 >>> Demo.klassmeth()           ③
9 (<class '__main__.Demo'>,
10 >>> Demo.klassmeth('spam')
11 (<class '__main__.Demo'>, 'spam')
12 >>> Demo.statmeth()           ④
13 ()
14 >>> Demo.statmeth('spam')
15 ('spam',)

```

- ① klassmeth 返回所有位置参数。
- ② statmeth 也返回所有位置参数。
- ③ 无论如何调用, Demo.klassmeth 的第一个参数都是 Demo 类。
- ④ Demo.statmeth 的行为与普通的函数一样。



`@classmethod` 装饰器 非常有用, 但我从未经过必须使用 `@staticmethod` 装饰器的场景。有些函数即使不直接处理类, 也与类联系紧密。因此, 您可能会考虑将函数与类定义在一起。对于这种场景, 在类的前面或后面定义函数, 保持类与函数在同一个模块中, 基本就足够了^a。

^a本书的技术审校 Leonardo Rochael 不认同我对 staticmethod 的评价, 并向我推荐 Julien Danjou 的博文 “The Definitive Guide on How to Use Static, Class or Abstract Methods in Python” 作为反驳的依据。Danjou 的博文非常好, 我也向您推荐。但该博文不足以改变我对 staticmethod 的看法。所以, 您必须自己做出判断。

现在, 我们已经了解了 `@classmethod` 的优点(而 `@staticmethod` 不是很有用), 让我们回到对象表示形式的问题, 看看如何支持格式化输出。

11.6 格式化显示

f-strings、内置函数 `format()` 与 `str.format()` 方法会将各种类型的格式化方式委托给相应的`__format__(format_spec)` 方法。其中的 `format_spec` 是格式说明符, 它是:

- `format(my_obj, format_spec)` 的第二个参数;
- {} 中待换字段冒号后面的部分, 或者 `fmt.str.format()` 中的 `fmt`。

示例如下:

```

1 >>> brl = 1 / 4.82 # BRL to USD currency conversion rate
2 >>> brl
3 0.20746887966804978
4 >>> format(brl, '0.4f') ①
5 '0.2075'
6 >>> '1 BRL = {rate:0.2f} USD'.format(rate=brl) ②
7 '1 BRL = 0.21 USD'
8 >>> f'1 USD = {1 / brl:0.2f} BRL' ③

```

```
9 '1 USD = 4.82 BRL'
```

- ❶ 格式说明符是 '0.4f'。
 - ❷ 格式说明符是 '0.2f'。待换字段中的 rate 部分不属于格式化说明符, 它只用于决定将 format() 的哪个关键字参数传递给待换字段。
 - ❸ 同样, 格式说明符是 '0.2f'。'1 / brl' 表达式不属于格式说明符。
- ❷与❸说明了一个重要知识点: "0.mass:5.3e" 这样的格式化字符串实际上包含两部分。冒号 (:) 左侧的 "0.mass" 是待换字段语法中的字段名 (field_name) 部分, 可以是 f-strings 中的任意表达式。冒号 (:) 右侧的 "5.3e" 是格式说明符。格式说明符中使用的符号称为 “**格式化规范微语言 (Format Specification Mini-Language)**”。



如果你对 f-strings、format() 和 str.format() 不熟悉, 根据我的教学经验, 最好先学习 format() 内置函数, 它只使用 “Format Specification Mini-Language”。了解其要点之后, 再阅读 “[Formatted string literals](#)” 与 “[Format String Syntax](#)”, 以了解 fstrings 与 str.format() 方法中使用的 {} 待换字段表示法 (包括转换标志!s、!r 与!a)。f-strings 出现之后, str.format() 并没有被淘汰:f-strings 适用于大多数场景, 而有些特殊场景仍需使用 str.format() 方法。

格式化规范微语言 为一些内置类型提供了专用的表示代码。例如, b 与 x 分别表示二进制与十六进制的 int 类型; f 表示小数形式的 float 类型, 而 % 表示百分数形式的 float 类型。

```
1 >>> format(42, 'b')
2 '101010'
3 >>> format(2 / 3, '.1%')
4 '66.7%'
```

格式化规范微语言 是可扩展的, 每个类都可以自行决定如何解释 format_spec 参数。例如, `datetime` 模块 中的类, 在 `strftime()` 函数 及其 `__format__` 方法中使用相同的格式代码。以下是使用内置 format() 和 str.format() 方法的几个示例:

```
1 >>> from datetime import datetime
2 >>> now = datetime.now()
3 >>> format(now, '%H:%M:%S')
4 '18:49:05'
5 >>> "It's now {:.I:{M %p}".format(now)
6 "It's now 06:49 PM"
```

如果一个类未定义 `__format__` 方法, 则从 `object` 继承的 `__format__` 方法将返回 `str(my_object)`。由于 `Vector2d` 类定义了 `__str__` 方法, 因此可以这样做:

```
1 >>> v1 = Vector2d(3, 4)
2 >>> format(v1)
3 '(3.0, 4.0)'
```

但是, 如果传递格式说明符, `object.__format__` 就会引发 `TypeError`:

```

1 >>> format(v1, '.3f')
2 Traceback (most recent call last):...
3 TypeError: non-empty format string passed to object.__format__

```

我们将通过实现自己的 **格式化规范微语言** 来解决这个问题。第一步是假定用户提供的格式说明符旨在格式化向量的每个 float 分量。我们想达到的效果如下所示：

```

1 >>> v1 = Vector2d(3, 4)
2 >>> format(v1)
3 '(3.0, 4.0)'
4 >>> format(v1, '.2f')
5 '(3.00, 4.00)'
6 >>> format(v1, '.3e')
7 '(3.000e+00, 4.000e+00)'

```

示例 11.5 实现的 `__format__` 方法, 可产生上述的格式化效果。

</> 示例 11.5: `Vector2d.__format__` 方法: 版本 1

```

1 # 在 Vector2d 类中定义
2 def __format__(self, fmt_spec=''):
3     components = (format(c, fmt_spec) for c in self) ❶
4     return '({}, {})'.format(*components) ❷

```

❶ 用内置函数 `format()`, 将 `fmt_spec` 应用于向量的每个分量, 构建一个可迭代的格式化字符串。

❷ 将格式化字符串带入公式 '`x,y`' 中。

然后, 为 **格式化规范微语言** 添加一个自定义格式化代码: 如果格式说明符以 “p” 结尾, 就在极坐标中显示向量 (即 $\langle r, \theta \rangle$)。其中, r 是模, θ (theta) 是角度表示的弧度。格式说明符的其余部分 (“p” 之前的内容) 将像以前那样解释。



为自定义的格式化代码选择字母时, 我会避免使用与其他类型重复的字母。在 **格式化规范微语言** 中, 整数使用的代码是 ‘bcd0xXn’, 浮点数使用的代码是 ‘eEf-FgGn%’, 字符串使用 ‘s’。因此, 我选择了字母 ‘p’ 来表示极坐标。由于每个类对这些代码的解释都是独立的, 因此即使在新类型中重复使用自定义格式的代码字母也不会出错, 但可能会让用户感到困惑。

为了生成极坐标, 我们已经有了用于获取模的 `__abs__` 方法, 我们将使用 `math.atan2()` 函数编写一个简单的 `angle` 方法来获取角度。代码如下:

```

1 # 在 Vector2d 类中定义
2 def angle(self):
3     return math.atan2(self.y, self.x)

```

这样, 我们就可以增强我们的 `__format__` 方法, 以生成极坐标, 如示例 11.6 所示。

</> 示例 11.6: `Vector2d.__format__` 方法, 版本 2: 使用极坐标

```

1 def __format__(self, fmt_spec=''):

```

```
1  if fmt_spec.endswith('p'):      ❶
2      fmt_spec = fmt_spec[:-1]    ❷
3      coords = (abs(self), self.angle()) ❸
4      outer_fmt = '<{}, {}>' ❹
5
6  else:
7      coords = self
8      outer_fmt = '({}, {})'
9
10     components = (format(c, fmt_spec) for c in coords) ❺
11
12     return outer_fmt.format(*components) ❻
```

- ❶ 若格式化代码以 'p' 结尾, 这使用极坐标格式表示向量。
- ❷ 从 fmt_spec 中删除 'p' 后缀, 剩余的格式化代码还按往常那样解释。
- ❸ 构建一个元组, 以展示极坐标:(模, 角度)。
- ❹ 将外层格式 outer_fmt 设置为一对尖括号 “<>”。
- ❺ 若格式化代码不以 'p' 结尾, 则用 self 的 x 分量与 y 分量构建直角坐标。
- ❻ 将外层格式 outer_fmt 设置为一对圆括号 “()”。
- ❼ 用向量的各个分量, 生成可迭代对象, 并构建格式化字符串。
- ❽ 将格式化字符串带入外层格式 outer_fmt。

运行 [示例 11.6](#), 将得到如下格式的结果:

```
1 >>> format(Vector2d(1, 1), 'p')
2 '<1.4142135623730951, 0.7853981633974483>'
3 >>> format(Vector2d(1, 1), '.3ep')
4 '<1.414e+00, 7.854e-01>'
5 >>> format(Vector2d(1, 1), '0.5fp')
6 '<1.41421, 0.78540>'
```

如本节所示, 扩展 格式化规范微语言 以支持用户定义类型并不难。

现在, 让我们换一个与对象外在表现无关的话题: 我们将 Vector2d 变成 [可哈希 \(hashable\)](#) 的, 以便于构建基于向量的 set, 或将向量作为 dict 的键。

11.7 可哈希的 Vector2d

根据定义, 到目前为止, Vector2d 实例是不 [可哈希 \(hashable\)](#) 的, 因此无法将 Vector2d 实例放入 set (集合) 中:

```
1 >>> v1 = Vector2d(3, 4)
2 >>> hash(v1)
3 Traceback (most recent call last):
4 ...
5 TypeError: unhashable type: 'Vector2d'
6 >>> set([v1])
7 Traceback (most recent call last):
8 ...
9 TypeError: unhashable type: 'Vector2d'
```

为了使 `Vector2d` 实例可哈希 (hashable), 必须实现 `__hash__` (`__eq__` 也是必需的, 前面已经实现了)。此外, 还要让 `Vector2d` 实例不可变 (详见 “3.4.1 什么是可哈希<71页>”)。

现在, 任何人都可以执行 `v1.x = 7`, 而且代码中也没有内容表明不能这么做。但我们想要的行为如下所示:

```

1  >>> v1.x, v1.y
2  (3.0, 4.0)
3  >>> v1.x = 7    # 禁止修改 Vector2d 实例
4  Traceback (most recent call last):
5  ...
6  AttributeError: can't set attribute

```

为此, 我们需要将 `x` 和 `y` 分量都设置为只读属性 (如示例 11.7 所示)。

```

</> 示例 11.7: vector2d_v3.py: 仅给出使 Vector2d 不可变所需的代码, 完整代码见 “示例 11.11<307页>”

1  class Vector2d:
2      typecode = 'd'
3
4      def __init__(self, x, y):
5          self.__x = float(x) ①
6          self.__y = float(y)
7
8      @property
9      def x(self):          ②
10         return self.__x    ③
11
12     @property
13     def y(self):          ④
14         return self.__y
15
16     def __iter__(self):    ⑤
17         return (i for i in (self.x, self.y)) ⑥
18
19     # 其余方法: 与先前的 Vector2d 相同

```

① 仅使用 2 个前导下划线 (尾部用 0 个或 1 个下划线), 即可将属性设为私有⁴。

② `@property` 装饰器 标记 特性 (property) 的 读值 (getter) 方法。

③ 读值 (getter) 方法的名称与通过该方法访问的属性同名, 都是 `x`。

④ 仅返回 `self.__x`。

⑤ 处理属性 `y` 的方式, 与处理 `x` 的方式相同。

⑥ 需要读取分量 `x` 与 `y` 的方法可以保持不变, 仍然可通过 `self.x` 与 `self.y` 读取公开属性, 而无需读取私有属性。因此, 本示例省略了 `Vector2d` 类的其余代码。

⁴ 关于私有属性的利弊, 即将在 “11.10 Python 私有属性与 “受保护” 的属性<310页>” 中探讨



实例中的属性 (attributes) 是对象所拥有的数据, 可以是变量、方法或其他的对象。而 `property` 是一种特殊的属性 (为了以示区分, 简称“特性”), 通过 `@property` 装饰器来定义, 它可以通过 `getter` 和 `setter` 方法控制对属性 (attributes) 的访问和赋值操作。

`Vector.x` 和 `Vector.y` 是只读属性的示例。读/写属性将在“[二十二 动态属性和特性](#)”中介绍, 届时将深入探讨 `@property` 装饰器。

现在, 我们的 `Vector2d` 实例已比较安全, 不会被意外更改。接下来, 可以实现 `__hash__` 方法了, 该方法应该返回一个 `int` 值, 理想情况还要考虑对象属性的哈希码⁵ (`__eq__` 方法也是如此, 因为相等的对象应具有相同的哈希码)。特殊方法 `__hash__` 的文档 建议根据向量分量的元组来计算哈希码, 如示例 11.8 所示。

</> [示例 11.8: vector2d_v3.py: 实现 __hash__ 方法](#)

```
1 # 在 Vector2d 类中定义 :
2 def __hash__(self):
3     return hash((self.x, self.y))
```

实现了 `__hash__` 方法之后, `Vector2d` 实例就变成 [可哈希 \(hashable\)](#) 的了。

```
1 >>> v1 = Vector2d(3, 4)
2 >>> v2 = Vector2d(3.1, 4.2)
3 >>> hash(v1), hash(v2)
4 (1079245023883434373, 1994163070182233067)
5 >>> {v1, v2}
6 {Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```



严格来说, 创建 [可哈希 \(hashable\)](#) 类型并不需要实现 [特性 \(property\)](#) 或以其他方式保护实例属性 (attributes)。只需正确地实现 `__hash__` 和 `__eq__` 即可。但是, [可哈希 \(hashable\)](#) 对象的值是绝不会改变的, 因此借机提到了只读特性 (`property`)。

如果要创建一个具有标量数值的类型, 还可以实现特殊方法 `__init__` 与 `__float__` (分别由构造函数 `int()` 与 `float()` 调用), 以便在某些情况下强制转换类型。此外, 还有一个特殊方法 `__complex__` 用于支持内置构造函数 `complex()`。`Vector2d` 类或许也应该实现 `__complex__` 方法, 这就留给读者当作练习吧。

11.8 支持位置的模式匹配

到目前为止, `Vector2d` 实例已兼容“[关键字类模式 \(Class Pattern\)](#)”匹配 (详见“[5.8.2 关键字类模式](#)<157页>”)。

如 [示例 11.9](#) 所示, 所有这些关键字模式匹配都可按预期正常工作。

</> [示例 11.9: Vector2d 实例的关键字模式匹配 \(Python≥3.10\)](#)

⁵ 哈希码: 为了避免与映射对象中的“值”相混淆, 本书用“哈希码”代替“哈希值”。

```

1 def keyword_pattern_demo(v: Vector2d) -> None:
2     match v:
3         case Vector2d(x=0, y=0):
4             print(f'{v!r} is null')
5         case Vector2d(x=0):
6             print(f'{v!r} is vertical')
7         case Vector2d(y=0):
8             print(f'{v!r} is horizontal')
9         case Vector2d(x=y) if x==y:
10            print(f'{v!r} is diagonal')
11        case _:
12            print(f'{v!r} is awesome')

```

但是,如果您尝试使用类似这样的位置 模式匹配 (Pattern Match) :

```

1 case Vector2d(_, 0):
2     print(f'{v!r} is horizontal')

```

则会得到如下错误提示:

```

1 TypeError: Vector2d() accepts 0 positional sub-patterns (1 given)

```

为了使 Vector2d 能够使用位置 模式匹配 (Pattern Match),需要为其添加类属性 `__match_args__`,按照位置 模式匹配 (Pattern Match) 的顺序列出实例属性:

```

1 class Vector2d:
2     __match_args__ = ('x', 'y')
3     # 等等

```

现在,编写匹配 Vector2d 实例的 模式 (Pattern) 时,可以少敲几次键盘,如 [示例 11.10](#) 所示。

</> [示例 11.10](#): Vector2d 实例的位置模式匹配 (Python ≥ 3.10)

```

1 def positional_pattern_demo(v: Vector2d) -> None:
2     match v:
3         case Vector2d(0, 0):
4             print(f'{v!r} is null')
5         case Vector2d(0):
6             print(f'{v!r} is vertical')
7         case Vector2d(_, 0):
8             print(f'{v!r} is horizontal')
9         case Vector2d(x, y) if x==y:
10            print(f'{v!r} is diagonal')
11        case _:
12            print(f'{v!r} is awesome')

```

类属性 `__match_args__` 不需要包含所有公开实例属性。特别是,如果类 `__init__` 具有分配给实例属性的必需参数和可选参数,那么类属性 `__match_args__` 中应该仅列出必需参数,而不必列出可选参数。

现在,让我们回顾一下迄今为止为 Vector2d 类编写的所有代码。

11.9 第3版 Vector2d 的完整代码

我们开发 `Vector2d` 已经有一段时间了, 但每次仅展示一些代码片段。[示例 11.11](#) 是 `vector2d_v3.py` 的完整代码 (包括使用的 doctests)。

</> 示例 11.11: vector2d_v3.py:完整版

```

43     >>> Vector2d(1, 0).angle()
44     0.0  >>> epsilon = 10**-8
45     >>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
46     True
47     >>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
48     True
49
50 用极坐标测试 ``format()``::
51     >>> format(Vector2d(1, 1), 'p') # doctest:+ELLIPSIS
52     '<1.414213..., 0.785398...>'
53     >>> format(Vector2d(1, 1), '.3e')
54     '<1.414e+00, 7.854e-01>'
55     >>> format(Vector2d(1, 1), '0.5fp')
56     '<1.41421, 0.78540>'
57
58 测试只读特性 x 与 y::
59     >>> v1.x, v1.y
60     (3.0, 4.0)
61     >>> v1.x = 123
62     Traceback (most recent call last):
63     ...
64     AttributeError: can't set attribute 'x'
65
66 测试哈希::
67     >>> v1 = Vector2d(3, 4)
68     >>> v2 = Vector2d(3.1, 4.2)
69     >>> len({v1, v2})
70     2
71     """
72
73 from array import array
74 import math
75
76 class Vector2d:
77     __match_args__ = ('x', 'y')
78
79     typecode = 'd'
80
81     def __init__(self, x, y):
82         self._x = float(x)
83         self._y = float(y)
84
85     @property
86     def x(self):
87         return self._x
88
89     @property
90     def y(self):
91         return self._y
92

```

```
93     def __iter__(self):
94         return (i for i in (self.x, self.y))
95     def __repr__(self):
96         class_name = type(self).__name__
97         return '{}({!r}, {!r})'.format(class_name, *self)
98
99     def __str__(self):
100        return str(tuple(self))
101
102    def __bytes__(self):
103        return (bytes([ord(self.typecode)]) +
104                bytes(array(self.typecode, self)))
105
106    def __eq__(self, other):
107        return tuple(self) == tuple(other)
108
109    def __hash__(self):
110        return hash((self.x, self.y))
111
112    def __abs__(self):
113        return math.hypot(self.x, self.y)
114
115    def __bool__(self):
116        return bool(abs(self))
117
118    def angle(self):
119        return math.atan2(self.y, self.x)
120
121    def __format__(self, fmt_spec=''):
122        if fmt_spec.endswith('p'):
123            fmt_spec = fmt_spec[:-1]
124            coords = (abs(self), self.angle())
125            outer_fmt = '<{}, {}>'.format
126        else:
127            coords = self
128            outer_fmt = '({}, {})'.format
129        components = (format(c, fmt_spec) for c in coords)
130        return outer_fmt(*components)
131
132    @classmethod
133    def frombytes(cls, octets):
134        typecode = chr(octets[0])
135        memv = memoryview(octets[1:]).cast(typecode)
136        return cls(*memv)
```

回顾一下，在本节和前几节中，我们介绍了一些必要的特殊方法。要想得到功能完整的对象，这些方法可能是必备的。



只有在应用程序需要时,才应该实现这些特殊方法。最终用户并不关心组成应用程序的对象是否 Pythonic。另外,如果您的类是供其他 Python 程序员使用的库的一部分,那么您肯定猜不到程序员会对您的对象做哪些操作,他们或许更希望您的代码是 Pythonic。

如“[示例 11.11<307页>](#)”所示,Vector2d 只是一个教学示例,其中包含一系列与对象表示相关的特殊方法,而并不是每个用户定义类的模板。

下一节,我们将从 Vector2d 中抽身,讨论 Python 中私有属性机制(双下划线前缀,如 `self.__x`)的设计和缺点。

11.10 Python 私有属性与“受保护”的属性

在 Python 中,无法像 Java 中那样使用 `private` 修饰符创建私有变量。Python 中有一个简单的机制来防止子类意外覆盖父类中的“私有”属性。

举个例子,有人编写了一个 Dog 类,其内部用到了 `mood` 实例属性,但未将该属性公开。现在,您创建了 Dog 类的子类 Beagle。如果您在毫不知情的情况下,又创建了自己的 `mood` 实例属性。那么,您就会破坏从 Dog 类集成的方法所使用的 `mood` 实例属性(即子类覆盖了父类的 `mood` 实例属性)。这是难以调试的问题。

为了避免这种情况,如果以 `__mood` 形式(2个前导下划线,尾部0或1个下划线)命名实例属性,Python 会将该属性名存储在实例的 `__dict__` 中,并以1个前导下划线与类名为属性名前缀。因此,Dog 类中的实例属性 `__mood` 变成了 `_Dog__mood`,而 Beagle 中的实例属性 `__mood` 则变成了 `_Beagle__mood`。这种语言特性叫做“名称改写(Name Mangling)”。

[示例 11.12](#) 以在“[示例 11.7<304页>](#)”中定义的 Vector2d 类为例,展示了 Python 对私有实例属性的名称改写。

</> [示例 11.12: 私有属性的名称会被改写,在前面添加“_类名”前缀](#)

```

1  >>> v1 = Vector2d(3, 4)
2  >>> v1.__dict__          # 将私有属性存储在 __dict__ 中
3  {'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
4  >>> v1._Vector2d__x     # 为实例属性 __x 增加了 `__类名` 前缀, 即 `__Vector2d__
5  3.0

```

“名称改写(Name Mangling)”是一种保护措施,而不是安全措施:它旨在防止意外访问,而无法防止恶意操作(窥探)。[图 11.1](#) 以实物的形式展示了这种保护措施。

任何了解名称改写机制的人,都可以直接读取私有实例属性(如[示例 11.12](#)最后一行所示)——实际上,这对调试与序列化也很有用。此外,还可以通过编写“`v1._Vector2d__x=7`”来直接为 Vector2d 实例的私有分量赋值。但如果在生产中这么做,那么出现问题时就不要抱怨了(自找麻烦)。

并不是所有 Python 程序员都喜欢“名称改写”功能,也不是所有程序员都喜欢 `self.__x` 这种写法。有些人喜欢避免使用这种双下划线(`__`)语法,而约定仅使用单下划线(`_`)前缀来“保护”实例属性(例如 `self._x`)。反对双下划线名称改写的人认为,应该通过命名约定来避免对实例属性的意外覆盖。`pip` 与 `virtualenv` 等项目的创建者 Ian Bicking 指出:

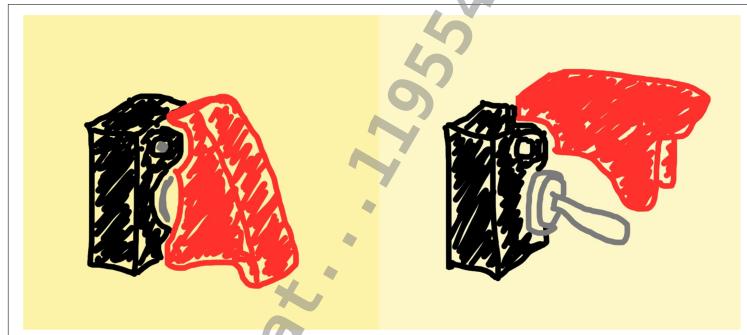


图 11.1: 开关的盖子是一种保护装置,而不是安全装置。
它能防止误操作,但无法防止恶意操作。

千万不要使用双前导下划线(`__`),这是个令人恼火的私人行为。如果担心实例属性名称冲突,应该使用单下划线(`_`)实现显式的名称改写(例如`_MyThing_blahtblah`)。这本质上与隐式的双下划线改写相同,但是却拥有更好的可读性^a。

^a摘自“*Paste Style Guide*”

Python 解释器不会对使用单下划线(`_`)前缀的属性名做特殊处理,但这是很多 Python 程序员严格遵守的约定,即不应该从类的外部访问这种属性⁶。遵守使用单下划线(`_`)标记实例的私有属性很容易,就像遵守使用全大写字母来定义常量一样容易。

在 Python 文档的某些地方⁷,带有单个`_`前缀的属性被称为“protected”属性。通过约定俗成的`self._x`形式来“保护”属性的做法很普遍,但将其称为“受保护(protected)”属性并不常见。有些人甚至称之为“私有(private)”属性。

总之,Vector2d 分量是“私有的(private)”,而且 Vector2d 实例都是“不可变”的。注意,我用了双引号(“”),因为并不能实现真正的私有与不可变⁸。

现在,我们回到 Vector2d 类。在下一节中,我们将介绍一个特殊属性(不是方法)。它将影响对象的内部存储,还可能对内存的使用产生重大影响,但是对对象的公开接口影响不大。这个属性就是`__slots__`。

11.11 用 `__slots__` 节省内存

默认情况下,Python 将每个实例的属性都存储在一个名为`__dict__`的 dict 中。正如“[3.9 dict 实现方式对实践的影响<84页>](#)”所述,即使执行了该节提到的优化措施,`dict` 也会产生很大的内存开销。但是,如果为类定义了`__slots__`类属性,该属性会以序列的形式保存实例属性名称,Python 会为实例属性使用另一种存储模型——`__slots__`中的属性名称存储在一个隐藏的引用数组中,内存消耗比`dict`更少。让我们从[示例 11.13](#)开始,通过简单的示例,来了解类属性`__slots__`的工作原理。

</> 示例 11.13: 使用`__slots__`的 Pixel 类

1 >>> `class Pixel:`

⁶在模块中,在顶层名称前使用单下划线(`_`)前缀,确实会有影响:例如,“`from mymod import *`”不会从`mymod`中导入带有单下划线前缀的名称。但是,依旧可以用“`from mymod import _privatefunc`”的方式导入。详见《*The Python Tutorial*》的“[6.1 More on Modules](#)”。

⁷[gettext 模块文档](#)中就有一个示例。

⁸如果您对这种说法感到沮丧,并希望 Python 在这方面向 Java 看齐,请不要阅读本章“[11.14 延伸阅读<318页>](#)”中的“杂谈”。在那里,我描述了 Java 中`private`修饰符的相对优势。

```

2 ...     __slots__ = ('x', 'y') ❶
3 ...>>> p = Pixel() ❷
4 >>> p.__dict__ ❸
5 Traceback (most recent call last):
6 ...
7 AttributeError: 'Pixel' object has no attribute '__dict__'
8 >>> p.x = 10 ❹
9 >>> p.y = 20
10 >>> p.color = 'red' ❺
11 Traceback (most recent call last):
12 ...
13 AttributeError: 'Pixel' object has no attribute 'color'

```

- ❶ 必须在定义类时,就声明 **类属性 __slots__**,后续再添加或修改均无效。属性名称可以存储在一个 tuple 或 list 中。不过,我更倾向于用 tuple,因为它可以明确表明 **__slots__** 是无法修改的。
- ❷ 创建一个 Pixel 实例,因为 **类属性 __slots__** 的效果要通过实例体现。
- ❸ 第一个效果:Pixel 实例没有 **__dict__** 属性。
- ❹ 正常设置 p.x 与 p.y 属性。
- ❺ 第二个效果:尝试设置 **__slots__** 中未列出的属性,会引发 **AttributeError** 异常。

到目前为止,一切都很顺利。现在,在 [示例 11.14](#) 中定义一个 Pixel 的子类,看看 **类属性 __slots__** 违反直觉的一面。

</> **示例 11.14:** OpenPixel 是 Pixel 的子类

```

>>> class OpenPixel(Pixel): ❶
2 ...     pass
3 ...
4 >>> op = OpenPixel() ❷
5 >>> op.__dict__ ❸
6 {}
7 >>> op.x = 8 ❹
8 >>> op.__dict__ ❺
9 {}
10 >>> op.x ❻
11 8
12 >>> op.color = 'green' ❾
13 >>> op.__dict__ ❿
14 {'color': 'green'}

```

- ❶ OpenPixel 类自身未声明任何属性。
- ❷ 惊讶:OpenPixel 实例有一个 **__dict__** 属性。
- ❸ 如果设置属性 x(在基类 Pixel 的 **__slots__** 中命名)...
- ❹ ... 属性 x 不会被存入实例的 **__dict__** 属性中...
- ❺ ... 而是将属性 x 存入实例的一个隐藏的引用数组中。
- ❻ 若设置的属性未包含在类属性 **__slots__** 中...
- ❼ ... 则会将其存入实例的 **__dict__** 属性中。

示例 11.14 表明, 子类只继承了类属性 `__slots__` 的部分效果。为了确保子类的实例也没有 `__dict__` 属性, 必须在子类中再次声明类属性 `__slots__`。

如果在子类中声明类属性 “`__slots__=()`” (即一个空元组), 则子类的实例将没有 `__dict__` 属性, 并且只接受基类的 `__slots__` 属性列出的属性名称。

如果希望子类拥有额外的属性, 则应将它们列在子类的 `__slots__` 类属性中, 如示例 11.15 所示。

</> 示例 11.15: `ColorPixel` 是 `Pixel` 的另一个子类

```

1  >>> class ColorPixel(Pixel):
2      ...      __slots__ = ('color',)
3  >>> cp = ColorPixel()
4  >>> cp.__dict__
5  Traceback (most recent call last):
6      ...
7  AttributeError: 'ColorPixel' object has no attribute '__dict__'
8  >>> cp.x = 2
9  >>> cp.color = 'blue'
10 >>> cp.flavor = 'banana'
11 Traceback (most recent call last):
12      ...
13 AttributeError: 'ColorPixel' object has no attribute 'flavor'
```

- ❶ 本质上, 超类的 `__slots__` 会被添加到当前类的 `__slots__` 中。不要忘记, 单项元组必须在项的后面加一个逗号, 如 `(t,)`。
- ❷ `ColorPixel` 实例没有 `__dict__` 属性。
- ❸ 可以设置已在当前类和超类的 `__slots__` 中声明的属性, 但不能设置其他属性。

但是, “节省的内存, 也可能被再次吃掉”: 如果将 “`__dict__`” 这一名称添加到类属性 `__slots__` 中, 则实例会在各个实例独有的引用数组中存储 `__slots__` 中的名称, 除此之外还支持动态创建的属性, 这些属性将存储在常规的 `__dict__` 中。如果要使用 `@cached_property` 装饰器 (详见 “22.3.5 第 5 步: 用 `functools` 缓存特性<678页>”), 就必须这样做。

当然, 将 “`__dict__`” 添加到类属性 `__slots__` 中可能完全违背了初衷, 具体取决于各个实例中静态属性与动态属性的数量, 以及它们的用法。草率地优化比过早地优化更糟糕, 往往会增加复杂性, 并且收效甚微。

每个实例需要注意的另一个特殊属性是 `__weakref__`, 它是对象支持弱引用 (Weak Reference) 所必需的 (详见 “6.6 `del` 与垃圾回收<177页>”)。该属性默认存在于用户定义类的实例中。但是, 如果类定义了类属性 `__slots__`, 并且需要将实例作为弱引用 (Weak Reference) 的目标, 则必须将 `__weakref__` 添加到类属性 `__slots__` 中。

现在, 看看为 `Vector2d` 类添加类属性 `__slots__` 的效果。

11.11.1 简单衡量 `__slot__` 节省的内存

示例 11.16 展示了 `Vector2d` 类中 `__slots__` 的实现。

</> 示例 11.16: `vector2d_v3_slots.py`: 只为 `Vector2d` 类增加 `__slots__` 属性

```
1  class Vector2d:
```

```

2     __match_args__ = ('x', 'y')
3     __slots__ = ('_x', '_y')
4     typecode = 'd'
5     # 方法与前面的版本一样

```

① `__match_args__` 列出了位置模式匹配的公开 (public) 属性名称。

② 与此相反, `__slots__` 列出了实例属性的名称。本例列出的是私有属性 (即双下划线为前缀)。

为了衡量节省的内存,我编写了 `mem_test.py` 脚本。该脚本以两版 `Vector2d` 类变体所在的模块名称为命令行参数,并用列表推导式构建一个包含 10,000,000 个 `Vector2d` 实例的列表。[示例 11.17](#) 中第 1 次运行时,使用 `vector2d_v3.Vector2d` (来自“[示例 11.7<304页>](#)”);第 2 次运行时,使用“[示例 11.16<313页>](#)”中带 `__slots__` 的版本,如 [示例 11.17](#) 所示。

</> [示例 11.17: mem_test.py 使用指定模块中定义的 Vector2d 类创建 10,000,000 个实例](#)

```

1 $ time python3 mem_test.py vector2d_v3
2 Selected Vector2d type: vector2d_v3.Vector2d
3 Creating 10,000,000 Vector2d instances
4 Initial RAM usage: 6,983,680
5 Final RAM usage: 1,666,535,424
6
7 real 0m11.990s
8 user 0m10.861s
9 sys 0m0.978s
10
11 $ time python3 mem_test.py vector2d_v3_slots
12 Selected Vector2d type: vector2d_v3_slots.Vector2d
13 Creating 10,000,000 Vector2d instances
14 Initial RAM usage: 6,995,968
15 Final RAM usage: 577,839,104
16
17 real 0m8.381s
18 user 0m8.006s
19 sys 0m0.352s

```

如 [示例 11.17](#) 所示,当 10,000,000 个 `Vector2d` 实例都使用 `__dict__` 属性时,脚本的 RAM 占用增加到 1.55GiB,但当 `Vector2d` 类具有 [类属性 __slots__](#) 时,RAM 占用降到了 551MiB。而且 `__slots__` 版本的速度也更快。本测试中的 `mem_test.py` 脚本主要处理加载模块、检查内存使用情况和格式化结果。`mem_test.py` 脚本的源代码见 [example-code-2e/11-pythonic-obj /mem_test.py](#)。



处理数百万个具有数值数据的对象,其实应该使用 NumPy 数组 (见“[2.10.3 NumPy<55页>](#)”)。NumPy 数组不仅节约内存,而且具有高度优化的数值处理函数。其中,很多函数可一次性处理整个数组。我设计 `Vector2d` 类只是为了在讨论特殊方法时提供上下文,因为我会尽可能避免使用含糊不清的示例。

11.11.2 总结 `__slots__` 的问题

如果使用得当,类属性 `__slots__` 可显著节省内存,但也有一些注意事项:

- 必须记住在每个子类中重新声明类属性 `__slots__`,以防止它们的实例中出现 `__dict__`。
- 除非在 `__slots__` 中包含“`__dict__`”(但这样做可能会抵消节省的内存)。否则,实例只能拥有类属性 `__slots__` 中列出的属性。
- 使用 `__slots__` 的类不应用 `@cached_property` 装饰器,除非显式将“`__dict__`”添加到类属性 `__slots__` 中。
- 如果要将实例用作弱引用(Weak Reference)的目标对象,必须将“`__weakref__`”显式添加到类属性 `__slots__` 中。

下一节,将讨论如何在实例和子类中重写类属性。

11.12 覆盖类属性 (Class Attribute)

Python 的一个独特功能是类属性可以用作实例属性的默认值。在 `Vector2d` 类中有一个类属性 `typecode`,被 `__bytes__` 方法使用了 2 次,并且每次都用 `self.typecode` 读取它的值。因为 `Vector2d` 实例在创建时本身没有 `typecode` 实例属性,因此 `self.typecode` 将默认获得类属性 `Vector2d.typecode`。

但如果为不存在的实例属性赋值,则会动态创建一个新的实例属性。例如,为 `typecode` 实例属性赋值,则同名的类属性则不会受赋值的影响。然而,从那刻起,实例读取的 `self.typecode` 其实是动态创建的实例属性,而不是同名的类属性 `typecode`,从而有效地遮蔽了同名的类属性。这样,就有可能为单个实例定制不同的 `typecode`。

默认的 `Vector2d.typecode` 为 ‘d’,这意味着在导出到字节序列时,每个向量分量都将以 8 字节双精度浮点表示。如果我们在导出前将 `Vector2d` 实例的 `typecode` 设置为 ‘f’,则每个分量将以 4 字节单精度浮点数的形式导出。如 [示例 11.18](#) 所示。



此处讨论添加自定义实例属性,因此 [示例 11.18](#) 使用了“[示例 11.11<307页>](#)”中不含类属性 `__slots__` 的 `Vector2d` 实现。

</> [示例 11.18](#): 设定原本从类继承的类属性 `typecode`,自定义一个实例

```
1  >>> from vector2d_v3 import Vector2d
2  >>> v1 = Vector2d(1.1, 2.2)
3  >>> dumpd = bytes(v1)
4  >>> dumpd
5  b'd\x9a\x99\x99\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
6  >>> len(dumpd)           ❶
7  17
8  >>> v1.typecode = 'f'     ❷
9  >>> dumpf = bytes(v1)
10 >>> dumpf
```

```

11 b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
12 >>> len(dumpf)          ❸
13 9>>> Vector2d.typecode ❹
14 'd'
15 >>> v2 = Vector2d(3.3, 4.4)
16 >>> v2.typecode          ❺
17 'd'

```

- ❶ 默认的字节序列表示形式长度为 17 个字节。
- ❷ 在实例 v1 中, 将 typecode 设置为 'f'。
- ❸ 现在得到的字节序列长度是 9 个字节。
- ❹ Vector2d.typecode 不受实例 v1.typecode='f' 赋值的影响; 只有实例 v1 使用了 typecode 'f'。
- ❺ 实例 v2 读取的仍为类属性 typecode。

现在应该清楚为何 Vector2d 导出的字节序列要以 typecode 为前缀了——因为我们希望 Vector2d 可以支持不同的导出格式 (typecode)。

如果要更改类属性的值, 必须直接通过类来修改, 而不能通过实例修改。可以通过执行如下操作, 更改所有实例 (未单独为 typecode 实例属性赋值) 的 typecode 默认值:

```

1 >>> Vector2d.typecode = 'f'

```

然而, 有一种 Python 惯用方法可以实现更持久的重写效果, 也更有针对性。因为类属性是公开 (public) 的, 会被子类继承, 所以通常的做法是创建一个子类来定制类的数据属性。Django 基于类的视图广泛使用了这种技术。如示例 11.19 所示。

</> 示例 11.19: ShortVector2d 是 Vector2d 的子类, 仅重写了默认的 typecode

```

1 >>> from vector2d_v3 import Vector2d
2 >>> class ShortVector2d(Vector2d):
3     ...     typecode = 'f'
4     ...
5 >>> sv = ShortVector2d(1/11, 1/27) ❷
6 >>> sv
7 ShortVector2d(0.09090909090909091, 0.037037037037037035) ❸
8 >>> len(bytes(sv))           ❹
9 9

```

- ❶ 将 ShortVector2d 创建为 Vector2d 的子类, 仅为了重写类属性 typecode。
- ❷ 构建 ShortVector2d 实例 sv, 以用于演示。
- ❸ 检查 sv 的表示形式。
- ❹ 检查导出的字节序列长度是否为 9, 而不是之前的 17。

这个例子也解释了我为何没在 Vector2d.__repr__ 中硬编码 class_name 的值, 而是从 type(self).__name__ 中获取, 如下所示 (摘自“示例 11.11<307 页>”):

```

1 # 在 Vector2d 类中定义:
2 def __repr__(self):
3     class_name = type(self).__name__

```

4 | `return '{}({!r}, {!r})'.format(class_name, *self)`

如果硬编码了 `class_name` 的值, 则 `Vector2d` 的子类(如 `ShortVector2d`)将必须重写 `__repr__` 方法才能更改 `class_name`。通过从实例的类型中读取类名(`__name__`), 使 `__repr__` 方法可被子类放心地继承。

至此, 我们完成了对构建一个简单类的介绍。该类可利用数据模型与 Python 的其他功能完美结合, 包括提供不同的对象表示形式、提供自定义格式化代码、公开只读属性, 以及通过支持 `hash()` 函数以与 `set` 或映射集成。

11.13 本章小结

本章的目的是, 演示如何使用特殊方法和约定来构建一个行为良好且符合 Python 风格的类。

`vector2d_v3.py`(见“[示例 11.11<307页>](#)”)比 `vector2d_v0.py`(见“[示例 11.2<297页>](#)”)更符合 Python 风格么? 显然 `vector2d_v3.py` 中的 `Vector2d` 类展现了更多的 Python 特性。但是, 二者之中哪个更符合 Python 风格, 要取决于它的使用环境。Tim Peter 在《Python 之禅》中写道:

简单总比复杂好。

一个符合 Python 风格的对象应该刚好满足所需, 而不应该堆砌语言特性。开发应用程序时, 代码应专注于支持终端用户所需的功能, 仅此而已。开发供其他程序员使用的库时, 应实现一些特殊方法, 以便为 Python 程序员提供预期的行为。例如, `__eq__` 方法可能不是满足业务需求所必需的, 但它会使类更便于测试。

本章中, 我逐步扩展 `Vector2d` 代码的目的是为讨论 Python 特殊方法与编码约定提供上下文。本章中的示例演示了在“[表 1.1<14页>](#)”中首次见到的几个特殊方法:

- 字符串/字节表示方法: `__repr__`、`__str__`、`__format__` 和 `__bytes__`。
- 将对象转换为数字的方法: `__abs__`、`__bool__` 与 `__hash__`。
- 用于支持测试与哈希的 `__eq__` 与 `__hash__`。

在支持转换为字节序列的同时, 还实现了一个备选构造函数 `Vector2d.frombytes()`, 它为讨论 `@classmethod`(非常方便)和 `@staticmethod`(不太有用, 使模块级函数更简单)提供了背景。`frombytes` 方法的灵感来源于 `array.array` 类中的同名方法。

我们了解到, [格式化规范微语言](#) 可通过实现 `__format__` 方法进行扩展。该方法可以解析提供给内置函数 `format(obj, format_spec)` 或 f-strings 或 `str.format()` 方法的待换字段“`{: "format_spec"}`”中的 `format_spec`。

为了使 `Vector2d` 实例 [可哈希 \(hashable\)](#), 先使实例不可变。至少通过将 `x` 和 `y` 属性设为私有(`private`)属性, 并将它们公开为只读特性(`property`), 防止意外更改。然后, 使用实现了 `__hash__`, 该方法使用推荐的 `xor`(异或)运算符计算实例属性的哈希码。

接着, 本章讨论了如何使用 [类属性 __slots__](#) 节省内存, 以及注意事项。由于使用 `__slots__` 会产生副作用, 因此只有在处理大量(数百万个, 而非数千个)实例时才建议使用。但是, 在这种情况下, 使用 `pandas` 或许是最好的选择。

最后一个主题是重写类属性, 包括两种实现方式:

- 在实例中创建一个与类属性同名的实例属性。在此情况下, 实例会优先使用实例属性, 而不会访问类属性。此种方式适用于只想为特定实例更改属性值的场景。

- 创建一个子类，并在子类中重新定义类属性的值，从而覆盖父类中的类属性。此种方式适用于想要在整个类层次结构中更改属性值的场景。

在本章中，我提到了“本章所有示例的设计灵感，都源自对标准 Python 对象 API 的研究。”如果用一句话来概括本章，那就是：

要构建 Pythonic 对象，请仔细观察真正 Python 对象的行为。

——中国古代谚语

11.14 延伸阅读

本章涵盖了数据模型的几种特殊方法，因此主要参考资料自然与“[一 Python 数据模型](#)”中提供的相同。“[一 Python 数据模型](#)”对同一主题进行了高层次的阐述。为方便起见，我将在此重复前面的 4 条建议，并补充一些其他建议：

- 《[The Python Language Reference](#)》中的“[3. Data model](#)”
本章中使用的大多数方法都见于“[3.3.1. Basic customization](#)”。
- 《[Python in a Nutshell, 3rd Ed](#)》(Alex Martelli、Anna Ravenscroft、Steve Holden 合著)
深入介绍了特殊方法。
- 《[Python Cookbook, 3rd Ed](#)》(David Beazley、Brian K.Jones 合著)
通过经典实例演示了现代 Python 编程实践。特别是第 8 章“[Classes and Objects](#)”，提供了与本章相关的几个解决方案。
- 《[Python Essential Reference, 4th ed](#)》(David Beazley 著)
详细介绍了数据模型（只涵盖 Python2.6 与 3.0）。自 Python2.2 统一了内置类型与用户定义类以来，基础概念都是一样的，大多数数据模型 API 也没什么变化。

2015 年，也就是我完成《流畅的 Python（第 1 版）》的那一年，Hynek Schlawack 启动了 [attrs](#) 包的开发。以下内容摘自 [attrs 文档](#)：

[attrs](#) 是一个 Python 包，它能让你从实现对象协议（又称 [dunder 方法](#)）的繁重工作中解脱出来，重新享受编写类的乐趣。

我在“[5.10 延伸阅读<159页>](#)”中提到过，[attrs](#) 包是 [@dataclass](#) 的强大替代品。“[五 数据类构建器](#)”中的数据类构建器和 [attrs](#) 会自动为你的类配备一些特殊方法。但是，只有了解如何自己编写这些特殊方法，才能理解这些包的作用，才能判断是否真的这些包，才能在必要时覆盖这些包生成的方法。

在本章中，涵盖了与对象表示形式相关的所有特殊方法（[__index__](#) 与 [__fspath__](#) 除外）。[__index__](#) 方法将在“[12.5.2 能处理切片的 __getitem__ 方法<327页>](#)”中讨论。而关于 [__fspath__](#) 方法，请参阅“[PEP 519 -Adding a file system path protocol](#)”。

Smalltalk 语言很早就意识到了为对象提供不同字符串表示形式的必要性。Bobby Woolf 在发表的文章“[How to Display an Object as a String: print□ String and displayString](#)”（1996 年）中，讨论了该语言中 [printString](#) 与 [displayString](#) 方法的实现。在“[11.2 对象表示形式<296页>](#)”中定义 [repr\(\)](#) 和 [str\(\)](#) 时，我借用了该文章中“便于开发人员理解的方式”和“便于用户理解的方式”这两个精辟的描述。

杂谈

特性 (property) 有助于减少前期投入

在 Vector2d 的初始版本中, 属性 x 和属性 y 是公开 (public) 的。默认情况下, 所有 Python 实例属性和类属性都是公开 (public) 的。自然地, 向量实例的用户需要访问向量的分量 (即 x 与 y)。尽管向量实例是可迭代对象, 可解包为一对变量, 但还是希望通过 my_vector.x 和 my_vector.y 这种形式来获取各个分量。

如果我们认为有必要避免意外更新属性 x 与属性 y 时, 则可将二者实现为 特性 (property)。代码的其他部分无需更改, Vector2d 的公开接口也不受影响, 这一点已通过 doctests 进行了验证。我们仍然可以访问 my_vector.x 与 my_vector.y。

这表明, 可以先用最简单的方式 (即用公共属性) 定义我们的类。因为如果稍后需要对 读值 (getter) 方法与 设值 (setter) 方法施加更多控制, 可以通过 特性 (property) 来实现。这样做, 无需更改任何已通过公开属性名 (如 x 和 y) 与对象交互的代码。

而 Java 中所采取的方式却截然相反: Java 程序员不能先定义简单的公开属性, 然后再根据需要将属性实现为 特性 (property), 因为 Java 语言中没有 特性 (property)。因此, 在 Java 中编写 读值 (getter) 与 设值 (setter) 方法是一种规范 (即便这些方法无任何实际用处), 因为 API 无法在不影响用户代码的情况下, 从简单的公开属性转变成 读值 (getter) 与 设值 (setter) 方法。

此外, Martelli、Ravenscroft 和 Holden 在《Python in a Nutshell 第 3 版》中指出, 到处输入 getter/ setter 调用是非常愚蠢的 (如下所示):

```
1 >>> my_object.set_foo(my_object.get_foo() + 1)
```

只需写成这样 (如下所示) 就可以了。

```
1 >>> my_object.foo += 1
```

维基百科的发明者和极限编程先驱 Ward Cunningham, 建议编码时要自问: “做这件事最简单的方法是什么?” 言外之意, 我们应将焦点放在目标上^a, 而提前实现 读值 (getter) 方法与 设值 (setter) 方法会偏离目标。在 Python 中, 我们可以先简单地使用公开属性, 因为我们知道: 根据需要, 我们可以在后期将这些公开属性实现为 特性 (property)。

私有属性的保护性与安全性

Perl 语言对强制隐私没有特别的喜好, 它更希望你不要进入它的“客厅”, 并不是因为它拿着霰弹枪, 而是因为你没有被邀请。

——Larry Wall, Perl 创始者

Python 和 Perl 在很多方面都是截然相反的, 但 Guido 和 Larry 似乎在对象隐私方面达成了一致。

多年来, 我曾教过很多 Java 程序员学习 Python, 我发现很多人过于相信 Java 提供的隐私保证。事实证明, Java 的 private 和 protected 修饰符通常只提供意外保护 (即保护措施 (safety))。仅当应用程序经过特殊配置, 并部署在 Java SecurityManager 之上时, 才能防止恶意访问。但在实践中, 很少有人这么做, 即便在企业环境中也很少见。

下面通过一个 Java 类来证实这一点(如示例 11.20 所示)。

</> **示例 11.20:** confidential.java:一个拥有名为 secret 的私有字段的 Java 类

```

1  public class Confidential {
2
3      private String secret = "";
4
5      public Confidential(String text) {
6          this.secret = text.toUpperCase();
7      }
8  }

```

在 **示例 11.20** 中, 我将文本转换为大写后存储在字段 secret 中。这样做, 只是为了表明字段 secret 中的值全都是大写形式。

要使用 Jython 运行 expose.py 脚本, 才能真正说明问题。该脚本使用 **内省**(Introspection)(在 Java 中称为“反射”)读取私有字段的值。如**示例 11.21** 所示。

</> **示例 11.21:** expose.py: 用于读取另一个类中私有字段内容的 Jython 代码

```

1  #!/usr/bin/env jython
2  # NOTE: Jython is still Python 2.7 in late2020
3
4  import Confidential
5
6  message = Confidential('top secret text')
7  secret_field = Confidential.getDeclaredField('secret')
8  secret_field.setAccessible(True) # break the lock!
9  print 'message.secret =', secret_field.get(message)

```

运行 **示例 11.21** 将得到如下结果:

```

1  $ jython expose.py
2  message.secret = TOP SECRET TEXT

```

字符串“TOP SECRET TEXT”是从 Confidential 类的私有字段 secret 中读取的。

这里没有什么黑魔法: expose.py 使用 Java 反射 API 获取对私有字段 secret 的引用, 然后调用“secret_field.setAccessible(True)”以使其可读。当然, 同样的事情也可以用 Java 来完成(但需要 3 倍以上的代码量; 请参阅**随书代码库**中的文件 `Expose.java`)。

仅当 Jython 脚本或 Java 主程序(例如, `Expose.class`)在安全管理器的监督下运行时, 关键的调用`.setAccessible(True)`才会失败。但在实践中, Java 应用程序很少使用安全管理器进行部署——除非浏览器还支持 Java applet。

我的观点是: 在 Java 中, 访问控制修饰符主要是为了保护(safety), 而不是为了安全(security), 至少在实践中是这样。所以, 安心享受 Python 提供的强大功能。放心地去使用吧。

^a详见“*The Simplest Thing that Could Possibly Work A Conversation with Ward Cunningham, Part V.*”

序列的特殊方法

不要直接检查它是否是一只鸭子,而是要检查的特征与行为。例如,是否像鸭子一样嘎嘎叫、是否像鸭子一样走路等等。具体检查哪些特征,取决于你需要在语言游戏中使用哪些鸭子行为。

——Alex Martelli

在本章中,将创建一个表示多维向量的 `Vector` 类——与“[十一 Pythonic 对象](#)”中的二维 `Vector2d` 相比,这是一个重大进步。`Vector` 类的行为类似于标准的 Python 不可变扁平序列。`Vector` 实例中的元素是浮点数,本章结束后 `Vector` 类将支持如下功能:

- 基于序列协议: `__len__` 与 `__getitem__`。
- 安全地表述拥有多个元素的实例。
- 适当的切片支持,生成新的 `Vector` 实例。
- 综合各个元素的值,计算实例的哈希码。
- 自定义格式化语言扩展。

还将使用 `__getattr__` 实现动态属性访问,以取代在 `Vector2d` 中使用的只读属性——不过,这在序列类型中并不常见。

在大量代码的演示中间,将穿插讨论一个概念——将协议用作非正式接口。届时,将讨论[协议 \(protocol\)](#)与[鸭子类型 \(Duck Typing\)](#)之间的关系,以及对自定义类型的实际意义。

12.1 本章新增内容

本章没有重大变化,在“[12.4 协议与鸭子类型<324页>](#)”结尾处新增的“提示框”中,对 `typing.Protocol` 进行了简短介绍。

“[12.5.2 能处理切片的 __getitem__ 方法](#)”中,“[示例 12.6<328页>](#)”实现的 `__getitem__` 方法比第 1 版(见“[示例 12.2<323页>](#)”)更简洁且更健壮,这要得益于[鸭子类型 \(Duck Typing\)](#)与[operator.index](#)。本章后面与“[十六 运算符重载](#)”中实现的 `Vector` 类也做了同样的改动。

下面开始吧!

12.2 Vector 类: 用户定义序列类型

本章实现 Vector 类的策略是使用组合, 而不使用继承。Vector 类将向量分量存储在 floats 数组中, 并且还将实现不可变扁平序列所需的特殊方法。

不过, 在实现序列方法之前, 要先确保有一个与先前的 Vector2d 类 (“[示例 11.11<307页>](#)”) 兼容的 Vector 基准实现——除非这种兼容没有意义。

三维以上的向量应用程序

谁需要 1,000 维的向量呢? N 维向量 (N 值较大) 广泛应用于信息检索, 其中文档与文本查询以向量表示, 每个单词一个维度。这就是“向量空间模型”。在此模型中, 一个关键的相关性指标是“余弦相似度” (即代表查询的向量与代表文档的向量之间的夹角的余弦)。夹角减小, 余弦值越接近最大值 1, 文档与查询的相关性也越接近最大值。

不过, 本章中的 Vector 类只是一个教学示例, 不会涉及过多的数学运算。我们的目标只是以序列类型为背景, 演示一些 Python 特殊方法。

如果在实践中需要做向量运算, 则应该使用 NumPy 与 SciPy。由 Radim Čehák 开发的 PyPI 包 [gensim](#), 用 NumPy 和 SciPy 为自然语言处理和信息检索实现了向量空间建模。

12.3 Vector 类第 1 版: 与 Vector2d 类兼容

Vector 类的第 1 版应尽可能与先前定义的 Vector2d 类 (“[示例 11.11<307页>](#)”) 兼容。

然而, 在设计上, Vector 构造函数与 Vector2d 构造函数并不兼容。可以通过在 `__init__` 中用 `*args` 获取任意个参数来使 `Vector(3, 4)` 和 `Vector(3, 4, 5)` 正常工作。但序列类型构造函数的最佳实践是接受可迭代对象为参数, 就像所有内置序列类型一样。[示例 12.1](#) 展示了 Vector 类的几种实例化方式。

</> [示例 12.1: 测试 Vector.__init__ 方法与 Vector.__repr__ 方法](#)

```

1 >>> Vector([3.1, 4.2])
2 Vector([3.1, 4.2])
3 >>> Vector((3, 4, 5))
4 Vector([3.0, 4.0, 5.0])
5 >>> Vector(range(10))
6 Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])

```

除了新的构造函数签名之外, 我还确保使用 Vector2d (如 `Vector2d(3,4)`) 进行的每个测试都能通过, 并且可产生与 2 个分量的 `Vector(3,4)` 相同的结果。



当 Vector 的分量超过 6 个时, `repr()` 生成的字符串会缩写为 ..., 如 [示例 12.2](#) 最后一行所示。这对于任何可能包含大量元素的容器类型都至关重要。因为 `repr` 是用于调试的, 您肯定不希望一个大对象在控制台或者日志中输出数千行内容。[replib](#) 模块可用于生成长度有限的表示形式, 如 [示例 12.2](#) 所示。在 Python 2.7 中, `replib` 模块的名称是 `repr`。

示例 12.2 列出了 Vector 类的第 1 版实现代码（以“示例 11.2<297 页>”与“示例 11.3<299 页>”为基础）。

</> 示例 12.2: vector_v1.py: 从 vector2d_v1.py 衍生而来

```
1  from array import array
2  import reprlib
3  import math
4  class Vector:
5      typecode = 'd'
6
7      def __init__(self, components):
8          self._components = array(self.typecode, components) ❶
9
10     def __iter__(self):
11         return iter(self._components) ❷
12
13     def __repr__(self):
14         components = reprlib.repr(self._components)
15         components = components[components.find('['):-1] ❸
16         return f'Vector({components})' ❹
17
18     def __str__(self):
19         return str(tuple(self))
20
21     def __bytes__(self):
22         return (bytes([ord(self.typecode)]) +
23                 bytes(self._components)) ❺
24
25     def __eq__(self, other):
26         return tuple(self) == tuple(other)
27
28     def __abs__(self):
29         return math.hypot(*self) ❻
30
31     def __bool__(self):
32         return bool(abs(self))
33
34     @classmethod
35     def frombytes(cls, octets):
36         typecode = chr(octets[0])
37         memv = memoryview(octets[1:]).cast(typecode)
38         return cls(memv) ❼
```

❶ self._components 是“受保护（protected）”的实例属性。

❷ 为了使 Vector 实例可迭代，用 self._components 构建一个迭代器¹，作为返回值。

❸ 用 reprlib.repr() 为 self._components 获取长度有限的表示形式（例如 array('d', [0.0, 1.0, 2.0, 3.0, 4.0,

¹“十七 迭代器、生成器和经典协程”将介绍 iter() 函数和 __iter__ 方法。

...])。

- ④ 在将字符串插入 Vector 构造函数调用之前, 去除前缀 “array('d',” 与尾部的 “)”。
- ⑤ 直接用 self._components 构建一个 bytes 对象。
- ⑥ 从 Python 3.8 开始, `math.hypot` 接受 N 维坐标点。之前使用的表达式是: `math.sqrt(sum(x * x for x in self))`。
- ⑦ 与先前的 `frombytes` 方法 (见 “[示例 11.3<299页>](#)”) 相比, 仅需改动最后一行: 直接将 `memoryview` 传给构造函数, 无需像先前那样使用星号 (*) 解包。

`reprlib.repr()` 函数 用于为大型结构或递归结构生成安全的 `repr` 表示形式, 它会限制输出字符串的长度, 并用 “...” 表示被截断的部分。Vector 类的底层实现是 `array`, 因此默认的 `repr` 表示形式是 `Vector(array('d', [3.0, 4.0, 5.0]))`。我要用 `reprlib.repr()` 函数 将 `Vector` 实例的表示形式更改为 `Vector([3.0, 4.0, 5.0])`。为了使代码简洁, 我更倾向于让构造函数仅接受一个列表类型的参数。

在编写 `__repr__` 方法时, 本可以使用表达式 `reprlib.repr(list(self._components))` 生成简化的分量显示。但是, 这么做会浪费资源——需要将 `self._components` 中的每个元素都复制到一个 `list` 中, 然后使用列表的 `repr` 表示形式。相反, 我直接为数组 `self._components` 应用了 `reprlib.repr()` 函数, 然后去除中括号 `[]` 之外的字符。如 [示例 12.2](#) 中 `__repr__` 方法的第 2 行所示。



由于 `repr()` 在调试中的作用, 因此 `repr()` 调用决不能引发异常。如果在实现 `__repr__` 时出了问题, 则必须处理这个问题, 并尽可能生成有用的输出, 让用户能够识别接收者 (self)。

请注意, `__str__`、`__eq__` 和 `__bool__` 方法与先前的 `Vector2d` (见 “[示例 11.2<297页>](#)”) 一致, 而 `frombytes` 只更改了 1 个字符 (删除了最后一行的 *)。这是使先前的 `Vector2d` 可迭代的一个好处——即在新的 `Vector` 类中只需进行简单的修改, 便能直接复用 `Vector2d` 的方法。

顺便说一下, 我们本可以让 `Vector` 类继承 `Vector2d`, 但我没这么做, 主要有两点原因: 其一, 二者的构造函数不兼容, 不适合使用继承。这一点通过适当调整 `__init__` 方法的参数可以解决。其二 (更重要的原因), 我希望 `Vector` 成为实现序列协议的类的独立示例。接下来, 我们先讨论 [协议 \(protocol\)](#) 这一术语。然后, 再实现序列协议。

12.4 协议与鸭子类型

如 “[Python 数据模型](#)” 所述, 在 Python 中创建一个功能完备的序列类型, 无需继承任何特殊的类: 仅需实现满足序列协议的方法即可。但此处谈论的哪种协议呢?

在面向对象编程中, 协议是一种非正式的接口, 仅在文档中定义, 而不是在代码中定义。例如, Python 中的序列协议仅需要 `__len__` 与 `__getitem__` 方法。任何以标准签名和语法实现这些方法的类 (如 `Spam`), 都可以在需要序列的地方使用。至于这个类 (如 `Spam`) 是哪个类的子类, 并不重要, 重要的是它提供了必要的方法即可。“[示例 1.1<5页>](#)” 就是一个例子, 这里再次给出相关代码, 如 [示例 12.3](#) 所示。

</> [示例 12.3](#): 为方便起见, 此处转转 “[示例 1.1<5页>](#)” 的代码

```
1 import collections
2
```

```

3 Card = collections.namedtuple('Card', ['rank', 'suit'])
4 class FrenchDeck:    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
5 suits = 'spades diamonds clubs hearts'.split()
6
7     def __init__(self):
8         self._cards = [Card(rank, suit) for suit in self.suits
9                         for rank in self.ranks]
10
11    def __len__(self):
12        return len(self._cards)
13
14    def __getitem__(self, position):
15        return self._cards[position]

```

示例 12.3 中的 FrenchDeck 类利用了许多 Python 功能,因为它实现了序列协议,即便代码中未声明该协议。经验丰富的 Python 程序员只要看一眼代码,就知道 FrenchDeck 是一个序列,即便它是 object 的子类也无妨。我们说 FrenchDeck 是序列,是因为其行为(实现了 `__len__` 与 `__getitem__`)像序列,这才是最重要的。

根据本章开头处 Alex Martelli 的引文,这就是所谓的“[鸭子类型 \(Duck Typing \)](#)”。

由于协议是非正式的,不强制执行。因此,如果知道一个类的具体使用环境,通常仅需实现协议的一部分即可。例如,为了支持迭代,仅需实现 `__getitem__` 方法即可,而无需实现 `__len__`。



实现“[PEP 544 –Protocols: Structural subtyping \(static duck typing\)](#)”之后,Python 开始支持 [协议类 \(Protocol Class\)](#),即“[8.5.10 静态协议<233页>](#)”中的 typing 结构。Python 中的“协议”与传统意义上的协议有关系,但又不完全相同。为了以示区分,我会用 [静态协议 \(static protocol\)](#) 指代 Python 中协议类规定的协议,而用 [动态协议 \(dynamic protocol\)](#) 指代传统意义上的协议。其中的主要区别是,静态协议的实现必须提供协议类中定义的所有方法。更多信息,详见“[13.3 两种协议](#)”。

现在,我们将在 Vector 中实现序列协议,该类最初不完全支持切片功能,但稍后会完善。

12.5 Vector 类第 2 版: 可切片的序列

如“[示例 12.3<324页>](#)”中的 FrenchDeck 类所示,如果能将实现序列协议所需的具体操作(如获取对象长度、获取特定索引处的元素等)委托给对象中的序列属性,那么实现序列协议将变得非常简单。下述代码块中仅有 1 行代码的 `__len__` 与 `__getitem__` 方法就是一个很好的开始。

```

1 class Vector:
2     # 省略多行
3     # ...
4
5     def __len__(self):
6         return len(self._components)
7     def __getitem__(self, index):
8         return self._components[index]

```

新增了这两个方法之后,如下操作都可正常执行:

```

1  >>> v1 = Vector([3, 4, 5])
2  >>> len(v1)
3  3
4  >>> v1[0], v1[-1]
5  (3.0, 5.0)
6  >>> v7 = Vector(range(7))
7  >>> v7[1:4]
8  array('d', [1.0, 2.0, 3.0])

```

如您所见,连切片功能都支持了,只是尚不完美。如果 Vector 的切片也是一个 Vector 实例而不是数组,那就更好了。先前的 FrenchDeck 类(“[示例 12.3<324页>](#)”)也有类似的问题:当你对 FrenchDeck 实例进行切片时,得到的是一个列表。对 Vector(“[示例 12.2<323页>](#)”)来说,如果切片生成普通的数组,将会失去很多功能。

想想内置的序列类型:每种类型在执行切片后,都会生成一个原类型的新实例,而不是其他类型的实例。

为了使 Vector 切片操作也生成一个新的 Vector 实例,不能简单的将切片操作委托给 array 数组。需要分析在 `__getitem__` 方法中得到的参数,并做适当地处理。

现在,看看 Python 如何将 `my_seq[1:3]` 语法转化为 `my_seq.__getitem__(...)` 的参数。

12.5.1 切片原理

一例胜千言,下面来看一下??。

```

</> 示例 12.4: 检查 __getitem__ 与切片的行为
1  >>> class MySeq:
2  ...     def __getitem__(self, index):
3  ...         return index  ❶
4  ...
5  >>> s = MySeq()
6  >>> s[1]          ❷
7  1
8  >>> s[1:4]        ❸
9  slice(1, 4, None)
10 >>> s[1:4:2]      ❹
11 slice(1, 4, 2)
12 >>> s[1:4:2, 9]   ❺
13 (slice(1, 4, 2), 9)
14 >>> s[1:4:2, 7:9] ❻
15 (slice(1, 4, 2), slice(7, 9, None))

```

❶ 在此示例中, `__getitem__` 直接返回传给它的值。

❷ 单个索引,没什么新鲜的。

❸ “`1:4`” 表示法变成了 “`slice(1, 4, None)`”。

❹ “`slice(1, 4, 2)`” 表示从 1 开始,到 4 停止,步长为 2。

❺ 神奇的事情发生了:如果 [] 中有逗号,则 `__getitem__` 接收到的将是一个元组。

⑥ 元组中甚至可以保存多个 slice 对象。

现在, 仔细看看 slice 本身, 如 [示例 12.5](#) 所示。

</> [示例 12.5: 检查 slice 类的属性](#)

```

1  >>> slice          ❶
2  <class 'slice'>
3  >>> dir(slice)    ❷
4  ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
5  '__format__', '__ge__', '__getattribute__', '__gt__',
6  '__hash__', '__init__', '__le__', '__lt__', '__ne__',
7  '__new__', '__reduce__', '__reduce_ex__', '__repr__',
8  '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
9  'indices', 'start', 'step', 'stop']

```

❶ slice 是一种内置类型 (首次出现在 “[2.7.2 切片对象](#)”)。

❷ 检查 slice 时, 会发现它有 3 种数据属性 start、stop 和 step, 以及方法 indices。

在 [示例 12.5](#) 中, 调用 dir(slice) 会显示一个 indices 属性, 它是一个非常有趣但鲜为人知的方法。 help(slice.indices) 给出的信息如下:

S.indices(len) -> (start, stop, stride)

假定序列长度为 len, 计算由 S 所描述的扩展切片的起始 (start) 索引、结束 (stop) 索引以及步长 (stride)。超出边界的索引会被截掉, 就像常规切片一样。

换句话说, indices 方法开放了内置序列实现的棘手逻辑, 以便可优雅地处理缺失索引或负值索引, 以及长度超过原始序列的切片。该方法会“规范化”元组, 将 start、stop、stride 都变成非负值, 而且都落在指定长度序列的边界内。

下面举几个例子。假设有一个长度为 5 的序列, 例如 “ABCDE”。

```

1  >>> slice(None, 10, 2).indices(5)      ❶
2  (0, 5, 2)
3  >>> slice(-3, None, None).indices(5)    ❷
4  (2, 5, 1)

```

❶ ““ABCDE”[:10:2]” 等同于 ““ABCDE”[0:5:2]”。

❷ ““ABCDE”[-3:]” 等同于 ““ABCDE”[2:5:1]”。

在 Vector 中, 无需使用 slice.indices() 方法, 因为当我们收到切片参数时, 会将切片操作委托给 self._components 数组。但是, 如果你不能依靠底层序列的服务, slice.indices() 方法能节省大量时间。

现在, 知道如何处理切片了。让我们看一下改进后的 Vector.__getitem__ 实现。

12.5.2 能处理切片的 __getitem__ 方法

[示例 12.6](#) 列出了让 Vector 表现为序列所需的两个方法: __len__ 与 __getitem__ (后者现在可以正确处理切片了)。

```
</> 示例 12.6: vector_v2.py 部分代码: 为 Vector (“示例 12.2<323页>” ) 添加 __len__ 与 __getitem__
```

```

1 def __len__(self):
2     return len(self._components)
3
4 def __getitem__(self, key):
5     if isinstance(key, slice):
6         cls = type(self)
7         return cls(self._components[key])
8     index = operator.index(key)
9     return self._components[index]

```

- ❶ 如果参数 key 是一个 slice 对象 ...。
- ❷ ... 获取实例的类 (即 Vector), 然后 ...。
- ❸ ... 调用 Vector 类的构造函数, 以从 _components 数组的切片构建一个新的 Vector 实例。
- ❹ 如果从参数 key 中得到的是单个索引 ...。
- ❺ ... 从 _components 返回特定的元素。

函数 `operator.index()` 调用了 `__index__` 特殊方法。该函数与特殊方法在 `Travis Oliphant` 提出的 “[PEP 357 -Allowing Any Object to be Used for Slicing](#)” 中定义, 目的是允许将 NumPy 中多种类型的整数用作索引和切片参数。`operator.index()` 与 `int()` 的主要区别在于, 前者是为这一特定目的而设计的。例如, `int(3.14)` 返回 3, 但 `operator.index(3.14)` 会引发 `TypeError`, 因为浮点数不能用作序列的索引。



过度使用 `isinstance` 可能是糟糕的面向对象设计的标志, 不过在 `__getitem__` 使用它处理切片是合理的。本书第 1 版中, 还用 `isinstance` 测试了 `key`, 判断 `key` 是不是整数。使用 `operator.index()` 之后, 可以避免这个测试, 如果无法从 `key` 中获取索引, 则会引发 `TypeError`, 并给出一条丰富的异常信息。参见 [示例 12.7](#) 中的最后一条错误消息。

将 [示例 12.6](#) 中的代码添加到 `Vector` 类 (“[示例 12.2<323页>](#)”) 后, 就可以拥有正确的切片行为了, 如 [示例 12.7](#) 所示。

```
</> 示例 12.7: 测试 示例 12.6 中增强的 __getitem__
```

```

1 >>> v7 = Vector(range(7))
2 >>> v7[-1]      ❶
3 6.0
4 >>> v7[1:4]    ❷
5 Vector([1.0, 2.0, 3.0])
6 >>> v7[-1:]    ❸
7 Vector([6.0])
8 >>> v7[1,2]     ❹
9 Traceback (most recent call last):
10 ...
11 TypeError: 'tuple' object cannot be interpreted as an integer

```

- ❶ 单个整数索引, 仅获取一个 float 类型的分量。

- ② 切片索引, 将创建一个新的 Vector 实例。
- ③ 长度为 1 (即 `len==1`) 的切片, 也会创建一个 Vector 实例。
- ④ Vector 不支持多维索引, 因此索引元组或多维切片会引发异常。

12.6 Vector 类第 3 版: 动态访问属性

从 `Vector2d` (“[示例 11.11<307页>](#)”) 到 `Vector` 的演变过程中, 我们失去了通过名称 (如 `v.x`、`v.y`) 访问向量分量的能力。现在要处理的是可能含有大量分量的向量。不过, 如果能通过单个字母访问前几个分量的话, 会比较方便。例如, 用 `x`、`y`、`z` 代替 `v[0]`、`v[1]`、`v[2]`。

下面是我们要提供的另一种语法, 用于读取向量的前 4 个分量:

```

1 >>> v = Vector(range(10))
2 >>> v.x
3 0.0
4 >> v.y, v.z, v.t
5 (1.0, 2.0, 3.0)

```

在 `Vector2d` 中, 我们使用 `@property` 装饰器 将属性 `x` 与 `y` 标记为只读特性 (property) (见 “[示例 11.7<304页>](#)”)。可以在 `Vector` 中也编写 4 个特性 (property), 但这样做很繁琐。特殊方法 `__getattr__` 提供了一种更好的方法。

当属性查找失败时, 解释器会调用 `__getattr__` 方法。简单来说, 给定表达式 `my_obj.x`, Python 会检查 `my_obj` 实例是否拥有名为 `x` 的属性; 如果没有, 就到类 (`my_obj.__class__`) 中查找; 如果还没有, 就沿着继承关系图, 继续向上查找²。如果依旧未找到属性 `x`, 则调用 `my_obj` 所属的类中定义的 `__getattr__` 方法, 调用时为该方法传入 2 个参数: `self` 与字符串形式的属性名 (如 “`x`”)。

[示例 12.8](#) 列出了我们的 `__getattr__` 方法。该方法主要检查所查找的属性是否为 `xyzt` 字母之一, 如果是, 则返回相应的向量分量。

</> [示例 12.8: vector_v3.py 部分代码: 向 Vector 类添加 __getattr__ 方法](#)

```

1     __match_args__ = ('x', 'y', 'z', 't')           ❶
2
3     def __getattr__(self, name):
4         cls = type(self)    ❷
5         try:
6             pos = cls.__match_args__.index(name)    ❸
7         except ValueError:  ❹
8             pos = -1
9         if 0 <= pos < len(self._components):    ❺
10            return self._components[pos]
11
12     msg = f'{cls.__name__!r} object has no attribute {name!r}' ❻
13     raise AttributeError(msg)

```

- ❶ 设置 `__match_args__`, 以允许对 `__getattr__` 支持的动态属性进行位置模式匹配³。

² 属性查找机制要比这复杂得多, 具体细节在 “[五 延伸阅读](#)” 中讲解。目前, 仅需知道这种简单的机制即可。

³ 尽管 `__match_args__` 的存在是为了支持 Python 3.10 中的模式匹配, 但在以前的 Python 版本中设置该属性也是无害的。在本书第 1 版中, 我将其命名为 `shortcut_names`。第 2 版中使用的这个新名字, 它就身兼两职: 其一, 它支持 `case` 子句中的位置模式; 其二, 保存

- ❷ 获取 Vector 类, 以供以后使用。
- ❸ 尝试获取 name 在 `__match_args__` 中的位置。
- ❹ `.index(name)` 在未找到 name 时会引发 `ValueError`; 将 pos 设为-1。(我也想在此处使用类似 `str.find` 的方法, 但 `tuple` 没有实现这样的方法。)
- ❺ 如果 pos 在可用分量的范围内, 则返回该分量。
- ❻ 如果我们走到这一步, 就会引发带有标准信息文本的 `AttributeError`。

实现 `__getattr__` 方法并不难, 但只像 [示例 12.8](#) 这样实现, 还远远不够。详见 [示例 12.9](#) 中诡异的交互。

</> [示例 12.9](#): 不恰当的行为: 赋值给 `v.x` 不会引发错误, 但会导致前后不一致

```

1  >>> v = Vector(range(5))
2  >>> v
3  Vector([0.0, 1.0, 2.0, 3.0, 4.0])
4  >>> v.x      ❶
5  0.0
6  >>> v.x = 10    ❷
7  >>> v.x
8  10
9  >>> v
10 Vector([0.0, 1.0, 2.0, 3.0, 4.0]) ❸

```

- ❶ 用 `v.x` 获取第一个分量 (`v[0]`)。
- ❷ 为 `v.x` 分配新值。这一操作应该引发异常。
- ❸ 读取 `v.x` 显示了新值为 10。
- ❹ 然而, 向量的分量并没有改变。

您能解释为什么会这样吗? 特别是, 如果 `v.x` 的值没在向量分量数组中, 为何 `v.x` 第二次返回 10? 如果不能马上给出答案, 请仔细阅读 [示例 12.8](#) 前面对 `__getattr__` 方法的说明。虽然原因不是很明显, 但却是理解本书后续内容的重要基础。

仔细思考后, 请继续阅读, 余下内容将向您解释到底发生了什么。

[示例 12.9](#) 中的不一致是由于 `__getattr__` 的工作方式引起的: 只有在对象中没有给定名称的属性时, Python 才会调用 `__getattr__` 方法, 这是一种后备机制。但是, [示例 12.9](#) 在赋值 `v.x = 10` 之后, 对象 `v` 现在有了名为 `x` 的属性。所以, 不会再调用 `__getattr__` 检索 `v.x`: 解释器将只需返回绑定到 `v.x` 的值 10。另一方面, [示例 12.8](#) 中实现的 `__getattr__` 并不关注 `self.__components` 以外的实例属性, 而是从 `self.__components` 中获取 `__match_args__` 列出的“虚拟属性”。

为了避免 [示例 12.9](#) 中的这种前后不一致, 需要改写 `Vector` 类中设置属性(即向量分量)的逻辑。

回想“[十一 Pythonic 对象](#)”中最新的 `Vector2d` 示例(“[示例 11.7<304页>](#)”), 如果尝试赋值给 `.x` 或 `.y` 实例属性, 就会引发 `AttributeError`。在 `Vector` 中, 我们希望对所有单字母小写的属性名进行赋值时, 也引发同样的异常。为此, 要为 `Vector` 类实现 `__setattr__` 方法, 如 [示例 12.10](#) 所示。

</> [示例 12.10](#): `vector_v3.py` 部分代码: 在 `Vector` 类中实现 `__setattr__` 方法

```
def __setattr__(self, name, value):
```

了 `__getattr__` 与 `__setattr__` 中特殊逻辑所支持的动态属性名称。

```

2     cls = type(self)
3     if len(name) == 1:          ❶
4         if name in cls.__match_args__:
5             error = 'readonly attribute {attr_name!r}' ❷
6         elif name.islower():      ❸
7             error = "can't set attributes 'a' to 'z' in {cls_name!r}"
8         else:
9             error = ''          ❹
10        if error:
11            msg = error.format(cls_name=cls.__name__, attr_name=name)
12            raise AttributeError(msg)
13        super().__setattr__(name, value)          ❺

```

- ❶ 对单字符的属性名, 要进行特殊处理。
- ❷ 如果 name 在 `__match_args__` 中, 则设置特定的错误消息。
- ❸ 如果属性名称是小写字母, 则设置一个针对小写字母属性的错误消息。
- ❹ 否则, 设置空白的错误消息。
- ❺ 如果 error 不为空, 则引发 `AttributeError`。
- ❻ 默认情况: 在超类上调用 `__setattr__`, 以提供标准的赋值行为。



`super()` 函数 用于动态访问超类的方法, 这在 Python 这种支持多重继承的动态语言中是必需的。如 [示例 12.10](#) 所示, 它用来将子类方法的某些任务委托给超类中合适的方法。关于 `super()` 的更多信息, 请参阅 “[14.4 多重继承与方法解析顺序](#)”。

在为 `AttributeError` 选择错误消息的措辞时, 我参考了内置类型 `complex` 的行为, 因为 `complex` 对象是不可变的, 而且有一对数据属性: `real` 与 `imag`。如果尝试更改 `complex` 实例的任何一个属性, `complex` 实例都会引发 `AttributeError` 异常, 错误消息是 “`can't set attribute`”。另一方面, 如果尝试为受特性 (property) 保护 ([“11.7 可哈希的 Vector2d”](#)) 的只读属性赋值, 则会得到错误消息 “`read-only attribute`”。我从这两种措辞中汲取了灵感, 在 `__setattr__` 中设置了错误消息, 更明确地指出了禁止赋值的属性。

请注意, 我们并没有禁止为所有属性赋值, 只是禁止为单个小写字母属性赋值, 以避免与支持的只读属性 `x`、`y`、`z` 和 `t` 相混淆。



我们知道, 在类中声明类属性 `__slots__`, 可以防止动态创建实例属性。我们也很想用这一特性来代替我们的 `__setattr__` 实现。然而, 正如 “[11.11.2 总结 `__slots__` 的问题](#)” 所指, 不应只为了避免动态创建实例属性而使用 `类属性 __slots__`。
`__slots__` 应该只用于节省内存, 而且只有在内存严重不足时, 才启用 `类属性 __slots__`。

尽管 `Vector` 类还不够完善 (无法为分量赋值), 但却从该类中得到一个重要启示: 通常在实现 `__getattr__` 方法的同时, 还要编写代码实现 `__setattr__`, 以避免对象行为的不一致 (如 “[示例 12.9](#)” 所示)。

若想要 `Vector` 类支持分量赋值, 则可以实现 `__setitem__` 方法 以支持 `v[0]=1.1`, 以及 (或者) 实现 `__setattr__` 方法 以支持 `v.x=1.1`。不过, 要保持 `Vector` 类是不可变的, 因为 “[12.7 Vector 类第 4 版: 哈希与快](#)

“**速等值测试**”要将 `Vector` 类变成 **可哈希 (hashable)** 的。

12.7 `Vector` 类第 4 版: 哈希与快速等值测试

本节要再次实现 `__hash__` 方法。加上现有的 `__eq__` 方法, 这将使 `Vector` 实例变成 **可哈希 (hashable)** 的对象。

`Vector2d` 中的 `__hash__` (“[示例 11.8<305页>](#)”) 方法计算了包含 `self.x` 与 `self.y` 这两个分量的元组的哈希码。现在, `Vector` 类要处理的分量可能会多达上千个, 因此构建元组可能会消耗太多的资源。鉴于此, 我将会对向量实例每个分量的 **哈希码** 依次应用 `^` (异或) 运算符, 就像这样: “`v[0] ^ v[1] ^ v[2]`”。这也正是 `functools.reduce` 函数的作用。之前我说过 `reduce` 函数 不再像以前那样流行⁴, 但计算向量所有分量的哈希码非常适合使用 `reduce` 函数。该函数的整体思路如 [图 12.1](#) 所示。

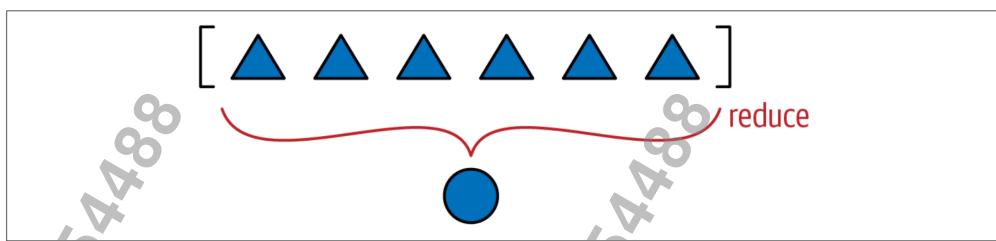


图 12.1: 归约函数 (`reduce`、`sum`、`any`、`all`) 将有限的可迭代对象聚合为单个结果

我们已知道, `sum()` 函数可以代替 `functools.reduce()`, 下面说说它的原理。`reduce` 函数的关键思想是, 将一系列值归约成单个值。`functools.reduce(func, iter)` 的参数 `func` 是一个可接受 2 个参数的函数, 参数 `iter` 是一个可迭代对象。假设有一个接受 2 个参数的函数 `fn` 与一个列表 `lst`。当调用 `reduce(fn, lst)` 时, `fn` 将被应用于 `lst` 中的第一对元素——`fn(lst[0], lst[1])`, 以生成第一个结果 `r1`。然后, `fn` 会被应用到 `r1` 与 `lst` 中的下一个元素——`fn(r1, lst[2])`, 以生成第二个结果 `r2`。再调用 `fn(r2, lst[3])` 生成 `r3` … 直至 `lst` 中的最后一个元素, 返回最终得到的结果 `rN`。

以下是如何使用 `reduce` 函数 计算 $5!$ (5 的阶乘):

```

1  >>> 2 * 3 * 4 * 5 # 想要的结果是 5! == 120
2  120
3  >>> import functools
4  >>> functools.reduce(lambda a,b: a*b, range(1, 6))
5  120

```

回到我们的可哈希 (hashable) 问题上。[示例 12.11](#) 展示了计算聚合异或 (`xor`) 的 3 种方式: 一种用 `for` 循环, 其余两种用 `reduce` 函数。

</> [示例 12.11: 计算 0 到 5 整数累加异或 \(xor\) 的三种方法](#)

```

1  >>> n = 0
2  >>> for i in range(1, 6): ❶
3  ...     n ^= i
4  ...
5  >>> n

```

⁴`sum`、`any` 和 `all` 涵盖了 `reduce` 最常见的用途。详见 “[7.3.1 map、filter 与 reduce 的替代品<190页>](#)” 一节的讨论。

```

1
2 1>>> import functools>>> functools.reduce(lambda a, b: a^b, range(6)) ②
3
4 1>>> import operator
5
6 1>>> functools.reduce(operator.xor, range(6)) ③
7
8 1

```

- ① 使用 for 循环和累加器变量计算聚合异或 (xor)。
- ② 使用 `functools.reduce` 函数, 传入匿名函数。
- ③ 使用 `functools.reduce` 函数, 将 `lambda` 表达式替换成 `operator.xor`。

示例 12.11 中的多个备选方案中, 我最喜欢最后一种, 其次是 for 循环。你呢?

如 “7.8.1 operator 模块<197页>” 所述, `operator` 模块 为所有中缀运算符提供了等效函数, 从而减少了使用 `lambda` 的必要。

为了使用我喜欢的方式编写 `Vector.__hash__` 方法, 需要导入 `functools` 模块 与 `operator` 模块。`Vector` 类的相关改动如??所示。

</> 示例 12.12: `vector_v4.py` 部分代码: 在 `vector_v3.py` 基础上导入 2 个模块, 并增加 `__hash__`

```

1 1>>> from array import array
2 2>>> import reprlib
3 3>>> import math
4 4>>> import functools
5 5>>> import operator
6
7 class Vector:
8     typecode = 'd'
9     # 省略了很多行...
10
11     def __eq__(self, other): ③
12         return tuple(self) == tuple(other)
13     def __hash__(self):
14         hashes = (hash(x) for x in self._components) ④
15         return functools.reduce(operator.xor, hashes, 0) ⑤
16         # 省略了很多行...

```

- ① 导入 `functools` 模块, 以使用 `reduce` 函数。
- ② 导入 `operator` 模块, 以使用 `xor` 函数。
- ③ `__eq__` 方法 没有变化。这里将它列出来, 是因为在源码中将 `__eq__` 与 `__hash__` 放在一起是一种很好的习惯, 因为它们要结合在一起使用。
- ④ 创建一个生成器表达式, 惰性计算向量实例中各个分量的哈希码。
- ⑤ 将序列 `hashes` 提供给 `reduce` 函数, 使用 `operator.xor` 函数计算聚合的哈希码。第 3 个参数 (0) 是初始值 (见下面的“警告栏”)。



在使用 `reduce(function, iterable, initializer)` 时, 最好提供第 3 个参数, 以防出现异常: “`TypeError: reduce() of empty sequence with no initial value`” (这是个很棒的错误消息, 不仅提出了问题, 还给出了解决方案)。若参数 `iterable` 为空, 则 `reduce` 函数的返回值就是 `initializer`。否则, 在归约循环中, 以 `initializer` 作为第一个参数。因此, `initializer` 应该是所执行操作的 **身份值 (Identity Value)**。例如, 对于 `+`、`l.^` 来说, `initializer` 应该是 `0`, 但对于 `*`、`&` 来说, `initializer` 应该是 `1`。

示例 12.12 实现的 `__hash__` 方法是一种完美的 map-reduce 计算, 如 图 12.2 所示。

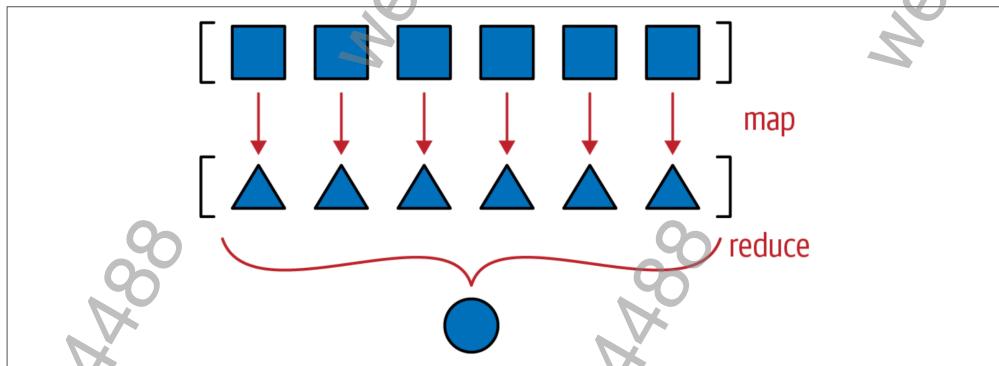


图 12.2: 映射 (map): 先将函数应用于每个项以生成新序列
归约 (reduce): 再计算聚合值

映射 (map) 步骤为向量的每个分量生成一个 **哈希码**, 而归约 (reduce) 步骤用 `xor` 运算符 聚合所有 **哈希码**。将生成器表达式替换为 map, 可使映射 (map) 步骤更加明显:

```
1 def __hash__(self):
2     hashes = map(hash, self._components)
3     return functools.reduce(operator.xor, hashes)
```



函数 `map` 在 Python 2 中的效率较低, 因为 `map` 要用计算结果构建一个新 `list`。但是, 在 Python 3 中, **函数 map** 是惰性的: 它会创建一个生成器, 按需产出结果, 因此更节省内存——这与“示例 12.12<333页>”中 `__hash__` 方法使用的生成器表达式一样。

说到归约 (reduce) 函数, 我们可以用一种更高效的解决方案来取代我们快捷的 `__eq__` (见“示例 11.2<297页>”实现)。至少对于大型向量来说, 新方案在处理时间和内存占用方面成本更低。先前的快捷 `__eq__` 实现代码如下所示:

```
1 def __eq__(self, other):
2     return tuple(self) == tuple(other)
```

这种快捷的 `__eq__` 实现方案对 `Vector2d` 与 `Vector` 都有效——但是该方案会认为 `Vector([1, 2])` 与 `(1, 2)` 相等, 这可能是个 bug, 但现在暂且忽略不计⁵。但对于可能具有数千个分量的 `Vector` 实例来说, 这种方法

⁵“节 16.2<452页>”将认真对待“`Vector([1, 2]) == (1, 2)`”的问题。

非常低效。它构建了 2 个元组, 完整复制了操作数的内容, 只是为了使用 tuple 类型的 `__eq__` 方法。对于只有 2 个分量的 `Vector2d` (只有 2 个分量) 而言, 这是个捷径。但对于大型多维向量来说, 不是一个好办法。将一个 `Vector` 与另一个 `Vector` (或可迭代对象) 进行比较的更好方式, 如示例 12.13 所示。

</> 示例 12.13: `Vector.__eq__` 的实现: 在 for 循环中使用 `zip` 函数, 以提高效率。

```

1  def __eq__(self, other):
2      if len(self) != len(other):
3          return False
4      for a, b in zip(self, other):
5          if a != b:
6              return False
7      return True

```

- ❶ 如果两个对象的长度不同, 则它们不相等。
- ❷ 函数 `zip` 会生成一个由 `tuple` 构成的生成器。`tuple` 中的元素来自参数传入的 2 个可迭代对象。有关函数 `zip` 的信息, 请参阅后文附注栏的“出色的 `zip` 函数”。❶ 处的长度判断是有必要的, 因为一旦其中一个输入耗尽, 函数 `zip` 就会在没有任何警告的情况下, 停止生成数值。
- ❸ 一旦有两个分量不同, 就退出比较并返回 `False`。
- ❹ 否则, 两个对象是相等的。



函数 `zip` 的名称取自“拉链 (zipper)”, 因为拉链会将两侧的一对链牙咬合在一起, 这形象地说明了函数 `zip(left,right)` 的作用。函数 `zip` 与文件压缩无关。

示例 12.13 的效率很高。不过, 用单行的 `all` 函数替换整个 `for` 循环, 可使代码更简洁。在 `all` 函数中, 如果所有对应分量的比较结果都是 `True`, 则 `all` 函数就会返回 `True`; 只有任意一次比较结果是 `False`, `all` 函数就会返回 `False`。用函数 `all` 实现 `__eq__` 方法的方式, 如示例 12.14 所示。

</> 示例 12.14: 用函数 `zip` 与 `all` 实现的 `Vector.__eq__`, 逻辑与示例 12.13 相同

```

1  def __eq__(self, other):
2      return len(self) == len(other) and all(a == b for a, b in zip(self, other))

```

请注意, 我们要先检查 2 个运算对象是否等长, 因为函数 `zip` 会在最短的运算对象耗尽时, 停止生成数值。

我们选择在 `vector_v4.py` 中采用示例 12.14 中实现的 `__eq__` 方法。

出色的 `zip` 函数

使用 `for` 循环来迭代元素, 无需处理索引变量, 还可以避免许多 bug, 但需要一些特殊的实用函数 (如 `zip`)。函数 `zip` 可以轻松地并行遍历两个或多个可迭代对象, 返回的元组可以解包为多个变量。每个变量分别对应到并行输入中的各个可迭代对象的一个元素。参见示例 12.15。

</> 示例 12.15: 内置函数 `zip` 的使用示例

```

1  >>> zip(range(3), 'ABC')
2  <zip object at 0x10063ae48>
3  >>> list(zip(range(3), 'ABC'))
4  [(0, 'A'), (1, 'B'), (2, 'C')]
5  >>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3]))
6  [(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
7  >>> from itertools import zip_longest
8  >>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3], fillvalue=-1))
9  [(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]

```

- ❶ zip 返回一个生成器, 可按需生成元组。
- ❷ 为了显示函数 zip 的结果, 构建一个列表; 实践中, 通常会遍历❶ 处返回的生成器。
- ❸ 当一个可迭代对象耗尽后, 函数 zip 会在没有警告的情况下, 停止生成元组。
- ❹ 函数 `itertools.zip_longest` 的行为与 zip 不同。`itertools.zip_longest` 用可选的 `fillvalue`(默认为 `None`) 来填充缺失的值, 因此可以在某个可迭代对象耗尽时继续生成元组, 直至最后一个可迭代对象耗尽。



Python 3.10 为函数 zip 新增的可选参数

本书第 1 版中说过, 函数 `zip` 在最短的可迭代对象处默默停止是令人惊讶的, 这不是一个良好的 API 特性。默默忽略部分输入可能会导致微妙的 bug。相反, 如果可迭代对象的长度不同, 函数 `zip` 应该引发 `ValueError`, 这也是在将一个可迭代对象解包到不同长度的变量元组时会发生的情况, 符合 Python 的 Fail-Fast (快速失败) 策略。“[PEP 618 –Add Optional Length-Checking To zip](#)”(Python 3.10 已实现) 为函数 `zip` 添加了一个可选参数 `strict`, 以表现这种行为。

函数 `zip` 还可用于对嵌套可迭代对象表示的矩阵进行转置。

```

1  >>> a = [(1, 2, 3),
2  ...      (4, 5, 6)]
3  >>> list(zip(*a))
4  [(1, 4), (2, 5), (3, 6)]
5  >>> b = [(1, 2),
6  ...      (3, 4),
7  ...      (5, 6)]
8  >>> list(zip(*b))
9  [(1, 3, 5), (2, 4, 6)]

```

如果您想了解 `zip`, 请花一些时间弄清楚这些示例是如何工作的。

内置函数 `enumerate` 是另一个常用于 `for` 循环的生成器函数, 以避免直接处理索引变量。详细介绍, 见“[Built-in Functions](#)”文档。内置函数 `zip`、`enumerate`, 以及标准库中其他几个生成器函数, 将在“[17.9 标准库中的生成器函数](#)”中讨论。

本章最后, 像 `Vector2d` (“[示例 11.11](#)”[307页](#)”那样, 为 `Vector` 类实现了 `__format__` 方法。

12.8 Vector类第5版:格式化

Vector的`__format__`方法与Vector2d(“[示例11.6<302页>](#)”)类似。但Vector不使用极坐标,而是使用球面坐标(又称“超球面”坐标)。因为Vector类支持n个维度,而超过四维(4D)后,球面就变成了“超球面”⁶。因此,我们将把自定义格式后缀从“p”改为“h”。



正如“[11.6 格式化显示<300页>](#)”所述,在扩展`格式化规范微语言`时,应避免重复使用内置类型所支持的格式化代码。特别是,本章对微语言的扩展还用到了`float`类型的格式化代码“`eEfFgGn%`”(保留原意),因此要避免使用这些代码。整数用“`bcdoXn`”,字符串用“`s`”。我为Vector2d(“[示例11.6<302页>](#)”)的极坐标选择了“`p`”,而为Vector的超球面坐标选择了代码“`h`”。

例如,给定一个4D空间中的Vector对象(`len(v) == 4`),代码“`h`”将产生类似`<r,Φ₁,Φ₂,Φ₃>`的显示。其中,`r`是Vector对象的模(即`abs(v)`的值),余下的数字为角度分量 $Φ_1, Φ_2, Φ_3$ 。

以下是4D中球面坐标格式的一些示例,摘自`vector_v5.py`的`doctests`([见示例12.16](#)):

```

1 >>> format(Vector([-1, -1, -1, -1]), 'h')
2 '<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
3 >>> format(Vector([2, 2, 2, 2]), '.3eh')
4 '<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
5 >>> format(Vector([0, 1, 0, 0]), '0.5fh')
6 '<1.00000, 1.57080, 0.00000, 0.00000>'
```

在小幅改动`__format__`方法之前,需要先编写一对辅助方法:`angle(n)`用于计算某个角度分量(如 $Φ_1$);`angles()`用于计算返回所有角度分量的可迭代对象。本书不会讲解其中所涉及的数学理论,若您对此好奇,可查看维基百科中的“`n-sphere`”词条。其中,包含了用于计算从Vector分量数组中的直角坐标转换为球面坐标的几个公式。

[示例12.16](#)是`vector_v5.py`脚本的完整代码⁷,包含自“[12.3 Vector类第1版:与Vector2d类兼容<322页>](#)”以来实现的所有代码,以及本节实现的自定义显示(`__format__`)格式。

</> [示例12.16: vector_v5.py:Vector类最终版的doctest与完整代码](#)

```

1 """
2 一个多维向量类 ``Vector``, 第5版
3
4 用数字可迭代对象构建 ``Vector`` 实例::
5 >>> Vector([3.1, 4.2])
6 Vector([3.1, 4.2])
7 >>> Vector((3, 4, 5))
8 Vector([3.0, 4.0, 5.0])
9 >>> Vector(range(10))
10 Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
11
12 测试二维向量(结果与 ``vector2d_v1.py`` (``示例11.11<307页>``) 相同)::
```

⁶Wolfram Mathworld网站有一篇关于超球面的文章:“[Hypersphere](#)”;维基百科关于超球面的介绍见“`n-sphere`”词条。

⁷带黑圈标号的那几行是为了支持`__format__`方法而添加的代码。


```
63  >>> v1 = Vector([3, 4, 5])
64  >>> v1_clone = Vector.frombytes(bytes(v1))      >>> v1_clone
65  Vector([3.0, 4.0, 5.0])
66  >>> v1 == v1_clone
67  True
68
69  测试序列行为::
70  >>> v1 = Vector([3, 4, 5])
71  >>> len(v1)
72  3
73  >>> v1[0], v1[len(v1)-1], v1[-1]
74  (3.0, 5.0, 5.0)
75
76  测试切片::
77  >>> v7 = Vector(range(7))
78  >>> v7[-1]
79  6.0
80  >>> v7[1:4]
81  Vector([1.0, 2.0, 3.0])
82  >>> v7[-1:]
83  Vector([6.0])
84  >>> v7[1,2]
85  Traceback (most recent call last):
86  ...
87  TypeError: 'tuple' object cannot be interpreted as an integer
88
89  测试动态属性访问::
90  >>> v7 = Vector(range(10))
91  >>> v7.x
92  0.0
93  >>> v7.y, v7.z, v7.t
94  (1.0, 2.0, 3.0)
95
96  动态属性查找失败::
97  >>> v7.k
98  Traceback (most recent call last):
99  ...
100 AttributeError: 'Vector' object has no attribute 'k'
101 >>> v3 = Vector(range(3))
102 >>> v3.t
103 Traceback (most recent call last):
104 ...
105 AttributeError: 'Vector' object has no attribute 't'
106 >>> v3.spam
107 Traceback (most recent call last):
108 ...
109 AttributeError: 'Vector' object has no attribute 'spam'
110
111 测试哈希::
112 >>> v1 = Vector([3, 4])
```

```

113     >>> v2 = Vector([3.1, 4.2])
114     >>> v3 = Vector([3, 4, 5])      >>> v6 = Vector(range(6))
115     >>> hash(v1), hash(v3), hash(v6)
116     (7, 2, 1)
117
118     大多数非整数的哈希码在 32-bit 与 64-bit 的 CPython 中不一样::
119     >>> import sys
120     >>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else 357915986)
121     True
122
123     测试用 ``format()`` 格式化二维直角坐标系::
124     >>> v1 = Vector([3, 4])
125     >>> format(v1)
126     '(3.0, 4.0)'
127     >>> format(v1, '.2f')
128     '(3.00, 4.00)'
129     >>> format(v1, '.3e')
130     '(3.000e+00, 4.000e+00)'
131
132     测试用 ``format()`` 格式化三维与七维直角坐标::
133     >>> v3 = Vector([3, 4, 5])
134     >>> format(v3)
135     '(3.0, 4.0, 5.0)'
136     >>> format(Vector(range(7)))
137     '(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'
138
139     测试用 ``format()`` 格式化二维、三维、四维球面坐标::
140     >>> format(Vector([1, 1]), 'h') # doctest:+ELLIPSIS
141     '<1.414213..., 0.785398...>'
142     >>> format(Vector([1, 1]), '.3eh')
143     '<1.41421, 0.78540>'
144     >>> format(Vector([1, 1]), '0.5fh')
145     '<1.41421, 0.78540>'
146     >>> format(Vector([1, 1, 1]), 'h') # doctest:+ELLIPSIS
147     '<1.73205..., 0.95531..., 0.78539...>'
148     >>> format(Vector([2, 2, 2]), '.3eh')
149     '<3.464e+00, 9.553e-01, 7.854e-01>'
150     >>> format(Vector([0, 0, 0]), '0.5fh')
151     '<0.00000, 0.00000, 0.00000>'
152     >>> format(Vector([-1, -1, -1, -1]), 'h') # doctest:+ELLIPSIS
153     '<2.0, 2.09439..., 2.18627..., 3.92699...>'
154     >>> format(Vector([2, 2, 2, 2]), '.3eh')
155     '<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
156     >>> format(Vector([0, 1, 0, 0]), '0.5fh')
157     '<1.00000, 1.57080, 0.00000, 0.00000>'
158
159
160     from array import array
161     import reprlib
162     import math

```

```
163 import functools
164 import operator import itertools
165
166 class Vector:
167     typecode = 'd'
168
169     def __init__(self, components):
170         self._components = array(self.typecode, components)
171
172     def __iter__(self):
173         return iter(self._components)
174
175     def __repr__(self):
176         components = reprlib.repr(self._components)
177         components = components[components.find('['):-1]
178         return f'Vector({components})'
179
180     def __str__(self):
181         return str(tuple(self))
182
183     def __bytes__(self):
184         return (bytes([ord(self.typecode)]) +
185                 bytes(self._components))
186
187     def __eq__(self, other):
188         return (len(self) == len(other) and
189                 all(a == b for a, b in zip(self, other)))
190
191     def __hash__(self):
192         hashes = (hash(x) for x in self)
193         return functools.reduce(operator.xor, hashes, 0)
194
195     def __abs__(self):
196         return math.hypot(*self)
197
198     def __bool__(self):
199         return bool(abs(self))
200
201     def __len__(self):
202         return len(self._components)
203
204     def __getitem__(self, key):
205         if isinstance(key, slice):
206             cls = type(self)
207             return cls(self._components[key])
208             index = operator.index(key)
209             return self._components[index]
210
211         __match_args__ = ('x', 'y', 'z', 't')
```

```

213     def __getattr__(self, name):
214         cls = type(self)          ❶
215         pos = cls.__match_args__.index(name)
216         except ValueError:
217             pos = -1
218         if 0 <= pos < len(self._components):
219             return self._components[pos]
220         msg = f'{cls.__name__!r} object has no attribute {name!r}'
221         raise AttributeError(msg)
222
223     def angle(self, n):          ❷
224         r = math.hypot(*self[n:])
225         a = math.atan2(r, self[n-1])
226         if (n == len(self) - 1) and (self[-1] < 0):
227             return math.pi * 2 - a
228         else:
229             return a
230
231     def angles(self):          ❸
232         return (self.angle(n) for n in range(1, len(self)))
233
234     def __format__(self, fmt_spec=''): ❹
235         if fmt_spec.endswith('h'): # hyperspherical coordinates
236             fmt_spec = fmt_spec[:-1]
237             coords = itertools.chain([abs(self)],          ❺
238                                     self.angles())
239             outer_fmt = '<{}>' ❻
240         else:
241             coords = self
242             outer_fmt = '({})' ❾
243             components = (format(c, fmt_spec) for c in coords) ❿
244             return outer_fmt.format(', '.join(components)) ❽
245
246     @classmethod
247     def frombytes(cls, octets):
248         typecode = chr(octets[0])
249         memv = memoryview(octets[1:]).cast(typecode)
250         return cls(memv)

```

- ❶ 导入 `itertools` 模块, 以便在 `__format__` 中使用函数 `itertools.chain`。
- ❷ 用 “n-sphere” 词条中的公式, 计算某个角度向量。
- ❸ 创建一个生成器表达式, 以按需计算所有角度向量。
- ❹ 用 `itertools.chain` 函数创建生成器表达式, 以无缝遍历向量的模与各个角度向量。
- ❺ 配置使用尖括号 “<>” 显示球面坐标。
- ❻ 配置使用圆括号 “()” 显示直角坐标。
- ❼ 创建一个生成器表达式, 以按需格式化显示各个坐标元素。
- ❽ 将以逗号分隔的格式化分量, 放入在尖括号 “<>” 或圆括号 “()” 内。



本书在 `__format__`、`angle`、`angles` 中大量使用了生成器表达式, 其目的是为了使 `Vector` 类的 `__format__` 方法与 `Vector2d` 类(“[示例 11.6<302页>](#)”)处于同一实现水平。“[十七 迭代器、生成器和经典协程](#)”在介绍生成器时, 将以 `Vector` 类中的部分代码为例, 详细说明生成器的技巧。

本章的任务到此结束。“[十六 运算符重载](#)”将使用中缀运算符对 `Vector` 类进行改进。本章的目的是探讨如何编写容器类广泛使用的几个特殊方法。

12.9 本章小结

本章中的 `Vector` 示例是为了与 `Vector2d` 兼容而设计的, 只是二者使用了不同的构造函数签名。`Vector` 类的构造函数接受一个可迭代对象, 这与内置序列类型一样。`Vector` 类仅通过实现 `__getitem__` 与 `__len__`, 就可表现得像一个序列一样。鉴于这一事实, 我们讨论了协议, 协议是[鸭子类型 \(Duck Typing\)](#)语言中使用的非正式接口。

然后, 本章说明了 `my_seq[a:b:c]` 语法背后的工作原理——即创建 `slice(a,b,c)` 对象, 并将其交给 `__getitem__` 方法处理。了解这一点后, 我们使 `Vector` 能正确地处理切片, 如同 Python 风格的序列那样返回新的 `Vector` 实例。

接下来, 通过定义 `__getattr__` 方法实现了对 `Vector` 实例前几个分量的只读访问(使用 `my_vec.x` 表示法)功能。实现这一功能后, 用户会想通过 `my_vec.x=7` 语法为特殊的分量赋值——这可能是一个潜在的 bug。我们通过实现 `__setattr__` 来修复这个 bug, 以禁止为单字母属性赋值。通常, 在编写 `__getattr__` 代码时, 需要同时添加 `__setattr__`, 以避免赋值与读值的行为不一致。

在为 `Vector` 类实现 `__hash__` 方法时, 特别适合使用 `functools.reduce` 函数。该函数将异或(`xor`)运算符`^`依次应用于 `Vector` 实例各个分量的[哈希码](#), 以生成整个 `Vector` 实例的聚合哈希码。在 `__hash__` 方法中应用了 `reduce` 函数之后, 又用内置归约函数 `all` 实现了更高效的 `__eq__` 方法。

`Vector` 类的最后一项改进是重新实现了 `Vector2d`(“[示例 11.6<302页>](#)”)的 `__format__` 方法。改进后的 `__format__` 方法除了支持直角坐标之外, 还支持了球面坐标。为了编码 `__format__` 方法及其辅助函数, 本章使用了大量的数学知识和几个生成器, 但这些都是实现细节——我们将在“[十七 迭代器、生成器和经典协程](#)”再次讨论生成器。“[12.8 Vector 类第 5 版: 格式化](#)”的目标是支持自定义格式, 从而兑现了承诺——让 `Vector` 可完成 `Vector2d` 的所有功能, 甚至可以做得更多。

与“[十一 Pythonic 对象](#)”一样, 本章经常研究 Python 标准对象的行为, 以模仿它们并为 `Vector` 提供符合 Python 风格的外观和感觉。

在“[十六 运算符重载](#)”中, 将为 `Vector` 类实现几个中缀运算符。那里用到的数学知识要比本章 `angle()` 用到的简单多了。但探索中缀运算符的工作原理, 可加深对面向对象设计的理解。但在讨论运算符重载之前, 我们将暂时告别单个类, 看看如何用接口与继承来组织多个类, 详见“[十三 接口、协议与抽象基类](#)”与“[十四 继承的利与弊](#)”。

12.10 延伸阅读

Vector 类中的多数特殊方法在“十一 Pythonic 对象”的 Vector2d 类(“示例 11.11<307页>”)中也有,因此 11.14 延伸阅读给出的参考资料同样适合本章。

强大的 `reduce` 高阶函数也被称为折叠 (fold) 函数、累加 (accumulate) 函数、聚合 (aggregate) 函数、压缩 (compress) 函数和注入 (inject) 函数。有关更多信息,请参阅维基百科的“[Fold \(higher-order function\)](#)”文章,其中介绍了该高阶函数的应用,重点是使用递归数据结构的函数式编程。该文章还包含了一个表格,列出了数十种编程语言中的类似折叠 (fold-like) 的函数。

“[What's New in Python 2.5](#)”简要概述了 `__index__` 的作用,旨在支持 `__getitem__` 方法(参见“[12.5.2 能处理切片的 __getitem__ 方法<327页>](#)”)。“[PEP 357 -Allowing Any Object to be Used for Slicing](#)”(Travis Oliphant, NumPy 的主要创建者)站在 C 语言扩展实现者的角度,详细说明了需要 `__index__` 的原因。Oliphant 对 Python 的诸多贡献,使 Python 成为领先的科学计算语言,进而使 Python 成为机器学习应用领域的引领者。

杂谈

将协议当作非正式接口

“协议”并非 Python 的发明。Smalltalk 团队(“面向对象”的发明者)将“协议”用作接口的同义词。一些 Smalltalk 编程环境允许程序员将一组方法标记为“协议”,但那只是一种文档和导航辅助工具,不受语言强制执行。因此,在与熟悉正式接口(并由编译器强制执行)的用户交流“协议”时,我会将“协议”简单地解释为“非正式接口”。

在使用动态类型的语言中,已经建立的协议会自然进化。所谓动态类型,是指在程序运行时进行类型检查,因为在方法签名和变量中没有静态类型信息。Ruby 是一种重要的面向对象语言,它也具有动态类型,并使用协议。

在 Python 文档中,当看到类似“类文件(file-like)对象”的表述时,通常指的就是“[协议\(protocol\)](#)”。“类文件(file-like)对象”是对“行为基本与文件一致,实现了部分文件接口,满足上下文相关需求的结构。”的一种简短表述。

您可能认为仅实现协议的一部分,是不够严谨的。但这样做会使事情变得更容易。《[The Python Language Reference](#)》的“3.3. Special method names”中给出了如下建议:

在实现模仿内置类型的类时,重要的是:仅实现到对所建模对象有意义的程度即可。例如,某些序列可能只需要能检索单个元素即可,而实现切片功能可能没有意义。

当不需要编写无意义的方法(即仅为满足某些过度设计的接口契约,并让编译器满意的方法)时,就更容易遵循 [KISS 原则](#)^a。

然而,如果要使用类型检查器验证协议的实现,则需要对协议进行更严格的定义。此时,可以用 `typing.Protocol`。

协议与接口是“[十三 接口、协议与抽象基类](#)”的重点内容,届时将会更详细地讨论协议与接口。

鸭子类型 (Duck Typing) 的起源

我相信,Ruby 社区在 [鸭子类型\(Duck Typing\)](#) 这个术语的推广过程中起到了主要作用。该社区向大

量 Java 用户宣扬了 鸭子类型 (Duck Typing) 这一概念。但早在 Ruby 与 Python 流行之前, Python 中就已经使用 鸭子类型 (Duck Typing) 这一术语了。根据维基百科的记录, 面向对象编程中最早使用“鸭子”做比喻的是: Alex Martelli 于 2000 年 7 月 26 日在 Python-list 中发布的一条消息:“polymorphism (was Re: Type checking in python?)”, 这也是本章开头那句引文的出处。若想了解“鸭子类型 (Duck Typing)”这一概念的文学起源, 以及此概念在多种编程语言中的应用, 请阅读维基百科的“Duck typing”词条。

安全的 `__format__` 方法, 增强可用性

在实现 `__format__` 方法时, 我没有采取任何预防措施来处理具有大量分量的 `Vector` 实例。而在 `__repr__` 中, 使用 `reprlib` 进行了防范处理。原因是, `repr()` 用于调试和日志记录, 因此它必须始终生成一些可用的输出, 而 `__format__` 用于向最终用户显示输出。最终用户可能想看到整个 `Vector` 实例, 如果你认为这样做存在风险, 则可为 `格式化规范微语言` 实现进一步的扩展, 以自定义 `Vector` 实例的显示形式。

我是这样做的: 默认情况下, 任何格式化的 `Vector` 实例都会显示数量合理且有限(如 30)个分量。若分量超出此数量, 默认行为将与 `reprlib` 类似, 截断超出的部分, 并用 `...` 表示。然而, 若格式化代码末尾是星号 (*, 表示全部), 则显示 `Vector` 实例的所有分量。因此, 用户在不知情的情况下不会被冗长的输出吓到。如果默认的上限碍事, 那么 `...` 的存在可能会引导用户去检索文档, 从而发现星号 (*) 这个自定义的格式化代码。

寻找符合 Python 风格的求和 (`sum`) 方式

“什么是 Pythonic(符合 Python 风格)?”这一问题, 没有标准答案。就像我说的那样“地道的 Python”并不能令人 100% 满意。因为对您来说“地道”的东西, 可能并不适合我。但可以肯定的是, “地道”并意味着要使用最晦涩的语言特性。

在 `Python-list` 中, 有一篇发表于 2003 年 4 月的文章“`Pythonic way to sum n-th list element?`”。该文章与本章讨论的 `reduce` 函数 有关。

在该文章中, 作者 Guy Middleton 要求改进这个解决方案, 并表示他不喜欢使用 `lambda`:^b

```
1 >>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
2 >>> import functools
3 >>> functools.reduce(lambda a, b: a+b, [sub[1] for sub in my_list])
4 60
```

此段代码中包含了许多“地道”的用法: `lambda`、`reduce`、列表推导式。这段代码在人气竞赛中可能会排在最后, 因为它冒犯了讨厌 `lambda` 与列表推导式的人——毕竟这类人可不少呢。

若想使用 `lambda`, 或许就不应该使用列表推导式——除非是为了过滤, 但这里的情况并非如此。

下面是我自己的解决方案, 希望能让 `lambda` 拥护者们满意:

```
1 >>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
2 60
```

我未参与那个话题, 也不会在真实的代码中使用上述方案。因为我非常不喜欢 `lambda` 表达式。这里只是为了展示如何不使用列表推导式 (Listcomp) 实现求和计算。

第一个答案来自 IPython 的创建者 Fernando Perez, 他强调 NumPy 支持 n 维数组和 n 维切片:

```

1 >>> import numpy as np
2 >>> my_array = np.array(my_list)
3 >>> np.sum(my_array[:, 1])
4 60

```

我认为 Perez 的方案很酷,但 Guy Middleton 却认可 Paul Rubin 与 Skip Montanaro 的方案,如下所示:

```

1 >>> import operator
2 >>> functools.reduce(operator.add, [sub[1] for sub in my_list], 0)
3 60

```

然后, Evan Simpson 问道:“这样做有什么错?”(代码如下所示)

```

1 >>> total = 0
2 >>> for sub in my_list:
3 ...     total += sub[1]
4 ...
5 >>> total
6 60

```

很多人都认为这也符合 Python 风格。Alex Martelli 甚至说, Guido (Python 创始人) 或许也会这样编码。

我很喜欢 Evan Simpson 的代码,不过也喜欢 David Eppstein 对这段代码给出的评论。

如果想计算列表中所有元素的和,那么代码的逻辑就应该看起来像是在“计算元素之和”。而不是“遍历元素,维护一个变量 t,再执行一系列求和操作。”若不能站在一定高度上表明意图,却让语言去关注实现意图所需的底层操作,那还要高级语言作甚?

而后, Alex Martelli 又给出了建议:

“求和”是一种常规操作,所以我不介意 Python 将其作为内置函数。但在我看来“reduce(operator.add,...)”并不是一个很好的表达方式。(作为一个读过《A Programming Language》的老程序员和函数式编程爱好者,我本应该喜欢这种表达方式。但是,实际上我并不喜欢。)

随后, Alex 建议提供并实现了 sum() 函数。在那次讨论之后 3 个月,Python 2.3 就内置了 sum() 函数。因此, Alex 喜欢的语法也就变成了标准(如下所示):

```

1 >>> sum([sub[1] for sub in my_list])
2 60

```

到了 2004 年 11 月,Python 2.4 与生成器表达式一起发布了。在我看来, Guy Middleton 那个问题最符合 python 风格的答案应该是这样的:

```

1 >>> sum(sub[1] for sub in my_list)

```

2

60

这不仅比 `reduce` 更易读, 而且避免了空序列的陷阱: `sum([])` 的结果为 0, 就是这么简单。

在该次讨论中, Alex Martelli 认为 Python 2 中的内置函数 `reduce` 弊大于利。因为函数 `reduce` 推荐的一些编码方式, 会让用户难以理解。他的观点最有说服力, 因此 Python 3 将函数 `reduce` 降级到了 `functools` 模块中。

尽管如此, `functools.reduce` 还是有其用武之地。它以一种符合 Python 风格的方式, 解决了 `Vector.__hash__` 的问题。

^aKISS 原则, 即“保持简单而愚蠢(Keep It Simple, Stupid)”, 强调简洁而有效的设计和实现, 避免过度复杂和不必要的功能。

^b为了此处的展示, 我对源代码做了轻微改动。因为在 2003 年, `reduce` 是内置函数。而在 Python 3 中, 需要导入 `functools` 模块。此外, 将 `x` 与 `y` 两个名称更换成了 `my_list` 与 `sub`(表示子列表)。

wechat: 119554488

接口、协议与抽象基类

针对接口编程,而不是针对实现编程。

——Gamma、Helm、Johnson、Vlissides, “面向对象设计第一原则”^a

^a出自《设计模式》“引言”部分。

面向对象编程的核心是接口。如“8.4 类型由支持的操作来定义<211页>”所述,在 Python 中理解一个类型的最佳方法就是了解它提供的方法,也就是它的接口。

在每种编程语言中,都有一种或多种定义和使用接口的方式。Python 自 3.8 以来,提供了 4 种类型实现方式(如图 13.1 所示),概述如下;

- 鸭子类型 (Duck Typing)

自 Python 诞生以来,默认使用的类型实现方式。从第 1 章开始,本书一直在研究 鸭子类型 (Duck Typing)。

- 大鹅类型 (Goose Typing)

自 Python 2.6 起,抽象基类 (ABCs) 支持这种类型实现方式,它依赖于运行时根据 抽象基类 (ABCs) 对对象进行检查。大鹅类型 (Goose Typing) 是本章的主要内容。

- 静态类型 (Static Typing)

C 和 Java 等静态类型语言使用的传统的类型实现方式;自 Python 3.5 起,由 `typing` 模块提供支持,由符合“PEP 484 -Type Hints”要求的外部类型检查器实施检查。本章不涉及此实现方式。“八 函数中的类型提示”的多数内容与“十五 类型注解进阶 1136”讨论了 静态类型 (Static Typing)。

- 静态鸭子类型 (Static Duck Typing)

一种因 Go 语言而流行的一种类型实现方式。由 `typing.Protocol` (Python 3.8 新增) 子类提供支持,由外部类型检查器实施检查。静态鸭子类型 (Static Duck Typing) 首次出现在“8.5.10 静态协议”中。

13.1 类型图

图 13.1 描述的 4 种类型实现方式是互补的:它们各有利弊。否定其中任何一种都是没有意义的。

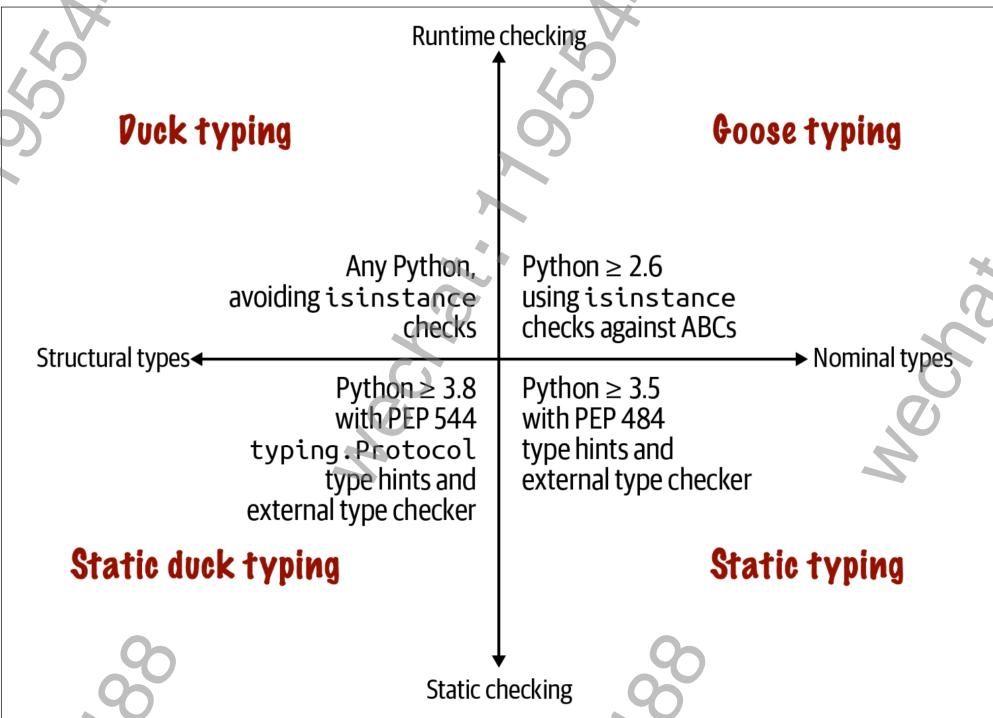


图 13.1: 上半部分只用 Python 解释器在运行时检查类型;
下半部分借助外部静态类型检查工具(如 MyPy 或 PyCharm)检查类型;
左半部分的类型基于对象的结构(对象提供的方法),与对象所属的类或超类无关;
右半部分的类型要求对象有明确的类型名称(对象所属类或超类的名称)。

这 4 种方法都要依赖于接口来工作,但静态类型(Static Typing)可以通过具体类型(Concrete Type)而非接口抽象(如协议(protocol)与抽象基类(ABCs))实现,但可能会存在不足。这 4 种方法都要依赖于接口来工作,但静态类型(Static Typing)可以通过具体类型(Concrete Type)而非接口抽象(如协议(protocol)与抽象基类(ABCs))实现,但可能会存在不足。本章将介绍鸭子类型(Duck Typing)、大鹅类型(Goose Typing)、静态鸭子类型(Static Duck Typing),这些都是围绕接口展开的类型约束方法。

本章分为 4 个主要部分,分别讨论了图 13.1 4 个象限中的 3 个。

- “13.3 两种协议”比较了两种(即图 13.1 的左半部分)依赖协议(protocol)的结构类型。
- “13.4 用鸭子类型编程”深入探讨了 Python 惯用的鸭子类型,包括如何使其更安全,同时保留其主要优势:灵活性。
- “13.5 大鹅类型(Goose Typing)”解释了如何使用抽象基类(ABCs)进行更严格的运行时类型检查。这是篇幅最长的一节,不是因为它更重要。而是因为本书其他章节已涵盖关于鸭子类型(Duck Typing)、静态鸭子类型(Static Duck Typing)和静态类型(Static Typing)的内容。
- “13.6 静态协议”介绍了 typing.Protocol 子类的用法、实现和设计,这些子类对静态类型检查与运行时类型检查非常有用。

13.2 本章新增内容

本章内容改动较大,与第 1 版相应的第 11 章相比,内容增加了约 24%。虽然某些标题没有变化,但新增了许多新内容。主要变化如下:

- 本章引言与图 13.1 都是新增的。这是本章许多新增内容,以及其他章节与类型相关($\text{Python} \geq 3.8$)

的内容的关键。

- “13.3 两种协议”解释了动态协议与静态协议的异同。
- “13.4.3 防御性编程与快速失败”的内容主要来自本书第 1 版,但进行了一些更新。为了突出重要性,重新修改了章节标题。
- “13.6 静态协议”是全新的内容,是“8.5.10 静态协议<233页>”的延续。
- 更新了“图 13.2<353页>”、“图 13.3<362页>”、“图 13.4<364页>”中 `collections.abc` 的类图,以包含 `Collection`(Python 3.6 新增的抽象基类(ABCs))。

本书第 1 版有一节建议用 `numbers` 模块中的抽象基类(ABCs)实现大鹅类型(Goose Typing)。但是,若要为大鹅类型(Goose Typing)使用静态类型检查器以及运行时检查,则应该使用 `typing` 模块中的数字静态协议。“13.6.8 number 模块中的抽象基类与 Numeric 协议<385页>”将解释背后的原因。

13.3 两种协议

在计算机科学领域中,根据语境的不同,协议(protocol)一词有不同的含义。例如,HTTP 网络协议规定了客户端可以向服务器发送的指令,如 GET、PUT 与 HEAD。如“12.4 协议与鸭子类型<324页>”所述,对象协议指明了对象为履行角色而必须提供的方法。“一 Python 数据模型”中的 FrenchDeck 示例(“示例 1.1<5页>”)演示了一个对象协议(即序列协议):一个 Python 对象为了表现得像一个序列,而需要提供的方法。

实现一个完整协议可能需要多个方法,但通常仅需实现其中的一部分即可,如示例 13.1 中的 `Vowels` 类。

</> 示例 13.1: 用 `__getitem__` 实现部分序列协议

```

1  >>> class Vowels:
2      ...     def __getitem__(self, i):
3      ...         return 'AEIOU'[i]
4      ...
5  >>> v = Vowels()
6  >>> v[0]
7  'A'
8  >>> v[-1]
9  'U'
10 >>> for c in v: print(c)
11 ...
12 A
13 E
14 I
15 O
16 U
17 >>> 'E' in v
18 True
19 >>> 'Z' in v
20 False

```

仅实现 `__getitem__` 就足以允许按索引获取项,还可以支持迭代与 `in` 运算符。特殊方法 `__getitem__` 实际上是序列协议的关键。以下内容摘自《Python/C API Reference Manual》的“Sequence Protocol”一节:

```
int PySequence_Check(PyObject *o)
```

如果对象提供序列协议，则返回 1，否则返回 0。注意，除了 dict 子类，如果一个 Python 类实现了 `__getitem__` 方法，则也返回 1。

我们预期一个序列也应该支持 `len()` 方法，也就是要实现 `__len__` 方法。尽管 `Vowels` 类没有实现 `__len__` 方法，但仍然可以在某些情景下表现为一个序列。而某些时候，这已经足够满足需求了。所以，我常说 [协议 \(protocol\)](#) 是一种“非正式接口”。这也是 Smalltalk 对协议的理解，Smalltalk 是第一个使用“协议”一词的面向对象编程语言。

在 Python 文档中，除了有关网络编程的内容之外，“[协议 \(protocol\)](#)”一词大多数都是指这些非正式接口。

现在，Python 3.8 采用了“[PEP 544 -Protocols: Structural subtyping \(static duck typing\)](#)”之后，“[协议 \(protocol\)](#)”一词在 Python 中又多了一种含义——密切相关，但又不同。如“[8.5.10 静态协议 <233页>](#)”所述，[PEP 544](#) 允许我们创建 `typing.Protocol` 的子类，来定义一个类必需实现（或继承）的一个或多个方法，以令静态类型检查工具满意。

当需要区分 [协议 \(protocol\)](#) 的不同含义时，我会使用如下这些术语：

- **动态协议**

Python 一直支持的非正式协议。动态协议是隐式的，由约定定义，并在文档中描述。Python 最重要的动态协议由解释器本身直接支持，并在《[The Python Language Reference](#)》的“3. Data model”中进行了说明。

- **静态协议**

由“[PEP 544 -Protocols: Structural subtyping \(static duck typing\)](#)”定义的协议，自 Python 3.8 开始支持。静态协议要用 `typing.Protocol` 子类显式定义。

二者之间的主要区别如下：

- 对象可以只实现动态协议的一部分，而且仍然有用；但若想实现静态协议，该对象必需实现协议类中声明的每一个方法，即使程序中用不到。
- 静态协议可以由静态类型检查器来验证，但动态协议则不行。

这两种协议有一个共同的基本特征，即“[类在实现某个接口或协议时，并不需要明确地声明或继承一个特定的接口或协议](#)”。

除了静态协议之外，Python 还提供了另一种在代码中定义显式接口的方法：[抽象基类 \(ABCs\)](#)。

本章余下的内容涵盖了动态协议、静态协议以及 [抽象基类 \(ABCs\)](#)。

13.4 用鸭子类型编程

让我们以 Python 中最重要的两个动态协议（序列协议、可迭代协议）为例，展开对动态协议的探讨。在 Python 中，解释器会不遗余力地处理提供了这两个协议的对象，即使是最基本的实现也会被解释器有效地处理（如下节所述）。

13.4.1 Python 深入理解序列

Python 数据模型的理念是尽可能地与基本动态协议配合。对序列来说,即使是最简单的实现,Python 也会努力与之配合。

图 13.2 展示了 Sequence 接口如何被确立 (formalize) 为一个 **抽象基类 (ABCs)**。Python 解释器与内置序列 (如 list、tuple、str 等) 完全不依赖于此 **抽象基类 (ABCs)**。此处,我只是用 Sequence 类描述一个完善的序列应该支持的功能。

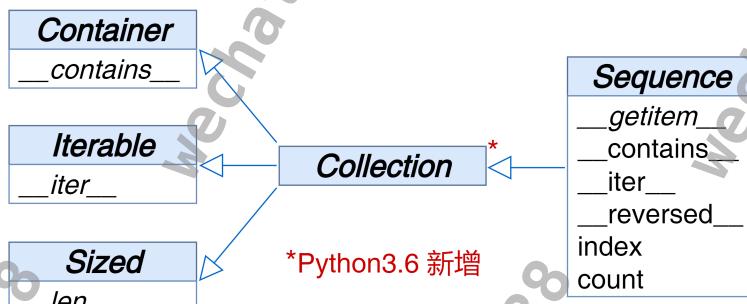


图 13.2: Sequence 及相关抽象基类的 UML 类图
 箭头由子类指向超类;斜体名称是抽象方法。
 Python 3.6 之前,没有 Collection 抽象基类
 Sequence 是 Container、Iterable、Sized 的直接子类

`collections.abc` 模块 中的多数**抽象基类 (ABCs)** 都是为了确立 (formalize) 接口而存在的。这些接口由内置类型实现,并由解释器隐式支持——内置类型与解释器都早于 **抽象基类 (ABCs)** 存在。这些**抽象基类 (ABCs)** 可作为新类的基础,并为运行时的显式类型检查(又称**大鹅类型 (Goose Typing)**)与静态类型检查工具用到的**类型提示 (Type Hints)**提供支持。

从 图 13.2 可以看出,Sequence 的一个正确子类必须实现 `__getitem__` 和 `__len__` (来自 `Sized`)。Sequence 中的所有其他方法都是具体的 (concrete),因此子类可以继承它们的实现,或者提供更好的实现。

现在,回顾一下“示例 13.1<351页>”中的 `Vowels` 类。它没有继承 `abc.Sequence`,而且只实现了 `__getitem__`。

虽然没有 `__iter__` 方法,但 `Vowels` 实例仍然是可迭代对象。因为作为一种后备机制,如果 Python 找到了对象的 `__getitem__` 方法,Python 就会尝试用从 0 开始的整数索引,调用对象的 `__getitem__` 方法来尝试迭代对象。因为 Python 足够聪明,可以迭代 `Vowels` 实例,所以即使缺少 `__contains__` 方法,Python 也使用 `in` 运算符:Python 会进行顺序扫描以检查是否存在某个项。

总之,考虑到序列类数据结构的重要性,在 `__iter__` 和 `__contains__` 不可用时,Python 会调用 `__getitem__` 方法,设法让迭代与 `in` 运算符可用。

“[Python 数据模型](#)”中的原始 `FrenchDeck` 也没有子类化 `abc.Sequence`,但它实现了序列协议的 2 个方法:`__getitem__` 和 `__len__` 方法。如 [示例 13.2](#) 所示,

</> [示例 13.2: 一副有序的纸牌 \(同 “\[示例 1.1<5页>\]\(#\)”\)](#)

```

1 import collections
2
3 Card = collections.namedtuple('Card', ['rank', 'suit'])

```

```

4
5 class FrenchDeck:
6     ranks = [str(n) for n in range(2, 11)] + list('JQKA')
7     suits = 'spades diamonds clubs hearts'.split()
8
9     def __init__(self):
10        self._cards = [Card(rank, suit) for suit in self.suits
11                      for rank in self.ranks]
12
13     def __len__(self):
14        return len(self._cards)
15
16     def __getitem__(self, position):
17        return self._cards[position]

```

“— Python 数据模型”中的几个示例之所以有效,是因为 Python 对任何类似于序列的对象都进行了特殊处理。Python 中的可迭代协议代表了**鸭子类型 (Duck Typing)**的一种极端形式:解释器会尝试调用 2 种不同的方法来迭代对象。

需要明确的是,本节中描述的行为是在解释器中实现的,主要是用 C 语言实现的,不依赖于**抽象基类 (ABCs)** Sequence 中的方法。例如,Sequence 类中的具体方法 `__iter__` 与 `__contains__` 模拟了 Python 解释器的内置行为。如果好奇,请查看 `Lib/_collections_abc.py` 中这些方法的源代码。

现在,我们来研究另一个例子,它强调了协议的动态本性,并解释了为什么静态类型检查器无法处理动态协议。

13.4.2 猴子补丁:在运行时实现协议

“**猴子补丁 (Monkey-patch)**”¹ 是指在运行时动态更改模块、类或函数,以新增功能或修复 bug。例如,gevent 网络库对 Python 标准库的部分内容打了**猴子补丁 (Monkey-patch)**,以实现没有线程或 `async/await` 的轻量级并发。

示例 13.2 中的 FrenchDeck 类缺少一个基本功能:无法洗牌。多年前,我首次编写 FrenchDeck 示例时,确实实现了一个 `shuffle()` 方法。在深刻理解了 Python 风格之后,我发现:既然 FrenchDeck 像一个序列,那么它就不需要实现自己的 `shuffle` 方法了。因为已经有现成可用的 `random.shuffle` 函数,根据文档介绍,该函数的作用是“就地打乱序列 x”。

标准 `random.shuffle` 函数 的使用方式如下:

```

1 >>> from random import shuffle
2 >>> l = list(range(10))
3 >>> shuffle(l)
4 >>> l
5 [5, 2, 9, 7, 8, 3, 1, 4, 0, 6]

```

¹维基百科上的“猴子补丁”词条,用 Python 制作了一个有趣的例子。



当你遵循既定的协议时,即可提高利用现有标准库和第三方代码的机会,这要归功于鸭子类型 (Duck Typing)。

但是,如果用 `random.shuffle` 函数 打乱一个 `FrenchDeck` 实例,则会出现异常,如 [示例 13.3](#) 所示。

</> [示例 13.3](#): `random.shuffle` 无法处理 `FrenchDeck` 实例

```

1 >>> from random import shuffle
2 >>> from frenchdeck import FrenchDeck
3 >>> deck = FrenchDeck()
4 >>> shuffle(deck)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   File ".../random.py", line 265, in shuffle
8     x[i], x[j] = x[j], x[i]
9 TypeError: 'FrenchDeck' object does not support item assignment

```

错误消息很明确:“`'FrenchDeck'` object does not support item assignment (`'FrenchDeck'` 不支持为项赋值)”。此问题的原因在于 `random.shuffle` 函数 是就地 (in place) 调换容器内部各项的位置。而 `FrenchDeck` 仅实现了不可变序列协议,无法支持就地操作。要想实现可变序列协议,还必需提供一个 `__setitem__` 方法。

由于 Python 是动态的,所以我们在运行时,甚至在交互式控制台中解决这个问题。[示例 13.4](#) 展示了如何做到这一点。

</> [示例 13.4](#): 为 `FrenchDeck` 应用猴子补丁,使其可变并与 `random.shuffle` 兼容 (接续[示例 13.3](#))

```

1 >>> def set_card(deck, position, card): ❶
2 ...     deck._cards[position] = card
3 ...
4 >>> FrenchDeck.__setitem__ = set_card ❷
5 >>> shuffle(deck) ❸
6 >>> deck[:5]
7 [Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4',
8 suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]

```

- ❶ 创建一个以 `deck`、`position` 和 `card` 为参数的函数。
- ❷ 将该函数分配给 `FrenchDeck` 类中名为 `__setitem__` 的属性。
- ❸ 现在可以对 `FrenchDeck` 实例洗牌了,因为我添加了可变序列协议所必需的方法。

特殊方法 `__setitem__` 的签名在《Python 语言参考》的“3.3.7. Emulating container types”中定义。此处,我将参数重命名为 `deck`、`position`、`card`,而不是《Python 语言参考》中的 `self`、`key`、`value`,以表明每个 Python 方法开始时都是一个普通函数,将第一个参数命名为 `self` 只是一种习惯。在控制台会话中使用这些重命名的参数没有问题,但在 Python 源文件中,最好按照文档那样使用 `self`、`key` 和 `value`。

这里的关键是, `set_card` 函数需要知道 `deck` 对象有一个 `_card` 属性,而且 `_card` 的值是一个可变序列。然后,将 `set_card` 函数作为 `__setitem__` 特殊方法附加到 `FrenchDeck` 类。这是猴子补丁 (Monkey-patch) 的一个示例:在运行时更改类或模块,而无需改动源码。虽然 猴子补丁 (Monkey-patch) 功能强大,但是执

行修补的代码与被修补的程序耦合度非常高,通常需要处理文档中未明确说明的私有属性。

示例 13.4 除了是一个 猴子补丁 (Monkey-patch) 的例子外,还强调了动态 鸭子类型 (Duck Typing) 中协议的动态本性:`random.shuffle` 函数并不关心 实参 (Argument) 对象所属的类,只要那个对象实现了可变序列协议中的方法即可。至于该方法是那个对象“与生俱来”的,还是后期通过某种方式获得的,都无关紧要。

13.4.3 防御性编程与快速失败

防御性编程就像防御性驾驶:提供了一套可提高安全性 (safety) 的实践,即使面对粗心的程序员或驾驶员,也能提高安全性。

许多 bug 只有在运行时才能被发现,即使在主流静态类型语言中也是如此。在动态类型语言中,Fail-Fast (快速失败) 是让程序更安全、更易于维护的绝佳建议。Fail-Fast (快速失败) 意味着尽快引发运行时错误,例如,在函数体开始时就拒绝无效的实参 (Argument)。

此处有一个例子:当编写代码时,如果需要在函数内部将一系列的项以 list 的形式来处理。那么,就不要直接通过类型检查来强制传入一个 list 类型的 实参 (Argument)。相反,应该接受 实参 (Argument) 并在函数主体中立即从 实参 (Argument) 构建一个 list。如“示例 13.10<370页>”中的 `__init__` 所示。

```
1 def __init__(self, iterable):      # 没有直接接收一个 list
2     self._balls = list(iterable)    # 而是在方法主体内,根据接收的 实参,构建一个 list
```

这么做,可以使代码更加灵活。因为 `list()` 函数可以处理任何适合内存的可迭代对象。如果传入的 实参 (Argument) 不是可迭代对象,那么在实例初始化时 `list()` 调用将会 Fail-Fast (快速失败),并引发意义明确的 `TypeError` 异常。若想让异常信息更加明确,可以用 `try/except` 包装 `list()` 调用来定制错误消息。但我只会在外部 API 上使用这种额外的代码,因为这样可以方便代码库的维护者发现问题。无论哪种方式,引发问题的调用都将在 `traceback` 的尾部,这样很容易修复。如果在 `__init__` 中未捕获到无效参数,而是当类的其他方法需要操作 `self._balls` 时,才发现它不是 `list`。那样会为时已晚,很难定位问题的根本原因。

当然,如果数据量过大,或按照设计,需要就地更改数据 (如 `random.shuffle`) 以满足调用方的需求。那么,调用 `list()` 复制数据数据将是一个很糟糕的主意。此时,应使用类似 `isinstance(x, abc.MutableSequence)` 的运行时检查。

如果害怕函数接收到一个无穷尽的生成器 (不常见),则可以先用 `len()` 获取 实参 (Argument) 的长度。这样可以拒绝迭代器,以安全地处理元组、数组以及其他实现了 `Sequence` 接口的类,该类可以是存量的,也可以是日后新增的。调用 `len()` 的开销很小,但是作用明显,在遇到无效参数时会立即引发异常。

另外,如果可接收任何类型的可迭代对象,则应尽早调用 `iter(x)` 以获得一个迭代器 (详见“17.3 序列可迭代的原因:`iter` 函数<481页>”)。同样,如果 `x` 不是可迭代对象, `iter(x)` 也会 Fail-Fast (快速失败) 并引发一个易于调试的异常。

在这两种情况下,类型提示 (Type Hints) 可以提前发现一些问题,但并不能发现所有问题。回想一下, `Any` 类型与所有类型都相容 (Consistent-with),而类型推断可能导致变量被标记为 `Any` 类型。此时,类型检查器会被蒙在鼓里。此外,类型提示 (Type Hints) 不会在运行时强制执行类型检查。Fail-Fast (快速失败) 是最后一道防线。

利用 鸭子类型 (Duck Typing) 做防御性编程,可以无需使用 `isinstance()` 或 `hasattr()` 测试,就能处理不

同的类型。

举个例子,模仿 `collections.namedtuple` 处理 `field_names` 参数的方式。参数 `field_names` 接受一个由空格或逗号分隔的标识符组成的单个字符串(如“‘a’, ‘b’, ‘c’”),或者一个标识符序列(如“[‘a’, ‘b’, ‘c’]”)。示例 13.5 展示了我如何使用鸭子类型来做到这一点。

</> 示例 13.5: 用鸭子类型处理字符串或由字符串构成的可迭代对象

```

1  try:                                ❶
2      field_names = field_names.replace(',', ' ').split() ❷
3  except AttributeError:               ❸
4      pass                                ❹
5  field_names = tuple(field_names)       ❺
6  if not all(s.isidentifier() for s in field_names):      ❻
7      raise ValueError('field_names must all be valid identifiers')

```

- ❶ 假设 `field_names` 是一个字符串(EAFP 原则:请求原谅比请求许可更容易)²。
- ❷ 将字符串中的逗号(,)转换为空格,并将结果拆分为 `list`。
- ❸ 抱歉,`field_names` 的行为不像 `str`,其原因可能是没有 `replace` 方法,或者返回的结果无法被拆分为 `list`。
- ❹ 如果引发 `AttributeError`,则 `field_names` 不是 `str`。那么,假定 `field_names` 已经是由名称构成的可迭代对象。
- ❺ 为了确保 `field_names` 是可迭代对象,也为了留存一个副本,根据 `field_names` 现有数据构建一个 `tuple`(元组)。`tuple` 比 `list` 更紧凑,还能防止代码意外更改名称。
- ❻ 使用 `str.isidentifier` 确保每个名称都是有效的标识符。

示例 13.5 展示的情况,说明:“**鸭子类型 (Duck Typing)** 比**静态类型提示 (Type Hints)** 更具表现力。”类型提示无法表达“`field_names` 必须是一个字符串,字符串内容是由空格或逗号分隔的标识符”这种约束。在 `typedesh` 项目中, `namedtuple` 签名的相关部分如下所示(完整代码见 `stdlib/collections/_init_.pyi`)。

```

1  def namedtuple(
2      typename: str,
3      field_names: Union[str, Iterable[str]],
4      *,
5      # 省略余下的签名内容

```

如您所见,`field_names` 被注解为 `Union[str, Iterable[str]]`,这个 **类型提示 (Type Hints)** 基本够用,但并不足以覆盖所有可能的场景。

在回顾了动态协议之后,接下来换个话题,讨论一下更明确的运行时类型检查:**大鹅类型 (Goose Typing)**

²EAFP 原则:表示“it's Easier to Ask for Forgiveness than Permission.”(请求原谅比请求许可更容易)。这意味着 Python 中的 `try:... except:...` 语法非常有用,可以放心使用!

13.5 大鹅类型 (Goose Typing)

抽象类代表一个接口。

——Bjarne Stroustrup, C++ 创始人^a

^a出自《The Design and Evolution of C++》(Bjarne Stroustrup 著)第 278 页。

Python 中没有接口 (interface) 这个关键字。我们使用 [抽象基类 \(ABCs\)](#) 来定义接口,以便在运行时进行显式类型检查,这种类型检查也可以被静态类型检查器支持。

Python 术语表中的 [抽象基类](#) 词条,很好地解释了 [抽象基类 \(ABCs\)](#) 给 [鸭子类型 \(Duck Typing\)](#) 语言带来的好处:

[抽象基类 \(ABCs\)](#) 是对 [鸭子类型 \(Duck Typing\)](#) 的补充,它提供了一种定义接口的方法,而其他技术(如 `hasattr()`)会显得笨拙或存在微妙的错误(如魔法方法)。[抽象基类 \(ABCs\)](#) 引入了 [虚拟子类 \(Virtual Subclass\)](#),虚拟子类不是从某个类继承而来,但仍能被 `isinstance()` 和 `issubclass()` 识别;参见 [abc 模块文档](#)。

[大鹅类型 \(Goose Typing\)](#) 是一种利用 [抽象基类 \(ABCs\)](#) 实现的运行时检查方法。接下来,将由 Alex Martelli 在后文的附注栏“[水禽与抽象基类](#)”中进行详细解释。



我非常感谢我的朋友 Alex Martelli 与 Anna Raven-scroft。我在 OSCON 2013 上向他们展示了《流畅的 Python (第 1 版)》的原始大纲,他们鼓励我将其提交给 O'Reilly 出版社。后来,他们还担任了该书的技术审校。Alex 已经是这本书中被提到次数最多的人。他主动提出撰写下面这篇文章。接下来,交给你了,Alex!

水禽与抽象基类

作者:Alex Martelli

我因协助了 [鸭子类型 \(Duck Typing\)](#)(忽略对象的实际类型,而是专注于对象是否实现了所需的方法、签名和语义)一词的传播,而在维基百科上受到了赞誉。

对 Python 来说,这基本上是指避免使用 `isinstance` 检查对象的类型(更别提“`type(foo) is bar`”这种更糟的类型检查方式了)。这样检查,没有任何好处,甚至禁止了最简单的继承方式。)

总体来说, [鸭子类型 \(Duck Typing\)](#) 在多数情况下仍然非常有用。然而,随着时间的推移,在某些情况下通常会进化出更好的类型实现方式。事情是这样的…

近些年,属与种^a(包括但不限于被称为 Anatidae(鸭科)的水禽家族)主要是根据 [表型学 \(phenetics\)](#) 来进行生物分类的。这种分类方法侧重于形态和行为的相似性——主要是指可观察的特征。因此,用 [鸭子类型 \(Duck Typing\)](#) 做比喻是最贴切的。

然而,平行进化经常会在实际上并不相关的物种之间产生相似的特征,无论是形态上的还是行为上的。这些不相关的物种只是恰好在相似但分离的“生态位”中进化而已^b。在编程中也会出现类似的“偶然相似性”,例如,考虑经典的面向对象编程示例:

```

1  class Artist:
2      def draw(self): ...
3
4  class Gunslinger:
5      def draw(self): ...
6
7  class Lottery:
8      def draw(self): ...

```

显然,即使两个对象 `x` 与 `y` 都有一个名为 `draw` 的方法,并且这个方法可以被无参数调用(如 `x.draw` 与 `y.draw`)。但仅仅这一点并不足以确保这两个对象在语义上是相似的或可互换的。也就是说,即使两个对象都能调用 `draw` 方法,它们之间的行为或语义可能完全不同。所以,不能仅仅依靠方法的存在和名称来判断对象的相似性或等价性。相反,我们需要依靠有经验的程序员主动分析和确认这种等价性,这可能涉及到更深层次的理解和比较对象的行为、属性等方面。

在生物学(以及其他学科)中,这个“偶然相似性”问题催生出另一种与 **表型学** (phenetics) 不同的生物分类方法——**宗族学** (cladistics)。宗族学将生物分类的焦点放在从共同祖先继承而来的特征上,而不是独立进化的特征上。(近些年,既便宜又快速的 DNA 测序技术,使得宗族学在很多情况下变得高度实用。)

例如,sheldgeese(曾被归类为更接近鹅类)与 shelducks(曾被归类为更接近鸭类)现在都被重新归为同一亚科(Tadornidae),这表明 sheldgeese 与 shelducks 彼此之间更接近,比起与其他鸭科动物¹ 的关系更近。此外,DNA 分析还发现,尽管白翅木鸭与美洲家鸭在外貌与行为上很相似,但在 DNA 方面二者并不接近。因此,白翅木鸭完全脱离了之前所属的亚科,被重新分类到了自己的属(genus)中。这说明了科学家们通过 DNA 分析对生物分类的重新评估,有时会推翻传统的外貌和行为相似性所建立的生物分类。

知道这些有什么用呢?这要视情况而定!比如,当抓到一只水禽后,决定如何烹制才最美味时,可考虑具体可观察到的特征(并非全部特征,例如羽毛在此场景下并不重要)主要是质地与味道,这比宗族学更有意义。但当研究飞禽针对不同病原体的易感性(无论是想要圈养,还是野外保护)时,DNA 的接近性可能更重要(这属于宗族学)。

因此,参照水禽的生物分类学演化,我建议在 **鸭子类型** (Duck Typing) 的基础上补充(不是完全取代,因为鸭子类型在某些时候还有它的作用) **大鹅类型** (Goose Typing)。

大鹅类型 (Goose Typing) 指的是,只要 `cls` 是**抽象基类** (ABCs) (换句话说, `cls` 的元类是 `abc.ABCMeta`),就可以使用 `isinstance(obj,cls)`。

可以在 `collections.abc` 与 `numbers` 模块中找到许多有用的**抽象基类** (ABCs)。

与**具体类** (Concrete Class) 相比, **抽象基类** (ABCs) 拥有许多理论上的优势(例如,“<https://ptgmedia.pearsoncmg.com/images/020163371x/items/item33.html>”条款 33:所有非叶子类,都应是抽象类”,见《More Effective C++》(Scott Meyer 著))。Python 的抽象基类还具有一个主要的实际优势:终端用户可以用类方法 `register()` 在代码中将某个类“声明”为抽象基类的**虚拟子类** (Virtual Subclass)。(为此,被注册的类必需满足抽象基类对方法名称与签名的要求,更重要的是必须符合底层语义契约。但是,开发子类时无需了解抽象基类,也无需继承抽象基类。)这在打破继承所带来的强耦合方面迈出了很大一步,这与多数面向对象程序员所掌握的继承知识有很大出入,因此使用继承时要格外谨慎。

有时候即使你没有显式地使用 `register()` 方法将某个类注册为 **抽象基类 (ABCs)** 的子类, **抽象基类 (ABCs)** 仍能够识别该类为其子类。

其实, **抽象基类 (ABCs)** 本质上就是几个特殊方法。例如:

```

1  >>> class Struggle:
2      ... def __len__(self): return 23
3      ...
4  >>> from collections import abc
5  >>> isinstance(Struggle(), abc.Sized)
6  True

```

如您所见, 无须注册, `abc.Sized` 也能将 `Struggle` 识别为自己的子类, 只要 `Struggle` 实现了特殊方法 `__len__` 即可。`__len__` 方法要使用正确的语法和语义。正确的语法要求该方法不带参数, 正确的语义要求该方法返回一个表示对象长度的非负整数。若使用不合规范的语法或语义实现诸如 `__len__` 之类特殊方法, 将导致非常严重的问题。)

最后我想说的是: 如果要实现的类中体现了 `numbers`、`collections.abc` 或其他框架中抽象基类的概念, 则要么继承相应的抽象基类(必要时), 要么将实现的类注册到相应的抽象基类中。

在使用某个库或框架开发程序时, 如果它们忽略了这一点, 请手动执行注册; 如果必需检查 **实参 (Argument)** 的类型(例如, 是否是序列), 则可这样做:

```
1  isinstance(the_arg, collections.abc.Sequence)
```

并且, 不要在生产代码中定义 **抽象基类 (ABCs)** (或元类)。如果您有这样做的冲动, 我敢打赌可能是因为您想“找茬”, 就像刚拿到新工具的人都有大干一场的冲动。如果能避开这些深奥的概念, 那么您(以及未来的代码维护人员)的生活将更轻松愉快, 因为代码会变得简洁明了。再会!

^a属 (genus) 和种 (species) 是两个重要的分类单位。属是一个相对较高的分类级别, 它包含具有共同特征的相关物种的集合。种是生物学中最基本的分类单位, 它代表着一组可以相互交配并产生生育后代的个体。属和种之间的关系可以用嵌套的方式来描述: 一个属可以包含多个种, 而一个种则只属于一个属。

^b“生态位”是指一个物种在其生态系统中所占据的特定角色或地位, 包括其在食物链中的位置、其与其他物种的相互作用方式以及其对环境的适应能力等。

^c传统上, 鸭科动物被分为鹅和鸭两个类群, 而鹅类和鸭类各自也有各自的分类。

综上所述, **大鹅类型 (Goose Typing)** 要求:

- 从**抽象基类 (ABCs)** 进行子类化, 以明确您正在实现先前定义的接口。
- 运行时类型检查时, 使用**抽象基类 (ABCs)** 而不是**具体类 (Concrete Class)** 作为 `isinstance()` 与 `issubclass()` 的第 2 个参数。

Alex 指出, 从**抽象基类 (ABCs)** 继承其实就是实现所需的方法, 这也明确表明了开发人员的意图。这种意图还可以通过注册**虚拟子类 (Virtual Subclass)** 来明确表达。



“13.5.6 抽象基类的虚拟子类<371页>”将详细介绍如何使用 `register()` 函数。此处先举一个简单的例子：对于 `FrenchDeck` 类（“[示例 13.2<353页>](#)”），如果我想让它通过类似 `issubclass(FrenchDeck, Sequence)` 的检查，可以用这几行代码使它成为抽象基类 `Sequence` 的虚拟子类（Virtual Subclass）：

```
1 from collections.abc import Sequence
2 Sequence.register(FrenchDeck)
```

使用 `isinstance()` 与 `issubclass()` 测试抽象基类（而不是具体类），更容易被人所接受。如果用于测试 **具体类**（Concrete Class），类型检查会限制多态性（polymorphism，面向对象编程的基本特征）。但如果用于**抽象基类**（ABCs），这些测试就更加灵活了。毕竟，如果一个组件没有通过子类化来实现**抽象基类**（ABCs），但实现了必要的方法。那么，它还可以在事后通过 `register()` 函数注册为**抽象基类**（ABCs）的子类，从而可通过显式类型检查。

不过，即使使用了**抽象基类**（ABCs），你也应该注意，过度使用 `isinstance` 检查可能会产生**代码异味**（Code Smell）。因为，这是糟糕的面向对象设计的症状之一。

通常情况下，在一连串 `if/elif/elif` 中用 `isinstance` 做类型检查，并根据对象类型执行不同的操作，往往是一种不好的做法。此时应该使用多态性（polymorphism）来实现这一点，即设计您的类，以便解释器实参（Argument）类型将函数调用分派给合适的方法（如“[示例 9.20<265页>](#)”所示），而不是在 `if/elif/elif` 块中硬编码分派逻辑。

另一方面，如果必须强制执行 API 契约，那么可以针对**抽象基类**（ABCs）执行 `isinstance` 检查。正如本书技术审校 Lennart Regebro 所说：“伙计，如果你想调用我，就必须实现这一点”。这对采用插入式架构的系统来说特别有用。在框架之外，**鸭子类型**（Duck Typing）通常比类型检查更简单、更灵活。

最后，Alex 在文章中不止一次地强调，在创建**抽象基类**（ABCs）时要克制。过度使用**抽象基类**（ABCs）会使这种因实用和务实而流行的语言太过注重表面形式，这不是什么好事。在对本书评审过程中，Alex 在一封电子邮件中写道：

抽象基类（ABCs）旨在封装由框架引入的泛化(general)^a概念和抽象。例如，“一个序列”和“一个确切的数字”。读者基本上不需要编写任何新的**抽象基类**（ABCs），只要正确使用现有的**抽象基类**（ABCs），就可获得 99.9% 的好处，而且几乎不存在严重的设计错误风险。

^a泛化（general）通常指的是编写具有通用性质的代码，以便能够适用于各种不同的情况和数据类型，而不是针对特定情况编写特定的代码。例如，“序列”就是一种泛化类型，而“list”则是一种特化（special）类型。

下面通过实例，讲解**大鹅类型**（Goose Typing）的实际应用。

13.5.1 子类化一个抽象基类

根据 Martelli 的建议，在大胆发明自己的**抽象基类**（ABCs）之前，先利用现有的 `collections.MutableSequence` 类。在[示例 13.6](#) 中，`FrenchDeck2` 被明确声明为 `collections.MutableSequence` 的子类。

</> [示例 13.6: frenchdeck2.py:FrenchDeck2 是 collections.MutableSequence 的子类](#)

```
1 from collections import namedtuple, abc
```

```

2
3 Card = namedtuple('Card', ['rank', 'suit'])
4 class FrenchDeck2(abc.MutableSequence):
5     ranks = [str(n) for n in range(2, 11)] + list('JQKA')
6     suits = 'spades diamonds clubs hearts'.split()
7
8     def __init__(self):
9         self._cards = [Card(rank, suit) for suit in self.suits
10                     for rank in self.ranks]
11
12     def __len__(self):
13         return len(self._cards)
14
15     def __getitem__(self, position):
16         return self._cards[position]
17
18     def __setitem__(self, position, value): ❶
19         self._cards[position] = value
20
21     def __delitem__(self, position): ❷
22         del self._cards[position]
23
24     def insert(self, position, value): ❸
25         self._cards.insert(position, value)

```

❶ 为了支持洗牌,仅需实现 `__setitem__` 方法即可...

❷ ...但是继承 `MutableSequence` 的子类必须实现 `__delitem__` 方法,这是 `MutableSequence` 类的一个抽象方法 (Abstract Method)。

❸ 此外,还需要实现 `insert()` 方法,该方法是 `MutableSequence` 类的第 3 个 抽象方法 (Abstract Method)。

Python 在导入时 (加载并编译 `frenchdeck2.py` 模块时) 不检查抽象方法的实现,而只在运行时 (实际尝试实例化 `FrenchDeck2` 时) 才检查抽象方法的实现。因此,如果没有正确实现某个 抽象方法 (Abstract Method),则会引发 `TypeError` 异常,错误消息为 “Can't instantiate abstract class FrenchDeck2 with abstract methods `__delitem__`, `insert`”。正是这个原因,即便 `FrenchDeck2` 类不需要 `__delitem__` 与 `insert` 提供的行为,也要实现这两个 抽象方法 (Abstract Method),这是 抽象基类 (ABCs) `MutableSequence` 的要求。

如 图 13.3 所示,在 `Sequence` 与 `MutableSequence` 中,并非所有方法都是 抽象方法 (Abstract Method)。

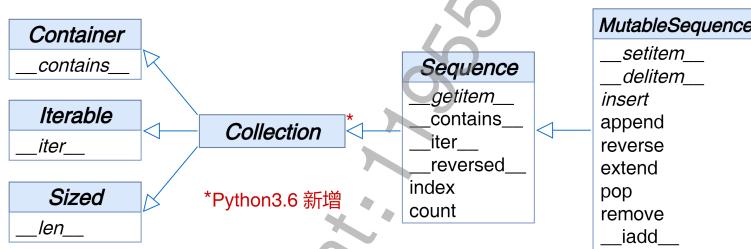


图 13.3: `MutableSequence` 抽象基类及其超类的 UML 类图
箭头由子类指向祖先;斜体名称是抽象类与抽象方法

为了将 FrenchDeck2 定义为 `MutableSequence` 的子类, 我不得不实现 `__delitem__` 与 `insert`, 但是 FrenchDeck2 类并不需要这 2 个 **抽象方法 (Abstract Method)**。FrenchDeck2 类从 `Sequence` 继承了 5 个 **具体方法 (Concrete Method)**: `__contains__`、`__iter__`、`__reversed__`、`index` 和 `count`。另外, 还从 `MutableSequence` 中继承了 6 个 **具体方法 (Concrete Method)**: `append`、`reverse`、`extend`、`pop`、`remove` 和 `__iadd__`³。

在 `collections.abc` 中, 每个 **抽象基类 (ABCs)** 中的 **具体方法 (Concrete Method)** 都是根据类的公开接口实现的, 因此它们在工作时无需了解实例的内部结构。



作为**具体子类 (Concrete Subclass)**的编码者, 你可以用更高效的方式重新实现从**抽象基类 (ABCs)**继承的**具体方法 (Concrete Method)**。例如, **抽象基类 (ABCs)**提供的`__contains__`方法是通过对序列进行全面扫描来工作的。如果你的具体序列会按顺序保存元素, 则可以重写`__contains__`方法, 利用标准库的**bisect**函数进行二分查找, 从而提升检索速度。详见本书配套网站的“*Managing Ordered Sequences with Bisect*”一文。

要想充分利用**抽象基类 (ABCs)**, 你需要知道有哪些可用的**抽象基类 (ABCs)**。接下来, 将将介绍 `collections` 中的抽象基类。

13.5.2 标准库中的抽象基类

自 Python 2.6 起, 标准库提供了多个**抽象基类 (ABCs)**。大多数都定义在 `collections` 模块中。不过, 其他地方也有一些 (如 `io` 包与 `numbers` 包)。但使用最广泛的是 `collections.abc` 模块。



标准库中有 2 个名为 `abc` 的模块。这里讨论的是 `collections.abc`。为了减少加载时间, 自 Python 3.4 起, 该模块在 `collections` 包之外单独实现 (即 `Lib/_collections_abc.py`) 中实现, 因此要与 `collections` 包分开导入。另一个 `abc` 模块就是 `abc` (即 `Lib/abc.py`), 其中定义了 `abc.ABC` 类。每个**抽象基类 (ABCs)**都依赖于 `abc` 模块, 但除了创建一个全新的**抽象基类 (ABCs)** 外, 我们自己并不需要导入 `abc` 模块。

图 13.4 是 `collections.abc` 中定义的 17 个**抽象基类 (ABCs)** 的 UML 类图 (简图, 未含属性名称)。`collections.abc` 文档中的表格“*Collections Abstract Base Classes*”对这些抽象基类做了总结, 描述了它们之间的继承关系以及各个基类提供的**抽象方法 (Abstract Method)**与**具体方法 (Concrete Method)** (又称“**混合 (Mixin) 方法**”)。图 13.4 中, 存在大量的多重继承。“**十四 继承的利与弊**”将着重介绍多重继承。现在, 仅需指出, 当涉及**抽象基类 (ABCs)**时, 多重继承通常不是问题⁴。

下面详细描述一下图 13.4 中的这些基类。

- `Iterable`、`Container` 和 `Sized`

每个容器 (collection) 都应该继承这些**抽象基类 (ABCs)**, 或实现兼容的协议。`Iterable` 通过 `__iter__` 支持迭代, `Container` 通过 `__contains__` 支持 `in` 运算符, `Sized` 通过 `__len__` 支持 `len()`。

³ `__iadd__` 方法为就地拼接运算符 `+=` 提供底层支持。

⁴ Java 认为多重继承有危害, 因此不支持多重继承, 但是接口除外。Java 接口可以扩展多个接口, 而且 Java 类可以实现多个接口。

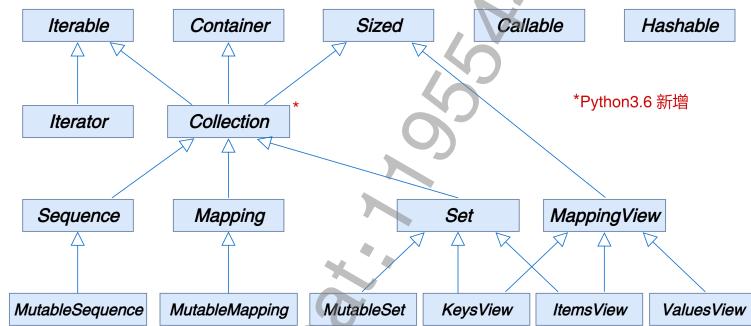


图 13.4: collections.abc 模块中抽象基类的 UML 类图

- Collection

`Collection` 是 Python 3.6 新增的, 自身未提供方法。该基类存在的目的是为了方便子类化 `Iterable`、`Container` 和 `Sized`。

- Sequence、Mapping 和 Set

这 3 个 **抽象基类 (ABCs)** 是主要的不可变容器类型, 各自都有一个可变的子类。“图 13.3<362页>”是 `MutableSequence` 类的详细 UML 类图; 关于 `MutableMapping` 类与 `MutableSet` 类的 UML 类图, 请参见“[三字典与集合](#)”中的“图 3.1<70页>”与“图 3.2<89页>”。

- MappingView

在 Python 3 中, 从映射方法 `.items()`、`.keys()` 和 `.values()` 返回的对象分别实现了在 `ItemsView`、`KeysView` 和 `ValuesView` 中定义的接口。前 2 个对象还实现了丰富的 `Set` 接口, 以及“[3.11.1 集合运算](#)<88页>”中描述的所有运算符。

- Iterator

请注意, **迭代器 (Iterator)** 是 `Iterable` 的子类。详见“[十七 迭代器、生成器和经典协程](#)”。

- Callable、Hashable

这两个不是容器, 只不过因为 `collections.abc` 是标准库中第一个定义 **抽象基类 (ABCs)** 的软件包, 而 `Callable` 与 `Hashable` 又太重要了, 因此被放在 `collections.abc` 内。二者可在类型检查中, 用于判断可调用对象与**可哈希 (hashable)**对象。

对于可调用检测, 内置函数 `callable(obj)` 比 `isinstance(obj, Callable)` 更方便。

如果 `isinstance(obj, Hashable)` 返回 `False`, 则可以肯定 `obj` 不是 **可哈希 (hashable)** 的。但如果返回 `True`, 则可能是误报。下面的附注栏将解释其原因。

用 `isinstance` 检查 `Hashable` 与 `Iterable`, 结果可能不准确

对 **抽象基类 (ABCs)** `Hashable` 与 `Iterable` 进行 `isinstance` 或 `issubclass` 测试的结果, 很容易让人误解。

若 `isinstance(obj, Hashable)` 返回 `True`, 这只意味着 `obj` 的类实现或继承了 `__hash__` 方法。例如, 若 `obj` 是一个包含 `list` 的元组, 即便 `isinstance(obj, Hashable)` 返回 `True`, 而 `obj` 仍是不 **可哈希 (hashable)** 的。本书技术审校 Jürgen Gmach 指出, [鸭子类型 \(Duck Typing\)](#) 提供了确定实例是否 **可哈希 (hashable)** 的最准确方法, 即 `hash(obj)`。如果 `obj` 不可哈希, 调用 `hash(obj)` 将引发 `TypeError`。

另外, 即便 `isinstance(obj, Iterable)` 返回 `False`, Python 仍然可通过 `__getitem__` (利用从 0 开始的整数索引) 迭代 `obj`, 参见“[一 Python 数据模型](#)”以及“[小节 13.4.1<353页>](#)”所示。`collections.abc.Iterable`

文档指出：“确定对象 `obj` 是否可迭代，唯一可靠的方式是调用 `iter(obj)`”。

在了解了一些现有的 **抽象基类 (ABCs)** 之后，下面从头开始实现一个 **抽象基类 (ABCs)**，并将其投入使用，以练习 **大鹅类型 (Goose Typing)**。这么做的目的不是鼓励所有人自己动手创建 **抽象基类 (ABCs)**，而是借此学习如何阅读标准库和其他软件包中 **抽象基类 (ABCs)** 的源码。

13.5.3 定义并使用一个抽象基类

本书第 1 版，在讲“接口”一章给出了如下警告：

抽象基类 (ABCs) 与 **描述符 (Descriptor)**、**元类 (MetaClass)** 一样，都是用于构建框架的工具。因此，只有少数 Python 开发者能够创建 **抽象基类 (ABCs)**，而不会给其他程序员带来不合理的限制和不必要的工作。

如今，**抽象基类 (ABCs)** 使用范围更广，可支持静态类型的 **类型提示 (Type Hints)**。正如“8.5.7 抽象基类<226页>”所述，在函数 **形参 (Parameter)** 的类型提示中用**抽象基类 (ABCs)** 取代**具体类 (Concrete Class)**，可为函数调用者提供更大的灵活性。

为了证明创建 **抽象基类 (ABCs)** 的必要性，需要提供一个在框架中使用 **抽象基类 (ABCs)** 的场景。因此，我们的场景是：您需要在网站或移动 app 上随机显示广告，但在整个广告清单显示完毕之前，不会重复显示广告。假设我们正在构建一个名为 ADAM 的广告管理框架，框架的职责之一是，支持用户提供的非重复随机选取类⁵。为了让 ADAM 用户清楚地了解“非重复随机选取”组件的预期行为，我们将定义一个 **抽象基类 (ABCs)**。

在有关数据结构的文献中，“堆栈 (stack)”与“队列 (queue)”以对象的物理排列方式，描述了抽象接口。我将效仿它们，并用一个现实世界的比喻，来命名我们的 **抽象基类 (ABCs)**。宾果机 (BingoCage) 与彩票机 (LotteryBlower) 是从有限的集合中随机抽取物品的机器，选出的物品没有重复，直至选完为止。

这里将待创建的 **抽象基类 (ABCs)** 命名为 `Tombola`——取自宾果机与混合数字滚动器的意大利语名称。

抽象基类 (ABCs) `Tombola` 共包含 4 个方法。其中，2 个 **抽象方法 (Abstract Method)** 是：

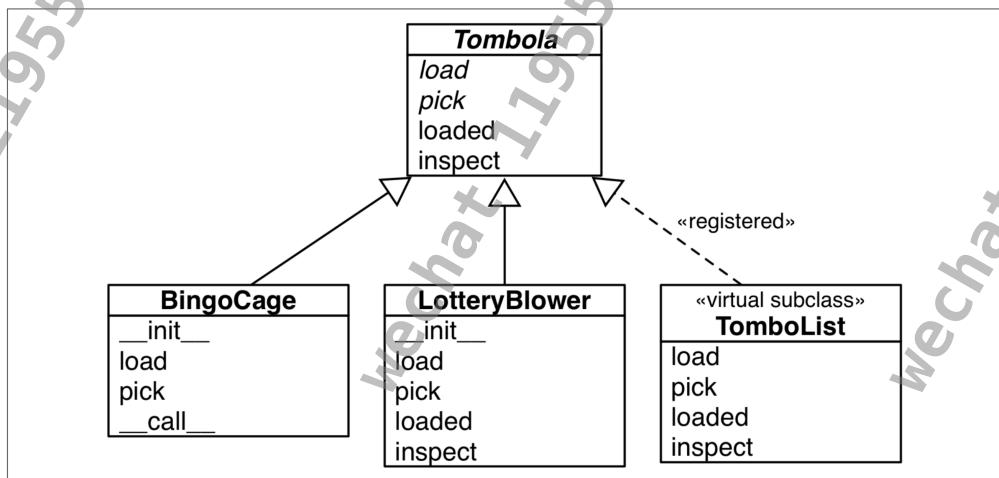
- `.load(...)`
将元素放入容器。
- `.pick()`
从容器中随机取出一个元素，再返回这个元素。

另外 2 个是 **具体方法 (Concrete Method)**：

- `.loaded()`
如果容器中至少有一个元素，则返回 `True`。
- `.inspect()`
返回由容器中现有元素构成的元组，不改变容器内容（内部的顺序不保留）。

⁵也许客户需要对随机生成器进行审核，或者代理机构想作弊。谁知道呢……

图 13.5 展示了抽象基类 Tombola 与 3 个具体实现。

图 13.5: 1 个抽象基类与 3 个子类的 UML 类图, 抽象基类 Tombola 及其抽象方法用斜体表示
虚线箭头表示接口实现, 这里表示 TomboList 不仅实现了 Tombola 接口, 还被注册为 Tombola 的虚拟子类 (详见后文)⁶

抽象基类 (ABCs) Tombola 的定义, 如示例 13.7 所示。

</> 示例 13.7: tombola.py: Tombola 是包含 2 个抽象方法与 2 个具体方法的抽象基类

```

1  import abc
2
3  class Tombola(abc.ABC):
4
5      @abc.abstractmethod
6      def load(self, iterable):
7          """从可迭代对象从添加元素"""
8
9      @abc.abstractmethod
10     def pick(self):
11         """随机删除元素, 再返回被删除的元素。
12
13         如果实例为空, 此方法应引发 LookupError 异常。
14         """
15
16     def loaded(self):
17         """若至少有一个元素, 就返回 True。否则, 返回 False。"""
18         return bool(self.inspect())
19
20     def inspect(self):
21         """返回由容器中的当前元素构成的有序元组"""
22         items = []
23         while True:
24             try:
25                 items.append(self.pick())
26             except LookupError:
  
```

⁶ «registered» 与 «virtual subclass» 不是标准的 UML 术语。我用它们表示 Python 特有的类关系。

```

27     break
28     self.load(items)
29     return tuple(items)
    ⑦

```

- ① 继承 `abc.ABC`, 定义一个 **抽象基类 (ABCs)**。
- ② **抽象方法 (Abstract Method)** 用 `@abstractmethod` 装饰器标记, 除了文档字符串外, 其主体通常是空的⁷。
- ③ 根据文档字符串描述: 若没有元素可选, 将引发 `LookupError` 异常。
- ④ **抽象基类 (ABCs)** 中可以包含**具体方法 (Concrete Method)**。
- ⑤ **抽象基类 (ABCs)** 中的 **具体方法 (Concrete Method)** 必需只依赖于 **抽象基类 (ABCs)** 定义的接口 (即抽象基类的其他具体方法或抽象方法或属性)。
- ⑥ 我们不知道 **具体子类 (Concrete Subclass)** 将如何存储元素, 但可以通过连续调用 `.pick()` 清空 `Tombola` 来构建检查结果 ...
- ⑦ ... 然后使用 `.load(...)` 将所有元素放回原处。



其实, **抽象基类 (ABCs)** 中的 **抽象方法 (Abstract Method)** 可以有实现代码, 但是即便如此, 子类也必须重写该抽象方法。但是, 子类可以用 `super()` 函数调用抽象方法, 为它添加功能, 而不是从头实现。有关 `@abstractmethod` 的使用方法, 请参阅“`abc` 模块文档”。

示例 13.7 中 `.inspect()` 方法的代码非常愚蠢, 但它表明我们可以依靠 `.pick()` 和 `.load(...)` 来检查 `Tombola` 中的内容, 方法是选出所有元素再加载回 `Tombola`, 而无需知道元素的实际存储方式。此示例的重点是强调, 在抽象基类中提供具体方法是没有问题的, 只要这些方法只依赖于接口中的其他方法即可。在了解 `Tombola` 内部数据结构后, `Tombola` 的 **具体子类 (Concrete Subclass)** 可以使用更智能的实现来重写 `.inspect()`, 但它们并非必须这样做。

示例 13.7 中的 `.loaded()` 方法只有一行, 但开销很大: 它调用 `.inspect()` 构建元组的目的, 只是为了在元组上调用 `bool()`。虽然这个方法也是可行的, 但是**具体子类 (Concrete Subclass)** 可以做的更好, 后文将详述。

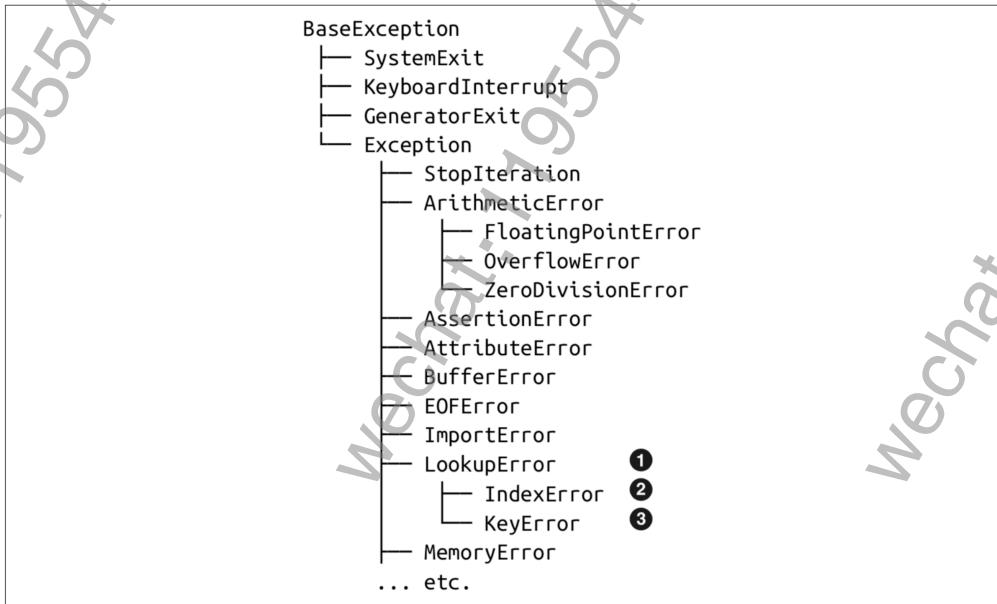
请注意, `.inspect()` 的这种迂回实现方式要求我们捕获由 `self.pick()` 引发的 `LookupError`。`self.pick()` 可能引发 `LookupError` 也是其接口的一部分, 但在 Python 中, 除了在文档 (docstring) 中明确说明外 (参见 **示例 13.7** 中抽象方法 `pick` 的 docstring), 没有办法做到这一点。

之所以选择 `LookupError` 异常, 是因为它在 Python 异常层次结构中的位置与 `IndexError` 和 `KeyError` 相关, 而 `IndexError` 和 `KeyError` 是具体实现 `Tombola` 所用的数据结构最有可能引发的异常。因此, 在实现代码中, 可以引发 `LookupError`、`IndexError`、`KeyError`, 或 `LookupError` 的自定义子类, 以符合要求。如 **图 13.6** 所示。

- ① `LookupError` 是我们在 `Tombola.inspect` 中处理的异常。
- ② `IndexError` 是 `LookupError` 子类, 当试图从序列中获取索引超过最后位置的元素时, 会引发 `IndexError` 异常。

⁷ 在**抽象基类 (ABCs)** 出现之前, 抽象方法会引发 `NotImplementedError`, 表明子类负责实现这些方法。在 Smalltalk-80 中, 抽象方法主体会调用 `subclassResponsibility` (一种从 `object` 继承的方法), 该方法会引发一个错误信息: “My subclass should have overridden one of my messages”。

⁸ 完整的 Exception 层次结构树, 详见 Python 标准库文档 中的“Exception hierarchy”一节。

图 13.6: Exception 类的层次结构¹⁸

- ③ 当试图使用不存在的“键 (Key)”从映射中获取元素时,会引发 `KeyError`。

现在,我们有了自己的 **抽象基类 (ABCs)** `Tombola`。为了验证 **抽象基类 (ABCs)** 的接口检查功能,让我们尝试用一个有缺陷的具体子类 (Concrete Subclass) 实现 (如示例 13.8 所示),来“愚弄”`Tombola` 类。

```

</>示例 13.8: 假冒的 Tombola 不会被识破
1  >>> from tombola import Tombola
2  >>> class Fake(Tombola): ❶
3      ...     def pick(self): ❷
4      ...         return 13
5      ...
6  >>> Fake
7  <class '__main__.Fake'>
8  >>> f = Fake() ❸
9  Traceback (most recent call last):
10  File "<stdin>", line 1, in <module>
11  TypeError: Can't instantiate abstract class Fake with abstract method load

```

- ❶ 将 `Fake` 类声明为 `Tombola` 的子类。`Fake` 类仅实现了 `Tombola` 中的抽象方法 `pick`,而未实现抽象方法 `load`。
- ❷ `Fake` 类已创建,目前未出现任何错误。
- ❸ 当尝试实例化 `Fake` 类时,会引发 `TypeError` 异常。错误消息很明确:Python 认为 `Fake` 是一个抽象类,因为它没有实现 **抽象基类 (ABCs)** `Tombola` 中声明的抽象方法之一 `load`。

这样,我们就定义了第一个 **抽象基类 (ABCs)**,并将其用于验证一个类。稍后会对 **抽象基类 (ABCs)** `Tombola` 进行子类化。但在此之前,必须先了解一些 **抽象基类 (ABCs)** 的编码规则。

13.5.4 抽象基类语法详解

声明 **抽象基类 (ABCs)** 的标准方式是继承 `abc.ABC` 或其它 **抽象基类 (ABCs)**。

除了 `ABC` 基类和 `@abstractmethod` 装饰器, `abc` 模块 还定义了 `@abstractclassmethod`、`@abstractstaticmethod` 和 `@abstractproperty` 装饰器。但是, 后 3 个装饰器在 Python 3.3 中已被弃用。因为可以在 `@abstractmethod` 上叠放装饰器, 这使得其他装饰器变得多余。例如, 声明 **抽象方法 (Abstract Method)** 的首选方法是:

```
1 class MyABC(abc.ABC):  
2     @classmethod  
3     @abc.abstractmethod  
4     def an_abstract_classmethod(cls, ...):  
5         pass
```



叠放函数装饰器的顺序很重要, `@abstractmethod` 文档 明确指出:

当与其他方法描述符结合使用时, 应将 `abstractmethod()` 放在最内层

... a

^a 摘自 `abc` 模块文档 中的 `@abstractmethod` 词条。

换句话说, 在 `@abstractmethod` 与 `def` 语句之间, 不得出现其他装饰器。

在了解了 **抽象基类 (ABCs)** 语法之后, 让我们通过实现 `Tombola` 的 2 个**具体子类 (Concrete Subclass)** 来使用 `Tombola`。

13.5.5 子类化抽象基类

有了 **抽象基类 (ABCs)** `Tombola` 之后, 现在要开发 2 个满足其接口的 **具体子类 (Concrete Subclass)**。 “图 13.5<366页>” 展示了这 2 个**具体子类 (Concrete Subclass)**, 以及下一节将要讨论的**虚拟子类 (Virtual Subclass)**。

示例 13.9 的 `BingoCage` 类是 “示例 7.8<193页>” 的变体, 使用了更好的随机发生器。 `BingoCage` 实现了所需的抽象方法 `load` 和 `pick`。

</> 示例 13.9: `bingo.py`: `BingoCage` 是 `Tombola` 的具体子类

```
1 import random  
2  
3 from tombola import Tombola  
4  
5 class BingoCage(Tombola): ❶  
6  
7     def __init__(self, items):  
8         self._randomizer = random.SystemRandom() ❹  
9         self._items = []  
10        self.load(items) ❺
```

```

11
12     def load(self, items):      self._items.extend(items)
13     self._randomizer.shuffle(self._items) ❸
14
15     def pick(self):          ❶
16         try:
17             return self._items.pop()
18         except IndexError:
19             raise LookupError('pick from empty BingoCage')
20
21     def __call__(self):        ❷
22         self.pick()

```

- ❶ BingoCage 类明确地扩展了 Tombola 类。
- ❷ 假设将在线上游戏中使用这个随机发生器。`random.SystemRandom` 在 `os.urandom()` 函数之上实现随机 API, 根据 `os` 模块文档 描述, `os.urandom()` 函数 提供“适合加密使用的随机字节”。
- ❸ 将初始加载委托给 `load(...)` 方法。
- ❹ 使用 `SystemRandom` 实例的 `shuffle()` 方法, 而不是普通的 `random.shuffle()` 函数。
- ❺ `pick` 方法的实现方式与“[示例 7.8<193页>](#)”一致。
- ❻ `__call__` 方法的实现方式也与“[示例 7.8<193页>](#)”一致。为了满足 `Tombola` 接口的要求, 并无须实现这个方法, 不过额外增加方法也没什么坏处。

`BingoCage` 从 `Tombola` 继承了耗时的 `loaded` 方法和笨拙的 `inspect` 方法。这两种方法都可以用更快的单行方法重写(如 [示例 13.10](#) 所示)。此处想表达的观点是:我们可以偷懒, 直接从 [抽象基类 \(ABCs\)](#) 中继承次优的具体方法 (Concrete Method)。虽然从 `Tombola` 继承的方法并没有 `BingoCage` 的方法快。但是对于任何正确实现了 `pick` 和 `load` 的 `Tombola` 子类来说, 这些方法 (`loaded` 与 `inspect`) 都能提供正确的结果。

[示例 13.10](#) 展示了 `Tombola` 接口的另一种完全不同(但有效)的实现。`LottoBlower` 将“数字球 (`self._balls`)”打乱后, 没有弹出最后一个, 而是弹出一个随机位置上的球。

</> [示例 13.10: lottery.py](#): `LotteryBlower` 是 `Tombola` 的具体子类, 重写了继承的 `inspect` 与 `loaded` 方法

```

1 import random
2
3 from tombola import Tombola
4
5 class LottoBlower(Tombola):
6
7     def __init__(self, iterable):
8         self._balls = list(iterable) ❶
9
10    def load(self, iterable):
11        self._balls.extend(iterable)
12
13    def pick(self):
14        try:
15            position = random.randrange(len(self._balls)) ❷
16        except ValueError:

```

```
17     raise LookupError('pick from empty LottoBlower')
18     return self._balls.pop(position)          ③
19
20     def loaded(self):
21         return bool(self._balls)
22
23     def inspect(self):
24         return tuple(self._balls)             ⑤
```

- ① 初始化方法可接受任何可迭代对象,根据传入的实参 (Argument) 构建一个 list。
- ② 如果范围为空,random.randrange(...) 函数会引发 ValueError 异常。因此,我们需要捕捉 ValueError 异常,并重新抛出了 LookupError,以便与 Tombola 兼容 ...
- ③ ... 否则,从 self._balls 中弹出随机选择的项 (球)。
- ④ 重写 loaded 方法,以避免调用 inspect (“[示例 13.7<366页>](#)” 中 Tombola.loaded 就是这么做的)。直接处理 self._balls (无需构建一个全新的元组),可加快处理速度。
- ⑤ 仅用一行代码,重写 inspect 方法。

[示例 13.10](#) 展示了一个值得一提的习惯用法:在 `__init__` 中, `self._balls` 存储的是 `list(iterable)`, 而不是对 `iterable` 的引用 (即未直接将 `iterable` 赋值给 `self._balls`, 为实参创建别名⁹)。如 “[13.4.3 防御性编程与快速失败<356页>](#)” 所述,这么做使得 `LottoBlower` 更为灵活,因为实参 (Argument) `iterable` 可以是任何可迭代类型。同时,将元素存储在一个列表中,还可以确保能弹出元素。即使为 实参 (Argument) 传入列表, `list(iterable)` 操作也会为传入的列表生成一个副本。考虑到我们将从实参中删除元素,而客户可能不希望自己提供的列表被修改,因此 `list(iterable)` 仍然是一种绝佳的做法。

接下来,介绍 [大鹅类型 \(Goose Typing\)](#) 的关键动态特性:使用 `register` 方法声明 [虚拟子类 \(Virtual Subclass\)](#)。

13.5.6 抽象基类的虚拟子类

[大鹅类型 \(Goose Typing\)](#) 的一个基本特征 (也是它值得被称为“水禽”的原因之一) 是即便没有继承关系,也能将一个类注册为“[抽象基类](#)”的“[虚拟子类](#)”。这样做时,我们将承诺该 [虚拟子类 \(Virtual Subclass\)](#) 忠实地实现了 [抽象基类 \(ABCs\)](#) 中定义的接口,而 Python 会相信我们的承诺,不再检查。如果我们撒谎,则常规的运行时异常会将我们捕获。

注册 [虚拟子类 \(Virtual Subclass\)](#) 的方式是在 [抽象基类 \(ABCs\)](#) 上调用 `register` 类方法。注册后的类将成为 [抽象基类 \(ABCs\)](#) 的 [虚拟子类 \(Virtual Subclass\)](#),可被 `issubclass()` 识别,但它不会继承 [抽象基类 \(ABCs\)](#) 的任何方法或属性。

⁹“[6.5.2 可变参数的防御性编程<175页>](#)”专门讨论了我们在此处避免的别名问题。



虚拟子类 (Virtual Subclass) 不会从其注册的 抽象基类 (ABCs) 中继承。而且在任何时候, 虚拟子类 (Virtual Subclass) 都不会被检查是否符合 抽象基类 (ABCs) 的接口, 甚至在实例化时也不会被检查。此外, 静态类型检查器目前也无法处理虚拟子类 (Virtual Subclass)。详情见“[Mypy issue 2922——ABCMeta.register support](#)”。

类方法 register() 通常作为普通函数调用 (见“[13.5.7 register 的实际使用](#)”), 但也可以作为装饰器使用。

在[示例 13.11](#) 中, 使用装饰器语法实现了 Tombola 的虚拟子类 TomboList (如图 13.7 所示)。

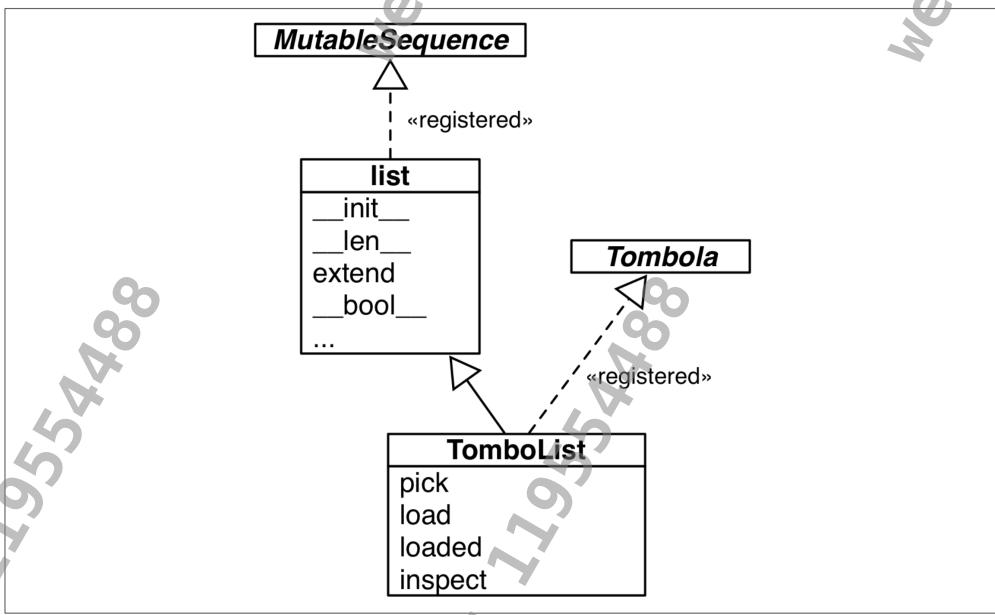


图 13.7: 虚拟子类 TomboList 的 UML 类图
它是 list 的真实子类, 是 Tombola 的虚拟子类

</> [示例 13.11: tombolist.py:TomboList 是 Tombola 的虚拟子类](#)

```

1  from random import randrange
2
3  from tombola import Tombola
4
5  @Tombola.register          ❶
6  class TomboList(list):      ❷
7
8      def pick(self):          ❸
9          if self:              ❹
10             position = randrange(len(self)) ❺
11             return self.pop(position) ❻
12         else:
13             raise LookupError('pop from empty TomboList')
14
15     load = list.extend        ❼
16
17     def loaded(self):         ❼
18         return bool(self)    ❽
  
```

```

19
20     def inspect(self):      return tuple(self)
21
22     # Tombola.register(TomboList) ❷

```

- ❶ Tombolist 被注册为 Tombola 的 [虚拟子类 \(Virtual Subclass\)](#)。
- ❷ Tombolist 扩展了 list, 是 list 的真实子类。
- ❸ Tombolist 继承了 list 的布尔值行为, 如果列表不为空, 则返回 True。
- ❹ pick 调用 self.pop (继承自 list), 并传入一个随机的元素索引。
- ❺ Tombolist.load 等同于 list.extend。
- ❻ 将 loaded 委托给 bool¹⁰。
- ❼ 始终可以这样调用 register。若需要注册不是自己维护 (但能满足接口要求) 的类时, 可以这么做。

请注意, 由于注册的原因, 函数 `issubclass()` 和 `isinstance()` 的行为就像 TomboList 是 Tombola 的子类一样:

```

1  >>> from tombola import Tombola
2  >>> from tombolist import TomboList
3  >>> issubclass(TomboList, Tombola)
4  True
5  >>> t = TomboList(range(100))
6  >>> isinstance(t, Tombola)
7  True

```

然而, 类的继承关系是由类属性 `__mro__` 指定的, 即 [MRO \(方法解析顺序\)](#)。MRO 按照 Python 解释器用于搜索方法的顺序, 列出了类及其超类¹¹。如果检查 TomboList 的 `__mro__` 属性, 你会发现它只列出了“真实的”超类, 即 list 与 object:

```

1  >>> TomboList.__mro__
2  (<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)

```

Tombola 不在 Tombolist.`__mro__` 类属性中, 因此虚拟子类 Tombolist 不会从抽象基类 Tombola 中继承任何方法。

对 [抽象基类 \(ABCs\)](#) Tombola 的案例研究到此结束。下一节, 将介绍 `register()` 函数的实际使用。

13.5.7 register 的实际使用

在 [示例 13.11](#) 中, 将 `Tombola.register` 当作类装饰器来使用。在 Python 3.3 之前, `register` 不能这么用, 而是必须在类定义之后作为普通函数来调用 (如 [示例 13.11](#) 末尾❷处的注释所示)。然而, 即便是现在, 仍然经常将 `register` 函数当作普通函数来调用, 以便注册在其他地方定义的类。例如, 在 `collections.abc` 模块的源码中, 内置类型 `tuple`、`str`、`range` 与 `memoryview` 被注册为 [Sequence](#) 的 [虚拟子类 \(Virtual Subclass\)](#), 如下所示:

¹⁰`loaded()` 中不能使用 `load()` 的那种处理方式, 因为 `list` 类型没有实现 `loaded` 所需的 `__bool__` 方法。而内置的 `bool()` 不需要 `__bool__` 也能工作, 因为它还可以使用 `__len__`。参见 Python 文档 [“Built-in Types”](#) 一节中的 [“Truth Value Testing”](#)。

¹¹“[14.4 多重继承与方法解析顺序](#)”[<398页>](#)”专门讲解了 `__mro__` 类属性。现在, 只需要了解这个简单的解释就足够了。

```

1 Sequence.register(tuple)
2 Sequence.register(str)Sequence.register(range)
3 Sequence.register(memoryview)

```

其他几个内置类型在 `Lib/_collections_abc.py` 中也被注册为 **抽象基类 (ABCs)** 的 **虚拟子类 (Virtual Subclass)**。注册过程仅在导入该模块时才发生,这是没有问题的,因为必需导入模块才能使用模块提供的**抽象基类 (ABCs)**。例如,从 `collections.abc` 中导入 `MutableMapping` 之后,才能执行 `isinstance(my_dict, MutableMapping)` 检查。

子类化 **抽象基类 (ABCs)** 或将类注册到 **抽象基类 (ABCs)**,都能让类通过 `issubclass` 与 `isinstance` 检查(后者依赖前者)。但是,有些**抽象基类 (ABCs)**还支持结构化类型,详见下一节。

13.5.8 用抽象基类实现结构类型

抽象基类 (ABCs) 主要用于实现 **名义类型 (Nominal Typing)**。当 `Sub` 类明确继承自 `AnABC` 或向 `AnABC` 注册时, `AnABC` 的名称将被链接到 `Sub` 类中。所以,在运行时 `issubclass(AnABC, Sub)` 会返回 `True`。

相比之下, **结构类型 (Structural Typing)** 是通过观察对象公开接口的结构来确定其类型——如果一个对象实现了类型中定义的方法,那么它就与该类型**相容 (Consistent-with)**¹²。**动态鸭子类型 (Duck Typing)** 与 **静态鸭子类型 (Static Duck Typing)** 是实现结构类型的两种方式。

事实证明,有些**抽象基类 (ABCs)**也支持**结构类型 (Structural Typing)**。如 Alex 在“**水禽与抽象基类**”(见“**13.5 大鹅类型 (Goose Typing) <358页>**”)中所述:未注册的类也可能被识别为**抽象基类 (ABCs)**的子类。下面是他的示例(增加了 `issubclass` 测试):

```

1 >>> class Struggle:
2     ...     def __len__(self): return 23
3     ...
4 >>> from collections import abc
5 >>> isinstance(Struggle(), abc.Sized)
6 True
7 >>> issubclass(Struggle, abc.Sized)
8 True

```

因为 `abc.Sized` 实现了一个名为 `__subclasshook__` 的特殊类方法。所以,通过 `issubclass` 函数将 `Struggle` 类视为 `abc.Sized` 的子类(`isinstance` 也是如此)。

`abc.Sized` 的 `__subclasshook__` 类方法会检查通过 `cls` 传入的类是否有 `__len__` 属性。如果是,则认为它是 `Sized` 的**虚拟子类 (Virtual Subclass)**。如**示例 13.12** 所示。

</> **示例 13.12: Sized 类的定义 (源码见 `Lib/_collections_abc.py`)**

```

1 class Sized(metaclass=ABCMeta):
2
3     __slots__ = ()
4
5     @abstractmethod
6     def __len__(self):

```

¹²有关“类型相容”的概念,详见“**8.5.1.1 子类型与相容 <217页>**”。

```
7     return 0
8     @classmethod  def __subclasshook__(cls, C):
9
10    if cls is Sized:
11        if any("__len__" in B.__dict__ for B in C.__mro__): ❶
12            return True ❷
13    return NotImplemented ❸
```

- ❶ 如果在 C.__mro__ 中列出的某个类 (即 C 及其超类) 的 __dict__ 中有名为 __len__ 的属性 ...
- ❷ ... 就返回 True, 表明 C 是 Sized 的 虚拟子类 (Virtual Subclass)。
- ❸ 否则, 返回 NotImplemented, 以便让子类继续检查。



如果对子类检查的细节感兴趣, 请参阅 Python 3.6 中 ABCMeta.__subclasscheck__ 方法的源代码: [Lib/abc.py](#)。注意: 它有大量的 if 和两个递归调用。在 Python 3.7 中, 为了提高性能, Ivan Levkivskyi 和 Inada Naoki 用 C 重写了 abc 模块的大部分逻辑。参见 [Python issue #31333](#)。当前的 ABCMeta.__subclasscheck__ 实现只是简单地调用 __abc_subclasscheck__。相关的 C 语言源码见 [cpython/Modules/_abc.c](#) 第 605 行。

抽象基类 (ABCs) 对 结构类型 (Structural Typing) 的支持, 就是通过类方法 __subclasshook__ 实现的。可以用 抽象基类 (ABCs) 确立 (定义) 一个接口, 然后对这个 抽象基类 (ABCs) 进行 issubclass 检查。即使完全不相关的类也可通过 issubclass 检查, 只要它实现了某个方法 (或者不惜一切代价说服 __subclasshook__ 为它担保) 即可。

那么, 我们自己定义的 抽象基类 (ABCs) 应该实现 __subclasshook__ 类方法么? 或许不应该。在 Python 源码中, 所有的 __subclasshook__ 方法实现都是在类似 Sized 这样声明了一个特殊方法的 抽象基类 (ABCs) 中, 而且它们只是简单地检查了这个特殊方法的名称。由于 __len__ 是“特殊”方法, 因此可以确信任何名为 __len__ 的方法, 都会按预期运行。但是, 即便是特殊方法和基本的 抽象基类 (ABCs), 这种假设也是有风险的。例如, 虽然映射实现了 __len__、__getitem__ 和 __iter__, 但不能将它视作 Sequence 的子类型, 因为无法通过整数偏移或切片来检索映射中的元素。这就是为什么 abc.Sequence 类没有实现 __subclasshook__ 的原因。

而对于你我编写的 抽象基类 (ABCs), 类方法 __subclasshook__ 就更不可靠了。我可不认为任何名为 Spam 的类, 只要实现或继承了 load、pick、inspect 和 loaded 方法, 就一定会表现得像 Tombola (见“[示例 13.7<366页>](#)”一样)。最好的做法是让程序员通过从 Tombola 中派生 Spam, 或用 Tombola.register(Spam) 来注册 Spam, 这样才能万无一失。当然, 你的 __subclasshook__ 也可以检查方法签名和其他特性, 但我认为这不值得。

13.6 静态协议



“8.5.10 静态协议<233页>”中介绍过静态协议。我曾考虑过将协议 (protocol) 的所有内容都推迟到本章来介绍,但是在初始介绍函数的类型提示 (Type Hints) 时,又不得不提到协议。因为 鸭子类型 (Duck Typing) 是 协议 (protocol) 的重要组成部分,而没有协议的静态类型检查无法很好的处理 Pythonic (Python 风格) API。

在本章最后,将通过两个简单的示例来介绍静态协议,并讨论 numeric 抽象基类 和 协议 (protocol)。首先,说明如何利用静态协议来注解 double() 函数(首次出现在“8.4 类型由支持的操作来定义<211页>”),并对它做类型检查。

13.6.1 为函数 double 添加类型提示

在向习惯于静态类型语言的程序员介绍 Python 时,我最喜欢用这个简单的 double() 函数举例。

```

1  >>> def double(x):
2      ...
3      ...
4  >>> double(1.5)
5  3.0
6  >>> double('A')
7  'AA'
8  >>> double([10, 20, 30])
9  [10, 20, 30, 10, 20, 30]
10 >>> from fractions import Fraction
11 >>> double(Fraction(2, 5))
12 Fraction(4, 5)

```

在引入静态协议之前,几乎无法在不限制 double() 函数用途¹³的前提下,为其添加 类型提示 (Type Hints)。

得益于 鸭子类型 (Duck Typing),函数 double 甚至可以支持未来可能出现的类型。例如,“16.5 重载标量乘法运算符 *”新增的增强型 Vector 类(“示例 16.11<461页>”)

```

1  >>> from vector_v7 import Vector
2  >>> double(Vector([11.0, 12.0, 13.0]))
3  Vector([22.0, 24.0, 26.0])

```

Python 中 类型提示 (Type Hints) 的最初实现是一个名义类型 (Nominal Typing) 系统:注解中的类型名称必须与 实参 (Argument) 的类型 (或其超类) 名称相匹配。在 Python 3.8 之前,由于无法通过支持所需操作来命名实现了某个协议的所有类型,因此无法通过 类型提示 (Type Hints) 来明确表示 鸭子类型 (Duck Typing)。

¹³好吧,double() 函数除了可用于演示,并没什么大用处。不过在 Python 3.8 添加静态协议之前,Python 标准库中也有许多函数无法准确注释。我通过使用 协议 (protocol) 添加 类型提示 (Type Hints),帮助修复了 Typeshed 项目 中的几个 bug。例如,修复了“issue#4061 Should Mypy warn about potential invalid arguments to max?”,利用 _SupportsLessThan 协议,改进了 max,min,sorted 和 list.sort 的注解。

有了 `typing.Protocol` 之后, 我们可以告诉 Mypy 类型检查器: 函数 `double` 接受一个支持 $x * 2$ 运算的实参 `x`, 如 [示例 13.13](#) 所示。

</> [示例 13.13: double_protocol.py: 用 Protocol 定义 double 函数](#)

```

1  from typing import TypeVar, Protocol
2
3  T = TypeVar('T') ❶
4
5  class Repeatable(Protocol):
6      def __mul__(self: T, repeat_count: int) -> T: ... ❷
7
8  RT = TypeVar('RT', bound=Repeatable) ❸
9
10 def double(x: RT) -> RT: ❹
11     return x * 2

```

- ❶ 将在 `__mul__` 函数签名中, 使用这个类型变量 `T`。
- ❷ `__mul__` 是 `Repeatable` 协议的核心。参数 `self` 通常不需要注解——默认被假定为当前实例所属的类。此处使用 `T` 是为了确保返回值类型与 `self` 类型相同。此外, 请注意 `repeat_count` 在此协议中仅限于 `int` 类型。
- ❸ 类型变量 `RT` 的上边界由 `Repeatable` 协议限定: 类型检查工具将要求具体使用的类型实现 `Repeatable` 协议。
- ❹ 现在, 类型检查器可以验证参数 `x` 是一个可以与整数相乘的对象, 而返回值的类型与 `x` 相同。

通过此示例可看出, 为何 [PEP 544](#) 的标题为 “Protocols: Structural subtyping (static duck typing)”。提供给函数 `double` 的实参 `x` 是什么 [名义类型](#) (`Nominal Typing`), 并不重要。重要的是, 只要它实现了 `__mul__` 即可——也就是说 “它会呱呱叫” 就行了。

13.6.2 运行时可检查的静态协议

在 [类型图](#) (见 “[图 13.1<350页>](#)”) 中, `typing.Protocol` 位于静态检查区域 (即图的下半部分)。然而, 在定义 `typing.Protocol` 子类时, 可以使用 `typing.runtime_checkable` 装饰器 使该协议支持运行时的 `isinstance()` / `issubclass()` 检查。其背后的原因是, `typing.Protocol` 是一个抽象基类, 因此它支持特殊的类方法 `__subclasshook__` (详见 “[13.5.8 用抽象基类实现结构类型<374页>](#)”)。

从 Python 3.9 开始, `typing` 模块 包含了 7 个可在运行时检查的即用协议。以下是其中的 2 个, 直接引自 `typing` 模块文档:

```

1  class typing.SupportsComplex
2      An ABC with one abstract method, __complex__.
3
4  class typing.SupportsFloat
5      An ABC with one abstract method, __float__.

```

这些 [协议](#) 旨在检查数字类型是否可转换类型: 如果一个对象 `o` 实现了 `__complex__` 方法, 那么通过调用 `complex(o)` 即可得到一个复数——因为特殊方法 `__complex__` 的存在, 是为了给内置函数 `complex()` 提

供支持。

示例 13.14 展示了 `typing.SupportsComplex` 协议的源代码（见 `Lib/typing.py` 1751 行）。

```
</> 示例 13.14: typing.SupportsComplex 协议源代码
1 @runtime_checkable
2 class SupportsComplex(Protocol):
3     """An ABC with one abstract method __complex__."""
4     __slots__ = ()
5
6     @abstractmethod
7     def __complex__(self) -> complex:
8         pass
```

`typing.SupportsComplex` 协议的核心¹⁴是抽象方法 `__complex__`。在静态类型检查过程中，如果一个对象实现了 `__complex__` 方法，而该方法只接收 `self` 并返回一个复数。那么该对象将被视为与 `SupportsComplex` 协议 **相容** (*Consistent-with*)。

由于在协议 `SupportsComplex` 中应用了 `@runtime_checkable` 类装饰器，该协议也可以与 `isinstance()` 检查一起使用，如 ?? 所示。

```
>>> from typing import SupportsComplex
>>> import numpy as np
>>> c64 = np.complex64(3+4j) ❶
>>> isinstance(c64, complex) ❷
False
>>> isinstance(c64, SupportsComplex) ❸
True
>>> c = complex(c64) ❹
>>> c
(3+4j)
>>> isinstance(c, SupportsComplex) ❺
False
>>> complex(c)
(3+4j)
```

- ❶ `complex64` 是 NumPy 提供的 5 种复数类型之一。
- ❷ NumPy 中的复数类型均不是内置类型 `complex` 的子类。
- ❸ 但 NumPy 的复数类型实现了 `__complex__` 方法，因此它们遵守 `SupportsComplex` 协议。
- ❹ 因此，可以用 NumPy 中的复数类型创建内置的 `complex` 对象。
- ❺ 遗憾的是，内置类型 `complex` 没有实现 `__complex__` 方法。不过，当 `c` 是一个 `complex` 值时，`complex(c)` 可以得到正确的结果。

根据最后一点，若要测试对象 `c` 是否是 `complex` 或 `SupportsComplex`。那么，可以为 `isinstance` 的第 2 个参数提供一个类型元组，如下所示：

```
1 isinstance(c, (complex, SupportsComplex))
```

¹⁴类属性 `__slots__` 与当前讨论的内容无关，它是“[11.11 用 `__slots__` 节省内存<311页>](#)”中介绍的一种内存优化措施。

另一种方法是使用由 `Numbers` 模块 定义的 **抽象基类** `Complex`。内置类型 `complex` 以及 NumPy 中的 `complex64` 和 `complex128` 都被注册为 `numbers.Complex` 的**虚拟子类** (Virtual Subclass)，因此可以使用：

```

1 >>> import numbers
2 >>> isinstance(c, numbers.Complex)
3 True
4 >>> isinstance(c64, numbers.Complex)
5 True

```

本书第 1 版建议使用 `Numbers` 模块 中的**抽象基类** (ABCs)。但是，现在这个建议已经过时，因为静态类型检查器无法识别这些**抽象基类** (ABCs)，详见“13.6.8 `number` 模块中的抽象基类与 Numeric 协议<385页>”。

本节主要目的是：演示运行时可检查协议可以与 `isinstance` 检查一起使用。但事实证明，这个示例并不是特别适合使用 `isinstance`，具体原因见后文附注栏“**充分利用鸭子类型**”。



如果使用外部类型检查工具，则使用 `isinstance` 显式检查类型会有一个好处：当编写条件为 `isinstance(o, MyType)` 的 `if` 语句时，Mypy 可以 `if` 语句块内推断出 `o` 对象的类型与 `MyType` 相容 (Consistent-with)。

充分利用鸭子类型

在运行时，**鸭子类型** (Duck Typing) 通常是类型检查的最佳方法：无需调用 `isinstance` 或 `hasattr`，只需尝试需要对对象执行的操作，并根据需要处理异常。示例如下：

继续之前的讨论——假设我想将对象 `o` 当作复数来使用，则可以：

```

1 if isinstance(o, (complex, SupportsComplex)):
2     # 当 o 可以转换为 complex 时，执行一些操作。
3 else:
4     raise TypeError('o must be convertible to complex')

```

对于**大鹅类型** (Goose Typing)，则要使用**抽象基类** `numbers.Complex`：

```

1 if isinstance(o, numbers.Complex):
2     # 当 o 是 Complex 实例时，执行一些操作。
3 else:
4     raise TypeError('o must be an instance of Complex')

```

然而，我更喜欢利用**鸭子类型** (Duck Typing) 来做到这一点，因为“请求原谅比请求许可更容易” (EAFP 原则)。

```

1 try:
2     c = complex(o)
3 except TypeError as exc:
4     raise TypeError('o must be convertible to complex') from exc

```

而且，如果你要做的只是引发 `TypeError`，那么可以省略 `try/except/raise` 语句，直接这样写：

```

1 c = complex(o)

```

在最后一种情况下,如果 `o` 不是可接受的类型,Python 会引发一个异常,并给出非常明确的信息。例如,如果 `o` 是一个元组,我就会得到这样的结果:

```
1  TypeError: complex() first argument must be a string or a number, not 'tuple'
```

我觉得在这种情况下,使用 鸭子类型(Duck Typing) 效果要好得多。

现在,我们已经了解了:在运行时,如何利用静态协议检查诸如 `complex`、`numpy.complex64` 之类的现有类型。接下来,将讨论运行时可检查 (checkable) 协议的局限性。

13.6.3 运行时协议检查的局限性

如前所述,类型提示 (Type Hints) 通常会在运行时被忽略,这也会影响针对静态协议的 `isinstance` 或 `issubclass` 检查的使用。

例如,任何具有 `__float__` 方法的类在运行时都会被视为 `SupportsFloat` 的虚拟子类 (Virtual Subclass),即使 `__float__` 方法不返回 `float` 值。

请看如下的控制台会话:

```
1  >>> import sys
2  >>> sys.version
3  '3.9.5 (v3.9.5:0a7dcdbb13, May 3 2021, 13:17:02) \n[Clang 6.0 (clang-600.0.57)]'
4  >>> c = 3+4j
5  >>> c.__float__
6  <method-wrapper '__float__' of complex object at 0x10a16c590>
7  >>> c.__float__()
8  Traceback (most recent call last):
9  File "<stdin>", line 1, in <module>
10 TypeError: can't convert complex to float
```

在 Python 3.9 中, `complex` 类型确实有一个 `__float__` 方法,但它的存在只是为了引发一个带有明确错误信息的 `TypeError`。如果 `__float__` 方法有注解,返回类型将会是 `NoReturn` (详见“8.5.12 `NoReturn`<240页>”)。

但是 `typeshed` 项目中 `complex.__float__` 的类型提示并不能解决这个问题,因为 Python 的运行时通常会忽略类型提示 (Type Hints),并且根本无法访问 `typeshed` 项目中的存根 (stub) 文件。

继续前面的 Python 3.9 控制台会话:

```
1  >>> from typing import SupportsFloat
2  >>> c = 3+4j
3  >>> isinstance(c, SupportsFloat)
4  True
5  >>> issubclass(complex, SupportsFloat)
6  True
```

因此,我们得到了误导性的结果:针对 `SupportsFloat` 的运行时检查表明,您可以将 `complex` 值转换为 `float` 值,但事实上,这种类型转换会引发类型错误。



Python 3.10.0b4 删除了 `complex.__float__` 方法, 解决了 `complex` 类型的这个问题。

但总体问题仍然存在: `isinstance / issubclass` 检查仅查看方法是否存在, 而不检查它们的签名, 更不会检查方法的类型注释。这种行为不会改变, 因为在运行时大规模类型检查所带来的性能损耗是无法接受的^a。

^a感谢 Ivan Levkivskyi (PEP 544 作者之一) 指出, 类型检查不是检查 `x` 的类型是不是 `T`, 而是判断 `x` 的类型是否与 `T` 相容 (Consistent-with)。检查相容 (Consistent-with) 的开销更大。难怪即便是很短的 Python 脚本, Mypy 也要用几秒才能完成类型检查。

现在, 让我们看看如何在用户定义的类中实现静态协议。

13.6.4 支持静态协议

回顾一下“十一 Pythonic 对象”种构建的 `Vector2d` 类 (见“[示例 11.11<307页>](#)”)。鉴于复数与 `Vector2d` 实例都由一对 `float` 值组成。因此, 支持从 `Vector2d` 到复数的转换是很有意义的。

[示例 13.15](#) 展示了 `__complex__` 方法的实现, 以增强上一个版本的 `Vector2d` 类 (见“[示例 11.11<307页>](#)”)。为了完整起见, 还定义了类方法 `fromcomplex`, 以执行逆运算, 即根据 `complex` 值构建 `Vector2d` 实例。

</> [示例 13.15: vector2d_v4.py: 与 complex 相互转换的方法](#)

```
1 def __complex__(self):
2     return complex(self.x, self.y)
3
4 @classmethod
5 def fromcomplex(cls, datum):
6     return cls(datum.real, datum.imag) ❶
```

❶ 假设 `datum` 具有 `.real` 和 `.imag` 属性。更好的实现方法, 见 [示例 13.16](#)。

根据上述代码以及“[示例 11.11<307页>](#)”中 `Vector2d` 已有的 `__abs__` 方法, 我们得到以下功能:

```
1 >>> from typing import SupportsComplex, SupportsAbs
2 >>> from vector2d_v4 import Vector2d
3 >>> v = Vector2d(3, 4)
4 >>> isinstance(v, SupportsComplex)
5 True
6 >>> isinstance(v, SupportsAbs)
7 True
8 >>> complex(v)
9 (3+4j)
10 >>> abs(v)
11 5.0
12 >>> Vector2d.fromcomplex(3+4j)
13 Vector2d(3.0, 4.0)
```

对于运行时类型检查, [示例 13.15](#) 可以胜任。但是为了让 Mypy 更好地做静态检查与错误报告, 应为 `__abs__`、`__complex__` 和 `fromcomplex` 方法提供 [类型提示 \(Type Hints\)](#), 如 [示例 13.16](#) 所示。

</> 示例 13.16: vector2d_v5.py: 为当前研究的方法添加注解

```

1 def __abs__(self) -> float: ①
2     return math.hypot(self.x, self.y)
3
4 def __complex__(self) -> complex: ②
5     return complex(self.x, self.y)
6
7 @classmethod
8 def fromcomplex(cls, datum: SupportsComplex) -> Vector2d: ③
9     c = complex(datum) ④
10    return cls(c.real, c.imag) ⑤

```

- ① 需要将返回值类型注解为 float, 否则 Mypy 推断出的类型是 Any, 而且不检查方法主体。
- ② 即使没有注解, Mypy 也能推断出该方法将返回一个 complex。根据 Mypy 的配置, 此注解可以防止出现警告。
- ③ SupportsComplex 确保 datum 可以转换成要求的类型。
- ④ ⑤ 这种显式转换是必要的, 因为 SupportsComplex 类型没有声明 ④ 处使用的 .real 和 .imag 属性。例如, Vector2d 没有这些属性, 但实现了 __complex__。

若在模块顶部出现 “from __future__ import annotations”, 那么 fromcomplex 的返回类型可以是 Vector2d。该导入会使类型提示以字符串形式存储, 而在导入时不会对函数定义进行求解。若没有 “from __future__ import annotations”, Vector2d 此时 (类尚未完全定义) 就是一个无效引用, 应写成字符串 ‘Vector2d’, 就像它是一个前向引用 (Forward Reference) 一样。这个 __future__ 导入是在 Python 3.7 实现的 “PEP 563 – Postponed Evaluation of Annotations” 中引入的。该行为原计划在 Python 3.10 中成为默认行为, 但这一更改被推迟到下一个版本了¹⁵。到那时, 这个导入语句就是多余的了, 但是也是无害的。

下一节, 让我们来介绍如何创建一个静态协议, 并在稍后对其进行扩展。

13.6.5 设计静态协议

在介绍 大鹅类型 (Goose Typing) 时, 我们定义了 抽象基类 (ABCs) Tombola (见 “示例 13.7<366页>”)。此处, 我们将看到如何使用静态协议定义与 Tombola 类似的接口。

抽象基类 (ABCs) Tombola 定义了 2 个抽象方法 (Abstract Method): pick 与 load。我们也可以使用这 2 个方法定义静态协议。但我从 Go 语言社区了解到, 单方法协议会使 静态鸭子类型 (Static Duck Typing) 更有用、更灵活。Go 标准库中有几个接口, 例如 Reader, 这是个只需要一个 read 方法的 I/O 接口。以后, 如果觉得需要一个更完整的协议, 则可以将多个协议组合起来, 定义一个新的协议。

可以随机从中挑选元素的容器, 不一定需要重新加载容器, 但肯定需要一个执行挑选操作的方法。因此, 我决定实现一个最精简的 RandomPicker 协议。该协议的代码如 示例 13.17 所示, 其用法如 示例 13.18 所示。

</> 示例 13.17: randompick.py: 协议 RandomPicker 的定义

```
from typing import Protocol, runtime_checkable, Any
```

¹⁵阅读 Python 指导委员会关于 python-dev 的决定

```
2  
3 @runtime_checkable class RandomPicker(Protocol):  
4     def pick(self) -> Any: ...
```



pick 方法返回 Any 类型。“15.8 实现泛化静态协议<444页>”将介绍如何让 RandomPicker 支持泛型 (Generic Type) 参数, 以允许协议的用户指定 pick 方法的返回类型。

</> 示例 13.18: randompick_test.py: 使用协议 RandomPicker

```
1 import random  
2 from typing import Any, Iterable, TYPE_CHECKING  
3  
4 from randompick import RandomPicker ❶  
5  
6 class SimplePicker:  
7     def __init__(self, items: Iterable) -> None:  
8         self._items = list(items)  
9         random.shuffle(self._items)  
10  
11     def pick(self) -> Any:  
12         return self._items.pop()  
13  
14     def test_isinstance() -> None:  
15         popper: RandomPicker = SimplePicker([1])❶❷  
16         assert isinstance(popper, RandomPicker) ❸  
17  
18     def test_item_type() -> None:  
19         items = [1, 2]  
20         popper = SimplePicker(items)  
21         item = popper.pick()  
22         assert item in items  
23         if TYPE_CHECKING:  
24             reveal_type(item) ❹  
25         assert isinstance(item, int)
```

- ❶ 定义一个实现静态协议的类, 无需先导入静态协议。这里导入 RandomPicker, 是因为❷处的 test_isinstance 会用到。
- ❷ SimplePicker 实现了 RandomPicker 协议, 但并没有子类化 RandomPicker。这就是 静态鸭子类型 (Static Duck Typing) 在起作用。
- ❸ Any 是默认的返回类型。因此, 严格来说, 这个注解不是必要的。但它能让我们更清楚地知道, 我们正在执行“示例 13.17<382页>”中定义的 RandomPicker 协议。
- ❹ 如果想让 Mypy 执行检查, 请不要忘记在添加类型提示 (Type Hints) “-> None”。
- ❺ 为 popper 变量添加了 类型提示 (Type Hints), 以表明 Mypy 能够理解 SimplePicker 与 RandomPicker 是 相容 (Consistent-with) 的。

- ⑥ 该测试证明 SimplePicker 的实例也是 Random Picker 的实例。这是因为 RandomPicker 使用了 @runtime_checkable 装饰器,而且 SimplePicker 按要求实现了所需的 **抽象方法 (Abstract Method)** pick。
- ⑦ 该测试调用 SimplePicker 的 pick 方法,验证它是否返回了一个提供给 SimplePicker 的元素;然后,对返回的元素进行静态检查和运行时检查。
- ⑧ 该行会在 Mypy 输出中生成一条注解。

如“[示例 8.22<235页>](#)”所述,reveal_type 是能被 Mypy 识别的“魔法”函数。该函数无需导入,并且只能在受 typing.TYPE_CHECKING 条件保护的 if 块中调用。typing.TYPE_CHECKING 条件只在静态类型检查器的眼中为 True,在运行时为 False。

[示例 13.18](#)中的 2 个测试均能通过,Mypy 也未检查到任何错误。对于 pick 方法返回的 item,reveal_type 输出的结果如下所示:

```
1 $ mypy randompick_test.py
2 randompick_test.py:24: note: Revealed type is 'Any'
```

创建了第一个协议后,接下来介绍关于设计协议时的一些建议。

13.6.6 静态协议设计最佳实践

10 年的 Go 语言[静态鸭子类型 \(Static Duck Typing\)](#)经验表明,窄协议 (narrow protocol) 通常更有用。窄协议通常只有一个方法,很少超过两个。Martin Fowler 写的一篇定义“[角色接口](#)”的文章,可用作设计协议时的参考。

此外,有时您可能会发现,协议在使用它的函数附近定义,即“在客户端代码”中定义而不是在库中定义。这样,可以方便创建新类型来调用该函数,有利于可扩展性与测试。

窄协议和客户端代码协议的做法,都避免了不必要的紧密耦合,符合[接口隔离原则](#)。我们可以将其概括为“客户端不应被迫依赖于他们不使用的接口”。

[typeshed 项目](#) 建议对静态协议采用如下 3 种命名约定(摘自 typeshed/CONTRIBUTING.md):

- 为表明确概念的协议使用简单的名称(如 Iterator、Container)。
- 对提供可调用方法的协议¹⁶,使用 SupportsX 进行命名(例如 SupportsInt、SupportsRead、SupportsReadSeek)。
- 对于具有可读和/或可写属性或 getter/setter 方法的协议,使用 HasX 进行命名(例如 HasItems、HasFileno)。

Go 标准库有一个我很喜欢的命名约定:对于单一方法协议,如果方法名称是动词,则附加“-er”或“-or”使其成为名词。例如,用 Reader 代替 SupportsRead。更多的例子包括 Formatter、Animator 和 Scanner。如需灵感,请参阅《[Go \(Golang\) Standard Library Interfaces \(Selected\)](#)》(Asuka Kenji 著)。

创建精简协议的好处是,如果需要的话,以后还可以对其进行扩展。下一节,您会发现扩展现有协议,额外添加方法并不难。

¹⁶每个方法都是可调用的,这条建议说得不太准确。或许改成“提供一种或两种方法”合适。无论如何,这只是一个指导方针,而不是严格的规定。

13.6.7 扩展一个协议

如“13.6.6 静态协议设计最佳实践”开头所述,Go 程序员在定义接口(他们对静态协议的称呼)时,都更倾向于极简主义。许多使用广泛的 Go 语言接口都只有一个方法。

如果实践中发现协议需要多个方法时,与其在原协议上添加方法,不如从中派生出一个新的协议。在 Python 中扩展静态协议有一些注意事项,如示例 13.19 所示。

</> 示例 13.19: randompickload.py: 扩展 RandomPicker 协议

```

1  from typing import Protocol, runtime_checkable
2  from randompick import RandomPicker
3
4  @runtime_checkable ❶
5  class LoadableRandomPicker(RandomPicker, Protocol): ❷
6      def load(self, Iterable) -> None: ... ❸

```

- ❶ 如果您希望派生出的协议,在运行时可检查,则必须再次应用 `typing.runtime_checkable` 装饰器,因为该装饰器的行为不会被继承¹⁷。
- ❷ 除了要扩展的协议之外,每个协议都必须明确地将 `typing.Protocol` 命名为它的基类之一¹⁸。这与 Python 2 种的继承方式不同。
- ❸ 回到“常规”的面向对象编程:只需要声明这个派生协议中的新方法即可, `pick` 方法的声明会继承自 `RandomPicker`。

本章对静态协议的定义和使用,就介绍到这里。

在结束本章之前,再回顾一下 `numeric` 抽象基类,以及可能取而代之的 `numeric` 协议。

13.6.8 number 模块中的抽象基类与 Numeric 协议

如“8.5.7.1 数字塔的崩塌<227页>”所述,标准库的 `numbers` 包 中的 抽象基类 (ABCs) 可以很好地用于运行时类型检查。

例如,检查 `x` 是不是整数,可以使用 `isinstance(x, numbers.Integral)`。`int`、`bool` (`int` 的子类),以及外部库提供的其他整数类型(需注册为 `numbers` 抽象基类的虚拟子类),都可通过这个测试。例如,NumPy 提供了 21 个整数类型,以及注册为 `numbers.Real` 的多种 `float` 类型和注册为 `numbers.Complex` 的不同位宽的复数。



令人惊讶的是, `decimal.Decimal` 并没有注册为 `numbers.Real` 的子类。原因是,如果你需要在程序中使用精度较高的 `Decimal`,那么通常希望避免将 `Decimal` 类型的对象与浮点数等精度较低的数字类型混合使用,以免发生意外的精度丢失。

遗憾的是,数字塔 (Numeric Tower) 并不是为静态类型检查而设计的。根 (Root) 抽象基类 `numbers.Number` 中未定义任何方法。因此,对于 `x: Number` 声明, Mypy 不会允许对 `x` 执行任何算术运算或

¹⁷ 有关细节和原理,请参阅“PEP 544 –Protocols: Structural subtyping (static duck typing)”中的“`@runtime_checkable` decorator and narrowing types by `isinstance()`”。

¹⁸ 有关详情及原理,请参阅“PEP 544 –Protocols: Structural subtyping (static duck typing)”中的“Merging and extending protocols”。

调用任何方法。

如果 `numbers` 抽象基类无法用于静态类型检查, 那还有其他选择么?

`typeshed` 项目是寻找 [类型提示 \(Type Hints\)](#) 解决方案的好去处。作为 Python 标准库的一部分, `statistics` 模块有一个相应的 [存根 \(stub\)](#) 文件, 即 `stdlib/statistics.pyi`。在该文件的 33 行处可找到如下的类型定义,许多函数的类型提示都用到了这 2 个类型。

```
_Number: TypeAlias = float | Decimal | Fraction
_NumberT = TypeVar("_NumberT", float, Decimal, Fraction)
```

这种做法不错,但是存在局限性,即它不支持标准库以外的数字类型,而 `numbers` 抽象基类支持这些外部数字类型的运行时检查,前提是这些数字类型被注册为 [虚拟子类 \(Virtual Subclass\)](#)。

目前的趋势是推荐使用 `typing` 模块提供的 `numeric` 协议（参见“13.6.2 运行时可检查的静态协议<377页>”）。

但是，在运行时，numbers 协议可能会让您失望。如“[13.6.3 运行时协议检查的局限性<380页>](#)”所述，Python 3.9 中的 complex 类型实现了 `_float_` 方法，但该方法的存在只是为了引发带有明确消息的 `TypeError`，警告“`can't convert complex to float.`”。出于同样的原因，complex 类型还实现了 `_int_` 方法。这些方法的存在使得 `isinstance` 在 Python 3.9 中会返回误导性的结果。Python 3.10 删除了 complex 类型中无条件引发 `TypeError` 的方法¹⁹（即 `_float_` 与 `_int_`）。

另外,NumPy 中复数 (complex) 类型实现的 `_float_` 和 `_int_` 方法要好一些,仅在首次使用时会发出警告:

```
1 >>> import numpy as np
2 >>> cd = np.cdouble(3+4j)
3 >>> cd
4 (3+4j)
5 >>> float(cd)
6 <stdin>:1: ComplexWarning: Casting complex values to real
7 discards the imaginary part
8 3.0
```

相反的问题也会发生：内置类型 `complex`、`float`、`int` 以及 `numpy.float16` 与 `numpy.uint8` 都未实现 `__complex__` 方法，因此 `isinstance(x, SupportsComplex)` 会返回 `False`²⁰。NumPy 中的复数类型（如 `np.complex64`）确实实现了 `complex` 方法，因此可转换为内置类型 `complex`。

然而,在实践中,内置构造函数 `complex()` 可正确处理所有这些类型的实例,并且不报错,也不会发出警告:

```
1 >>> import numpy as np
2 >>> from typing import SupportsComplex
3 >>> sample = [1+0j, np.complex64(1+0j), 1.0, np.float16(1.0), 1, np.uint8(1)]
4 >>> [isinstance(x, SupportsComplex) for x in sample]
5 [False, True, False, False, False, False]
6 >>> [complex(x) for x in sample]
7 [(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j)]
```

¹⁹详见 Issue #41974 Remove complex._float_, complex._floordiv_, etc。

²⁰我未测试 NumPy 提供的其他浮点数类型与整数类型。

由上述代码可知, 虽然 `isinstance` 针对 `SupportsComplex` 的检查, 表明某些实例在转换为 `complex` 类型时应该会失败。但是, 实际上这些实例全部都可以成功转换为 `complex` 类型。在 `typing-sig` 邮件列表中, Guido van Rossum 指出内置函数 `complex()` 接受单个参数, 这就是为什么这些转换能够成功的原因。

另外, 针对下面的 `to_complex()` 函数, 使用 Mypy 执行静态类型检查时, 实参 (Argument) 可接受所有这 6 种类型²¹:

在写作本书时, NumPy 包还未提供类型提示, 因此 NumPy 中的所有数字类型都是 `Any`²²。另一方面, 尽管在 `typeshed` 项目中只有内置的类型 `complex` 定义了 `__complex__` 方法²³, 但是 Mypy 在某种程度上意识到内置类型 `int` 与 `float` 也都可以转换为 `complex` 类型。

综上所述, 虽然对数字类型执行类型检查应该不难, 但现状是 “[PEP 484 - Type Hints](#)” 有意避开 [数字塔 \(Numeric Tower\)](#), 并隐式建议类型检查工具对内置类型 `complex`、`float`、`int` 之间的子类型关系进行硬编码。Mypy 也确实是这么做的, 并且还从实用角度出发, 认定 `int`、`float` 与 `SupportsComplex` 是相容 (Consistent-with) 的, 尽管 `int` 与 `float` 都未实现 `__complex__` 方法。



我只是在使用数字协议 `SupportsComplex` 进行 `isinstance` 检查时, 发现了这些意想不到的结果。如果您不使用复数 (`complex`), 则可以依赖这些数字协议, 而无需使用 `numbers` 包中的抽象基类。

本节的主要结论如下:

- `numbers` 包中的抽象基类可以用于运行时类型检查, 但不适合静态类型检查。
- 数字静态协议 `SupportsComplex`、`SupportsFloat` 等适用于静态类型检查。但在涉及复数 (`complex`) 时, 对于运行时类型检查并不可靠。

现在, 我们快速回顾一下本章中所讲的所有内容。

13.7 本章小结

类型图 (见 “[图 13.1<350页>](#)”) 是理解本章内容的关键。在简要介绍了 4 种类型实现方式之后, 我们对比了动态协议 (支持鸭子类型) 与静态协议 (支持静态鸭子类型)。这两种协议有一个共同的基本特征: 类不需要显式声明对任何特定协议的支持。一个类若想支持某个协议, 只需要实现该协议要求的方法即可。

下一个主要章节是 “[13.4 用鸭子类型编程<352页>](#)”。这一节深入探讨了 Python 解释器实现序列与可迭代动态协议的方式, 包括对二者的部分实现。然后, 说明了如何通过 [猴子补丁 \(Monkey-patch\)](#) 为类添加额外的方法, 使一个类在运行时实现特定的协议。之后, 又介绍了防御型编程, 包括无需使用 `isinstance()` 或在 `try/except` 结构中使用 `hasattr()` 检查结构类型, 以及 [Fail-Fast \(快速失败\)](#) 原则。

Alex Martelli 在 “[13.5 大鹅类型 \(Goose Typing\) <358页>](#)” 的附注栏 “[水禽与抽象基类](#)” 中介绍 [大鹅类型 \(Goose Typing\)](#) 之后, 我们说明了如何子类化现有的 [抽象基类 \(ABCs\)](#), 调查了标准库中重要的 [抽象基类 \(ABCs\)](#), 并从头开始创建了一个 [抽象基类 \(ABCs\)](#); 然后, 通过传统的子类化和注册机制, 为抽象基类提供了具体实现。最后, 了解了如何通过特殊的类方法 `__subclashook__` 来让 [抽象基类 \(ABCs\)](#) 支持 [结构类型 \(Structural Typing\)](#)。

²¹ 此处的 6 种类型指前文所述的: 内置类型 `complex`、`float`、`int`, 以及 `numpy.float16`、`numpy.uint8` 与 `np.complex64`。

²²

²³ 这是 `typeshed` 善意的谎言: 截至 Python 3.9, 内置类型 `complex` 实际上并没有 `__complex__` 方法。

“13.6 静态协议<376页>”一节也很重要,这一节接着“8.5.10 静态协议<233页>”继续探讨了静态鸭子类型(Static Duck Typing)。我们了解到`@runtime_checkable`类装饰器也是利用`__subclasshook__`在运行时支持结构类型。不过,静态协议最好与静态类型检查工具结合使用,连同类型提示(Type Hints),可实现更可靠的结构类型(Structural Typing)。接下来,讨论了静态协议的设计与实现,以及如何扩展。最后的“13.6.8 number 模块中的抽象基类与 Numeric 协议<385页>”讲述了数字塔(Numeric Tower)被荒废的悲伤故事,还提出了现存替代方案的一些缺陷,包括`SupportsFloat`等数字静态协议,以及Python 3.8在typing模块中增加的数字静态协议。

本章的主旨是,在现代Python中,我们有4种互补的类型实现方式,每种方式都各有优缺点。对于现代的Python代码库,只要规模够大,每种方式都有用武之地。作为一名Python程序员,抛弃其中任何一种类型实现方式,您的日子都不会好过。

话虽如此,最初Python只支持鸭子类型(Duck Typing)时,就已是大受欢迎了。其他的流行语言,如JavaScript、PHP、Ruby,以及Lisp、Smalltalk等,都从鸭子类型(Duck Typing)的强大和简单中受益匪浅。

13.8 延伸阅读

若想快速了解类型的优缺点,以及`typing.Protocol`对代码库健康情况静态检查的重要性,强烈推荐阅读Glyph Lefkowitz写的文章“*I Want A New Duck: typing.Protocol and the future of duck typing*”。我还从他的“*Interfaces and Protocols: Comparing zope.interface and typing.Protocol*。”一文学到了很多东西²⁴。`zope.interface`是早期为松耦合的插件系统定义接口的一种机制,Plone CMS、Pyramid Web框架、Twisted异步编程框架(由Glyph创建)都采用了这种机制。

优秀的Python书籍几乎按照定义,都对鸭子类型(Duck Typing)进行了大量介绍。我最喜欢的两本Python书籍:《*The Quick Python Book, 3rd Ed*》(Naomi Ceder著)与《*Python in a Nutshell, 3rd Ed*》(Alex Martelli、Anna Ravenscroft、Steve Holden著),在本书第1版发布后,都发布了更新。

有关动态类型的利弊讨论,请参阅Guido van Rossum对Bill Venners的访谈,访谈内容记录在“*Contracts in Python: A Conversation with Guido van Rossum, Part IV*”。Martin Fowler在“*Dynamic Typing*”文章中,对该访谈进行了全面而深入的分析。Martin还写了文章“*Role Interface*”(“13.6.6 静态协议设计最佳实践<384页>”提到过),该文章虽然与鸭子类型(Duck Typing)无关,但是与Python协议设计密切相关,它对窄角色接口与类的公开接口进行了对比。

`Mypy`文档通常是Python中与静态类型(包括静态鸭子类型)相关信息的最佳资源,详见“*Protocols and structural subtyping*”。

余下的资料都是关于大鹅类型(Goose Typing)的。《*Python Cookbook, 3rd Ed*》(Beazley、Jones著)有一节是关于定义抽象基类(ABCs)的(Recipe 8.12)。这本书是在Python 3.4之前编写的,因此未使用现在首选的语法,即通过子类化`abc.ABC`来声明抽象基类(ABCs),而是使用了`metaclass`关键字。我们只在“*二十四类元编程(Class Metaprogramming)*”才真正需要用到`metaclass`关键字。除了这个细节之外,该书很好地涵盖了抽象基类(ABCs)的主要功能。

《*The Python 3 Standard Library by Example*》(Doug Hellmann著)中有一章是关于`abc`模块的,并且在Doug的Python Module of the Week(PyMOTW)网站²⁵中在线阅读,链接为“<https://pymotw.com/3/abc/>

²⁴感谢本书技术审校Jürgen Gmach推荐了“*Interfaces and Protocols: Comparing zope.interface and typing.Protocol*。”文章。

²⁵Python Module of the Week(PyMOTW)网站:<https://pymotw.com/3/>。

`index.html`”。Hellmann 也是使用了旧式语法声明 **抽象基类 (ABCs)**，即 `PluginBase(metaclass=abc.ABCMeta)`。从 Python 3.4 开始，可以简写成 `PluginBase(abc.ABC)`。

在使用 **抽象基类 (ABCs)** 时，多重继承是不可避免的，而且会经常用到。因为每个基本 (fundamental) 的容器类型的抽象基类（如 `Sequence`、`Mapping`、`Set`）都扩展自 `Collection` 类，而 `Collection` 类又扩展自多个 **抽象基类 (ABCs)**（如“图 13.4<364页>”所示）。“**十四 继承的利与弊**”将深入探讨这个重要话题。

“[PEP 3119 –Introducing Abstract Base Classes](#)”介绍了 **抽象基类 (ABCs)** 的基本原理。“[PEP 3141 –A Type Hierarchy for Numbers](#)”介绍了 `numbers` 模块中的 **抽象基类 (ABCs)**。Mypy “[Issue #3186 int is not a Number?](#)”展开了一场论战，探讨了 `numbers` 模块中的 **抽象基类 (ABCs)** 为何不适合在静态类型检查中使用。Alex Waygood 在 [StackOverflow](#) 中详述了注解数字类型的各种方式²⁶。我将继续关注 Mypy [Issue #3186](#) 的进展，期待会有一个圆满的结局，可以让静态类型与 **大鹅类型 (Goose Typing)** 完美兼容。

杂谈

Python 静态类型的 MVP 之旅

我供职于 Thoughtworks 公司，该公司是敏捷软件开发领域的全球领导者。在 Thoughtworks，我们经常建议客户应以创建和部署 MVP (Minimal Viable Products, 最简可用产品) 为目标。我同事 Paulo Caroli 发表在 [martinfowler.com](#) 的文章 “[Lean Inception](#)” 中，对 MVP 的定义是：“为验证关键业务假设，而提供给用户的简单版本的产品。”

自 2006 年以来，Guido van Rossum 与其他核心开发人员在设计和实现静态类型时，一直遵循 MVP 策略。首先，Python 3.0 实现的 “[PEP 3107 –Function Annotations](#)” 提供了非常有限的语法，只包含了为函数 **形参 (Parameter)** 与返回值添加注解的语法。这么做的目的是为了进行实验和收集反馈——这是 MVP 的主要优势。

8 年后，“[PEP 484 –Type Hints](#)”被提出并获得了批准。它在 Python 3.5 中的实现不需要对语言或标准库进行任何更改，只是增加了不被标准库其他部分依赖的 `typing` 模块。[PEP 484](#) 仅支持具有泛型的 **名义类型 (Nominal Typing)**（类似于 Java），但实际的静态类型检查由外部工具完成。重要的功能（如变量注解、内置泛型、协议）有所缺失。尽管存在这些限制，但这个最简可用的类型系统仍然具有足够的价值，吸引了拥有庞大 Python 代码库的公司（如 Dropbox、Google、Facebook）的投入与使用，以及专业 IDE（如 PyCharm、Wing、VS Code）的支持。

“[PEP 526 –Syntax for Variable Annotations](#)”是 Python 3.6 中需要对解释器进行演进更改的第一步。为了支持 “[PEP 563 –Postponed Evaluation of Annotations](#)” 与 “[PEP 560 –Core support for typing module and generic types](#)”，Python 3.7 对解释器做了更多改动。得益于 “[PEP 585 –Type Hinting Generics In Standard Collections](#)” 的实现，Python 3.9 中内置的容器类型与标准库中的容器类型都可接受开箱即用的 **泛型 (Generic Type)** 类型提示。

在那些年里，包括我在内的一些 Python 用户，并未对静态类型提起兴趣。在学习 Go 语言之后，我意识到 Python 中缺少 **静态鸭子类型 (Static Duck Typing)** 是令人难以理解的，毕竟 **鸭子类型 (Duck Typing)** 是 Python 语言的核心优势。

但是，这正是 MVP（最简可用产品）的本质所在：它们可能无法满足所有潜在用户的需求，但却可以事半功倍，并通过现场实际使用的反馈来指导进一步的开发。

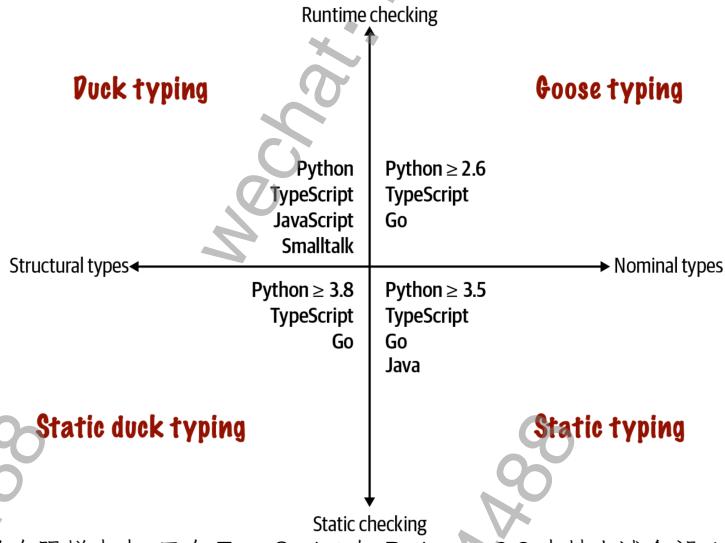
如果说我们从 Python 3 中学到了什么，那肯定 是循序渐进地开发比大刀阔斧地发布新功能要更安

²⁶ 问题标题为 “[How to hint at number *types* \(i.e. subclasses of Number\) - not numbers themselves?](#)”。

全。我很高兴不用等到 Python4(如果它真地到来),即可引起大公司对 Python 的注意。因为大公司发现,与静态类型带来的好处相比,增加的那点复杂性根本不值一提。

流行语言的类型实现方式

下图是 [类型图](#)(见“图 13.1<350页>”)的变体,其中标记了支持每种类型实现方式的几种流行语言。



在我随机考察的有限样本中,只有 TypeScript 与 Python ≥ 3.8 支持上述全部 4 种类型实现方式。

Go 显然是一种具有 Pascal 传统的静态类型语言,但它开创了[静态鸭子类型\(Static Duck Typing\)](#)的先河——至少在当今广泛使用的编程语言中是如此。我之所以将 Go 语言放在[大鹅类型\(Goose Typing\)](#)象限中,是因为它的类型断言允许在运行时检查,并适应不同的类型。

如果让我在 2000 年画一张类似的图,那么只有[鸭子类型\(Duck Typing\)](#)与[静态类型\(Static Typing\)](#)象限中会有编程语言。据我所知就,20 年前还没有支持[静态鸭子类型\(Static Duck Typing\)](#)与[大鹅类型\(Goose Typing\)](#)的编程语言。如上图所示,4 个象限中都至少包含 3 门流行的编程语言,这表明很多人认为这 4 种类型实现方式都有各自的价值。

猴子补丁 (Monkey-patch)

[猴子补丁\(Monkey-patch\)](#)的名声不太好。如果滥用猴子补丁,会导致系统难以理解和难以维护。补丁通常与被修补的目标紧密耦合,因此一旦目标代码发生了变化,补丁可能会失效。另一个问题是,如果两个库都应用了猴子补丁,二者可能会相互干扰,导致冲突和意外行为。例如,第二个库可能会覆盖第一个库的补丁,破坏第一个库所做的修改,从而引入新的问题。

不过猴子补丁也有它的作用。例如,可以使类在运行时实现某个协议。适配器(Adapter)设计模式通过实现一个全新的类,也可以解决运行时实现某个协议的问题。

为 Python 代码打猴子补丁很容易,但也有一些限制。与 Ruby 和 JavaScript 不同,Python 不允许对内置类型打补丁。实际上,我认为这是一个优点。因为可以确保 str 对象始终都具有相同的方法。这一限制减少了外部库使用冲突补丁的可能性。

接口中的隐喻与习惯用法

隐喻通过明确限制与功能特性,可让人更易于理解。这也是“堆栈”与“队列”这两个词在描述这些基本数据结构时的功效:这两个词明确了这些数据结构将允许哪些操作,即明确了添加或删除元素的方式。另外,Alan Cooper 等人在《About Face4: 交互设计精髓》^a一书中写道:

严格遵守隐喻设计毫无必要, 它会将用户界面与物理世界的运作机制死死地捆绑在一起。

Cooper 说的是用户界面, 但对 API 同样适用。不过, 他也认同当“真正合适的”隐喻“正中下怀”^b时, 我们也可以使用隐喻。我觉得本章示例(见“[示例 13.9<369页>](#)”)使用“宾果机(BingoCage)”做比喻很合适, 我相信自己。

到目前为止,《About Face4》是我读过的关于用户界面设计的最好的一本书。我从 Cooper 的作品中学到的最有价值的东西就是放弃隐喻这种设计范式, 取而代之的是“惯用的界面”。

在《About Face4》中, Cooper 并未提及 API。但是, 在深入思考他的观点之后, 我觉得可以将他的观点运用到 Python 中。Python 中的基本协议就是 Cooper 所说的“惯用方法”。比如, 一旦我们了解了“序列”是什么之后, 就可以将序列相关的知识应用到不同的场合。这正是本书的主要目的:着重讲解 Python 语言的基本惯用法, 让您的代码简洁、高效且易读, 将您打造成能流畅写出 Python 代码的程序员。

^a英文原版书名为:《About Face: The Essentials of Interaction Design, 4th Ed》。

^b他用的此是“正中下怀”, 因为合适的隐喻可遇而不可求。

wechat: 119554488

继承的利与弊

当前,继承的概念还不够清晰,需要为其提出一套更好的理论(现在也需要)。例如,继承与实例化混淆了实用层面(如优化代码以节省空间)与语法(如特化、泛化等)。

——Alan Kay, “The Early History of Smalltalk”^a

^aAlan Kay, “The Early History of Smalltalk”, 也可以在线阅读。感谢我的朋友 Christiano Anderson 在我撰写本章时,向我分享了这篇参考文献。

本章将探讨继承与子类化。阅读本章之前,需要对这些概念有基本的了解。可以通过阅读《[The Python Tutorial](#)》,或使用其他面向对象语言(如 Java、C# 或 C++)来了解继承与子类化的概念。本章我们重点关注 Python 的 4 个特性。

- `super()` 函数。
- 从内置类型进行子类化的陷阱。
- 多重继承与方法解析顺序。
- 混合 (Mixin) 类。

多重继承是指一个类拥有多个基类的能力。C++ 支持多重继承,而 Java 与 C# 不支持。许多人认为多重继承会得不偿失。早期的 C++ 代码库滥用了多重继承,因此 Java 排除了对多重继承的支持。

本章将为那些从未使用过多重继承的读者介绍多重继承,并为在使用继承时,如何处理单继承或多继承提供了一些指导。

截至 2021 年,人们普遍反对过度使用继承(不仅仅是多重继承),因为超类与子类是紧密耦合的。紧密耦合意味着对程序某一部分的修改,可能会导致牵一发而动全身,从而使系统变得脆弱且难以理解。

但是,有时仍须维护使用了复杂的类层次结构设计的存量系统,何况某些框架可能会强制必须使用继承(甚至是多重继承)。

本章将通过[Python 标准库](#)、[Django Web 框架](#)和 [Tkinter GUI 工具包](#)说明多重继承的实际应用。

14.1 本章新增内容

Python 没有为本章探讨的话题增加新的特性,但我根据本书技术审校(尤其是 Leonardo Rochael 与 Caleb Hattingh)的反馈,对本章做了大幅改动。

我重写了 [节 14.2](#),重点介绍了内置函数 `super()`。另外,还修改了 [节 14.4](#) 中的示例,深入探讨 `super()` 函数是如何支持 [协作多重继承](#)的。

[“14.5 混合类”](#) 也是新增的章节。[“14.6 多重继承的实际运用”](#) 重新做了内容编排,在介绍 Django 和 Tkinter 的复杂层次结构之前,先介绍了标准库中较简单的混合类示例。

正如本章标题所示,继承的注意事项一直是本章的主题之一。但是,越来越多的开发人员认为继承是一个非常棘手的问题。因此,本章在[“14.8 本章小结<413页>”](#) 与 [“14.9 延伸阅读<414页>”](#) 末尾增加了几段关于避免使用继承的内容。

接下来,先从神秘的 `super()` 函数开始。

14.2 `super()` 函数

坚持使用 `super()` 内置函数,对于面向对象 Python 程序的可维护性至关重要。

子类中重写超类的方法时,通常需要调用超类中相应的方法。[collections 模块文档](#) 的“[OrderedDict Examples and Recipes](#)”一节,有一个示例,演示了推荐的方式,代码如下所示¹:

```

1 class LastUpdatedOrderedDict(OrderedDict):
2     """按更新顺序,存储有序字典中的项"""
3
4     def __setitem__(self, key, value):
5         super().__setitem__(key, value)
6         self.move_to_end(key)

```

为了完成工作, `LastUpdatedOrderedDict` 重写了超类的 `__setitem__` 方法:

1. 使用 `super().__setitem__` 在超类上调用该方法,以使其插入或更新“键/值”对。
2. 调用 `self.move_to_end`,以确保更新的“键”位于最后一个位置。

调用被重写的 `__init__` 方法尤其重要,可以让超类完成它应负责的初始化任务。



如果学习过 Java 面向对象编程,您应该还记得,Java 构造方法会自动调用超类的无参数构造方法,而 Python 则不会这么做。因此,应像下面这样实现 `__init__` 方法。

```

1 def __init__(self, a, b):
2     super().__init__(a, b)
3     ... # 其他初始化代码

```

您或许见过不使用 `super()`,而是直接在超类上调用超类方法的代码,如下所示:

¹我只修改了示例中的文档字符串,因为原来的文档字符串容易令人误解。原来的内容是“Store items in the order the keys were last added”,这显然与类名表达的意思不符。

```

1 class NotRecommended(OrderedDict):
2     """这是一个反例"""
3
4     def __setitem__(self, key, value):
5         OrderedDict.__setitem__(self, key, value)
6         self.move_to_end(key)

```

这么做也可以,但并不是推荐的做法,原因有二:其一,它对基类进行了硬编码。OrderedDict 名称不仅出现在 class 语句中,还出现在 __setitem__ 方法内。若后续有人修改了 class 语句,更换了基类或又添加了其他基类。那么,说不定可能会忘记修改 __setitem__ 方法的主体,从而埋下 bug。

其二,super() 函数 实现的逻辑,可以处理多重继承涉及的类层次结构(详见“节 14.4<398页>”)。最后,回顾一下 Python 2 中调用super() 函数 的旧语法,该语法可接受 2 个 实参(Argument),这可以给我们一定启发。

```

1 class LastUpdatedOrderedDict(OrderedDict):
2     """Python 2 与 Python 3 中都可正常运行"""
3
4     def __setitem__(self, key, value):
5         super(LastUpdatedOrderedDict, self).__setitem__(key, value)
6         self.move_to_end(key)

```

现在,super(type, object_or_type=None) 函数 的 2 个 实参(Argument) 都是可选的。当在方法中调用 super(type, object_or_type=None) 函数 时,Python 3 字节码编译器通过检查 super() 周围的上下文,来自动提供如下 2 个参数:

- type

实现所需方法的超类的搜索路径起点。默认情况下,它是拥有 super() 调用的方法的类。

- object_or_type

要成为方法调用的接收者的对象(对于实例方法调用)或类(对于类方法调用),作为方法调用接收者的对象(例如方法调用)或类(对于类方法调用)。如果在实例方法中发生 super() 调用,则默认为 self。

无论是我们自己还是编译器提供这 2 个参数,super(type, object_or_type=None) 调用都会返回一个动态代理对象,该对象会在参数 type(即超类)中找到一个方法(如示例中的 __setitem__ 方法),并将方法绑定到 object_or_type 上。这样,我们在调用那个方法(如 __setitem__)时就无需显式传入接收者(self)了。

在 Python 3 中,仍然可以显式为 super() 提供第一个参数与第二个参数²。但是,只有在特殊情况下才必须这么做。例如,在测试或调试时跳过部分 MRO³,或者绕开不希望从超类得到的行为。

现在,让我们讨论对内置类型进行子类化时的注意事项。

²也可以只提供第一个参数,但这并不实用,而且可能很快就会被废弃,这也得到了最初创建 super() 的 Guido van Rossum 的支持。请参阅“Is it time to deprecate unbound super methods?”

³MRO 指的是方法解析顺序(Method Resolution Order),它是用于确定在多重继承情况下,Python 中类的方法调用顺序的规则。

14.3 子类化内置类型很棘手

在 Python 的早期版本中,无法子类化内置类型(如 list 或 dict)。自 Python 2.2 以来,子类化内置类型成为可能。但有一个重要的注意事项:内建类型(用 C 编写)的代码通常不会调用由用户自定义类重写的方法。PyPy 文档“[Differences between PyPy and CPython](#)”一节的“[Subclasses of built-in types](#)”中对这个问题有一个很好的简短描述:

CPython 并没有制定相关规则,指明内置类型子类的重写方法何时会被隐式调用。近似地讲,子类重写的方法永远不会被同一对象的其他内置方法调用。例如, dict 子类中重写的 `__getitem__()` 方法不会被内置 `get()` 方法调用。

示例 14.1 展示了这个问题。

```
</> 示例 14.1: 重写的 __setitem__ 方法,被 dict 的 __init__ 与 __update__ 忽略

1  >>> class DoppelDict(dict):
2      ...     def __setitem__(self, key, value):
3      ...         super().__setitem__(key, [value] * 2) ❶
4      ...
5  >>> dd = DoppelDict(one=1) ❷
6  >>> dd
7  {'one': 1}
8  >>> dd['two'] = 2 ❸
9  >>> dd
10 {'one': 1, 'two': [2, 2]}
11 >>> dd.update(three=3) ❹
12 >>> dd
13 {'three': 3, 'one': 1, 'two': [2, 2]}
```

- ❶ `DoppelDict.__setitem__` 方法将存入的值重复 2 次(仅为了提供易于观察的效果)。`super()` 函数将相关操作委托给了超类。
- ❷ 从 dict 继承的 `__init__` 方法显然忽略了子类中重写的 `__setitem__` 方法。因为,“one”对应的值没有被重复。
- ❸ 运算符 `[]` 调用了子类重写的 `__setitem__` 方法,并可按预期工作。因为,“two”对应的是重复的值,即 `[2,2]`。
- ❹ dict 的内置方法 `update`,也忽略了子类中重写的 `__setitem__` 方法。因为,“three”对应的值没有被重复。

这种内置行为违反了面向对象编程的基本规则:即使调用发生在超类实现的方法中,方法的搜索也应始终以接收者(`self`)的类为起点。这就是所谓的“[后期绑定](#)”,Alan Kay(因 Smalltalk 而出名)认为这是面向对象编程的一个关键特征:在 `x.method()` 形式的调用中,具体要调用的方法必须在运行时根据接收者 `x` 所属的类来确定⁴。这种糟糕的局面,导致了在“[3.5.3 标准库中 `__missing__` 的不一致用法](#)”中看到的问题。

不只实例内部的调用有问题(即 `self.get()` 不会调用 `self.__getitem__()`),内置类型的方法调用其他类

⁴有趣的是,C++ 有虚拟方法与非虚拟方法的概念。虚拟方法是后期绑定的,而非虚拟方法是在编译时绑定的。虽然我们在 Python 中编写的每个方法都像虚拟方法一样,是后期绑定的。但是用 C 语言编写的内置对象似乎默认具有非虚拟方法,至少在 CPython 中是这样的。

的重写方法时,也存在这个问题。示例 14.2 是改变自 PyPy 文档的一个示例。

```
</> 示例 14.2: AnswerDict 的 __getitem__ 被 dict.update 忽略
1  >>> class AnswerDict(dict):
2      ...     def __getitem__(self, key): ❶
3      ...         return 42
4      ...
5  >>> ad = AnswerDict(a='foo')
6  >>> ad['a'] ❷
7  42
8  >>> d = {}
9  >>> d.update(ad) ❸
10 >>> d['a'] ❹
11 'foo'
12 >>> d
13 {'a': 'foo'}
```

- ❶ 无论传入什么键,AnswerDict.__getitem__ 总是返回 42。
- ❷ ad 是一个加载了键值对 ('a', 'foo') 的 AnswerDict。
- ❸ ad['a'] 返回 42,符合预期。
- ❹ d 是普通 dict 的一个实例,我们用 ad 对其进行更新。
- ❺ dict.update 方法忽略了 AnswerDict.__getitem__ 方法。



直接子类化 dict、list 或 str 等内置类型容易出错,因为内置方法大多会忽略用户定义的重写。与其子类化内置类型,不如使用 UserDict、UserList 和 UserString 从 collections 模块 中派生类。这些类更易于扩展。

如果不使用 dict,而是子类化 collections.UserDict,则 [示例 14.1](#)与[示例 14.2](#) 中暴露的问题都能迎刃而解,如[示例 14.3](#)所示。

```
</> 示例 14.3: DoppelDict2 与 AnswerDict2 可按预期运行,因为它继承自 UserDict
```

```
1  >>> import collections
2
3  >>> class DoppelDict2(collections.UserDict):
4      ...     def __setitem__(self, key, value):
5      ...         super().__setitem__(key, [value] * 2)
6
7  >>> dd = DoppelDict2(one=1)
8
9  >>> dd
10 >>> dd['two'] = 2
11 >>> dd
12 {'two': [2, 2], 'one': [1, 1]}
13 >>> dd.update(three=3)
14 >>> dd
15 {'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
```

```

16  >>>
17  >>> class AnswerDict2(collections.UserDict):
18  ...     def __getitem__(self, key):
19  ...
20  ...
21  >>> ad = AnswerDict2(a='foo')
22  >>> ad['a']
23  42
24  ...
25  >>> d = {}
26  >>> d.update(ad)
27  >>> d['a']
28  42
29  >>> d
30  {'a': 42}

```

为了衡量子类化内置类型所需的额外工作量,我做了一个实验,将“[小节 3.6.5<81页>](#)”中原本继承 `collections.UserDict` 的 `StrKeyDict` 类,改为继承内置类型 `dict`。为了使新实现可通过相同的测试套件,我必须实现 `__init__`、`get` 和 `update` 方法。因为从 `dict` 继承的版本拒绝与重写的 `__missing__`、`__contains__` 与 `__setitem__` 方法配合。“[小节 3.6.5<81页>](#)”中那个 `UserDict` 子类有 16 行代码,而这个新实现(从 `dict` 继承)有 33 行代码。⁵

需要明确的是:本节涉及的问题仅适用于内置类型(C 语言实现的)的方法委托上,并且只影响直接从这些类型派生的类。如果子类化一个用 Python 编写的基类(如 `UserDict` 或 `MutableMapping`),则不会受到此问题的困扰。⁶

现在,让我们来关注与多重继承有关的一个问题:如果一个类有 2 个超类,但是这 2 个超类都有一个同名的属性 `attr`,那么 Python 该如何确定 `super().attr` 该使用哪个属性呢?下一节,将对此做出解答。

14.4 多重继承与方法解析顺序

任何实现多重继承的语言都需要处理超类实现同名方法时,可能出现的命名冲突。我们称之为“菱形问题”,如 [图 14.1](#)与[示例 14.4](#)所示。

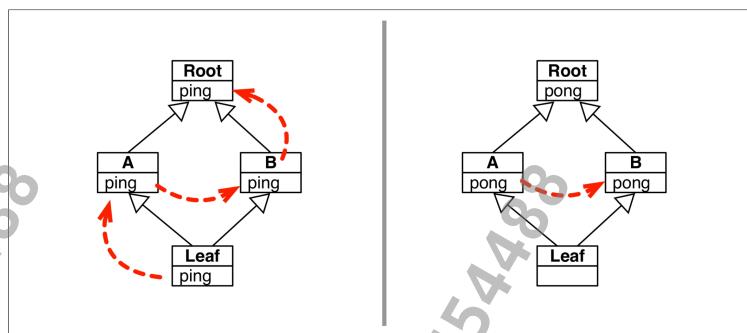


图 14.1: 左图:调用 `leaf1.ping()` 的唤醒序列;右图:调用 `leaf1.pong()` 的唤醒序列。

</> [示例 14.4](#): `diamond.py`:[图 14.1](#)中的 `Leaf` 类、`A` 类、`B` 类、`Root` 类

⁵若您感到好奇,可查看[随书代码库](#)中的 `14-inheritance/strkeydict_dictsub.py` 文件。

⁶顺便说一下,在这方面,PyPy 的行为比 CPython 更“正确”,但是引入了一些轻微的不兼容性。详见“[Differences between PyPy and CPython](#)”。

```
1  class Root:          ❶
2      def ping(self):
3          print(f'{self}.ping() in Root')
4
5      def pong(self):
6          print(f'{self}.pong() in Root')
7
8      def __repr__(self):
9          cls_name = type(self).__name__
10         return f'<instance of {cls_name}>'
11
12 class A(Root):        ❷
13     def ping(self):
14         print(f'{self}.ping() in A')
15         super().ping()
16
17     def pong(self):
18         print(f'{self}.pong() in A')
19         super().pong()
20
21 class B(Root):        ❸
22     def ping(self):
23         print(f'{self}.ping() in B')
24         super().ping()
25
26     def pong(self):
27         print(f'{self}.pong() in B')
28
29 class Leaf(A, B):    ❹
30     def ping(self):
31         print(f'{self}.ping() in Leaf')
32         super().ping()
```

❶ Root类不仅提供了ping、pong方法。同时,为了输出更易读的表示形式,还实现了__repr__方法。

❷ A类中的ping和pong方法都调用了super()。

❸ B类中只有ping方法调用了super()。

❹ Leaf类仅实现了ping方法,而且该方法调用了super()。

现在,让我们看看在Leaf实例上调用ping与pong方法的效果(如示例14.5所示)。

</>示例14.5: 在Leaf对象上调用ping与pong方法

```
1  >>> leaf1 = Leaf() ❶
2  >>> leaf1.ping() ❷
3  <instance of Leaf>.ping() in Leaf
4  <instance of Leaf>.ping() in A
5  <instance of Leaf>.ping() in B
6  <instance of Leaf>.ping() in Root
7  >>> leaf1.pong() ❸
```

```

8 <instance of Leaf>.pong() in A
9 <instance of Leaf>.pong() in B

```

- ① leaf1 是 Leaf 的一个实例。
- ② 调用 leaf1.ping() 会唤醒 Leaf、A、B 和 Root 中的 ping 方法, 因为前 3 个类中的 ping 方法都调用了 super().ping()。
- ③ 调用 leaf1.pong() 会通过继承图 (inheritance graph), 唤醒 A 中的 pong 方法; 然后, A.pong 通过调用 super.pong(), 唤醒了 B.pong 方法。

示例 14.5 与 图 14.1 所示的唤醒序列由如下 2 个因素决定:

- Leaf 类的方法解析顺序。
- 各方法中的 super() 调用。

每个类都有一个名为 `__mro__` 的属性, 它按MRO (方法解析顺序) 保存着一个指向超类的引用元组, 从当前类一直到 `object` 类 (自底向上)。⁷对于 Leaf 类, `__mro__` 的内容如下所示:

```

1 >>> Leaf.__mro__ # doctest:+NORMALIZE_WHITESPACE
2 (<class 'diamond1.Leaf'>, <class 'diamond1.A'>, <class 'diamond1.B'>,
3 <class 'diamond1.Root'>, <class 'object'>)

```



“图 14.1<398页>”可能会让您误以为 MRO (方法解析顺序) 使用 广度优先搜索 (Breadth-first search) 算法, 但这只是一种巧合。MRO 使用的是 C3 算法, Michele Simionato 的 “The Python 2.3 Method Resolution Order” 对 C3 算法在 Python 中的运用做了详细说明。这是一篇具有挑战性的读物, 但 Simionato 写道:“除非大量使用多重继承, 或继承关系较复杂。否则, 无需了解 C3 算法, 可以直接跳过这篇论文。”

MRO (方法解析顺序) 只决定唤醒顺序, 至于各个类中的特定方法是否会被唤醒, 取决于方法中是否调用了 `super()`。

以 “示例 14.5<399页>” 中的 pong 方法为例。`Leaf` 类没有重写 pong 方法, 因此调用 `leaf1.pong()` 会唤醒 `Leaf.__mro__` 的下一个类——`A` 类中的实现。方法 `A.pong` 调用 `super().pong()`。`B` 类是 MRO 中的下一个类, 因此 `B.pong` 被唤醒。但 `B.pong` 方法中未调用 `super().pong()`, 因此唤醒序列到此结束。

MRO (方法解析顺序) 不仅考虑了继承图 (inheritance graph), 还考虑了子类声明中罗列超类的顺序。换句话说, 如果在 `diamond.py` (“示例 14.4<398页>”) 中, `Leaf` 类被声明为 `Leaf(B,A)`。那么, 类 `B` 将出现在 `Leaf.__mro__` 中的 `A` 之前。这会影响 ping 方法的唤醒顺序, 也会导致 `leaf1.pong()` 通过继承图唤醒 `B.pong`。但 `A.pong` 和 `Root.pong` 永远不会运行, 因为 `B.pong` 中没有调用 `super()`。

当一个方法调用了 `super()` 时, 它就是一个 协作方法 (Cooperative Method)。利用协作方法可以实现 协作多重继承 (Cooperative Multiple Inheritance)。这些术语都是字面意思: 在 Python 中, 多重继承需要相关方法的积极协作才能生效。在 `B` 类 (“示例 14.4<398页>”) 中, `ping` 是协作方法, 但 `pong` 不是。

⁷类也有`.mro()`方法, 但这是元编程 (见 “24.2 身为对象的类<720页>”) 的高级功能。在类的正常使用过程中, `__mro__` 属性的内容才是最重要的。



非协作方法可能会导致不易察觉的 bug。许多程序员在阅读“[示例 14.4<398页>](#)”之后，可能会想当然地认为：当方法 A.ping 调用 super().ping() 时，最终会唤醒 Root.ping。但如果 B.ping 在 Root.ping 之前被唤醒，那么 ping 方法的唤醒序列就到此结束了^a。鉴于此，建议非根类的每一个方法 m 都应该调用 super().m()。

^a因为 B.ping 方法中没有 super() 调用，所以唤醒序列终止。

协作方法（Cooperative Method）必须具有兼容的函数签名，因为您永远不知道 A.ping 是在 B.ping 之前被调用，还是在 B.ping 之后被调用。当同时继承 A 类与 B 类时，唤醒顺序取决于子类声明语句中罗列超类 A 与 B 的顺序。

Python 是一种动态语言，因此 super() 与 MRO（方法解析顺序）的交互也是动态的。[示例 14.6](#) 展示了这种动态行为的惊人结果。

</> [示例 14.6: diamond2.py:演示 super\(\) 动态行为的类](#)

```

1  from diamond import A      ❶
2
3  class U():
4      def ping(self):
5          print(f'{self}.ping() in U')
6          super().ping() ❸
7
8  class LeafUA(U, A):        ❹
9      def ping(self):
10         print(f'{self}.ping() in LeafUA')
11         super().ping()

```

- ❶ A 类来自 diamond.py（“[示例 14.4<398页>](#)”）。
- ❷ U 类与 diamond.py 中的类（A 或 Root）无关。
- ❸ super().ping() 会做什么？答：这要视情况而定。请继续阅读。
- ❹ LeafUA 类按顺序子类化 U 类与 A 类。

如果创建 U 类的实例并尝试调用 ping，则会出现错误：

```

1  >>> u = U()
2  >>> u.ping()
3  Traceback (most recent call last):
4  ...
5  AttributeError: 'super' object has no attribute 'ping'

```

报错信息指出 super() 返回的“super”对象没有“ping”属性。因为 U 的 MRO（方法解析顺序）中有 2 个类：U 类与 object 类，而 object 类中没有名为“ping”的属性。

不过，U.ping 方法也并非毫无用处。请看下面的测试：

```

1  >>> leaf2 = LeafUA()
2  >>> leaf2.ping()
3  <instance of LeafUA>.ping() in LeafUA
4  <instance of LeafUA>.ping() in U

```

```

5 <instance of LeafUA>.ping() in A
6 <instance of LeafUA>.ping() in Root>>> LeafUA.__mro__ # doctest:+
NORMALIZE_WHITESPACE
7 (<class 'diamond2.LeafUA'>, <class 'diamond2.U'>,
8 <class 'diamond.A'>, <class 'diamond.Root'>, <class 'object'>)

```

LeafUA 中的 super().ping() 调用将唤醒 U.ping, 而 U.ping 也通过调用 super().ping() 进行协作, 继续唤醒 A.ping, 最终唤醒了 Root.ping。

请注意, LeafUA 的基类顺序是 (U,A)。如果基类是 (A,U), 那么 leaf2.ping() 将永远不会唤醒 U.ping。因为 A.ping 中的 super().ping() 将唤醒 Root.ping, 而 Root.ping 中未调用 super()。

在实际程序中, 像 U 这样的类可能是一个 **混合类 (mixin class)**: 一个旨在与其他类一起使用的多重继承类, 以提供额外的功能。有关混合类的内容, 参见 “[14.5 混合类<403页>](#)”。

在结束对 MRO (方法解析顺序) 的讨论之前, 看一下 “[Python 标准库](#)” 中 “[Tkinter GUI 工具包](#)” 的复杂多重继承图, 如 [图 14.2](#) 所示。

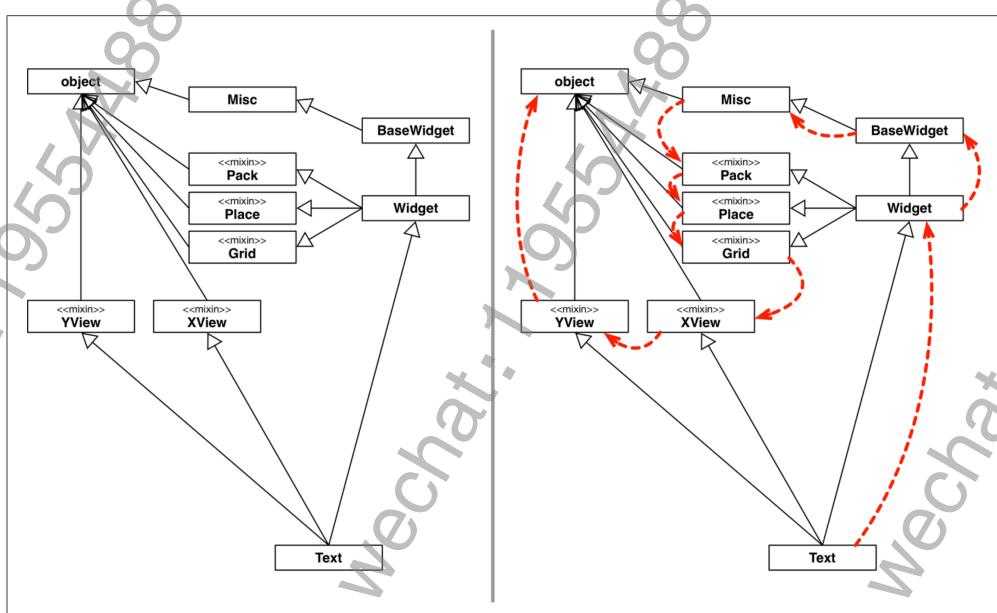


图 14.2: 左图: Tkinter Text 小部件类和超类的 UML 图。
右图: 用虚线箭头画出了 Text.__mro__ 的曲折路径。

研究 [图 14.2](#) 时, 请从图中底部的 Text 类开始。Text 类实现了一个功能完善的多行可编辑文本 widget。它本身提供了丰富的功能, 不过还从其他类继承了许多方法。左侧显示的是一个普通的 UML 类图。右侧则用箭头显示了 MRO (方法解析顺序), 如 [示例 14.7](#) 中 print_mro 函数的输出所示。

```

</> 示例 14.7: tkinter.Text 的 MRO (方法解析顺序)

1 >>> def print_mro(cls):
2 ...     print(''.join(c.__name__ for c in cls.__mro__))
3 >>> import tkinter
4 >>> print_mro(tkinter.Text)
5 Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object

```

下一节, 将讨论 “混合类”。

14.5 混合类

混合 (Mixin) 类的设计目的是在多重继承中连同至少一个其他类一起被子类化。混合类不能作为具体类 (Concrete Class) 的唯一基类。因为混合类并不为具体对象 (Concrete Object) 提供全部功能, 而只是添加或定制子类或同级类的行为, 而只是添加或定制子类或同级类的行为。



在 Python 与 C++ 中, 混合类只是一种没有明确语言支持的约定。Ruby 允许显式定义和使用作为“混合类”的模块。此类模块是一些方法的集合, 用于为类增加新功能。C#、PHP 和 Rust 实现了 traits^a, 这也是一种显式的混合类。

^aTraits 是一种代码重用机制, 它允许将一组方法或行为组合到一个单独的单元中, 并可以在不同的类型中重复使用。

让我们来看一个简单但实用的混合类示例。

14.5.1 不区分大小写的映射

示例 14.8 展示了一个映射类——UpperCaseMixin, 该类在新增或查询字符串“键 (key)”时会将“键 (key)”转换为大写形式, 以实现不区分大小写的“键 (key)”访问。

</> 示例 14.8: `uppermixin.py`: `UpperCaseMixin` 支持不区分大小写的映射

```

1  import collections
2
3  def _upper(key):      ❶
4      try:
5          return key.upper()
6      except AttributeError:
7          return key
8
9  class UpperCaseMixin: ❷
10     def __setitem__(self, key, item):
11         super().__setitem__(_upper(key), item)
12
13     def __getitem__(self, key):
14         return super().__getitem__(_upper(key))
15
16     def get(self, key, default=None):
17         return super().get(_upper(key), default)
18
19     def __contains__(self, key):
20         return super().__contains__(_upper(key))

```

- ❶ 辅助函数 `_upper` 接受一个任意类型的 `key`, 并尝试返回 `key.upper()`; 如果失败, 则将接收的 `key` 原样返回。
- ❷ 混合类 `UpperCaseMixin` 实现了映射的 4 个基本方法, 并且每个方法中都调用了 `super()`, 尽量传入大写形式的 `key`。

由于 `UpperCaseMixin` 类的每个方法都会调用 `super()`, 因此该混合类依赖于一个同级类——该同级类实现或继承了具有相同签名的方法。为了让混合类发挥作用, 在子类的 MRO 中, 混合类要出现在其他类的前面⁸。也就是说, 在类声明语句中, 混合类必需出现在基类元组的第一位, 如 ?? 所示。

</> 示例 14.9: `uppermixin.py`: 使用 `UpperCaseMixin` 的两个类

```

1 class UpperDict(UpperCaseMixin, collections.UserDict): ❶
2     pass
3
4 class UpperCounter(UpperCaseMixin, collections.Counter): ❷
5     """一个特殊计数器, 键 (key) 是大写形式"""
6

```

- ❶ `UpperDict` 不需要自己的实现, 但 `UpperCaseMixin` 必须是第一个基类, 否则将调用 `UserDict` 中的方法。
- ❷ `UpperCaseMixin` 类还可与 `Counter` 配合使用。
- ❸ 与其只放占位符 `pass`, 不如提供一个 `docstring`, 满足 class 语句语法对类主体的要求。

以下是来自 `uppermixin.py` 针对 `UpperDict` 的一些文档测试 (doctest):

```

1 >>> d = UpperDict([('a', 'letter A'), (2, 'digit two')])
2 >>> list(d.keys())
3 ['A', 2]
4 >>> d['b'] = 'letter B'
5 >>> 'b' in d
6 True
7 >>> d['a'], d.get('B')
8 ('letter A', 'letter B')
9 >>> list(d.keys())
10 ['A', 2, 'B']

```

以及 `UpperCounter` 的快速演示:

```

1 >>> c = UpperCounter('BaNanA')
2 >>> c.most_common()
3 [('A', 3), ('N', 2), ('B', 1)]

```

`UpperDict` 与 `UpperCounter` 看起来似乎很神奇, 但我必须仔细研究 `UserDict` 和 `Counter` 的代码, 才能使 `UpperCaseMixin` 类与它们一起工作。

例如, 我的第 1 版 `UpperCaseMixin` 类没有提供 `get` 方法。该版本适用于 `UserDict`, 但不适用于 `Counter`。
`UserDict` 类继承了 `collections.abc.Mapping` 中的 `get` 方法, 该 `get` 方法调用了我实现的 `__getitem__` 方法。但在 `__init__` 加载 `UpperCounter` 时, “键 (key)” 没有大写。发生这种情况是因为 `Counter.__init__` 使用了 `Counter.update`, 而 `Counter.update` 又依赖于从 `dict` 继承的 `get` 方法。然而, `dict` 类中的 `get` 方法并没有调用 `__getitem__`。这就是 “3.5.3 标准库中 `__missing__` 的不一致用法<78页>” 中讨论的问题的核心。这也清楚地提醒我们, 即使是小规模的继承, 利用继承的程序也是脆弱和令人费解的。

下一节, 将介绍几个多重继承的示例, 大多数都使用了混合 (Mixin) 类。

⁸MRO (Method Resolution Order, 方法解析顺序) 是用于确定在多重继承情况下, Python 中类的方法调用顺序的规则。

14.6 多重继承的实际运用

在《设计模式》⁹一书中,几乎所有示例都是用 C++ 编写的,其中“Adapter 设计模式”是唯一用到多重继承的示例。在 Python 中,多重继承也不常用,但也有一些重要的示例,本节将加以评述。

14.6.1 抽象基类也是混合类

在 Python 标准库中,多重继承最明显的用途是用于 `collections.abc` 包。这并无争议:毕竟,即使是 Java 也支持接口的多重继承,而 **抽象基类 (ABCs)** 是接口的声明,偶尔会提供 **具体方法 (Concrete Method)** 的实现¹⁰。

Python 的 `collections.abc` 官方文档 中使用了术语“**混合 (mixin) 方法**”来表示在许多容器 **抽象基类 (ABCs)** 中实现的 **具体方法 (Concrete Method)**。提供混合方法的 **抽象基类 (ABCs)** 扮演两种角色:既是接口定义,也是混合类。例如, `collections.UserDict` 的实现 (见 `Lib/collections/_init_.py` 1084 行) 就依赖 `collections.abc.MutableMapping` 提供的多个混合方法。

14.6.2 ThreadingMixIn 与 ForkingMixIn

`http.server` 包 提供了 `HTTPServer` 和 `ThreadingHTTPServer` 类,后者是在 Python 3.7 中新增的。`ThreadingHTTPServer` 的文档描述如下:

```
class http.server.ThreadingHTTPServer(server_address, RequestHandlerClass)
```

此类与 `HTTPServer` 作用相同,但通过 `ThreadingMixIn` 使用线程来处理请求。特别适合处理 Web 浏览器预先打开的套接字,如果使用 `HTTPServer` 处理,则等待时间无法确定。

在 Python 3.10 中, `ThreadingHTTPServer` 类的完整源码 (见 `Lib/http/server.py` 144 行) 如下所示:

```
1 class ThreadingHTTPServer(socketserver.ThreadingMixIn, HTTPServer):  
2     daemon_threads = True
```

`socketserver.ThreadingMixIn` 的源码有 38 行 (包括注释和文档字符串)。[示例 14.10](#) 仅显示了其概览。

</> [示例 14.10](#): Python 3.10 中 `Lib/socketserver.py` 文件的部分内容

```
1 class ThreadingMixIn:  
2     """混合 (Mixin) 类处理新线程中的每个请求"""  
3  
4     # 省略 8 行  
5  
6     def process_request_thread(self, request, client_address): ❶  
7         ... # 省略 6 行  
8  
9     def process_request(self, request, client_address): ❷
```

⁹此书英文原名为《Design Patterns: Elements of Reusable Object-Oriented Software》(Addison-Wesley 出版),由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著,是一本具有里程碑意义的经典著作。

¹⁰如前所述,Java 8 也允许接口提供方法实现。这个新功能在官方 Java 教程中被称为“**默认方法**”。

```

10 ... # 省略 8 行
11 def server_close(self):
12     super().server_close()
13     self._threads.join()

```

③

- ❶ `process_request_thread` 没有调用 `super()`, 因为它是一个新方法, 而不是对现有方法的重写。该方法的实现调用了 `HTTPServer` 提供或继承的 3 个实例方法。
- ❷ 这会重写 `HTTPServer` 从 `socketserver.BaseServer` 继承的 `process_request` 方法, 启动一个线程并将实际工作委托给在该线程中运行的 `process_request_thread`。它不调用 `super()`。
- ❸ `server_close` 调用 `super().server_close()` 停止接受请求; 然后, 等待 `process_request` 启动的线程完成其工作。

`socketserver` 模块文档将 `ThreadingMixIn` 与 `ForkingMixIn` 罗列在一起。`ForkingMixIn` 基于 `os.fork()` 为服务器提供并发支持。`os.fork()` 是一个用于启动子进程的 API, 在兼容 POSIX 的类 Unix 系统中可用。

14.6.3 Django 泛化视图混合类



无需了解 Django^a 即可阅读本节内容。我使用该框架的一小部分作为多重继承的实际示例, 所有必要的背景知识我都会讲解清楚, 不过您需要有一定的服务器端 Web 开发经验, 任何语言或框架都可以。

^aDjango 是一个 Python 编写的开源 Web 应用程序框架, 详见: <https://docs.djangoproject.com>

在 Django 中, 视图是一个可调用对象, 接受一个表示 HTTP 请求对象的 `request` 参数, 返回一个表示 HTTP 响应的对象。本次讨论中主要关注不同类型的响应。它们既可以像重定向一样简单, 没有内容主体; 也可以像在线商店中的目录页面一样复杂, 由 HTML 模板渲染, 并列出商品, 带有购买按钮和指向详情页面的链接。

最初, Django 提供了一组称为“泛化视图”的函数, 用于实现一些常见的需求。例如, 许多网站需要展示来自众多项的信息的搜索结果, 结果列表跨越多个页面, 每一项都含有指向其详情页面的链接。为解决此问题, Django 提供了一个列表视图和一个详情视图, 列表视图用于渲染搜索结果, 详情视图用于为各个条目生成详情页面。

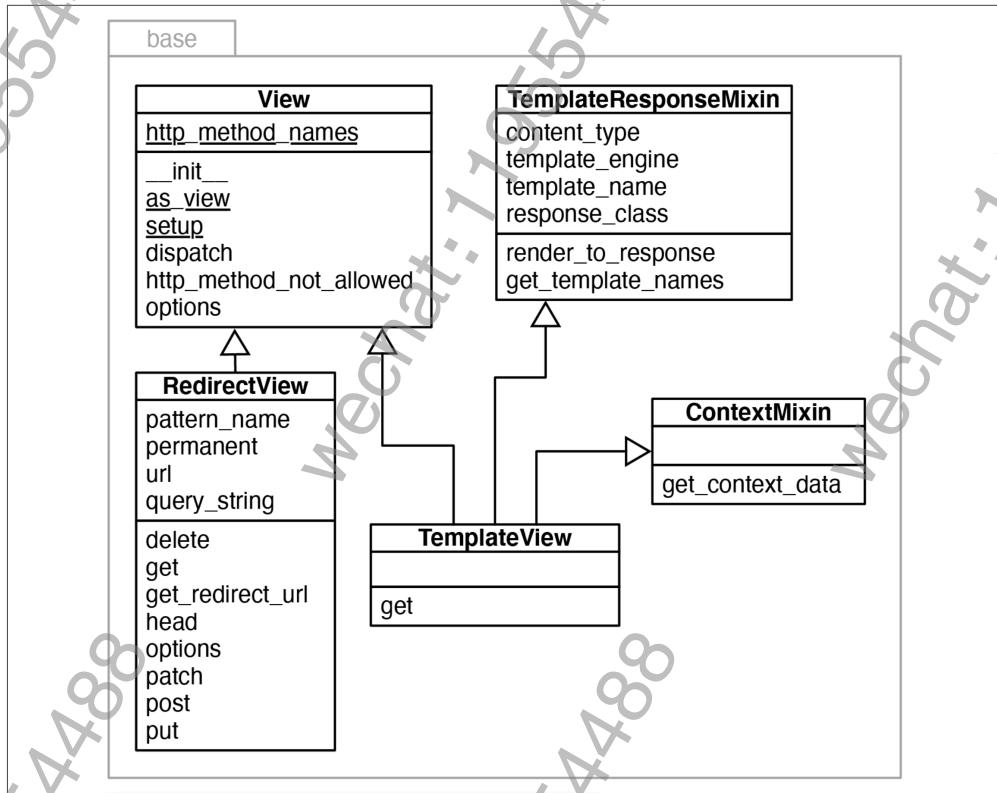
然而, 最初的泛化视图都是函数, 不具有可扩展性。如果您需要做类似但不完全像泛化列表视图的事情, 则必须从头开始实现。

Django 1.3 引入了基于类的视图的概念, 同时提供了一组泛化视图类。这些视图类被组织为基类、混合类和可直接使用的具体类 (Concrete Class)。在 Django 3.2 中, 这些基类和混合类位于 `django.views.generic` 包的 `base` 模块中, 如 图 14.3 所示。图的上半部分是职责完全不同的 2 个类: `View` 与 `TemplateResponseMixin`。这些类提供了更灵活和可扩展的视图实现方式, 使开发人员能够更轻松地定制和管理视图逻辑。



研究这些类的最佳资源是 [Classy Class-Based View 网站^a](https://ccbv.co.uk)。在此网站中可以轻松地浏览这些类, 查看每个类提供的所有方法 (继承的方法、重写的方法、新增的方法)、查看图表、浏览文档, 以及跳转到 [Github 中的源码](#)。

^aClassy Class-Based View 网站地址: <https://ccbv.co.uk/>

图 14.3: `django.views.generic.base` 模块的 UML 类图

`View` 是所有视图 (可以是抽象基类) 的基类, 它提供了核心功能, 如 `dispatch` 方法。`dispatch` 方法会将职责委托给具体子类实现的“处理器 (handler)”方法, 如 `get`、`head`、`post` 等, 以处理不同的 HTTP 动词¹¹。`RedirectView` 类仅从 `View` 继承, 并且您可以看到它实现了 `get`、`head`、`post` 等方法。

`View` 的具体子类 (Concrete Subclass) 应该实现处理器 (handler) 方法, 那么为什么这些方法不是 `View` 接口的一部分呢? 原因在于: 子类可以自由选择, 只需实现想要支持的处理器。例如, `TemplateView` 仅用于显示内容, 因此它只实现了 `get` 方法。如果向 `TemplateView` 发送 HTTP POST 请求, 则继承的 `View.dispatch` 方法在检查到该类未实现 `post` 处理程序时, 会生成 HTTP 405 Method Not Allowed 响应¹²。

`TemplateResponseMixin` 只为需要使用模板的视图提供相关功能。例如, `RedirectView` 没有内容主体, 因此它不需要模板, 也不会继承 `TemplateResponseMixin`。`TemplateResponseMixin` 为 `TemplateView` 和其他模板渲染视图 (如 `django.views.generic` 子包中的 `ListView`、`DetailView` 等) 提供行为。图 14.4 描述了 `django.views.generic.list` 模块和部分 `base` 模块。

对于 Django 用户来说, 图 14.4 中最重要的类是 `ListView`, 它是一个聚合类, 没有任何代码 (其主体只是 docstring)。实例化后, `ListView` 对象有一个 `object_list` 实例属性。模板通过迭代这个属性来显示页面内容 (通常是返回多个对象的数据库查询结果)。至于如何生成模板迭代的对象, 全部由 `MultipleObjectMixin` 来处理。该混合类还提供了复杂的分页逻辑——在一个页面中显示部分结果, 并通过链接指向更多页面。

假设您要创建一个视图, 它无需渲染模板, 但会生成一个 JSON 格式的对象列表。此时, 需要用到 `BaseListView`。该类提供了一个易于使用的扩展点, 可将 `View` 和 `MultipleObjectMixin` 功能整合在一起, 免去了模板机制的开销。

¹¹Django 程序员都知道, 类方法 `as_view` 是 `View` 接口中最显眼的部分, 不过它与本节的讨论无关。

¹²热衷于设计模式的人会发现, Django 的分派机制是模板方法模式的动态变体。之所以说它是动态的, 是因为 `View` 类并不强制子类实现所有的处理器, 而是在运行时由 `dispatch` 检查具体处理器是否可用于处理特定请求。

Django 基于类的视图 API 是比 Tkinter 更好的多重继承示例。尤其是它的混合类更易于理解：每个类都有明确的用途，并且名称都以 ...Mixin 为后缀。

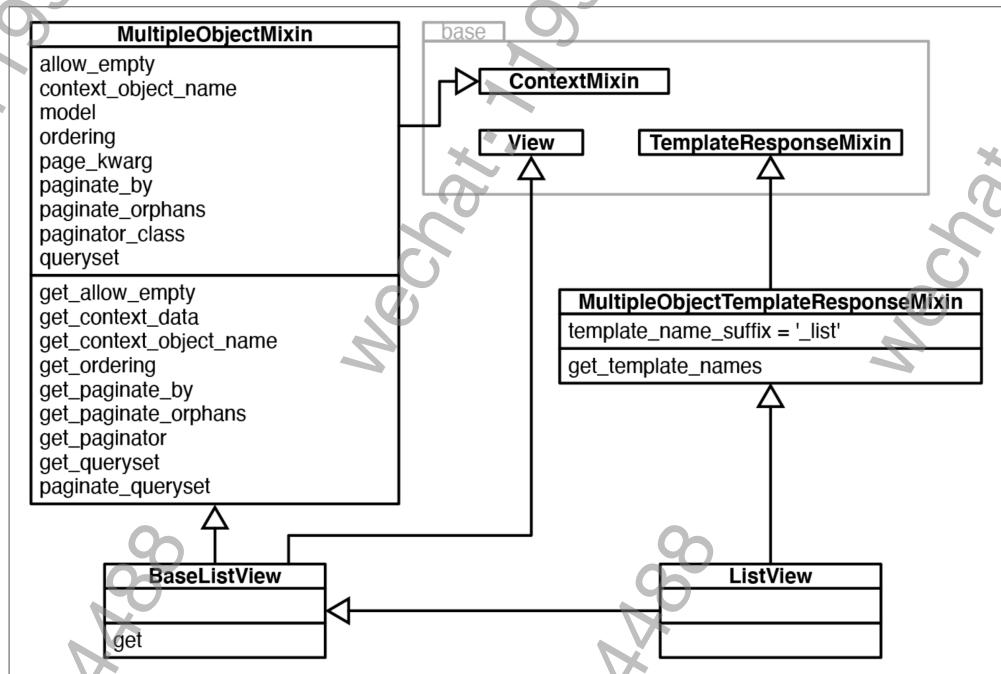


图 14.4: `django.views.generic.list` 模块的 UML 类图

折叠了 `base` 模块中的 3 个类 (见图 14.3)

`ListView` 是一个聚合类，没有方法与属性

基于类的视图并没有被 Django 用户普遍接受。许多人将它们视为不透明的盒子，仅以有限的方式使用。当需要实现新效果时，许多 Django 程序员会继续编写负责所有职责的单一视图函数，而不是尝试重用基本视图类与混合类。

学习如何利用以及如何扩展基于类的视图，以满足特定应用程序需求，确实需要一些时间，但我认为这是值得的。基于类的视图消除了大量的样板代码，使得重用解决方案更容易，并且甚至改进了团队之间的沟通。例如，为模板以及传递给模板上下文的变量定义标准名称。基于类的视图将 Django 视图带上了“正轨”。

14.6.4 Tkinter 中的多重继承

Python 标准库中多重继承的一个极端示例是 `Tkinter GUI 工具包`。图 14.2 展示的 MRO (方法解析顺序) 是 Tkinter 小部件 (widget) 层次结构的一部分。图 14.5 显示了 `tkinter` 基本包中的所有 widget 类 (在 `tkinter.ttk` 子包中有更多的 widget)。

在我写这篇文章的时候，Tkinter 已经 25 岁了，因此不能代表当下的最佳实践。但它展示了在不了解多重继承的缺点时，程序员是如何使用多重继承的。下一节在介绍多重继承最佳实践时，会将 Tkinter 作为反面教材。

看一下图 14.5 中的几个类：

- ① `Toplevel`: Tkinter 应用程序中顶层窗口的类。
- ② `Widget`: 可以放置在窗口上的每个可见对象的超类。
- ③ `Button`: 普通的按钮小部件。
- ④ `Entry`: 单行可编辑文本域。

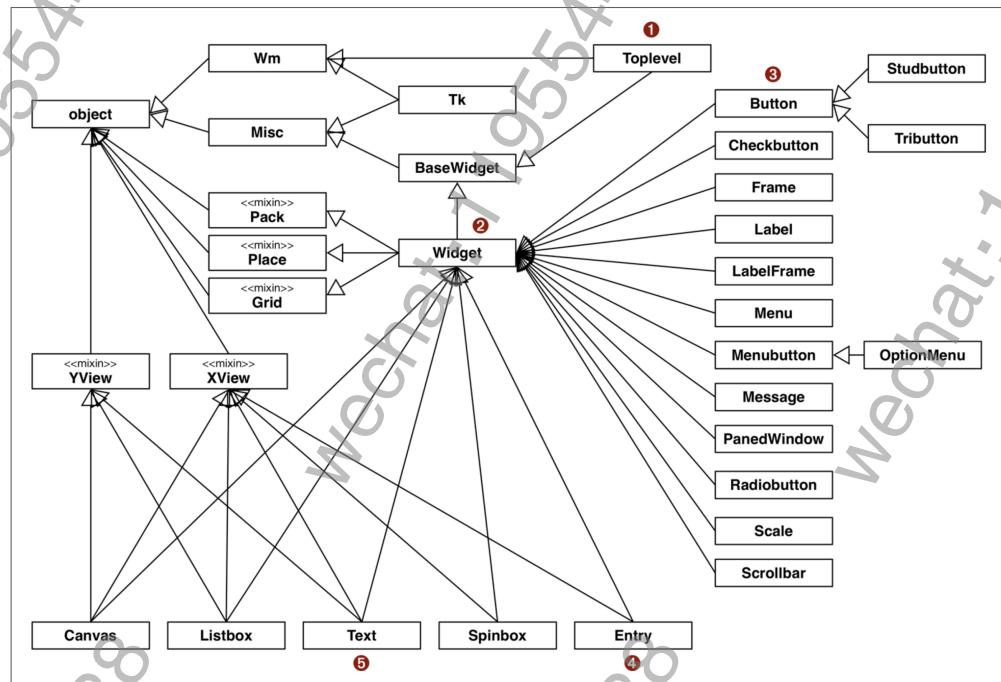


图 14.5: Tkinter GUI 类层次结构的汇总 UML 图
标记为“mixin”的类旨在通过多重继承为其他类提供具体方法

⑤ Text: 多行可编辑文本域。

这些类的MRO (方法解析顺序) 如下所示。这些输出是由 `print_mro` 函数 (见 “示例 14.7<402页>”) 得到的。

```

1  >>> import tkinter
2  >>> print_mro(tkinter.Toplevel)
3  Toplevel, BaseWidget, Misc, Wm, object
4  >>> print_mro(tkinter.Widget)
5  Widget, BaseWidget, Misc, Pack, Place, Grid, object
6  >>> print_mro(tkinter.Button)
7  Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
8  >>> print_mro(tkinter.Entry)
9  Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
10 >>> print_mro(tkinter.Text)
11 Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object

```



按照当前的标准, Tkinter 的类层次结构非常深。Python 标准库中很少有超过 3 到 4 层的具体类 (Concrete Class), Java 类库也是如此。不过, 有趣的是, Java 类库中层次最深的一些类恰恰位于与 GUI 编程相关的包中: `java.awt` 和 `javax.swing`。Squeak 是 Smalltalk 的现代免费版本, 它包括功能强大、创新性强的 Morphic GUI 工具包, 同样具有很深的类层次结构。根据我的经验, GUI 工具包是最能发挥多重继承的作用的地方。

请注意这些类之间的相互关系:

- `Toplevel` 是唯一一个不继承自 `Widget` 的图形类。因为它是顶层窗口, 其行为与 `Widget` 不同。例如,

它不能附加到窗口 (window) 或框架 (frame) 上。Toplevel 继承自 Wm, 后者提供了直接访问主机窗口管理器的功能, 如设置窗口标题和配置窗口边框。

- Widget 直接继承自 BaseWidget 以及 Pack、Place 和 Grid。后 3 个类是几何管理器, 负责在窗口 (window) 或框架 (frame) 内排列小部件 (widget)。每个类都封装了不同的布局策略和部件放置 API。
- Button 与多数 widget 一样, 仅从 Widget 派生, 但间接从 Misc 派生, 而 Misc 为每个 widget 提供了数十种方法。
- Entry 是 Widget 与 XView 的子类, XView 支持水平滚动。
- Text 是 Widget、XView 与 YView 的子类, YView 支持垂直滚动。

下面将讨论多重继承的一些最佳实践, 并看看 Tkinter 是否遵循了这些实践。

14.7 应对多重继承

本章开篇引用 Alan Kay 的那段内容现在依然正确: 目前, 仍然没有关于继承的通用理论可用于为程序员提供实践指导。我们有的只是一些经验法则、设计模式、最佳实践、巧妙的缩略语、禁忌等。虽然, 其中一些提供了有用的指导原则, 但是没有一条是被普遍接受或始终适用的。

使用继承 (即使避开多重继承) 很容易创建出令人费解且脆弱的设计。因为我们没有一套全面的继承理论, 因此本节提供了一些避免类图混乱的一些技巧。

14.7.1 优先使用对象组合, 而不是类继承

本小节的标题是《设计模式》一书中面向对象设计的第二条原则, 也是我们在此能提供的最佳建议。一旦您习惯了继承, 就很容易过度使用继承。按整齐的层次结构放置对象, 可唤起我们的秩序感, 而程序员这样做只是为了好玩。

利于组合的设计可以带来更多的灵活性。例如, 以 tkinter.Widget 类为例, 与其继承所有几何图形管理器的方法, 不如为 Widget 实例保存一个指向几何图形管理器的引用, 并调用其方法。毕竟, Widget 不应该是几何图形管理器, 但可以通过委托方式使用几何图形管理器的服务。这样, 您就可以添加新的几何管理器, 而无需触及 Widget 类的层次结构, 也不必担心名称冲突。即便是单一继承, 这一原则也能增强灵活性, 因为子类化是一种紧密耦合的形式, 而高大的继承树往往都很脆弱。

组合与委托可以取代混合 (mixin) 的使用, 将行为带给不同的类。但是, 不能取代使用接口继承来定义类型的层次结构。

14.7.2 理解不同场景下使用继承的原因

在处理多重继承时, 有必要弄清楚在每种特定场景下进行子类化的原因。主要原因有:

- 继承接口, 创建子类型, 实现“是什么”的关系。此种场景最好使用 **抽象基类 (ABCs)**。
- 继承实现, 通过重用避免代码重复。此种场景可以使用混合 (Mixin) 类。

在实践中, 这两种用途往往是同时存在的, 但只要能明确意图, 就一定要这么做。代码重用的继承是一种实现细节, 通常可被组合与委托所取代。另一方面, 接口继承是框架的支柱。接口继承应尽可能只使用 **抽象基类 (ABCs)** 作为基类。

14.7.3 使用抽象基类显式表示接口

在现代 Python 编程中,如果一个类的作用是定义接口,则应显式将其定义为一个 **抽象基类 (ABCs)** 或 **typing.Protocol** 子类。**抽象基类 (ABCs)** 只能子类化 `abc.ABC` 或其他 **抽象基类 (ABCs)**。继承多个 **抽象基类 (ABCs)** 是没有问题的。

14.7.4 用显式混合类实现代码重用

如果一个类的作用是提供方法实现,以供多个不相关的子类重用,但不体现“是什么”关系。那么,就应该将此类显式定义为“**混合 (Mixin) 类**”。从概念上讲,混合类并没有定义新的类型;它只是将方法捆绑在一起以便重复使用。**混合类**绝不应该被实例化,具体类也不应该只从混合类继承。每个混合类只应提供一种特定的行为,实现少数几种密切相关的方法。混合类应避免保留任何内部状态;也就是说,混合类不应有实例属性。

在 Python 中没有正式的方法来说明混合类,因此强烈建议在命名混合类时使用 `Mixin` 后缀。

14.7.5 为用户提供聚合类

主要通过继承混合类而构建,不添加自身的结构或行为的类,被称为“**聚合类**”。

——Booch 等人^a

^a《Object-Oriented Analysis and Design with Applications, 3rd ed》(Grady Booch 等) 第 109 页。

如果**抽象基类 (ABCs)**或混合类的某些组合,对客户端代码特别有用,则应该提供一个类,以合理的方式将它们组合在一起。

例如,如下是“图 14.4<408页>”右下角 `Django ListView` 类的完整源代码 (`django/views/generic/list.py` 194 行):

```
1 class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
2     """
3     Render some list of objects, set by `self.model` or `self.queryset`.
4     `self.queryset` can actually be any iterable of items, not just a queryset.
5     """
```

`ListView` 类的主体是空的,但该类提供了一项有用的服务:它将混合类与基类整合在一起,以作为一个整体来使用。

聚合类的另一个例子是 `tkinter.Widget`,它有 4 个基类,没有自己的方法与属性——仅有一个 `docstring`(源码见 `Lib/tkinter/_init__.py` 2618 行)。得益于聚合类 `Widget`,我们可以创建一个带有所需混合类的新 `widget`,而无需考虑声明语句中混合类的顺序,即可达到预期效果。

请注意,聚合类不必完全为空的,只是经常为空而已。

14.7.6 仅子类化为扩展而设计的类

技术审校 Leonardo Rochael 在审阅本章时,提出如下警告:



子类化复杂的类并重写其方法很容易出错,因为超类方法可能会以意想不到的方式忽略子类的重写。应尽可能避免重写方法,或至少限制自己仅子类化那些为扩展而设计的类,并且只以它们期望的方式进行子类化。

这是个很棒的建议,但是如何判断一个类是否是为了扩展而设计的呢?

首先,应查阅文档(或者代码中的注释与 docstring)。例如,Python 的 `socketserver` 软件包被描述为“a framework for network servers”。它的 `BaseServer` 类顾名思义是为子类化而设计的。更重要的是,该类的文档和源码中的 docstring 都明确指出了哪些方法可以被子类重写。

在 Python ≥ 3.8 中,“PEP 591 –Adding a final qualifier to typing”提供了一种新的方法可使这些设计约束更明确。该 PEP 引入了 `@final` 装饰器可用于标记类或单个方法,这样 IDE 或类型检查工具见到它后,就知道不应该子类化被它标记的类,或者不应该重写被它标记的方法¹³。

14.7.7 避免子类化具体类

子类化具体类(Concrete Class)比子类化抽象基类或混合类更危险,因为具体类的实例通常具有内部状态,当重写依赖于该状态的方法时,很容易破坏这些内部状态。即使是调用 `super()` 协作方法(Cooperative Method),并且内部状态使用 `_x` 语法保存在私有属性中,重写方法时仍然有无数种引入 bug 的可能。

在“13.5 大鹅类型(Goose Typing)<358页>”的附注栏“水禽与抽象基类”中,Alex Martelli 引用了《More Effective C++》(Scott Meyer 著)中的一句话:“所有非叶子类,都应是抽象类”。言外之意,Meyer 建议只应该对抽象类进行子类化。

如果必须用子类化来实现代码重用,那么想要重用的代码应放入抽象基类的混合方法中,或放入明确命名(如以...Mixin 为后缀)的混合类中。

接下来,从这些建议的角度来分析 Tkinter。

14.7.8 Tkinter 的优与劣

前几小节提出的建议,Tkinter 大都没有遵从。仅遵从了“14.7.5 为用户提供聚合类<411页>”,即便如此,Tkinter 做得也不够好。因为如“14.7.1 优先使用对象组合,而不是类继承<410页>”所述,使用组合模式将几何管理器继承到 Widget 中或许更好。

请记住,自 Python 1.1 于 1994 年发布以来,Tkinter 一直是标准库的一部分。Tkinter 的底层是 Tcl 语言优秀的 Tk GUI 工具包。Tcl/Tk 组合原本不是面向对象的,因此 Tk API 基本上就是一大堆函数。不过,该工具包在设计上是面向对象的,即使最初的 Tcl 实现未使用面向对象。

`tkinter.Widget` 的 docstring 介绍说 Widget 是一个内部类,这表明 Widget 很可能是一个抽象基类(ABCs)。虽然 Widget 没有自己的方法,但它定义了一个接口。它传达的信息是“除了会提供 3 个几何管理器的所有方法外,每个 Tkinter widget 也都会提供基本的 widget 方法(如 `_init_`、`destroy` 和几十个 Tk API 函数)”。我们可以认为这样定义接口的方式不是很好,但它确实是一个接口——即 Widget 将接口“定义”为其超类接口的并集。

封装 GUI 应用程序逻辑的 Tk 类继承自 Wm 与 Misc,二者既不是抽象类,也不是混合类。Misc 类的名称本身就明显是一种代码异味(Code Smell)。Misc 有 100 多个方法,所有的 widget 都继承自 Misc。为何每

¹³PEP 591 还引入了 Final 注解,用于标记不应被赋值或覆盖的变量或属性。

个 widget 都要处理剪贴板、文本选择、计时器管理等方法呢？我们不能将文本粘贴到按钮上，也不能选择滚动条上的文字。应该将 Misc 拆分成几个专用的混合类，并且不是所有 widget 都应继承这些混合类。

公平地说，作为 Tkinter 用户，您根本无需知道或使用多重继承。这些都是隐藏在 Widget 类后面的实现细节，在您自己的代码中仅需实例化或子类化 Widget 类即可。不过，当在控制台输入 `dir(tkinter.Button)`，并在列出的多达 214 个属性中，尝试查找所需的方法时，您会发现过度使用多重继承是多么痛苦。



尽管存在这些问题，但 Tkinter 仍然稳定、灵活，而且如果使用 `tkinter.ttk` 软件包及其主题部件，还能提供现代的外观和感觉。此外，一些原始的部件，如 `Canvas` 和 `Text`，功能都非常强大。你可以在几小时内将 `Canvas` 对象变成一个简单的拖放绘图应用程序。如果对 GUI 编程感兴趣，Tkinter 和 Tcl/Tk 绝对值得一看。

对继承的剖析，到此结束。

14.8 本章小结

本章首先回顾了在单一继承场景下 `super()` 函数的用法。然后，讨论了子类化内置类型时遇到的问题：内置类型在 C 中实现的原生方法不会调用子类中的重写方法，只有极少数例外。因此，在需要定制 `list`、`dict` 或 `str` 类型时，更容易的做法是子类化 `UserList`、`UserDict` 或 `UserString`。它们都定义在 `collections` 模块中，实际上是对相应内置类型的封装，并将操作委托给内置类型——这是标准库中优先使用组合而不是继承的 3 个例子。如果期望的行为与内置类型提供的行为差别很大，则更容易的做法是子类化 `collections.abc` 中相应的抽象基类（ABCs），并编写自己的实现。

本章的其余部分着重讲解了多重继承这把“双刃剑”。首先，介绍了 MRO（方法解析顺序），它保存在类属性 `__mro__` 中，解决了继承方法可能出现的命名冲突问题。随后，介绍了内置函数 `super()` 在具有多重继承的层次结构中的行为（有时会令人意外）。`super()` 函数的行为设计是为了支持混合类。通过 `UpperCaseMixin` 的简单示例（“[示例 14.8<403页>](#)”）展开了对混合类的研究。

介绍了多重继承和混合方法在 Python 抽象基类（ABCs）中的应用，以及在 `socketserver` 包中 `ThreadingMixIn` 与 `ForkingMixIn` 中的应用。Django 基于类的视图和 Tkinter GUI 工具包展示了多重继承的更复杂用法。尽管 Tkinter 不能代表多重继承的最佳实践，但它展示的复杂的类层次结构也可能出现在遗留系统中。

为了结束本章，我提出了 7 项应对继承的建议，并以 Tkinter 类层次结构为例做了补充说明。

拒绝继承（甚至是单一继承）是现代编程的趋势。Go 语言是 21 世纪创建的最成功的语言之一，它甚至都没有“class”这种结构，但你可以通过封装字段的结构体来构建类型，并且还可以将方法附加到这些结构体上。Go 允许定义接口，这些接口由编译器使用结构类型（Structural Typing）来检查。结构类型（Structural Typing），即静态鸭子类型（Static Duck Typing）——与 Python 3.8 以后的协议非常相似。Go 有专门的语法来通过组合构建类型和接口，但它不支持继承——甚至在接口之间也不支持继承。

因此，关于继承的最佳建议也许是：如果可以的话，尽量避免使用。但通常情况下，我们无法选择：因为我们使用的框架有它自己的设计选择。

14.9 延伸阅读

就可读性而言,适当的组合要优于继承。由于读代码的频率比写代码要高得多,因此一般情况下要避免使用继承。尤其是不要混合使用多种类型的继承,也不要使用继承来共享代码。

——Hynek Schlawack, 出自“Subclassing in Python Redux”

在本书的终审过程中,技术审校 Jürgen Gmach 向我推荐了文章“Subclassing in Python Redux”(Hynek Schlawack)——这也是前面引文的出处。Schlawack 是广受欢迎的 attrs 软件包 的作者,也是 Twisted 异步编程框架 的核心贡献者,该框架由 Glyph Lefkowitz 于 2002 年发起。Schlawack 说,随着时间的推移,核心团队意识到他们在设计中过度使用了子类化。他的文章很长,还引用了其他重要文章和演讲。强烈推荐阅读。

在那篇文章的结论中,Hynek Schlawack 还写道:“不要忘记,更多时候,你需要的只是一个函数而已。”我也认同这一观点,正因如此,本书才在类与继承之前先深入探讨了函数。我的目标是告诉你:利用标准库中的现有类,通过函数也可以完成许多工作,不要动不动就创建自己的类。

“Unifying types and classes in Python 2.2”(Guido van Rossum)一文介绍了子类化内置类型、super() 函数,以及描述符 (Descriptor) 和元类 (MetaClass) 等高级功能。自那以后,这些功能基本都没有太大的变化。Python 2.2 是语言进化史上一项了不起的壮举,它在不破坏向后兼容性的前提下,增加了一整套强大的新功能。这些新功能都是 100% 可选的,用不用全有您自己决定。如果想使用,仅需显式地 (间接或直接) 继承 object 类即可——这样即可创建所谓的“新型类”。在 Python3 中,所有类都是 object 类的子类。

《Python Cookbook, 3rd Ed》(David Beazley、Brian K. Jones) 中的几个经典示例展示了 super() 与混合类的使用。您可以从“8.7. Calling a Method on a Parent Class”开始阅读,在受到启发之后,在遵循其内部引用,阅读其他内容。

Raymond Hettinger 的博文“Python’s super() considered super!”,从积极的角度解读了 Python 中 super() 函数与多重继承的工作原理。此文章是为了回应“Python’s Super is nifty, but you can’t use it”而写的。Martijn Pieters 在回答“How to use super() with one argument?”这一问题时,对 super() 函数做了间接而深入的解释,包括它与描述符 (Descriptor) (详见“[二十三 属性描述符](#)”) 的关系。super 函数就是这样,它在基础场景中用法很简单,但是当涉及到 Python 那些最高级的动态功能时,又变成了强大而复杂的工具,这在其他语言中很少见。

尽管上述那些文章的题目都提到了内置函数 super(),但它不是问题的根源,况且 Python 3 中的 super 函数并不像 Python 2 中那么令人讨厌了。问题的根源是多重继承,它本身就很复杂、棘手。Michele Simionato 在他的“Setting Multiple Inheritance Straight”一文中,不仅批评了多重继承,还提供了一个解决方案:他实现了 traits,一种源自 Self 语言的明确形式的混合体。Simionato 写了一系列关于 Python 多重继承的博文,包括“The wonders of cooperative inheritance, or using super in Python 3”、“Mixins considered harmful”(第 1 部分与第 2 部分)、“Things to Know About Python Super”(第 1 部分、第 2 部分与第 3 部分)。最早的博文使用的是 Python 2 的 super 语法,不过依然值得一读。

我阅读过《面向对象分析与设计 (第 3 版)》(Grady Booch 等),并强烈推荐给您作为面向对象思想的通用 (与编程语言无关) 入门读物。这是一本罕见的书,它很可观地介绍了多重继承。

现在,人们更倾向于摒除继承,这里有两份关于如何避免使用继承的资料:一份是《Python Design Patterns》(Brandon Rhodes 著) 中的“The Composition Over Inheritance Principle”一文;另一份是 PyCon 2013 上题为“The End Of Object Inheritance & The Beginning Of A New Modularity”(Augie Fackler、

Nathaniel Manista) 的演讲。Fackler 与 Manista 的演讲讨论了如何围绕接口与处理实现这些接口的函数来组织系统, 避免类与继承的紧密耦合与失效模式。这让我想起了 Go 语言的许多处理方式, 他们提倡在 Python 中也采用这种方式。

杂谈

想想哪些类是真正需要的

起初, 我们推动继承的思想, 是为了让新手可以顺利使用框架(由专家设计)来开发作品。

——Alan Kay, 摘自 “The Early History of Smalltalk”

绝大多数程序员编写的都是应用程序, 而不是框架。即便是那些编写框架的程序员, 大多数时间也是在编写应用程序。在编写应用程序时, 通常不需要编写类的层次结构。如果需要自己编写类, 基本上也是子类化 [抽象基类\(ABCs\)](#) 或子类化框架提供的其他类。作为应用程序开发人员, 我们很少需要编写超类。我们自己编写的类几乎都是叶子类(即继承树中的叶子端)。

作为应用开发人员, 如果您发现自己正在构建多层的类层次结构。那么, 很可能出现以下一种或多种情况:

- 正在重新造轮子。去寻找框架或库, 它们提供的组件可在应用程序中重用。
- 正在使用设计糟糕的框架。去寻找替代方案吧。
- 设计过度。记住要遵守 [KISS 原则](#)。
- 对编写应用程序感到厌倦, 并决定新造一个框架。恭喜你, 祝你好运!

这些情况可能都会遇到: 感到无聊, 并决定通过构建自己的过度设计和设计糟糕的框架来重新发明轮子。这迫使您一次又一次地编写代码来解决琐碎的问题。希望你玩得开心, 或者至少能得到回报。

行为不当的内置类型: 是 bug 还是功能?

内置的 dict、list 和 str 类型是 Python 本身的基本构建块, 因此它们必须很快——这些内置类型有关的任何性能问题, 都会几乎对其他所有代码产生重大影响。于是, CPython 走了捷径, 故意让内置类型的方法行为不当, 即 [内置类型不会调用被其子类重写的方法](#)。摆脱这种困境的一种可能方式是为每种类型提供 2 种实现: 一种是内部实现, 针对解释器而优化; 另一种是供外部使用的实现, 便于扩展。

但是等等, 我们已经有了: UserDict、UserList 和 UserString 虽然没有内置类型那么快, 但却很容易扩展。CPython 采取的务实方法意味着, 我们也可以在自己的应用程序中使用高度优化(但难以子类化)的实现。考虑到我们很少需要定制映射、列表或字符串, 但每天都在使用 dict、list 和 str, 这也是合情合理的。我们只需了解其中的利弊得失即可。

跨语言继承

Alan Kay 创造了“面向对象”这一术语, 而 Smalltalk 只有单一继承, 不过也有支持多种形式多重继承的分支, 包括支持 traits 的现代 Squeak 和 Pharo Smalltalk 方言。traits 是一种语言结构, 它既能发挥混合类的作用, 又能避免多重继承的一些问题。

第一个实现多重继承的流行语言是 C++。该功能被滥用, 以至于 Java(旨在替代 C++) 在设计时就没有支持多重继承的实现(即没有混合类)。不过, Java 8 引入了默认方法, 这使得 Java 接口与 C++ 和 Python 用于定义接口的抽象类十分相似。在 Java 之后, 部署最广泛的 JVM 语言应该是 Scala, 它实现了

traits。

其他支持 traits 的语言还有 PHP 和 Groovy 的最新稳定版本, 以及 Rust 和 Raku(以前称为 Perl 6^a)。所以, 可以公平地说 Trait 在 2021 年很流行。

Ruby 在多重继承方面独辟蹊径: 它不支持多重继承, 但引入了混合模块作为语言特性。Ruby 类可以在其主体中包含一个模块, 这样模块中定义的方法就会成为类实现的一部分。这是一种“纯粹”的混合体形式, 不涉及继承, 而且很明显, Ruby 混合体不会影响使用它的类的类型。这提供了混合的好处, 同时也避免了许多常见问题。

有两种新的面向对象语言受到了广泛关注, 它们严重限制了继承性: Go 和 Julia。这两种语言都是以“对象”为编程对象, 支持多态性, 但都避免了使用“类”这个术语。

Go 完全没有继承。Julia 有一个类型层次结构, 但子类型不能继承结构, 只能继承行为, 而且只有抽象类型可以被子类化。此外, Julia 方法使用“多分派”来实现——这是“9.9.3 单分派泛化函数<264页>”所述的单分派机制的高级形式。

^a我的朋友兼技术评论员 Leonardo Rochael 比我解释得更清楚: “Perl 6 的持续存在和姗姗来迟, 耗尽了 Perl 自身发展的意志力。现在, Perl 作为一种独立的语言继续发展(目前已发展到 5.34 版本), 并没有因为以前的 Perl 6 语言而有任何衰退的迹象。”

类型注解进阶 1136

惨痛的教训告诉我, 对于小型的程序, 动态类型就足够了。而大型程序则需要更规范的方式。如果语言能提供这种规范, 那当然比“放任自流”要好。

——Guido van Rossum, Monty Python 的粉丝^a

^a摘自 YouTube 于 2019 年 4 月 2 日的直播视频“*A Language Creators’ Conversation: Guido van Rossum, James Gosling, Larry Wall, and Anders Hejlsberg*”, 引用的内容自 1:32:05 开始, 为简洁起见, 做了修改。完整内容见 <https://github.com/fluent-python/language-creators>

本章接续“[八 函数中的类型提示<205页>](#)”, 涵盖了 Python [渐进式类型系统 \(Gradual Type System \)](#) 的更多内容。主要内容包括:

- 重载的函数签名。
- 使用 `typing.TypedDict` 为用作记录的字典添加 [类型提示 \(Type Hints \)](#)。
- 类型校正 (Cast)。
- 运行时访问 [类型提示 \(Type Hints \)](#)。
- [泛型 \(Generic Type \)](#)
 - 声明一个泛化类。
 - [型变 \(variance \)](#): 不变、[协变 \(covariance \)](#) 与 [逆变 \(contravariance \)](#)。
 - 泛化静态协议。

15.1 本章新增内容

本章是《流畅的 Python》第 2 版新增的一章。下面从重载开始讲解。

15.2 重载的签名

Python 函数可以接受不同的参数组合。`@typing.overload` 装饰器允许注解这些不同的组合。当函数的返回类型取决于两个或多个参数的类型时,这一功能尤为重要。

请看内置函数 `sum`。如下是 `help(sum)` 的文本:

```
>>> help(sum)
1 sum(iterable, /, start=0)
2     Return the sum of a 'start' value (default: 0) plus an iterable of numbers
3
4
5     When the iterable is empty, return the start value.
6     This function is intended specifically for use with numeric values and may
7     reject non-numeric types.
```

内置函数 `sum` 是用 C 语言编写的, `typeshed` 项目在 `stdlib/2and3/builtins.pyi` 文件中为其提供了重载的 **类型提示** (Type Hints):

```
1 @overload
2 def sum(__iterable: Iterable[_T]) -> Union[_T, int]: ...
3 @overload
4 def sum(__iterable: Iterable[_T], start: _S) -> Union[_T, _S]: ...
```

首先,来看一下重载的整体语法。这就是**存根 (stub)** 文件 (.pyi) 中有关 `sum` 的所有代码。具体实现将在另一个文件中。省略号 (...) 除了满足函数体的语法要求外,没有其他作用,类似于 `pass`。所以,.pyi 文件是有效的 Python 文件。

如“8.6 注解仅限位置参数与变长参数<240页>”所述, `__iterable` 中的 2 个前导下划线是 PEP 484 为表示**仅限位置参数 (Positional-only Parameter)** 而制定的约定,供 Mypy 强制执行检查时使用。也就是说,可以调用 `sum(my_list)`,但不能调用 `sum(__iterable = my_list)`。

类型检查器会尝试按顺序将给定参数与每个重载的签名进行匹配。调用 `sum(range(100), 1000)` 与第一个重载不匹配,因为该签名只有一个参数。但它与第二个重载匹配。

在常规的 Python 模块中也可以使用 `@typing.overload`,方式是将重载的签名放在函数实际签名与函数实现之前。示例 15.1 展示了 `sum` 在 Python 模块中的注释和实现方式。

</> 示例 15.1: mysum.py: 具有重载签名的 `sum` 函数的定义

```
1 import functools
2 import operator
3 from collections.abc import Iterable
4 from typing import overload, Union, TypeVar
5
6 T = TypeVar('T')
7 S = TypeVar('S') ❶
8
9 @overload
10 def sum(it: Iterable[T]) -> Union[T, int]: ... ❷
11 @overload
12 def sum(it: Iterable[T], /, start: S) -> Union[T, S]: ... ❸
```

```
13 def sum(it, /, start=0): ④
14     return functools.reduce(operator.add, it, start)
```

- ① 第二个重载的签名中需要这个 TypeVar。
- ② 此签名适用于简单情况,即 `sum(my_iterable)`。结果类型可以是 `T`,与 `my_iterable` 产生的元素类型相同;也可以是 `int`,此时可迭代对象为空,因为参数 `start` 的默认值为 0。
- ③ 当给定 `start` 时,它可以是任何类型 `S`,因此结果类型是 `Union[T, S]`。这就是需要 `S` 的原因。如果要重用 `T`,则 `start` 需要与 `Iterable[T]` 元素的类型相同。
- ④ 函数具体实现中的签名没有 [类型提示 \(Type Hints\)](#)。

一个单行函数却用了这么多行注解。我知道,这可能有点“劳民伤财”了。但是,至少这不是随便举的例子。

如果您想通过阅读代码来了解 `@overload`,[typeshed 项目](#) 中有数百个示例。在写作本书时,[typeshed 项目](#) 中的[存根文件](#)为 Python 内置函数提供了 186 个重载的签名,比标准库还要多。



充分利用渐进式类型系统 (Gradual Type System)

追求[类型提示 \(Type Hints\)](#)全覆盖可能会导致代码充斥太多噪音,但价值却很小。为简化[类型提示 \(Type Hints\)](#)而进行重构,可能会导致 API 变得繁琐。有时,应该务实一些,允许部分代码没有[类型提示 \(Type Hints\)](#)也没关系。

符合 Python 风格的 API 通常很难注解。下一节,将给出一个示例:为了正确注解灵活的内置函数 `max`,需要 6 个重载的签名。

15.2.1 重载 `max` 函数

利用 Python 强大动态特性的函数,往往难以添加[类型提示 \(Type Hints\)](#)。

在研究 [typeshed 项目](#) 的过程中,我发现了 [bug 报告 #4051](#):内置函数 `max()` 的[实参 \(Argument\)](#) 不能是 `None`,但是当传入 `None` 或者传入可能产出 `None` 的可迭代对象时,Mypy 并未发出警告。而是会得到如下所示的运行时异常:

```
1  TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

`max` 文档的开头有这样一句话:

返回可迭代项中的最大项,或返回两个或多个参数中的最大一个。

对我来说,这是一个非常直观的描述。

但是,如果按照这样的说法为函数增加类型提示,我不得不问:注解什么?是一个可迭代对象,还是两个或两个以上参数?

实际情况更为复杂,因为 `max` 还包含两个可选的关键字参数:`key` 和 `default`。

我用 Python 编写了 `max`,以便更容易理解其工作方式与重载注解之间的关系(内置 `max` 是用 C 语言编写的);如[示例 15.2](#)所示。

</> 示例 15.2: mymax.py: 用 Python 重写 max 函数

```

1 # 省略导入语句与变量定义, 详见 示例 15.3
2
3 MISSING = object()
4 EMPTY_MSG = 'max() arg is an empty sequence'
5
6 # 省略重载的类型提示, 详见 示例 15.3
7 def max(first, *args, key=None, default=MISSING):
8     if args:
9         series = args
10        candidate = first
11    else:
12        series = iter(first)
13    try:
14        candidate = next(series)
15    except StopIteration:
16        if default is not MISSING:
17            return default
18        raise ValueError(EMPTY_MSG) from None
19    if key is None:
20        for current in series:
21            if candidate < current:
22                candidate = current
23        else:
24            candidate_key = key(candidate)
25        for current in series:
26            current_key = key(current)
27            if candidate_key < current_key:
28                candidate = current
29                candidate_key = current_key
30    return candidate

```

本例的关注点不是 max 函数的逻辑, 因此不会过多地解释具体实现。重点是 MISSING 常量, 该常量是个独特的 object 实例, 用作 **哨兵 (Sentinel)** 对象。它是 default= 关键字参数的默认值, 以让 max 函数可以接受 default=None, 并且还可以区分如下两种情况:

- 用户没有为 default= 提供值, 因此它是 MISSING, 如果 first 是一个空的可迭代对象, max 将引发 ValueError。
- 用户为 default= 提供了某个值, 包括 None, 因此如果 first 是一个空的可迭代对象, max 将返回提供的值。

为了修复 bug #4051, 我编写了如 [示例 15.3](#) 所示的代码¹。

</> 示例 15.3: mymax.py: 模块顶部, 包含导入、定义和重载

```

1 from collections.abc import Callable, Iterable
2 from typing import Protocol, Any, TypeVar, overload, Union

```

¹感谢 Jelle Zijlstra ([typedeshed](#) 项目的维护人之一), 他教会我许多, 还将我最初写的 9 个重载签名精简到了 6 个。

```
3
4     class SupportsLessThan(Protocol):      def __lt__(self, other: Any) -> bool: ...
5
6     T = TypeVar('T')
7     LT = TypeVar('LT', bound=SupportsLessThan)
8     DT = TypeVar('DT')
9
10    MISSING = object()
11    EMPTY_MSG = 'max() arg is an empty sequence'
12
13    @overload
14    def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
15        ...
16
17    @overload
18    def max(__arg1: T, __arg2: T, *args: T, key: Callable[[T], LT]) -> T:
19        ...
20
21    @overload
22    def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
23        ...
24
25    @overload
26    def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
27        ...
28
29    @overload
30    def max(__iterable: Iterable[LT], *, key: None = ...,
31            default: DT) -> Union[LT, DT]:
32        ...
33
34    @overload
35    def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
36            default: DT) -> Union[T, DT]:
37        ...
38
39    ...
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
```

我在 Python 中实现的 max 与那些类型导入和声明的长度差不多。得益于 鸭子类型 (Duck Typing)，我的代码没有使用 isinstance 检查，但是达到的类型检查效果与类型提示 (Type Hints) 相当——当然只是运行时检查。

@typing.overload 的关键优势是，可根据给定的参数类型，尽量精确地声明返回值类型。接下来，将以一个或两个为一组，深入研究 max 函数重载的签名。

15.2.1.1 参数实现了 SupportsLessThan，但未提供 key 与 default

```
1 @overload
2 def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
3     ...
4     # ... 省略多行 ...
5
6     @overload
7     def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
8         ...
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
```

在这两种情况下，输入要么是实现了 SupportsLessThan 协议的 LT 类型的多个单独实参；要么是单个可

迭代对象 `Iterable[LT]`, 其中的项是实现了 `SupportsLessThan` 协议的 `LT` 类型。`max` 函数的返回值类型与实参 (Argument) 类型或可迭代对象中项的类型相同 (详见 “8.5.9.2 有届的 `TypeVar`<231页>”)。

下面是可匹配这两个重载签名的调用示例:

```
1 max(1, 2, -3) # 返回 2
2 max(['Go', 'Python', 'Rust']) # 返回 'Rust'
```

15.2.1.2 提供了 `key`, 但未提供 `default`

```
1 @overload
2 def max(__arg1: T, __arg2: T, *_args: T, key: Callable[[T], LT]) -> T:
3 ...
4 # ... 省略多行 ...
5 @overload
6 def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
7 ...
```

输入可以是任何类型 `T` 的多个单独实参; 也可以是单个可迭代对象 `Iterable[T]`, 其中的项是任意类型 `T`; 参数 `key` 必须是一个可调用对象, 接受同为 `T` 类型的参数, 并返回实现了 `SupportsLessThan` 协议的值。`max` 函数的返回值类型与传入的实参 (Argument) 类型相同。

下面是可匹配这两个重载签名的调用示例:

```
1 max(1, 2, -3, key=abs) # 返回 -3
2 max(['Go', 'Python', 'Rust'], key=len) # 返回 'Python'
```

15.2.1.3 提供了 `default`, 但未提供 `key`

```
1 @overload
2 def max(__iterable: Iterable[LT], *, key: None = ...,
3         default: DT) -> Union[LT, DT]:
4 ...
```

输入是一个可迭代对象 `Iterable[LT]`, 其中的项是实现了 `SupportsLessThan` 协议的 `LT` 类型。参数 `default=` 是可迭代对象为空时的返回值。因此, `max` 的返回值类型必须 `LT` 类型与 `default` 参数类型 `DT` 的联合。

下面是可匹配这个重载签名的调用示例:

```
1 max([1, 2, -3], default=0) # 返回 2
2 max([], default=None) # 返回 None
```

15.2.1.4 提供了 `key` 与 `default`

```
1 @overload
2 def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
3         default: DT) -> Union[T, DT]:
```

4

输入包括以下内容：

- 一个可迭代对象 `Iterable[T]`, 其中的项可以是任意类型 `T`。
- 一个可调用对象 `Callable[[T], LT]`, 该可调用对象接收类型为 `T` 的参数, 返回实现了 `SupportsLessThan` 协议的 `LT` 类型值。
- 一个默认值, 类型为任意类型 `DT`。

`max` 的返回值类型必须是 `T` 类型与 `default` 参数类型 `DT` 的联合, 可匹配此重载签名的调用示例如下:

```
1 max([1, 2, -3], key=abs, default=None) # 返回 -3
2 max([], key=abs, default=None) # 返回 None
```

15.2.2 重载 `max` 函数的启示

类型提示 (Type Hints) 允许 Mypy 用如下错误消息标记类似 `max([None, None])` 的调用。

```
1 mymax_demo.py:109: error: Value of type variable "LT" of "max" cannot be "None"
```

另一方面, 要编写这么多行注解才能支持类型检查器, 可能会阻碍人们编写像 `max` 这样方便灵活的函数。如果我不得不重新发明 `min` 函数, 那么我可以重构和重用 `max` 的大部分实现。但我必须复制并粘贴所有重载声明——尽管除了函数名称外, 它们与 `min` 函数完全相同。

我的朋友 João S. O. Bueno (我认识的最聪明的 Python 开发者之一) 在推特上写道:

尽管表达 `max` 的签名很难, 但它很容易印在人们的脑海中。在我看来, 注解标记的表达能力非常有限, 与 Python 没法比。

接下来, 开始研究 `TypedDict` 类型结构。它并不像我最初认为的那么有用, 但也不是一点用都没有。对 `TypedDict` 的实验证明了静态类型在处理动态结构 (如 JSON 数据) 时的局限性。

15.3 TypedDict



在处理 JSON API 响应等动态数据结构时, 使用 `TypedDict` 来防止出错很有诱惑力。但这里的示例清楚地表明, JSON 的正确处理必须在运行时完成, 而不是通过静态类型检查。要使用类型提示 (Type Hints) 对 JSON 类结构进行运行时检查, 请查看 PyPI 上的 `pydantic` 包。

Python 字典有时被用作记录, “键 (key)” 被用作字段名, “值 (value)” 被用作不同类型的字段值。例如, 考虑用 JSON 或 Python 描述一本书的记录:

```
1 {
2     "isbn": "0134757599",
3     "title": "Refactoring, 2e",
4     "authors": ["Martin Fowler", "Kent Beck"],
```

```

5     "pagecount": 478
6 }
```

在 Python 3.8 之前,没有太好的方法可以注解这样的记录,因为如“8.5.6 泛化映射<225页>”所述,映射类型中的所有值 (value) 都必须是相同类型。

下面是对类似前面的 JSON 对象的记录进行注解的两种蹩脚尝试:

- Dict[str, Any]
“值 (value)” 可以是任意类型。
- Dict[str, Union[str, int, List[str]]]
可读性差,而且未保留字段名与各自字段类型之间的关系;如 title 应该是 str,而不能是 int 或 List[str]。

“PEP 589 -TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys” 解决了这个问题。示例 15.4 展示了一个简单的 TypedDict。

</> 示例 15.4: books.py: 定义 BookDict

```

1 from typing import TypedDict
2
3 class BookDict(TypedDict):
4     isbn: str
5     title: str
6     authors: list[str]
7     pagecount: int
```

乍一看,typing.TypedDict 好像是一个数据类构建器,类似于“五 数据类构建器”中介绍的 typing.NamedTuple。语法上的相似性会引起误解。TypedDict 与数据类构建器千差万别。它只是为了类型检查工具而存在,在运行时没有任何作用。

typing.TypedDict 提供了 2 个功能:

- 用类似于 Class 的语法,为 dict 中各个“字段 (field)”的值提供类型提示 (Type Hints)。
- 通过一个构造函数告诉类型检查工具,dict 应具有指定“键 (key)”与指定类型的“值 (value)”。

在运行时,TypedDict 构造函数 (如 BookDict) 相当于一种安慰剂,其作用与使用同样实参调用 dict 构造函数相同。

事实上,BookDict 创建的是一个普通的 dict,这也意味着:

- 伪类声明中的“字段 (field)”不会创建实例属性。
- 不能为“字段 (field)”指定初始化的默认值。
- 不允许定义方法。

下面看一下 BookDict 在运行时的行为 (如示例 15.5 所示)。

</> 示例 15.5: 使用 BookDict

```

1 >>> from books import BookDict
2 >>> pp = BookDict(title='Programming Pearls', ❶
3 ...           authors='Jon Bentley', ❷
4 ...           isbn='0201657880', ❸
5 ...           pagecount=256) ❹
```

```
6  >>> pp
7  {'title': 'Programming Pearls', 'authors': 'Jon Bentley', 'isbn': '0201657880', 'pagecount': 256}>>> type(pp)
8  <class 'dict'>
9  >>> pp.title
10 <Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 AttributeError: 'dict' object has no attribute 'title'
13 >>> pp['title']
14 'Programming Pearls'
15 >>> BookDict.__annotations__
16 {'isbn': <class 'str'>, 'title': <class 'str'>, 'authors': typing.List[str], 'pagecount': <class 'int'>}
```

- ❶ 可以像 dict 构造函数那样, 传入关键字参数来调用 BookDict; 也可以传入一个包含 dict 字面量的 dict 参数。
- ❷ 哇呀, 我忘了“authors”的值应该是一个 list (如“[示例 15.4<424页>](#)”所示)。不过, [渐进式类型系统 \(Gradual Type System\)](#) 意味着“在运行时不会检查类型”。
- ❸ 调用 BookDict 得到的结果是一个普通的字典 ...
- ❹ ... 因此, 无法使用 object.field 表示法读取数据。
- ❺ 类型提示位于 BookDict.__annotations__ 中, 而不是在实例 pp 中。

如果没有类型检查工具, TypedDict 的作用与代码注释一样, 可以为阅读代码的人提供些许帮助, 仅此而已。相比之下, “[五 数据类构建器](#)”探讨的类构建器, 即使不使用类型检查工具, 也是很有用的。因为它们会在运行时生成或增强一个可以实例化的自定义类。它们还提供了“[表 5.1<135页>](#)”中列出的多个有用的方法或函数。

[示例 15.6](#) 构建了一个有效的 BookDict, 并对其进行了一些操作。这展示了 TypedDict 如何让 Mypy 捕获错误, 如 [示例 15.7](#) 所示。

</> [示例 15.6: demo_books.py: 对 BookDict 进行的合法和非法操作](#)

```
1  from books import BookDict
2  from typing import TYPE_CHECKING
3
4  def demo() -> None:      ❶
5      book = BookDict(      ❷
6          isbn='0134757599',
7          title='Refactoring, 2e',
8          authors=['Martin Fowler', 'Kent Beck'],
9          pagecount=478
10     )
11     authors = book['authors'] ❸
12     if TYPE_CHECKING:        ❹
13         reveal_type(authors) ❺
14     authors = 'Bob'          ❻
15     book['weight'] = 4.2
16     del book['title']
```

```

17
18 if __name__ == '__main__':     demo()

```

- ❶ 记得添加返回值类型,这样 Mypy 就不会忽略该函数。
- ❷ 这是一个有效的 BookDict:所有“键(key)”都存在,而且“值(value)”也都是正确的类型。
- ❸ Mypy 将根据 BookDict 中“authors”键的注解,推断 authors 的类型。
- ❹ typing.TYPE_CHECKING 只有在对程序进行类型检查时才为 True。在运行时,它总是 false。
- ❺ 前面的 if 语句阻止了在运行时调用 reveal_type(authors)。reveal_type 不是 Python 运行时函数,而是 Mypy 提供的调试工具,因此无需使用 import 导入。输出见 [示例 15.7](#)。
- ❻ demo 函数的最后 3 行是非法的,它们将导致 [示例 15.7](#) 中的错误信息。

对 [示例 15.6](#) 中的 demo_books.py 执行类型检查,将得到如 [示例 15.7](#) 所示的输出:

```

</> 示例 15.7: 类型检查 demo_books.py
1  …/typeddict/ $ mypy demo_books.py
2  demo_books.py:13: note: Revealed type is 'built-ins.list[built-ins.str]' ❶
3  demo_books.py:14: error: Incompatible types in assignment
4          (expression has type "str", variable has type "List[str]") ❷
5  demo_books.py:15: error: TypedDict "BookDict" has no key 'weight' ❸
6  demo_books.py:16: error: Key 'title' of TypedDict "BookDict" cannot be deleted ❹
7  Found 3 errors in 1 file (checked 1 source file)

```

- ❶ 这条 note 信息是 reveal_type(authors) 的结果。
- ❷ authors 变量的类型是从初始化该变量的 book['authors'] 表达式的类型推断出来的。你不能将一个 str 赋值给一个类型为 List[str] 的变量。类型检查工具通常不允许改变变量的类型²。
- ❸ 无法为 BookDict 定义中不存在的“键(key)”赋值。
- ❹ 无法删除 BookDict 定义中存在的“键(key)”。

现在,看一下在函数签名中使用 BookDict,以对函数调用做类型检查。

假设需要根据图书记录生成类似于如下格式的 XML:

```

1 <BOOK>
2   <ISBN>0134757599</ISBN>
3   <TITLE>Refactoring, 2e</TITLE>
4   <AUTHOR>Martin Fowler</AUTHOR>
5   <AUTHOR>Kent Beck</AUTHOR>
6   <PAGECOUNT>478</PAGECOUNT>
7 </BOOK>

```

如果要编写 MicroPython 代码嵌入到微型微控制器中,可以编写类似 [示例 15.8](#) 所示的函数³。

```

</> 示例 15.8: books.py:to_xml 函数
1 AUTHOR_ELEMENT = '<AUTHOR>{}</AUTHOR>'

```

² 截至 2020 年 5 月,pytype 允许这样做。但它的 FAQ 中说,未来将不允许这样做。请参阅 pytype FAQ 中的问题:“为什么 pytype 不能捕获到我改变了注解变量的类型?”

³ 我更喜欢使用 lxml 软件包来生成和解析 XML:它容易上手、功能齐全、速度快。遗憾的是,lxml 和 Python 自带的 ElementTree 并不适合用在内存有限的微型控制器中。

```

2
3 def to_xml(book: BookDict) -> str: ①
4     elements: list[str] = [] ②
5     for key, value in book.items():
6         if isinstance(value, list): ③
7             elements.extend(
8                 AUTHOR_ELEMENT.format(n for n in value)) ④
9         else:
10            tag = key.upper()
11            elements.append(f'<{tag}>{value}</{tag}>')
12    xml = '\n\t'.join(elements)
13    return f'<BOOK>\n\t{xml}\n</BOOK>'

```

- ① 该示例的要点是:在函数签名中使用 BookDict。
- ② 一开始为空的容器类型通常需要注解。否则,Mypy 无法推断出元素的类型⁴。
- ③ Mypy 能理解 isinstance 检查,并在此代码块中将 value 视为 list。
- ④ 最初,用 key == 'authors' 作为这个代码块的 if 保护条件时, Mypy 报错: "object" has no attribute "_iter_”,因为它推断 book.items() 的返回值类型是 object,而 object 不支持生成器表达式所需的方法。而用 isinstance 就不会报错,因为 Mypy 知道 value 在这个代码块中是一个 list。

示例 15.9 展示了一个解析 JSON 字符串并返回 BookDict 的函数。

</> 示例 15.9: books_any.py:from_json 函数

```

1 def from_json(data: str) -> BookDict:
2     whatever = json.loads(data) ①
3     return whatever ②

```

- ① json.loads() 返回值类型为 Any⁵。
- ② 可以返回 Any 类型的任何内容,因为 Any 与每种类型都相容 (Consistent-with),包括声明的返回值类型 BookDict。

示例 15.9 中的 ② 要特别注意: Mypy 不会标记那行代码有问题,但是在运行时 whatever 的值可能不符合 BookDict 结构——甚至可能根本就不是 dict。

如果使用 --disallow-any-expr 运行 Mypy,它就会抱怨 from_json 主体中的 2 行内容:

```

…/typeddict/ $ mypy books_any.py --disallow-any-expr
books_any.py:30: error: Expression has type "Any"
books_any.py:31: error: Expression has type "Any"
Found 2 errors in 1 file (checked 1 source file)

```

上述代码段中提到的第 30 和 31 行是 from_json 函数的主体。为了消除这两个类型错误,可以在 whatever 变量的初始化语句中添加类型提示,如示例 15.10 所示。

</> 示例 15.10: books.py:带变量注解的 from_json 函数

⁴Mypy 文档在“Common issues and solutions”页面的“Types of empty collections”部分讨论了这一问题。

⁵自 2016 年以来,Brett Cannon、Guido van Rossum 等人一直讨论如何为 json.loads() 添加类型提示 (Type Hints)。详见“Mypy Issue#182:Define a JSON type”。

```

1 def from_json(data: str) -> BookDict:
2     whatever: BookDict = json.loads(data) ❶
3     return whatever ❷

```

❶ 当立即将 Any 类型的表达式分配给具有类型提示 (Type Hints) 的变量时, 即使使用 `-disallow-any-expr` 选项, Mypy 也不会报错。

❷ 现在, `whatever` 的类型是 `BookDict`, 即声明的返回值类型。



不要被 [示例 15.10](#) 中的代码所欺骗, 产生一种错误的类型安全意识! 在静态代码的情况下, 类型检查器无法预测 `json.loads()` 将返回类似 `BookDict` 结构的内容。只有运行时验证才能保证这一点。

静态类型检查无法防止与代码自身不确定性有关的错误, 比如 `json.loads()`, 它在运行时会构建不同类型的 Python 对象, 正如[示例 15.11](#)、[示例 15.12](#)和[示例 15.13](#)所示。

</> [示例 15.11](#): `demo_not_book.py`: 返回一个无效的 `BookDict`, 但对 `to_xml` 有效

```

1 from books import to_xml, from_json
2 from typing import TYPE_CHECKING
3
4 def demo() -> None:
5     NOT_BOOK_JSON = """
6         {"title": "Andromeda Strain",
7          "flavor": "pistachio",
8          "authors": true}
9         """
10
11     not_book = from_json(NOT_BOOK_JSON) ❶
12     if TYPE_CHECKING: ❷
13         reveal_type(not_book)
14         reveal_type(not_book['authors'])
15
16     print(not_book) ❸
17     print(not_book['flavor']) ❹
18
19     xml = to_xml(not_book) ❺
20     print(xml) ❻
21
22 if __name__ == '__main__':
23     demo()

```

❶ 此行不会生成有效的 `BookDict`——请参阅 `NOT_BOOK_JSON` 的内容。

❷ 让 MyPy 揭示几种类型。

❸ 这里应该没问题:`print` 可以处理 `object` 等各种类型。

❹ `BookDict` 没有 “`flavor`” 键, 但 JSON 源中有, 会发生什么情况?

❺ 记住签名:`def to_xml(book: BookDict) -> str`。

❻ XML 输出结果是什么样的?

现在,用 Mypy 工具对 demo_not_book.py 执行检查,如示例 15.12。

</>示例 15.12: demo_not_book.py 的 Mypy 报告(为便于阅读,重新格式化)

```

1  .../typeddict/ $ mypy demo_not_book.py
2  demo_not_book.py:12: note: Revealed type is
3      'TypedDict('books.BookDict', {'isbn': built-ins.str,
4          'title': built-ins.str,
5          'authors': built-ins.list[built-ins.str],
6          'pagecount': built-ins.int})'
7  demo_not_book.py:13: note: Revealed type is 'built-ins.list[built-ins.str]' ①
8  demo_not_book.py:16: error: TypedDict "BookDict" has no key 'flavor' ②
9  Found 1 error in 1 file (checked 1 source file) ③

```

- ① 揭示的类型是**名义类型 (Nominal Typing)**,而不是 not_book 的运行时内容。
- ② 同 BookDict 中定义的类型一样,这是 not_book['authors'] 的**名义类型 (Nominal Typing)**。而不是运行时类型。
- ③ 此错误针对 print(not_book['flavor']) 行:“flavor”键在**名义类型 (Nominal Typing)**中不存在。

现在,运行 demo_not_book.py,得到的输出如示例 15.13 所示:

</>示例 15.13: 运行 demo_not_book.py 的输出

```

1  .../typeddict/ $ python3 demo_not_book.py
2  {'title': 'Andromeda Strain', 'flavor': 'pistachio', 'authors': True}  \ding{202}
3  pistachio  \ding{203}
4  <BOOK>  \ding{204}
5      <TITLE>Andromeda Strain</TITLE>
6      <FLAVOR>pistachio</FLAVOR>
7      <AUTHORS>True</AUTHORS>
8  </BOOK>

```

- ① 这其实不是 BookDict 结构。
- ② not_book['flavor'] 的值
- ③ to_xml 接收一个 BookDict **实参 (Argument)**,但没有相关的运行时检查:即无法对函数的输入输出做出限制。

如示例 15.13 所见,demo_not_book.py 的输出没有实际意义,但在运行时没有报错。由此可知,在使用 TypedDict 处理 JSON 数据时,无法得到太多的类型安全保障。

如果以**鸭子类型 (Duck Typing)**的角度来看“示例 15.8<426页>”中 to_xml 的代码,则参数 book 必须提供一个 items() 方法,该方法会返回一个类似 (key, value) 的可迭代元组,其中:

- key 必须有 .upper() 方法。
- value 可以是任何内容。

本节演示的重点是:在处理动态结构(如 JSON 或 XML 等)的数据时,TypedDict 绝对不能替代运行时的数据验证。如要此类数据验证,请使用 **pydantic**。

TypedDict 还有其他更多功能,包括支持可选键、有限形式的继承以及另一种声明语法。如想进一步了解,

请查看 “[PEP 589 - TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys](#)”

现在,让我们将注意力转向一个最好避免使用,但有时又无法避免的函数:`typing.cast`。

15.4 类型校正

任何类型系统都不是完美的,静态类型检查器 `typeshed` 项目中的[类型提示 \(Type Hints\)](#),以及具有[类型提示 \(Type Hints\)](#)的第三方包都不例外。

特殊函数 `typing.cast()` 提供了一种方法,可以处理类型检查故障或代码中无法修复的不正确[类型提示 \(Type Hints\)](#)。[Mypy 0.930 文档](#)指出:

类型校正用于消除类型检查器发出的虚假警告,并在类型检查器无法完全理解正在发生的情况时,为类型检查器提供一些帮助。

在运行时,`typing.cast()` 绝对不执行任何操作。这是它的实现(见 `Lib/typing.py` 1340 行):

```

1 def cast(typ, val):
2     """Cast a value to a type.
3     This returns the value unchanged. To the type checker this
4     signals that the return value has the designated type, but at
5     runtime we intentionally don't check anything (we want this
6     to be as fast as possible).
7     """
8     return val

```

[PEP 484](#) 要求类型检查器要“盲目地相信”`cast` 中声明的类型。[PEP 484](#) 中的“[Casts](#)”一节给出了一个例子,说明类型检查器需要 `cast` 提供的指引:

```

1 from typing import cast
2
3 def find_first_str(a: list[object]) -> str:
4     index = next(i for i, x in enumerate(a) if isinstance(x, str))
5     # We only get here if there's at least one string
6     return cast(str, a[index])

```

对生成器表达式执行 `next()` 调用时,要么返回 `str` 项的索引,要么引发 `StopIteration` 异常。因此,如果没有异常,`find_first_str` 将始终返回一个 `str`,而且 `str` 是声明的返回值类型。

但如果最后一行只是 `return a[index]`,[Mypy](#) 会推断返回值类型为 `object`,因为参数 `a` 被声明为 `list[object]`。因此,需要使用 `cast()` 来引导 [Mypy](#)⁶。

下面是另一个使用 `cast` 的示例,这次是为了纠正 Python 标准库中过时的[类型提示 \(Type Hints\)](#)。在“[示例 21.12<640页>](#)”中,我创建了一个 `asyncio` 服务器对象,并想获得服务器正在监听的地址。我编写了这样一行代码:

```
addr = server.sockets[0].getsockname()
```

⁶示例中故意使用 `enumerate` 是为了迷惑类型检查器。如果直接生成字符串(不使用 `enumerate` 生成索引),[Mypy](#) 则可以正确分析,也就无需 `cast()` 的指引。

但是, Mypy 报告了如下错误:

```
1 Value of type "Optional[List[socket]]" is not indexable
```

2021 年 5 月, [typeshed 项目](#) 中 `Server.sockets` 的类型提示对 Python 3.6 有效, `sockets` 属性可以是 `None`。但在 Python 3.7 中, `sockets` 变成了一个特性 (property), 其读值 (getter) 方法总是返回一个 `list` (服务器没有 `sockets` 时为空)。从 Python 3.8 开始, 该读值 (getter) 方法返回一个 `tuple` (作为不可变序列使用)。

由于我现在无法修复 [typeshed](#)⁷, 因此我为其添加了如下所示的 `cast` 调用:

```
1 from asyncio.trsock import TransportSocket
2 from typing import cast
3
4 # ... 省略多行 ...
5
6 socket_list = cast(tuple[TransportSocket, ...], server.sockets)
7 addr = socket_list[0].getsockname()
```

为了编写这个 `cast` 调用, 我花了好几个小时。因为不只要弄清问题, 还要阅读 `asyncio` 源码以找到 `socket` 的正确类型——`socket` 来自无文档的 `asyncio.trsock` 模块中的 `TransportSocket` 类。为了提高可读性, 我还添加了 2 条导入语句和 1 行代码⁸。现在, 代码更安全了。

细心的读者可能会注意到, 如果 `sockets` 为空, `sockets[0]` 可能会引发 `IndexError`。不过, 以我对 `asyncio` 包的理解, “[示例 21.12<640页>](#)” 不会出现这种情况, 因为在我读取服务器的 `sockets` 属性时, 服务器已经做好接受连接的准备, 因此 `sockets` 不可能为空。况且, `IndexError` 还是一个运行时错误, 即便是 `print([][0])` 这么明显的运行时问题, Mypy 不是也发现不了么?



不要轻易使用 `cast` 来让 Mypy 保持沉默, 因为当 Mypy 报告错误时, 通常都是有原因的。经常使用 `cast` 也是一种 [代码异味 \(Code Smell\)](#)。比如, 团队可能用了错误的 [类型提示 \(Type Hints\)](#), 或者代码库中存在低质量的依赖关系。

尽管 `cast` 有这样或那样的缺点, 但还是有其合理的用途。下面是 Guido van Rossum 写的关于 `cast` 的文章:

偶尔调用 `cast()` 或用 `# type:` 忽略注释, 有何不妥?^a

^a 摘自 2020 年 5 月 19 日发送至 [typing-sig 邮件列表](#) 的信息。

完全禁止使用 `cast` 是不明智的, 主要原因是其他变通方法更糟:

- “`# type: ignore`” 提供的信息量更少⁹。
- 使用 `Any` 具有传染性: 由于 `Any` 与所有类型 [相容 \(Consistent-with\)](#), 滥用 `Any` 可能会导致类型推断

⁷ 我向 [typeshed](#) 报告了这个问题: “[Issues #5535: Wrong type hint for asyncio.base_events.Server.sockets attribute](#)”。Sebastian Ritter很快就修复了这个问题。不过, 我还是决定保留这个示例, 因为它说明了 `cast` 的一个常见用例, 而且我编写的 `cast` 是无害的。

⁸ 老实说, 我最初在 `server.sockets[0]` 行后面加上了 `#type:ignore` 注释。因为我在 `asyncio` 包的[文档](#)与一个[测试用例](#)发现了类似的行。所以, 我怀疑问题并不是由我的代码引起的。

⁹ 语法 “`# type:ignore[code]`” 允许指定要消除的 Mypy 错误代码, 但这些代码并不总是易于解释。请参阅 [Mypy 文档](#) 的 “[Error codes](#)” 页面。

产生连带效应,削弱类型检查工具检测错误的能力。

当然,并不是所有的类型错误都可以通过 `cast` 来修复。有时需要 “`# type: ignore`”,有时需要 `Any`,有时甚至需要去掉函数的 [类型提示 \(Type Hints\)](#)。

接下来,讨论在运行时,类型提示 (Type Hints) 的使用。

15.5 在运行时,读取类型提示

在 `import` 导入时,Python 会读取函数、类和模块中的 [类型提示 \(Type Hints\)](#),并将它们存储在名为 `__annotations__` 的属性中。下面以 [示例 15.14](#) 中的 `clip` 函数为例,进行说明¹⁰。

</> [示例 15.14: clipannot.py: 函数 clip 带注释的签名](#)

```
1 def clip(text: str, max_len: int = 80) -> str:
```

类型提示 (Type Hints) 以 `dict` 的形式,存储在函数的 `__annotations__` 属性中(如下所示):

```
1 >>> from clip_annot import clip
2 >>> clip.__annotations__
3 {'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

在[示例 15.14](#)中, `return` 键被映射到 `->` 符号后面的返回值类型提示。

请注意,Python 解释器在导入时会对 [类型提示 \(Type Hints\)](#) 进行求解,如同求解参数默认值一样。因此,类型提示 (Type Hints) 中的值是 Python 类 `str` 和 `int`,而不是字符串 “`str`” 和 “`int`”。在导入时,求解类型提示是截至 Python 3.10 的标准行为。如果 [PEP 563](#) 或 [PEP 649](#) 成为标准行为,此情况可能有变。

15.5.1 运行时的注解问题

类型提示 (Type Hints) 使用量的增加引发了 2 个问题:

- 当使用许多 [类型提示 \(Type Hints\)](#) 时,导入模块会占用更多的 CPU 和内存。
- 在引用尚未定义的类型时,需要使用字符串而不是实际类型。

这 2 个问题多是有内在原因的。第一个问题的原因如前文所述,类型提示 (Type Hints) 在执行导入时由解释器求解,并存储在 `__annotations__` 属性中。现在,重点分析第二个问题。

为了解决 [前向引用 \(Forward Reference\)](#) 的问题(类型提示需要引用同一模块下定义的类),有时需要将注解存储为字符串,而不是直接引用类名。然而,该问题在源码中的常见表象——类中的方法返回同一类的新对象,这看起来根本不像前向引用 (Forward Reference)。因为在 Python 完全求解类主体之前,类对象是未定义的。所以,类型提示 (Type Hints) 必须使用类名的字符串形式。这样做可以避免因类定义顺序而导致的问题。示例如下所示:

```
1 class Rectangle:
2     # ... 省略多行 ...
3     def stretch(self, factor: float) -> 'Rectangle':
4         return Rectangle(width=self.width * factor)
```

¹⁰ 我不会详细介绍 `clip` 的实现,但是如果你好奇的话,可以在 `clip_annot.py` ([15-more-types/clip_annot.py](#)) 中阅读整个模块。

截至 Python 3.10,涉及 [前向引用 \(Forward Reference\)](#) 的类型提示都必须写成字符串形式,这是标准做法。静态类型检查器从一开始就是为了处理这个问题而设计的。

但在运行时,如果编写代码读取 `stretch` 的 `return` 注解,得到的将是字符串“`Rectangle`”,而不是实际类型(`Rectangle`类)的引用。现在,你的代码需要设法确定得到的字符串的含义。

`typing` 模块 包括 3 个函数和 1 个归类为 `Introspection` 助手的类。其中,最重要的是 `get_type_hints`。该模块的部分文档指出:

```
typing.get_type_hints(obj, globalns=None, localns=None, include_extras=False)
```

... 得到的结果通常与 `obj.__annotations__` 的值相同。此外,以字符串字面量编码的 [前向引用 \(Forward Reference\)](#) 是通过在 `globals`、`locals` 和 [类型参数 \(Python 3.12 新增\)](#) 命名空间中求值来处理的...



从 Python 3.10 开始,应该使用新的 `inspect.get_annotations(...)` 函数来代替 `typing.get_type_hints`。然而,有些读者可能还没有使用 Python 3.10。所以,本节示例仍然使用 `typing.get_type_hints`,它在 Python 3.5 中添加了 `typing` 模块之后就可用。

“[PEP 563 –Postponed Evaluation of Annotations](#)”已获批准,使得无需再将注解写成字符串,并降低[类型提示 \(Type Hints\)](#)的运行时成本。“摘要”中的如下两句话描述了该 PEP 的主要目的:

本 PEP 建议修改函数注解和[变量注解 \(Variable Annotation\)](#),使其不在函数定义时进行求解。相反,它们以字符串形式保存在 `__annotations__` 中。

从 Python 3.7 开始,任何以如下 `import` 语句开头的模块都会按引文中的方式处理注解:

```
1 from __future__ import annotations
```

为了演示其效果,我将“[示例 15.14<432页>](#)”中的 `clip` 函数复制到 `clip_annot_post.py` 模块中,并在顶部增加了 `__future__ import annotations`。

在控制台中,导入 `clip_annot_post.py` 模块,读取函数 `clip` 的注解,得到的结果如下:

```
1 >>> from clip_annot_post import clip
2 >>> clip.__annotations__
3 {'text': 'str', 'max_len': 'int', 'return': 'str'}
```

如您所见,现在所有类型提示都是普通字符串,尽管在函数 `clip` 的定义(见“[示例 15.14<432页>](#)”)中并为将这些注解写成带引号的字符串。

`typing.get_type_hints` 函数能够解析许多[类型提示 \(Type Hints\)](#),包括 `clip` 中的类型提示:

```
1 >>> from clip_annot_post import clip
2 >>> from typing import get_type_hints
3 >>> get_type_hints(clip)
4 {'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

调用 `typing.get_type_hints` 可以获得真正的类型——即便在某些情况下,原始的[类型提示 \(Type Hints\)](#)是以带引号的字符串形式编写的。这是在运行时读取类型提示的推荐方法。

PEP 563 行为计划在 Python 3.10 中成为默认行为, 无需导入 `__future__`。然而, FastAPI 和 pydantic 的维护者提出了警告, 认为这一修改会破坏他们的代码, 因为这些代码在运行时依赖于类型提示, 无法可靠地使用 `typing.get_type_hints`。

在随后的 python-dev 邮件列表讨论中, Łukasz Langa (PEP 563 作者) 总结了函数 `get_type_hints` 的一些局限性:

... 结果表明, `typing.get_type_hints()` 有一定的局限, 在运行时使用通常开销较大, 并且更重要的是无法解析所有类型。最常见的问题是在非全局上下文 (例如, 内部类、函数内的类等) 中生成类型时, `typing.get_type_hints` 方法无法准确地解析类型。此外, 对于 [前向引用 \(Forward Reference\)](#) 的典型情况之一——具有接受或返回其自身类型对象的方法的类——如果使用类生成器, `typing.get_type_hints` 也无法正确处理。尽管我们可以通过一些技巧来解决这些问题, 但总体来说效果并不理想^a。

^a消息 “[PEP 563 in light of PEP 649](#)”, 发布于 2021 年 4 月 16 日。

Python 指导委员会决定将 PEP 563 作为默认行为推迟到 Python 3.11 或更高版本, 为开发者留出更多时间来提出解决方案, 以便既能解决 PEP 563 试图解决的问题, 又不会破坏运行时 [类型提示 \(Type Hints\)](#) 的广泛使用。[PEP 649 -Deferred Evaluation Of Annotations Using Descriptors](#) 是一种正在商讨的可行方案, 不过最终或许会给出其他折中方案。

综上所述, 截至 Python 3.10, 在运行时读取 [类型提示 \(Type Hints\)](#) 并不是 100% 可靠, 而且很可能在 2022 年发生变化。



大规模使用 Python 的公司希望从静态类型中获益, 但又不想为导入时的类型提示求解付出太多代价。静态检查发生在开发工作站和专用的 CI 服务器上, 但在生产容器中加载模块的频率和数量要高得多, 在大规模使用时, 这种代价不可忽视。这就造成了 Python 社区中的矛盾: 有人希望只将 [类型提示 \(Type Hints\)](#) 存储为字符串, 以降低加载成本; 有人则希望在运行时使用 [类型提示 \(Type Hints\)](#), 比如 FastAPI 和 pydantic 的创建者和用户, 他们更希望存储类型对象, 而不是求解这些注解, 这是一项具有挑战性的任务。

15.5.2 解决这个问题

鉴于目前的不稳定局势, 如果您需要在运行时读取类型提示, 我建议:

- 避免直接读取 `__annotations__`; 而应使用 `inspect.get_annotations` (从 Python 3.10 起) 或 `typing.get_type_hints` (从 Python 3.5 起)。
- 自己编写一个自定义函数, 作为 `inspect.get_annotations` 或 `typing.get_type_hints` 的封装器。在代码中调用该自定义函数, 这样当以后行为有变时, 只需修改这个自定义函数即可。

为了演示第 2 点, 下面是 “[示例 24.5<727页>](#)” 中定义的 `Checked` 类的第 1 行, 我们将在 “[二十四类元编程 \(Class Metaprogramming\)](#)” 研究该类。

```
1 class Checked:
2     @classmethod
3     def _fields(cls) -> dict[str, type]:
```

```
4     return get_type_hints(cls)
5     # ... 省略余下代码 ...
```

Checked._fields 类方法可以防止模块的其他部分直接依赖 `typing.get_type_hints`。如果将来 `get_type_hints` 的行为发生变化,需要额外的逻辑,或者你想用 `inspect.get_annotations` 取而代之。那么,只需修改 `Checked._fields` 方法即可,程序的其他部分不受影响。



鉴于对 [类型提示 \(Type Hints\)](#) 的运行时检查的持续讨论和修改建议,请务必阅读官方文档中的“[Annotations Best Practices](#)”。在通往 Python 3.11 的路上,该文档可能会被更新。该文档由 Larry Hastings 撰写,他是“[PEP 649 –Deferred Evaluation Of Annotations Using Descriptors](#)”的作者,PEP 649 是解决“[PEP 563 –Postponed Evaluation of Annotations](#)”所提出的运行时问题的替代方案。

本章其余各节将介绍 [泛型 \(Generic Type\)](#)。首先,介绍如何定义可由用户参数化的 [泛化类 \(Generic Classes\)](#)。

15.6 实现泛化类

“[示例 13.7<366页>](#)”中定义的 [抽象基类 \(ABCs\)](#) `Tombola`,为类似宾果机的类定义了接口。而“[示例 13.10<370页>](#)”中的 `LottoBlower` 类是该接口的一个具体实现。本节将研究 `LottoBlower` 的一个泛化版本,其用法如 [示例 15.15](#) 所示。

</> [示例 15.15: generic_lotto_demo.py](#): 使用一个泛化版的 `LottoBlower` 类

```
1 from generic_lotto import LottoBlower
2
3 machine = LottoBlower[int](range(1, 11)) ①
4
5 first = machine.pick() ②
6 remain = machine.inspect() ③
```

- ① 实例化一个泛化类,需要为其提供具体的类型参数,例如这里的 `int`。
- ② Mypy 会正确推断出 `first` 是一个 `int` 值 ...
- ③ ... 而 `remain` 是一个 `int` 元组。

此外,Mypy 还会报告参数化类型的违规情况,并提供有用的消息,如 [示例 15.16](#) 所示。

</> [示例 15.16: generic_lotto_errors.py](#): Mypy 报告的错误

```
1 from generic_lotto import LottoBlower
2
3 machine = LottoBlower[int]([1, .2]) ①
4 ## error: List item 1 has incompatible type "float";
5 ## expected "int"
6
7 machine = LottoBlower[int](range(1, 11))
```

```

8
9 machine.load('ABC')
10 ## error: Argument 1 to "load" of "LottoBlower" ②
11 ##     has incompatible type "str";
12 ##     expected "Iterable[int]"
13 ## note: Following member(s) of "str" have conflicts:
14 ## note:     Expected:
15 ## note:         def __iter__(self) -> Iterator[int]
16 ## note:     Got:
17 ## note:         def __iter__(self) -> Iterator[str]

```

- ❶ 在实例化 LottoBlower[int] 时, Mypy 会标记列表中的元素类型不能为 float。
 ❷ 在调用.load('ABC') 时, Mypy 解释了不能用 str 值的原因: str.__iter__ 会返回一个 Iterator[str], 但 LottoBlower[int] 需要一个 Iterator[int]。

泛化版 LottoBlower 类的实现,如 [示例 15.17](#) 所示。

```

</> 示例 15.17: generic_lotto.py: 泛化版的 LottoBlower 类

1 import random
2
3 from collections.abc import Iterable
4 from typing import TypeVar, Generic
5
6 from tombola import Tombola
7
8 T = TypeVar('T')
9
10 class LottoBlower(Tombola, Generic[T]): ❶
11
12     def __init__(self, items: Iterable[T]) -> None: ❷
13         self._balls = list[T](items)
14
15     def load(self, items: Iterable[T]) -> None: ❸
16         self._balls.extend(items)
17
18     def pick(self) -> T: ❹
19         try:
20             position = random.randrange(len(self._balls))
21         except ValueError:
22             raise LookupError('pick from empty LottoBlower')
23         return self._balls.pop(position)
24
25     def loaded(self) -> bool:
26         return bool(self._balls)
27
28     def inspect(self) -> tuple[T, ...]: ❺
29         return tuple(self._balls)

```

- ❶ 泛化类 (Generic Classes) 的声明通常要使用多重继承,因为需要子类化 Generic 类 以声明形式 (For-

mal) 类型参数 (这里的类型变量 T)。

- ② `__init__` 中的参数 `items` 类型是 `Iterable[T]`。当实例声明为 `LottoBlower[int]` 时, `Iterable[T]` 就变成了 `Iterable[int]`。
- ③ `load` 方法也有与 `__init__` 同样的约束。
- ④ 对于 `LottoBlower[int]`, 返回值类型 T 将变成 `int`。
- ⑤ 此方法未使用类型变量。
- ⑥ 最后, 返回的元组中项的类型都被设置为 T。



typing 模块文档 的“User-defined generic types”一节内容简短, 包含了一些示例, 而且还涵盖了此处未提及的一些细节。

现在, 已经了解了如何实现泛化类 (Generic Classes)。接下来, 定义一下与泛化类 (Generic Classes) 有关的术语。

15.6.1 泛型的基础术语

以下是我认为对研究泛型 (Generic Type) 比较有用的一些术语定义¹¹:

- **泛型 (Generic Type)**

具有一个或多个类型变量的类型。例如, `LottoBlower[T]` 与 `abc.Mapping[KT, VT]`。

- **形式 (Formal) 类型参数**

出现在泛型 (Generic Type) 类型声明中的类型变量。例如, `abc.Mapping[KT, VT]` 中的 `KT` 与 `VT`。

- **参数化 (Parameterized) 类型**

用实际 (Actual) 类型参数声明的类型。例如, `LottoBlower[int]` 与 `abc.Mapping[str, float]`。

- **实际 (Actual) 类型参数**

声明参数化 (Parameterized) 类型时, 为参数提供的实际类型。例如, `LottoBlower[int]` 中的 `int`, 以及 `abc.Mapping[str, float]` 中的 `str` 与 `float`。

下一节的主题是如何使泛型 (Generic Type) 更灵活, 介绍了协变 (covariance)、逆变 (contravariance) 和不变 (invariance) 等概念。

¹¹这些术语来自经典著作 “Effective Java, 3rd Ed” (Joshua Bloch)。其中的定义和示例均出自我的手笔。

15.7 型变



如果没有泛型 (Generic Type) 的使用经验, 本节可能是本书中最具挑战性的部分。型变 (variance) 的概念非常抽象, 其严谨的表述导致本节就像数学书中的内容一样晦涩难懂。

在实践中, 真正需要关注型变 (variance) 的用户, 主要都是一些代码库的作者。因为只有他们才可能需要支持新的泛化容器类型, 或者需要提供基于回调的 API。不过, 也可以支持不变 () 容器, 以降低复杂度——Python 标准库基本上就是这么做的。所以, 首次阅读本书时, 可跳过本节, 或者只阅读关于不变 (invariance) 类型的部分。

本书中首次接触 型变 (variance) 这一概念, 是在 “subsubsection 8.5.11.1<238页>” 一节。当时, 它被应用于参数化泛型的 Callable 类型。本节将扩展这个概念, 使其涵盖泛化容器类型, 利用现实中的具体事物来类比这个抽象的概念。

假设学校食堂规定只能安装果汁售货机 (简称“果汁机”), 而通用饮料售货机 (简称“饮料机”) 是不允许安装的¹²。因为饮料机可能提供苏打水, 而苏打水是被学校董事会明令禁止的¹³。

15.7.1 不变的自动售货机

下面尝试为学校食堂的规定建模: 定义一个泛化的 BeverageDispenser 类, 该类可根据饮料类型进行参数化。如 [示例 15.18](#) 所示。

</> [示例 15.18: invariant.py](#): 类型定义和 install 函数

```

1  from typing import TypeVar, Generic
2
3  class Beverage: ❶
4      """任何饮料"""
5
6  class Juice(Beverage):
7      """任何果汁"""
8
9  class OrangeJuice(Juice):
10     """美味的巴西橙汁"""
11
12 T = TypeVar('T') ❷
13
14 class BeverageDispenser(Generic[T]): ❸
15     """根据饮料类型进行参数化的自动售货机"""
16     def __init__(self, beverage: T) -> None:
17         self.beverage = Beverage
18

```

¹²我首次在《The Dart Programming Language》(Gilad Bracha 著) 一书的前言中, 看到关于用自助餐厅的饮料售货机来类比型变 (variance)。

¹³这可比禁止卖书要好多了!

```
19     def dispense(self) -> T:
20         return self.beverage
21     def install(dispenser: BeverageDispenser[Juice]) -> None: ④
22     """安装果汁自动售货机"""

```

- ① Beverage、Juice 和 OrangeJuice 构成一种类型层次结构（从超类到子类）。
- ② 简单的 TypeVar 声明。
- ③ BeverageDispenser 根据饮料类型进行参数化。
- ④ install 是一个模块全局函数。它的类型提示（Type Hints）强制执行只接受果汁机的规定。

根据 [示例 15.18](#) 中的定义，以下代码是合法的：

```
1 juice_dispenser = BeverageDispenser(Juice())
2 install(juice_dispenser)
```

但是，以下代码是不合法的：

```
1 beverage_dispenser = BeverageDispenser(Beverage())
2 install(beverage_dispenser)
3 ## mypy: Argument 1 to "install" has
4 ## incompatible type "BeverageDispenser[Beverage]"
5 ##           expected "BeverageDispenser[Juice]"
```

不接受可售卖任何饮料（Beverage）的售货机，因为食堂规定自动售货机只能售卖果汁（Juice）。

但是，令人不解的是，以下代码也不合法：

```
1 orange_juice_dispenser = BeverageDispenser(OrangeJuice())
2 install(orange_juice_dispenser)
3 ## mypy: Argument 1 to "install" has
4 ## incompatible type "BeverageDispenser[OrangeJuice]"
5 ##           expected "BeverageDispenser[Juice]"
```

专售橙汁（OrangeJuice）的售货机也不允许安装，只允许安装果汁机（BeverageDispenser[Juice]）。

当 BeverageDispenser[OrangeJuice] 与 BeverageDispenser[Juice] 不兼容（尽管 OrangeJuice 是 Juice 的子类型）时，用类型术语说：“BeverageDispenser(Generic[T]) 是不变（invariant）的”。

Python 的可变容器类型（如 list 和 set）是不变的。[“示例 15.17”](#) 中的 LottoBlower 类也是不变的。

15.7.2 协变的自动售货机

如果想要灵活一些，将售货机建模为可接受某种饮料类型及其子类型的泛化类（Generic Classes），则必须让该它支持协变（covariance）。支持协变（covariance）的泛化类（Generic Classes）BeverageDispenser 如 [示例 15.19](#) 所示：

</> [示例 15.19: covariant.py](#): 类型定义和 install 函数

```
1 T_co = TypeVar('T_co', covariant=True)
2
```

```

3  class BeverageDispenser(Generic[T_co]):           ❷
4      def __init__(self, beverage: T_co) -> None:    self.beverage = beverage
5
6      def dispense(self) -> T_co:
7          return self.beverage
8
9  def install(dispenser: BeverageDispenser[Juice]) -> None:  ❸
10     """安装果汁自动售货机"""

```

- ❶ 在声明类型变量时, 设置 covariant=True。后缀 _co 是 [typeshed 项目](#)采用的一种约定, 表明这是**协变 (covariance)**的类型参数。
- ❷ 使用 T_co 参数化特殊的 [Generic](#) 类。
- ❸ `install` 的**类型提示 (Type Hints)**与“[示例 15.18<438页>](#)”中的相同。

如下代码可正常运行, 因为现在对可**协变 (covariance)**的 `BeverageDispenser` 类来说, 果汁机 (`Juice`) 与橙汁机 (`OrangeJuice`) 都是有效的自动售货机。:

```

1 juice_dispenser = BeverageDispenser(Juice())
2 install(juice_dispenser)
3
4 orange_juice_dispenser = BeverageDispenser(OrangeJuice())
5 install(orange_juice_dispenser)

```

但是, 不接受可售卖任何饮料 (`Beverage`) 的售货机, 如下所示:

```

1 beverage_dispenser = BeverageDispenser(Beverage())
2 install(beverage_dispenser)
3 ## mypy: Argument 1 to "install" has
4 ## incompatible type "BeverageDispenser[Beverage]"
5 ##           expected "BeverageDispenser[Juice]"

```

这就是**协变 (covariance)**: [参数化售货机的子类型关系](#), 与**类型参数的子类型关系**, 二者的变化方向相同¹⁴。

15.7.3 逆变的垃圾桶

现在, 对食堂配备垃圾桶的规则进行建模。假设食物与饮料的包装材料都可生物降解, 剩饭剩菜与一次性餐具也都可生物降解。食堂内的垃圾桶必须适合存放可生物降解的垃圾。

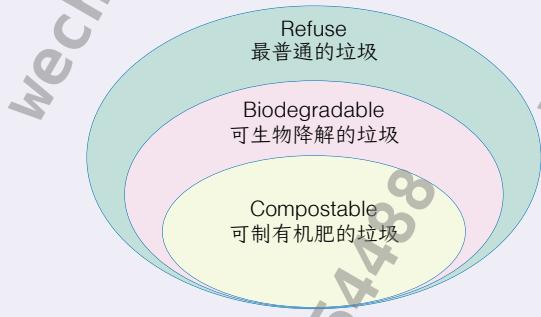
¹⁴译者注: 这句话是在说, 如果你有一种带有参数的类型 (比如 `List[T]`), 而且这个参数 T 代表的是某种类型 (比如列表里的元素类型), 那么当这个类型参数 T 的不同子类型之间有关系时, 这个参数化类型 `List[T]` 的子类型关系会以相同的方式变化。举个例子, 如果你有一个装水果的盒子, 可以装香蕉或苹果; 如果苹果是水果的子类型, 那么装有苹果的盒子也会是装有水果的盒子的子类型。即给定支持**协变 (covariance)**的泛化类 `C(T)`, 如果 `t1` 是 `t2` 的子类型, 那么 `C(t1)` 也是 `C(t2)` 的子类型; 反相, 如果 `t1` 是 `t2` 的父类型, 那么 `C(t1)` 也是 `C(t2)` 的父类型。即支持协变的类 `C(T)` 与参数化类型 T 会同向变化。



为简化教学,需要对此示例做一些约定,以整齐的层次结构对垃圾进行分类:

- Refuse 是最普通的垃圾类型,所有垃圾可归属于 Refuse。
- Biodegradable 是一种特定类型的垃圾,随着时间推移可被生物降解。某些 Refuse 垃圾不属于 Biodegradable(即无法被生物降解)。
- Compostable 是一种特定类型的 Biodegradable(可生物降解垃圾),可以在堆肥箱或堆肥设施中转化为有机肥料。根据定义,并非所有可生物降解垃圾(Biodegradable)都是可制成肥料的垃圾(Compostable)。

Refuse、Biodegradable、Compostable 之间的层次结构关系,如下图所示:



为了对食堂可接受的垃圾桶规则进行建模,需要引入逆变(contravariance)的概念,如示例 15.20 所示。

</> 示例 15.20: contravariant.py:类型定义与 install 函数

```
1  from typing import TypeVar, Generic
2
3  class Refuse:
4      """任意类型的垃圾"""
5
6  class Biodegradable(Refuse):
7      """可生物降解的垃圾"""
8
9  class Compostable(Biodegradable):
10     """可制成有机肥的垃圾"""
11
12 T_contra = TypeVar('T_contra', contravariant=True) ❶
13
14 class TrashCan(Generic[T_contra]):
15     def put(self, refuse: T_contra) -> None:
16         """在倾倒之前, 存放垃圾"""
17
18     def deploy(trash_can: TrashCan[Biodegradable]): ❷
19         """放置一个垃圾桶, 用于存放可生物降解的垃圾"""
20
```

❶ 一种废弃物的类型层次结构:Refuse 是最普通的类型,Compostable 是最具体的类型。

❷ 按约定,T_contra 用于表示逆变(contravariance)类型变量。

❸ TrashCan 对废弃物的类型实行逆变(contravariance)。

根据上述定义,如下类型的垃圾桶是可接受的:

```

1 bio_can: TrashCan[Biodegradable] = TrashCan()
2 deploy(bio_can)
3 trash_can: TrashCan[Refuse] = TrashCan()
4 deploy(trash_can)

```

最普通的 `TrashCan[Refuse]` 是可以接受的, 因为它可以存放任何类型的垃圾, 包括可生物降解垃圾 (`Biodegradable`)。但是, `TrashCan[Compostable]` 不可接受, 因为它不能存放可生物降解垃圾 (`Biodegradable`)。如下所示:

```

1 compost_can: TrashCan[Compostable] = TrashCan()
2 deploy(compost_can)
3 ## mypy: Argument 1 to "deploy" has
4 ## incompatible type "TrashCan[Compostable]"
5 ##           expected "TrashCan[Biodegradable]"

```

下面总结一下刚刚见到的概念。

15.7.4 型变总结

型变 (variance) 是一种微妙的性质。下面会总结不变 (invariant)、协变 (covariance)、逆变 (contravariance) 类型的概念, 并提供一些经验法则, 用于推断型变 (variance) 种类。

15.7.4.1 不变类型

无论实际 (Actual) 类型参数之间的类型关系如何, 只要 2 个参数化 (Parameterized) 类型之间不存在父子关系, 泛化类型 L 就是不变 (invariant) 的。换句话说, 如果 L 是不变 (invariant) 的, 则 $L[A]$ 与 $L[B]$ 就不存在父子关系。二者在两个方向都是不相容 (Consistent-with) 的。

如前所述, Python 的可变容器类型默认是不变 (invariant) 的。`list` 类型就是一个很好的例子: 虽然 `int` 是 `float` 的子类型, 但 `list[int]` 与 `list[float]` 不相容 (Consistent-with), 反之亦然。

一般来说, 如果形式 (Formal) 类型参数出现在方法的参数注解中, 并且相同的参数还出现在方法的返回类型中, 则该参数必须是不变的, 以确保在更新容器和从容器中读取时的类型安全。

例如, 如下是 `typeshed` 项目 上内置类型 `list` 的部分类型提示 (详见 `stdlib/builtins.pyi` 743 行):

```

1 class list(MutableSequence[_T], Generic[_T]):
2     @overload
3     def __init__(self) -> None: ...
4     @overload
5     def __init__(self, iterable: Iterable[_T]) -> None: ...
6     # ... 省略部分行 ...
7     def append(self, __object: _T) -> None: ...
8     def extend(self, __iterable: Iterable[_T]) -> None: ...
9     def pop(self, __index: int = ...) -> _T: ...
10    # ...

```

请注意, `_T` 既出现在 `__init__`、`append` 和 `extend` 等方法的参数注解中, 又是 `pop` 方法的返回值类型。如果这样的类在 `_T` 中是协变 (covariance) 或逆变 (contravariance) 的, 则无法保证该类的类型安全。

15.7.4.2 协变类型

考虑 2 种类型 A 和 B, 其中 B 与 A 相容 (Consistent-with), 并且它们都不是 Any。有些作者用 `<:` 和 `:>` 符号来表示类型之间的关系, 如下所示:

- $A :> B$
A 是 B 的超类型, 或者 A 与 B 相同。
- $B <: A$
B 是 A 的子类型, 或者 B 与 A 相同。

给定类型 $A :> B$, 如果 $C[A] :> C[B]$, 则泛型 C 是协变 (covariant) 的。注意, 此处符号 `:>` 的方向是相同的, 即 A 与 C(A) 都在 `:>` 的同侧。协变 (covariance) 的泛型 (Generic Type) 遵循实际 (Actual) 类型参数的子类型关系。

不可变容器可以是协变的。typing.FrozenSet 文档中用具有约定名称 (`T_co`) 的类型变量将 FrozenSet 类标记为可协变 (covariance) 的。如下所示:

```
1 class FrozenSet(frozensec, AbstractSet[T_co]):
```

将符号 `:>` 应用到参数化 (Parameterized) 类型, 可以得到:

```
1 float :> int
2 frozensec[float] :> frozensec[int]
```

迭代器是协变泛型的另一个例子: 迭代器与 frozensec 这种只读容器不同, 迭代器只产生输出。任何期望 `abc.Iterator[float]` 产生 `float` 数值的代码, 都可以安全地使用 `abc.Iterator[int]` 产生 `int` 数值。可调用类型的返回值类型也可以是协变 (covariance) 的, 原因与此类似。

15.7.4.3 逆变类型

给定类型 $A :> B$, 如果 $K[A] <: K[B]$, 则泛型 K 是逆变 (contravariant) 的。逆变 (contravariance) 的泛型 (Generic Type) 反转了实际 (Actual) 类型参数的子类型关系。

TrashCan 类就是逆变 (contravariance) 的一个例子。

```
1 Refuse :> Biodegradable
2 TrashCan[Refuse] <: TrashCan[Biodegradable]
```

逆变 (contravariance) 的容器类型通常是一种只写数据结构, 也称为 “sink”。标准库中没有这样的容器类型, 但是有一些类型的类型参数是逆变 (contravariance) 的。

Callable[[ParamType, …], ReturnType] 中的参数类型是逆变 (contravariance) 的, 不过 ReturnType 是协变 (covariance) 的 (详见 “8.5.11.1 Callable 类型的型变<238页>”)。此外, Generator、Coroutine 和 AsyncGenerator 都有一个可逆变 (contravariance) 的类型参数。Generator 类将在 “17.13.3 经典协程的泛型注解<524页>” 中进行介绍, 而 Coroutine 与 AsyncGenerator 将在 “二十一 异步编程<619页>” 中介绍。

以上关于型变 (variance) 的讨论, 重点在于逆变 (contravariance) 的形式 (Formal) 类型参数定义了用于调用对象或向对象发送 (send) 数据的实参类型, 而协变 (covariance) 的形式 (Formal) 类型参数定义了对象产生的输出类型——根据对象的不同, 可以是产出值的类型或者是返回值的类型。关于 “发送 (send)” 与 “产出 (yield)” 的含义, 详见 “17.13 经典协程<518页>”。

根据“可逆变的输出,可协变的输入”,可以得到一些有用的指导原则。

15.7.4.4 型变经验法则

最后,根据以下经验法则,可以推知具体的型变 (variance) 种类:

- 如果一个形式 (Formal) 类型参数定义的是从对象中产生的数据类型,那么该形式类型参数可能是协变 (covariance) 的。
- 如果一个形式 (Formal) 类型参数定义的是在对象初始化后,向对象中输入的数据类型,则该形式类型参数可能是逆变 (contravariance) 的。
- 如果形式 (Formal) 类型参数定义了从对象输出的数据类型,而同一形式 (Formal) 类型参数又定义了进入对象的数据类型,则该参数必须是不变 (invariant) 的。
- 为保证安全,形式 (Formal) 类型参数最好是不变 (invariant) 的。

Callable[[ParamType, …], ReturnType] 演示了规则 #1 和 #2: 即 ReturnType 是协变 (covariance) 的,而每个 ParamType 是逆变 (contravariance) 的。

默认情况下,TypeVar 创建的形式 (Formal) 类型参数是不变的,这也是标准库中可变容器类型 (如 list) 的注释方式。

“17.13.3 经典协程的泛型注解<524页>”中将继续当前关于型变 (variance) 的讨论。

下一节,将探讨如何定义泛化静态协议,将协变 (covariance) 的思想应用到几个新示例中。

15.8 实现泛化静态协议

Python 3.10 标准库提供了一些泛化静态协议。其中之一是 SupportsAbs 协议,在 typing 模块中的实现如下所示 (Lib/typing.py 1786 行):

```

1 @runtime_checkable
2 class SupportsAbs(Protocol[T_co]):
3     """An ABC with one abstract method __abs__ that is covariant in its
4         return type."""
5     __slots__ = ()
6
7     @abstractmethod
8     def __abs__(self) -> T_co:
9         pass

```

T_co 根据命名约定,声明如下:

```
1 T_co = TypeVar('T_co', covariant=True)
```

得益于 SupportsAbs 协议,Mypy 可以识别这段代码,并确认是有效的。如 ?? 所示。

</> 示例 15.21: abs_demo.py: 使用泛化协议 SupportsAbs,label=sec:lst:UseSupportsAbs

```

1 import math
2 from typing import NamedTuple, SupportsAbs
3

```

```
4  class Vector2d(NamedTuple):
5      x: float      y: float
6
7      def __abs__(self) -> float:
8          return math.hypot(self.x, self.y) ①
9
10     def is_unit(v: SupportsAbs[float]) -> bool: ②
11         """'True' if the magnitude of 'v' is close to 1."""
12         return math.isclose(abs(v), 1.0) ③
13
14     assert issubclass(Vector2d, SupportsAbs) ④
15
16     v0 = Vector2d(0, 1) ⑤
17     sqrt2 = math.sqrt(2)
18     v1 = Vector2d(sqrt2 / 2, sqrt2 / 2)
19     v2 = Vector2d(1, 1)
20     v3 = complex(.5, math.sqrt(3) / 2)
21     v4 = 1 ⑥
22
23     assert is_unit(v0)
24     assert is_unit(v1)
25     assert not is_unit(v2)
26     assert is_unit(v3)
27     assert is_unit(v4)
28
29     print('OK')
```

- ① 定义 `__abs__`, 使 `Vector2d` 与 `SupportsAbs` 相容 (`Consistent-with`)。
- ② 使用 `float` 类型对 `SupportsAbs` 协议进行参数化, 可确保 ...
- ③ ...Mypy 允许 `abs(v)` 作为 `math.isclose` 函数的第 1 个参数。
- ④ 得益于 `SupportsAbs` 定义中的 `@runtime_checkable`, 这是一个有效的运行时断言。
- ⑤ 其余代码都通过了 MyPy 检查和运行时断言。
- ⑥ `int` 类型也与 `SupportsAbs` 协议相容 (`Consistent-with`)。根据 `typeshed` 项目提供的类型提示 (`stdlib/builtins.pyi` 239 行), `int.__abs__` 返回一个 `int` 值, 而 `int` 与函数 `is_unit` 中参数 `v` 声明的 `float` 类型相容 (`Consistent-with`)。

同样, 可以为 “[示例 13.17<382页>](#)” 中的 `RandomPicker` 协议 (只定义了一个返回 `Any` 值的 `pick` 方法) 编写一个泛化版本。

[示例 15.22](#) 展示了泛化版 `RandomPicker` 协议的实现, `pick` 方法的返回值类型是可协变 (`covariance`) 的。

</> [示例 15.22: generic_randompick.py: 泛化协议 RandomPicker 的定义](#)

```
1  from typing import Protocol, runtime_checkable, TypeVar
2
3  T_co = TypeVar('T_co', covariant=True) ①
4
5  @runtime_checkable
```

```

6  class RandomPicker(Protocol[T_co]):      ②
7      def pick(self) -> T_co: ...          ③

```

- ① 声明可协变 (covariance) 的 `T_co`。
- ② 这使得 `RandomPicker` 协议成为具有可协变 (covariance) 的形式 (Formal) 类型参数的泛型 (Generic Type)。
- ③ 用 `T_co` 作为返回值类型。

泛化版 `RandomPicker` 协议可以 协变 (covariance) , 因为它只在返回值类型中使用了形式 (Formal) 类型参数。

15.9 本章小结

本章从使用 `@overload` 的简单示例开始, 随后详细研究了一个较为复杂的示例: 用多个重载的签名为内置函数 `max` 正确添加类型提示 (Type Hints)。

接下来, 介绍了特殊结构 `typing.TypedDict`。之所以未在 “五 数据类构建器” 中与 `typing.NamedTuple` 一起介绍, 是因为 `typing.TypedDict` 不是类构建器。它只用于为值是 `dict` 的变量或参数添加类型提示 (Type Hints), 指明各个字符串 “键 (key)” 对应的 “值 (value)” 是什么类型——当将 `dict` 用作记录时就会发生这种情况, 常用于处理 JSON 数据的场景。这部分内容较长, 因为使用 `typing.TypedDict` 会给人带来一种安全错觉。而且我想向您证明, 在试图从动态映射 (如 JSON 或 XML) 中创建静态结构化记录时, 运行时检查与错误处理都是不可避免的。

紧接着, 介绍了 `typing.cast()`, 该函数旨在为类型检查工具的工作提供指引。仔细考虑何时使用 `cast` 非常重要, 因为过度使用它会妨碍类型检查工具的正常工作。

之后, 介绍了如何在运行时访问类型提示 (Type Hints)。重点是要使用 `get_type_hints` 函数, 而不是直接读取 `__annotations__` 属性。然后, 该函数对于某些类型提示 (Type Hints) 或许不太可靠。Python 核心开发人员仍在研究一种新方法——使类型提示 (Type Hints) 在运行时可用的同时, 还可减少类型提示 (Type Hints) 对 CPU 与内存使用的影响。

最后几节是关于泛型 (Generic Type) 的内容。首先, 从泛型类 `LottoBlower` 开始, 这是一个 “不变 (invariant)” 的泛化类 (Generic Classes)。该示例之后, 是 4 个基本术语的定义: 泛型 (Generic Type) 、参数化 (Parameterized) 类型、形式 (Formal) 类型参数、实际 (Actual) 类型参数。

接下来, 介绍了型变 (variance) 相关的话题, 以现实中的 “饮料自动售货机” 与 “垃圾桶” 为例, 讲解了 “不变 (invariant) 类型”、“协变 (covariance) 类型”、“逆变 (contravariance) 类型” 的概念。之后, 总结了这些概念, 并给出了正式定义, 还分析了 Python 标准库中对这些概念的应用示例。

最后, 讲解了如何定义泛化静态协议。首先, 分析了标准库中的 `SupportsAbs` 协议。然后, “照猫画虎” 重新定义了泛化版的 `RandomPicker` 协议, 使其比 “示例 13.17<382页>” 中的原始版本更加严格。



Python 的类型系统是一个庞大而发展迅速的话题。本章内容并不全面。我只将重点都放在那些广泛适用、具有挑战性、概念上很重要的主题上, 因为这些主题可能会长期存在。

15.10 延伸阅读

Python 的静态类型系统在设计之初就很复杂, 随着时间的推移, 它的复杂性在逐年增加。表 15.1 列出了截至 2021 年 5 月我所知道的所有 PEP。如果要涵盖所有内容, 可能需要再写一本书。

表 15.1: 涉及 [类型提示 \(Type Hints\)](#) 的 PEP¹⁶

PEP	标题	Python 版本	年份
3107	Function Annotations	3.0	2006
483*	The Theory of Type Hints	n/a	2014
484*	Type Hints	3.5	2014
482	Literature Overview for Type Hints	n/a	2015
526*	Syntax for Variable Annotations	3.6	2016
544*	Protocols: Structural subtyping (static duck typing)	3.8	2017
557	Data Classes	3.7	2017
560	Core support for typing module and generic types	3.7	2017
561	Distributing and Packaging Type Information	3.7	2017
563	Postponed Evaluation of Annotations	3.7	2017
586*	Literal Types	3.8	2018
585	Type Hinting Generics In Standard Collections	3.9	2019
589*	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	3.8	2019
591*	Adding a final qualifier to typing	3.8	2019
593	Flexible function and variable annotations	?	2019
604	Allow writing union types as <code>X Y</code>	3.10	2019
612	Parameter Specification Variables	3.10	2019
613	Explicit Type Aliases	3.10	2020
645	Allow writing optional types as <code>x?</code>	?	2020
646	Variadic Generics	?	2020
647	User-Defined Type Guards	3.10	2021
649	Deferred Evaluation Of Annotations Using Descriptors	?	2021
655	Marking individual TypedDict items as required or potentially-missing	?	2021

Python 官方文档几乎很难跟进所有 PEP, 因此 [Mypy 文档](#) 是必不可少的参考资料。《[健壮的 Python](#)》(Patrick Viafore 著)¹⁷ 是我所知的第一本广泛介绍 Python 静态类型系统的书, 于 2021 年 8 月出版。您正在阅读的这本可能是第二本。

PEP 484 有专门的一节, 用于讲解 [型变 \(variance\)](#) 这一晦涩难懂的主题。另外, 在 [Mypy 文档](#) 的“[Generics](#)”页面及“[Common issues and solutions](#)”页面, 后包含对 [型变 \(variance\)](#) 的介绍。

¹⁶ 标有 * 号的 PEP 非常重要, 在 [typing 模块文档](#) 的开头段落中有所提及。Python 栏中的 “?” 表示正在讨论或尚未实现的 PEP; “n/a” 表示信息性 PEP, 不对应具体 Python 版本。

¹⁷ 英文原版书名为:《[Robust Python](#)》。

若打算使用与 `get_type_hints` 函数互补的 `inspect` 模块, “PEP 362 -Function Signature Object” 值得一读。

如果对 Python 的历史感兴趣, 可以阅读 Guido van Rossum 于 2004 年 12 月 23 日发表的 “All Things Pythonic: Adding Optional Static Typing to Python” 一文。

研究报告 “Python 3 Types in the Wild: A Tale of Two Type Systems” (Ingkarat Rak-amnouykit 等¹⁸), 调查了 Github 上开源项目中 `类型提示 (Type Hints)` 的使用情况, 结果表明大多数项目都未使用 `类型提示 (Type Hints)`, 而且大多数使用了 `类型提示 (Type Hints)` 的项目也都未使用类型检查器。我觉得报告中最有趣的部分, 是关于 Mypy 与 Google Pytype 语法差异的讨论。报告得出的结论是: “二者本质上是不同的类型系统”。

关于 `渐进式类型系统 (Gradual Type System)` 的 2 篇开创性论文是: “Pluggable Type Systems” (Gilad Bracha) 与 “Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages” (Eric Meijer 与 Peter Drayton)¹⁹。

在阅读其他语言书籍的相关部分时, 学到了许多知识, 这些语言也实现了一些相同的想法:

- 《Atomic Kotlin》(Bruce Eckel 与 Svetlana Isakov 著)
- 《Effective Java, 3rd ed》(Joshua Bloch)
- 《Programming with Types Examples in TypeScript》(Vlad Riscutu)
- 《Programming TypeScript》(Boris Cherny)
- 《The Dart Programming Language》(Gilad Bracha)²⁰

关于类型系统的一些批判性观点, 我推荐 Victor Youdaiken 的文章《Bad ideas in type theory》与《Types considered harmful II》。

最后, 我惊喜地发现了 Ken Arnold 撰写的 “Generics Considered Harmful”。Arnold 一开始就是 Java 的核心贡献者, 也是官方版《The Java Programming Language》前 4 版的合著者——与 Java 首席设计师 James Gosling 合著。

遗憾的是, Arnold 的批评也适用于 Python 的静态类型系统。在阅读有关类型系统的 PEP, 每每看到各种规则和特殊情况时, 都会想起 Gosling 帖子中的这段话:

这就引出了我经常提到的一个有关 C++ 的问题: 我称之为 “例外规则的 N 阶例外”。听起来像这样: “y 做了 z, 除了 y, 在这两种情况下, 你可以做 x, 如果 ...”^a

^a译者注: 在 C++ 中存在许多异常情况和例外情况, 而且这些情况还可能存在多层嵌套。因此, 即使按照规则来处理异常, 仍然存在各种复杂的情况和例外情况需要考虑。

幸运的是, 与 Java 和 C++ 相比, Python 有一个关键优势: 类型系统是可选的。当类型检查器和 `类型提示 (Type Hints)` 变得过于繁琐时, 可以静默类型检查器, 忽略 `类型提示 (Type Hints)`。

¹⁸Ingkarat Rak-amnouykit 等人, 来自伦斯勒理工学院 (Rensselaer Polytechnic Institute) 与 IBM TJ Watson 研究中心。

¹⁹阅读过脚注的读者应该记得, 以食堂饮料自动售货机来类比 `型变 (variance)`, 就是借鉴自 Erik Meijer。

²⁰这本书是为 Dart 1 编写的。在 Dart 2 中, 包括类型系统在内, 都有很大的变化。尽管如此, Bracha 仍是编程语言设计领域的重要研究者, 他对 Dart 设计的看法, 让我觉得这本书很有价值。

杂谈

类型无底洞

当使用类型检查器时, 有时会被迫发现并导入不需要了解的类, 而除了编写 [类型提示 \(Type Hints\)](#) 之外, 代码中也用不到这些类。这些类没有文档说明, 可能是因为它们被软件包的作者视为实现细节。如下是标准库中的两个示例:

为了在 “[15.4 类型校正 <430页>](#)” 中使用 `cast()`, 我不得不翻阅大量 `asyncio` 文档, 还浏览了该包中多个模块的源码。最后, 才在文档未记录的 `asyncio.trsock` 模块中找到了同样未记录的 `TransportSocket` 类。直接使用 `socket.socket`(而不用 `TransportSocket`)是不正确的, 因为根据源码([Lib/asyncio/trsock.py 5 行](#))中的 `docstring`, `TransportSocket` 显然不是 `socket.socket` 的子类型。

在为 “[示例 19.13 <579页>](#)” 添加类型提示时, 我也掉进了类似的无底洞。该示例使用 `SimpleQueue` 对象——调用 `multiprocessing.SimpleQueue()` 即可获得该对象。但是, 不能在 [类型提示 \(Type Hints\)](#) 中使用这个名称。因为 `multiprocessing.SimpleQueue` 并不是类, 而是文档未记录的 `multiprocessing.BaseContext` 类的一个绑定方法, 该方法构建并返回 `SimpleQueue` 类(在文档未记录的 `multiprocessing.queues` 模块中定义)的实例。

在这两种情况下, 为了编写一个类型提示, 我都要耗费大量时间去寻找应该导入的类, 而且文档中还未提供任何线索。当然, 我是为了写书, 所以必须研究透彻。如果是编写应用程序代码, 我应该不会仅为一行类型不全的代码, 浪费这么多精力。只需为该行代码加上 “`# type: ignore`” 即可搞定, 有时候, 这也是唯一合算的方案。

其他语言的型变 (variance) 表示法

[型变 \(variance\)](#) 是一个晦涩难懂的话题, 而且 Python 的类型提示语法也不尽如人意。引用自 [PEP 484](#) 中的一句话证明了这一点:

协变 (covariance) 或逆变 (contravariance) 不是类型变量的属性 (property), 而是使用该变量定义的泛化类 (Generic Classes) 的属性 (property)^a。

^a详见 [PEP 484](#) 中 “Covariance and contravariance” 部分的最后一段。

既然如此, 为何用 `TypeVar` 声明 [协变 \(covariance\)](#) 和 [逆变 \(contravariance\)](#), 而不是在 [泛化类 \(Generic Classes\)](#) 上声明?

[PEP 484](#) 作者对自己的工作提出了严格的约束——即在不对解释器进行任何更改的情况下支持 [类型提示 \(Type Hints\)](#)。正因如此, 需要引入 `TypeVar` 来定义类型变量, 还需要大量使用 `[]` 为 [泛型 \(Generic Type\)](#) 提供 `Klass[T]` 语法, 而其他流行语言(如 C#、Java、Kotlin 与 TypeScript)都使用 `Klass<T>` 表示法, 这些语音都不需要在使用前声明类型变量。

此外, Kotlin 和 C# 的语法会直接在类或接口声明中, 明确说明类型参数是 [协变 \(covariance\) 参数](#)、[逆变 \(contravariance\) 参数](#), 还是不变参数。

在 Kotlin 中, 可以这样声明 `BeverageDispenser`:

```
1 class BeverageDispenser<out T> {
2     // etc...
3 }
```

形式 (Formal) 类型参数中的 `out` 修饰符表示 `T` 是“out”类型, 因此 `BeverageDispenser` 是协变 (covariance) 类型。

您可能会猜到, 在 Kotlin 中, 将如何声明本章“[示例 15.20<441 页>](#)”中的 `TrashCan`:

```
1 class TrashCan<in T> {  
2     // etc...  
3 }
```

给定 `T` 作为“in”形式 (Formal) 类型参数, 那么 `TrashCan` 就是逆变 (contravariance) 的。

如果 `in` 和 `out` 都没有出现, 则该类在参数上是不变 (invariant) 的。

当在形式 (Formal) 类型参数中使用 `out` 和 `in` 时, 很容易让人想起“[15.7.4.4 型变经验法则](#)”的内容。

这表明 Python 中协变 (covariance) 与逆变 (contravariance) 类型变量的良好命名约定应该是:

```
1 T_out = TypeVar('T_out', covariant=True)  
2 T_in = TypeVar('T_in', contravariant=True)
```

然后, 就可以这样定义这些类:

```
1 class BeverageDispenser(Generic[T_out]):  
2     ...  
3  
4 class TrashCan(Generic[T_in]):  
5     ...
```

现在修改 PEP 484 中确立的命名约定是否为时已晚?

运算符重载

这里有一些让我感到纠结的事情, 比如运算符重载。不使用操作符重载是我的个人选择, 因为我在 C++ 中看到太多人滥用运算符重载。

——James Gosling, Java 之父^a

^a摘自 “The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling” 一文。

在 Python 中, 可以使用如下公式计算复利:

```
1 interest = principal * ((1 + rate) ** periods - 1)
```

出现在操作数之间 (如 `1 + rate`) 的运算符叫作 “中缀运算符”。在 Python 中, 中缀运算符可以处理任意类型。因此, 若想处理真实货币, 当确保 `principal`、`rate`、`periods` 都是精确的数值 (Python 类 `decimal.Decimal` 的实例), 则上述公式即可得到精确的结果。

但在 Java 中, 如果要从 `float` 转换到 `BigDecimal` 以获得精确结果, 就不能再使用中缀运算符。因为 Java 的中缀运算符只适用于原始类型。为了让计算公式正确处理 `BigDecimal` 数值, 在 Java 中要像下面这样写:

```
1 BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate))
2                     .pow(periods).subtract(BigDecimal.ONE));
```

显然, 中缀运算符可提高公式的可读性。为此, 支持运算符重载很有必要。这样可以使用户定义的类型或扩展的类型 (如 NumPy 数组) 也可以支持中缀运算符表示法。高级、易用的语言, 又实现了运算符重载, 这可能是 Python 在数据科学 (包括金融与科学应用) 领域取得巨大成功的关键因素。

在 “1.3.1 模拟数值类型⁹” 中, 为 `Vector` 类实现了一些简单的运算符 (详见 “[示例 1.2¹⁰](#)”)。示例中的 `__add__` 与 `__mul__` 方法是为了展示如何用特殊方法实现运算符重载, 不过忽视了一些小问题。另外, “[示例 1.2²⁹](#)” 中的 `Vector2d.__eq__` 方法认为 “`Vector(3, 4) == [3, 4]`” 为 `True`——这可能合理, 也可能不合理。本章将解决这些问题, 另外还将讨论如下这些问题:

- 中缀运算符方法应如何表明它无法处理某个操作数。
- 使用鸭子类型 (Duck Typing) 或大鹅类型 (Goose Typing), 处理各种类型的操作数。

- 丰富的比较运算符（如 `==`、`>`、`<` 等）的特殊行为。
- 增强赋值运算符（如 `+=`）的默认处理方式，以及如何重载这些运算符。

16.1 本章新增内容

“[示例 16.11<461页>](#)”更改为用[鸭子类型（Duck Typing）](#)取代对 `numbers.Real` 的显式 `isinstance` 检查¹。

在本书第 1 版出版时，Python 3.5 还处于内测阶段，矩阵乘法运算符 `@` 是在一个附注栏中进行介绍的。现在，哪部分内容已融入到本章正文，位于“[16.6 将 @ 当作中缀运算符使用<462页>](#)”。另外，还利用[大鹅类型（Goose Typing）](#)重新实现了 `__matmul__` 方法，使其比本书第 1 版中的 `__matutl__` 更安全，同时又不影响灵活性。

“[16.11 延伸阅读<473页>](#)”新增了一些参考资料，包括 Guido van Rossum 发表的一篇博文。还提到了 2 个库：`pathlib` 和 `Scapy`，它们展示了运算符重载在数学领域之外的有效应用：

16.2 运算符重载入门

运算符重载的作用是允许用户定义的对象也能使用中缀运算符（如 `+` 和 `l`）或一元运算符（如 `-` 和 `~`）。更广泛地讲，在 Python 中，函数调用 `(())`、属性访问 `(.)`，以及元素访问/切片 `([])` 也都是运算符，但本章仅涵盖一元运算符与中缀运算符。

在某些圈子中，运算符重载的名声不是很好。它是一种可能（而且已经）被滥用的语言特性，会为程序员带来困惑、bug 及意想不到的性能瓶颈。但是，如果使用得当，它能使 API 更易用，使代码更易读。Python 通过施加限制，使得运算符重载在灵活性、可用性、安全性之间取得了良好的平衡：

- 无法更改内置类型运算符的含义。
- 无法创建新的运算符，仅能重载现有的运算符。
- 某些运算符无法重载，如 `is`、`and`、`or`、`not`。但是，位运算符可以被重载，如 `&`、`l`、`~`。

“[示例 12.16<337页>](#)”已经为 `Vector` 类重载了一个中缀运算符，即由 `__eq__` 方法支持的 `==`。本章将改进 `__eq__` 方法的实现，以便可更好地处理除 `Vector` 类型以外的操作数。然而，丰富的比较运算符（`==`、`!=`、`>`、`<`、`>=`、`<=`）是运算符重载方面的特例。所以，本章将首先为 `Vector` 类重载 4 个算术运算符：即一元运算符 `-` 与 `+`，以及中缀运算符 `+` 与 `*`。

首先，介绍最简单的算术运算符：即一元运算符。

16.3 一元运算符

《Python 语言参考》的“[6.6. Unary arithmetic and bitwise operations](#)”一节，列出了 3 个一元运算符。此处列出了与它们相关的特殊方法：

- `-`，由 `__neg__` 实现

¹Python 标准库中的其余 [抽象基类（ABCs）](#) 对于[大鹅类型（Goose Typing）](#)与[静态类型（Static Typing）](#)仍然很有价值。关于 `numbers` 抽象基类的问题，请参阅“[13.6.8 number 模块中的抽象基类与 Numeric 协议<385页>](#)”。

“一元取负”算术运算符。例如,若 x 是 -2 ,则 $-x == 2$ 。

- $+$,由 `__pos__` 实现

“一元取正”算术运算符。例如,通常情况下 $x == +x$,但也有少数例外情况。如果感到好奇,请阅读本节后文的附注栏“[x 与 +x 何时不相等](#)”。

- \sim ,由 `__invert__` 实现

整数的“按位否”或“按位取反”²,公式为 $\sim x == -(x + 1)$ 。如果 x 是 2 ,则 $\sim x == -3$ 。具体计算细节见下方附注栏“[~ 的计算细节](#)”

~ 的计算细节

整数的补码表示

在介绍 \sim 计算细节之前,先了解一下有符号整数的补码表示。如下以 1 字节的有符号整数(-128 到 +127)为例。

• 正数的补码表示

- 正数的补码表示与其原码相同,即符号位为 0,后面跟着正数的二进制表示。
- 例如,十进制数 $+3$ 的二进制表示是 0000 0011,其补码表示也是 0000 0011。

• 负数的补码表示

- 负数的补码表示通过对其绝对值的原码进行取反操作,并加 1 得到。
- 例如,十进制数 -3 的绝对值为 3 ,二进制表示为 0000 0011,对其进行取反操作得到 1111 1100,然后加 1 得到 1111 1101,这就是 -3 的补码表示。

• 通过补码得原码

- 判断符号位:符号位为 1 表示负数;符号位为 0 表示正数。
- 负数情况:将除符号位以外的二进制,按位取反并加 1。
- 整数情况:源码与补码相同。

~ 的计算细节

当对一个整数 x 执行 \sim 运算(按位取反)时,实际上是将其二进制表示的每一位取反(即 0 变 1,1 变 0)。但是,注意 Python 中的整数是有符号的,并且使用补码表示。

下面看一下 ~ 2 的计算过程:

- ① 整数 2 的二进制表示是 0000 0010。
- ② ~ 2 (取反)后变为 1111 1101。最高位为符号位(0 为正,1 为负)。
- ③ 在 Python 中,这个结果是一个负数(因为最高位是 1)的二进制补码表示。
- ④ 为了得到 1111 1101 的原码:将除符号位以外的二进制 111 1101,按位取反,再加 1,得到 000 0011。
- ⑤ 这个二进制数 000 0011 转换为十进制是 3 。但是,因为它是负数。所以,结果是 -3 。

总结为公式,就是“ $\sim x == -(x + 1)$ ”。例如,“ $\sim 2 == -(2 + 1) == -3$ ”。

《Python 语言参考》的“3.3.8. Emulating numeric types”小节,还将内置函数 `abs()` 也列为一元运算符。

²有关“按位否”的解释,请参阅 https://en.wikipedia.org/wiki/Bitwise_operation#NOT。

如前文所述,与其相关的特殊方法为 `text`

支持一元运算符非常简单。只需实现相应的特殊方法,并且该方法只接受一个参数(`self`)即可。特殊方法的实现应符合所在类的逻辑,此外还要遵守运算符的一个基本规则:总是返回一个新对象。换句话说,不要修改接收者(`self`),而是创建并返回一个合适类型的新实例。

对于一元运算符`-`与`+`来说,结果可能是与`self`同属一类的实例。对于一元运算符`+`,若接收者(即调用方法的对象)是不可变对象,就应返回`self`;否则,应返回`self`的副本。对于`abs()`,结果应该是一个标量数值。

而对于一元运算符`~`来说,若不是处理整数中的位,则很难说什么结果是合理的。在数据分析包 `pandas` 中,运算符`~`表示取反布尔筛选条件,相关示例,请参阅 `pandas` 文档的“Boolean indexing”。

如“十二 序列的特殊方法”所述,本节将为“示例 1.2<10页>”中的 `Vector` 类实现几个新的一元运算符。

示例 16.1 展示了 `Vector` 类已有的 `__abs__` 方法,以及新增加的一元运算符方法 `__neg__` 与 `__pos__`。

</> 示例 16.1: `vector_v6.py`: 为“示例 1.2<10页>”的 `Vector` 类增加一元运算符`-`与`+`

```

1 def __abs__(self):
2     return math.hypot(*self)
3
4 def __neg__(self):
5     return Vector(-x for x in self) \ding{202}
6
7 def __pos__(self):
8     return Vector(self) \ding{203}
```

① 为了计算`-v`,需要构建一个新的 `Vector` 实例,并且将`self`的每个分量都取反。

② 为了计算`+v`,需要构建一个新的 `Vector` 实例,并且`self`的每个分量保持不变。

还记得吗? `Vector` 实例是可迭代对象,而且 `Vector.__init__` 方法接受一个可迭代对象作为参数。所以,`__neg__` 与 `__pos__` 方法的实现短小精悍。

我们暂时不实现 `__invert__` 方法,因此若视图对 `Vector` 实例执行`~v` 运算,将会引发 `TypeError` 异常。而且,会输出明确的错误消息“bad operand type for unary ~: 'Vector'”。

下方的附注栏讨论了一个奇怪的问题,可增加对一元运算符`+`的认识。

x 与`+x` 何时不相等

每个人都希望`x == +x`,这在 Python 中几乎总是正确的,但我在标准库中发现了 2 种例外(即`x != +x`)的情况。

第一种例外情况与 `decimal.Decimal` 类有关。若`x`是在算术运算上下文中创建的 `decimal.Decimal` 实例。然后,在不同的上下文中计算`+x`,则可能会出现`x != +x` 的情况。例如,`x`是在具有一定精度的上下文中计算的,但上下文的精度发生了变化,然后对`+x`进行求值。如示例 16.2 所示。

</> 示例 16.2: 算术上下文精度的变化,可能导致`x`与`+x` 不同

```

1 >>> import decimal
2 >>> ctx = decimal.getcontext() ❶
3 >>> ctx.prec = 40 ❷
4 >>> one_third = decimal.Decimal('1') / decimal.Decimal('3') ❸
5 >>> one_third ❹
```

- ❶ 获取对当前全局算术运算上下文的引用。
 - ❷ 将算术运算上下文的精度设置为 40。
 - ❸ 用当前精度计算 $1/3$ 。
 - ❹ 检查结果：小数点后有 40 位数字。
 - ❺ 此时, `one_third == +one_third` 为 `True`。
 - ❻ 将精度降至 28——这是 `Decimal` 算术运算的默认精度。
 - ❼ 现在, `one_third == +one_third` 为 `False`。
 - ❽ 检查 `+one_third`, 小数点后有 28 位数字, 比 ❹ 处 `one_third` 少了 12 位数字。

事实上, 表达式 `+one_third` 每次出现都会根据 `one_third` 的值生成一个新的 `Decimal` 实例, 但每次使用的都是当前算术上下文的精度。

第二种例外情况与 `collections.Counter` 类有关。`collections.Counter` 实现了多个算术运算符，包括用于将两个 `collections.Counter` 实例的计数相加的中缀运算符 `+`。不过，出于实用性原因，`Counter` 实例相加会从结果中剔除任何计数为零值或负值的项。而一元运算符 `+` 等同于加上一个空的 `Counter` 实例，因此会产生一个新的 `Counter` 实例，而且仅保留大于零的项(计数)。如 [示例 16.3](#) 所示。

</> 示例 16.3: 一元运算符 +, 将得到一个新的 Counter 实例 (不含计数为零与负值的项)

```
1 >>> ct = Counter('abracadabra')
2 >>> ct
3 Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
4 >>> ct['r'] = -3
5 >>> ct['d'] = 0
6 >>> ct
7 Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
8 >>> +ct
9 Counter({'a': 5, 'b': 2, 'c': 1})
```

如您所见, `+ct` 返回的 Counter 实例中, 只包含计数大于零的项。

下面回归正题。

16.4 重载向量加法运算符 +

Vector 类 (见“示例 12.16<337页>”) 是一种序列类型。按《The Python Language Reference》中的“3.3.7. Emulating container types”所述,序列应支持运算符 + (拼接) 与运算符 * (重复复制)。不过,这里我

们将把 + 与 * 实现为数学向量运算。对于 Vector 类型来说, 实现这两种运算难度稍大, 不过却更有意义。



若用户想要拼接或重复复制 Vector 实例, 可以将它们转换为元组或列表, 为其应用运算符。然后, 再将运算结果转换回 Vector 实例。这种实现细节要归功于 Vector 实例是可迭代对象, 并且可以由一个可迭代对象构建 Vector 实例。

```
1 >>> v_concatenated = Vector(list(v1) + list(v2))
2 >>> v_repeated = Vector(tuple(v1) * 5)
```

2 个欧几里得向量相加, 将得到一个新向量。其中的各分量是 2 个向量中相应分量之和。如下所示:

```
1 >>> v1 = Vector([3, 4, 5])
2 >>> v2 = Vector([6, 7, 8])
3 >>> v1 + v2
4 Vector([9.0, 11.0, 13.0])
5 >>> v1 + v2 == Vector([3 + 6, 4 + 7, 5 + 8])
6 True
```

若尝试将 2 个长度不同的 Vector 实例加在一起会怎样呢? 此时, 可以引发异常。不过, 根据实际情况 (如信息检索), 最好用 0 来填充最短的向量。我们想要的效果如下所示:

```
1 >>> v1 = Vector([3, 4, 5, 6])
2 >>> v3 = Vector([1, 2])
3 >>> v1 + v3
4 Vector([4.0, 6.0, 5.0, 6.0])
```

确定上述这些基本要求之后, `__add__` 方法的实现如 [示例 16.4](#) 所示。

</> [示例 16.4: Vector.__add__ 方法 \(第 1 版\)](#)

```
1 # inside the Vector class
2
3 def __add__(self, other):
4     pairs = itertools.zip_longest(self, other, fillvalue=0.0) ❶
5     return Vector(a + b for a, b in pairs) ❷
```

❶ `pairs` 是一个生成器, 能生成元组 (a,b), 其中 a 来自 `self`, b 来自 `other`。若 `self` 与 `other` 的长度不同, `fillvalue` 会为最短可迭代对象提供缺失值。

❷ 由生成器表达式构建一个新 Vector 实例, 生成器表达式将计算 `pairs` 中各个 (a,b) 之和。

注意, `__add__` 会返回一个新 Vector 实例, 不会更改 `self` 与 `other`。



实现一元或中缀运算符的特殊方法永远不应该更改操作数的值。使用此类运算符的表达式应通过创建新对象来产生结果。只有增量赋值运算符可以更改第一个操作数 (`self`), 详见 “[16.9 增量赋值运算符](#)”。

根据 [示例 16.4](#) 的实现, 可以将 Vector 加到 Vector2d, 也可以将 Vector 加到元组或任何可生成数字的可迭代对象上, 如 [示例 16.5](#) 所示。

</> 示例 16.5: Vector.__add__(第 1 版), 也支持 Vector 之外的对象

```

1 >>> v1 = Vector([3, 4, 5])
2 >>> v1 + (10, 20, 30)          # 将 Vector 实例加到元组
3 Vector([13.0, 24.0, 35.0])
4 >>> from vector2d_v3 import Vector2d
5 >>> v2d = Vector2d(1, 2)
6 >>> v1 + v2d                # 将 Vector 实例加到 Vector2d
7 Vector([4.0, 6.0, 5.0])

```

示例 16.5 中的 2 个加法 (+) 运算都有效, 要归功于 __add__ 使用了 `itertools.zip_longest()` 函数。zip_longest 可以处理任何可迭代对象, 而且构建新 Vector 实例的生成器表达式仅仅是将 zip_longest() 生成的数值对执行 $a+b$ 。所以, 可以使用任何能生成数字的可迭代对象。

但是, 如果对调操作数 (如示例 16.6 所示), 混合类型加法就会失败。

</> 示例 16.6: Vector.__add__(第 1 版), 左操作数不是 Vector 的混合类型加法, 将失败

```

1 >>> v1 = Vector([3, 4, 5])
2 >>> (10, 20, 30) + v1
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 TypeError: can only concatenate tuple (not "Vector") to tuple
6 >>> from vector2d_v3 import Vector2d
7 >>> v2d = Vector2d(1, 2)
8 >>> v2d + v1
9 Traceback (most recent call last):
10  File "<stdin>", line 1, in <module>
11 TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'

```

为了支持涉及不同类型对象的运算, Python 为中缀运算符特殊方法实现了一种特殊的分派机制。例如, 给定一个表达式 $a + b$, 解释器将执行以下几步操作 (另见 图 16.1)。

1. 若 a 具有 __add__ 方法, 而且不返回 NotImplemented。则调用 $a.__add__(b)$, 并返回结果。
2. 若 a 没有 __add__ 方法, 或调用 __add__ 返回 NotImplemented, 则检查 b 有没有 __radd__ 方法:
 - (a) 若 b 具有 __radd__ 方法, 而且不返回 NotImplemented。则调用 $b.__radd__(a)$, 并返回结果。
 - (b) 若 b 没有 __radd__ 方法, 或者调用 __radd__ 方法返回 NotImplemented。则引发 TypeError 异常, 并在错误消息中指明 “unsupported operand types message.”



__radd__ 方法被称为 __add__ 的“反射 (reflected)”或“反向 (reversed)”版本。

我更喜欢称其为“反向 (reversed)”特殊方法^a。

^aPython 文档中, 同时使用这 2 个术语。“3. Data model”中使用了“反射 (reflected)”, 而 numbers 模块的 “Implementing the arithmetic operations” 提到了“正向 (forward)”方法与“反向 (reversed)”方法, 我觉得这 2 个术语更好。因为“正向 (forward)”与“反向 (reversed)”明确指出了方向, 而“反向 (reversed)”没有这个效果。

因此, 要使“示例 16.6<457页>”中的混合类型加法有效, 需要实现 Vector.__radd__ 方法, 如果左操作数没有实现 __add__, 或者实现了 __add__ 但返回 NotImplemented, 以表示它不知道如何处理右操作数, Python 将调用 Vector.__radd__ 方法, 这是一种后备机制。

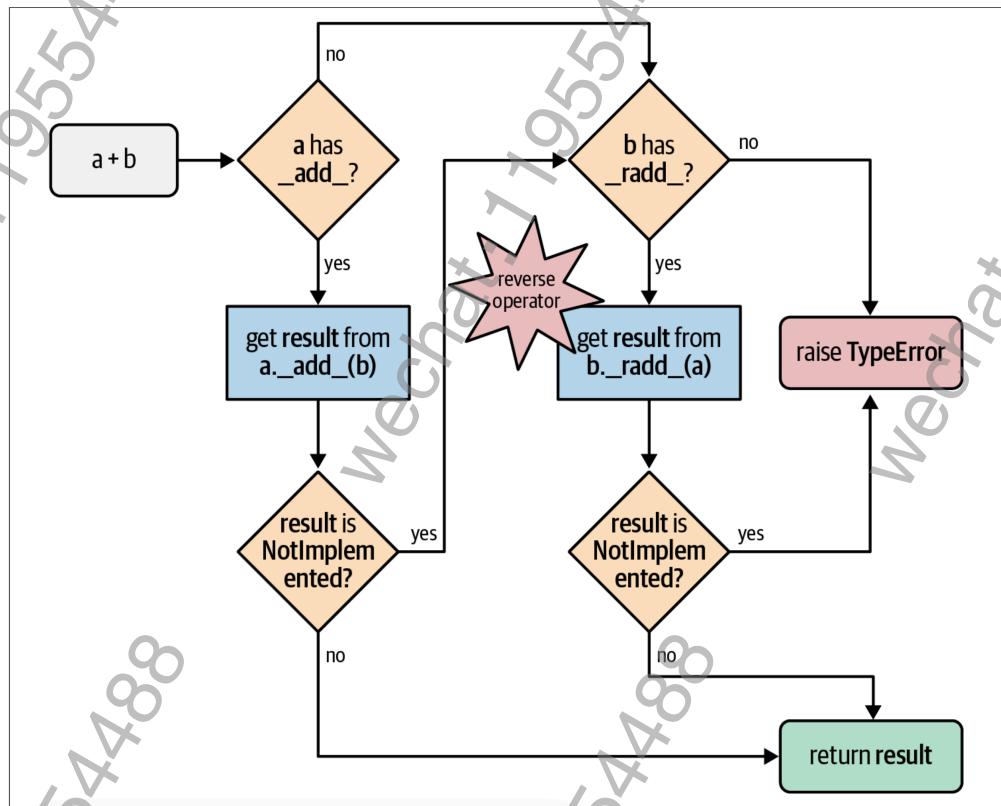


图 16.1: 用 __add__ 与 __radd__ 计算 a+b 的流程图



不要混淆 NotImplemented 和 NotImplemented。NotImplemented 是一个特殊的单例 (Singleton) 值, 中缀运算符特殊方法应返回该值, 以告诉解释器它无法处理给定的操作数。相反, NotImplemented 是抽象类中的存根方法可能引发的异常, 以警告子类必须实现它们。

Vector.__radd__ 的最简单实现如示例 16.7 所示。

</> 示例 16.7: Vector 的 __add__ 与 __radd__ 方法

```

1 # 在 Vector 类中定义
2
3 def __add__(self, other): ❶
4     pairs = itertools.zip_longest(self, other, fillvalue=0.0)
5     return Vector(a + b for a, b in pairs)
6
7 def __radd__(self, other): ❷
8     return self + other

```

❶ “示例 16.4<456页>” 中的 __add__ 方法没有变化。在此列出 __add__ 是因为 __radd__ 需要它。

❷ __radd__ 只是将计算委托给 __add__。

通常, __radd__ 方法就是这么简单: 只需调用适当的运算符, 本例就是委托 __add__。任何满足交换律的运算符都可以这么做。在处理数字或向量时, 运算符 + 满足交换律, 但在拼接 Python 序列时, 则不满足交换律。

如果 `_radd__` 只是简单地调用 `_add__`, 这里还有另一种方法, 可以达到同样效果:

```
1 def __add__(self, other):
2     pairs = itertools.zip_longest(self, other, fillvalue=0.0)
3     return Vector(a + b for a, b in pairs)
4
5 __radd__ = __add__
```

示例 16.7 中的方法可以处理 `Vector` 对象, 或任何包含数字项的可迭代对象, 如 `Vector2d`、整数元组或浮点数元组。但是, 如果提供了不可迭代对象, `_add__` 将引发异常, 并且会给出一条用处不大的错误消息, 如示例 16.8 所示。

</> 示例 16.8: `Vector.__add__` 方法需要一个可迭代的操作数

```
1 >>> v1 + 1
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "vector_v6.py", line 328, in __add__
5     pairs = itertools.zip_longest(self, other, fillvalue=0.0)
6 TypeError: zip_longest argument #2 must support iteration
```

更糟的是, 如果操作数是可迭代对象, 但是生成的项无法与 `Vector` 中的 `float` 项相加, 也会给出用处不大的错误消息, 如示例 16.9 所示。

</> 示例 16.9: `Vector.__add__` 方法需要一个包含数字的可迭代对象

```
1 >>> v1 + 'ABC'
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "vector_v6.py", line 329, in __add__
5     return Vector(a + b for a, b in pairs)
6   File "vector_v6.py", line 243, in __init__
7     self._components = array(self.typecode, components)
8   File "vector_v6.py", line 329, in <genexpr>
9     return Vector(a + b for a, b in pairs)
10 TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

我尝试将 `Vector` 与字符串相加, 但得到的错误消息确实 `float` 与 `str`。

示例 16.8 与示例 16.9 的问题不仅仅是晦涩的错误消息, 更深层次的问题是: 如果一个操作数的特殊方法由于类型不兼容而无法返回有效结果, 那么它应该返回 `NotImplemented`, 而不是引发 `TypeError` 异常。如果返回 `NotImplemented`, 那么另一个操作数所属的类型还有机会执行计算, 即 Python 会尝试调用“反向 (reversed)”方法。

本着遵守 鸭子类型 (Duck Typing) 的精神, 应避免测试操作数 `other` 的类型或其元素的类型, 而是应该捕获异常并返回 `NotImplemented`。如果解释器还未反转操作数 (即调用反向方法), 那么它将尝试去反转操作数。如果调用反向方法也返回 `NotImplemented`, 则 Python 会引发 `TypeError` 异常, 并返回一个标准的错误消息。例如, “`unsupported operand type(s) for +: Vector and str.`”。整个逻辑如图 16.1 所示。

`Vector` 加法 (`_add__` 与 `_radd__`) 的最终实现版本, 如示例 16.10 所示。

</> 示例 16.10: vector_v6.py: 添加到 vector_v5.py (见“[示例 12.16<337页>](#)”) 的 + 运算符方法

```

1  def __add__(self, other):
2      try:
3          pairs = itertools.zip_longest(self, other, fillvalue=0.0)
4          return Vector(a + b for a, b in pairs)
5      except TypeError:
6          return NotImplemented
7
8  def __radd__(self, other):
9      return self + other

```

请注意, `__add__` 现在会捕获 `TypeError` 并返回 [单例 \(Singleton\)](#) `NotImplemented`。



如果中缀运算符方法引发异常, 它将中止运算符分派算法。在 `TypeError` 的特殊情况下, 通常最好捕获该异常并返回 `NotImplemented`。这样解释器就可以尝试调用反向运算符方法 (如 `__radd__`), 如果交换的操作数属于不同类型, 则该方法可能会正确处理交换操作数的计算。

至此, 我们已经通过编写 `__add__` 和 `__radd__` 安全地重载了 + 运算符。接下来, 将重载另一个中缀运算符: *。

16.5 重载标量乘法运算符 *

`Vector([1, 2, 3]) * x` 是什么意思? 如果 `x` 是一个数字, 那么这将是一个“标量乘积 (scalar product)”, 结果将是一个新的 `Vector` 实例, 其中每个分量都被乘以 `x`——也称为“元素级乘法 (elementwise multiplication)”。

```

1  >>> v1 = Vector([1, 2, 3])
2  >>> v1 * 10
3  Vector([10.0, 20.0, 30.0])
4  >>> 11 * v1
5  Vector([11.0, 22.0, 33.0])

```



另一种涉及 `Vector` 操作数的乘积, 是 2 个向量的“点积 (dot product)”, 也称为“矩阵乘法 (matrix multiplication)”。若将一个向量视作 $1 \times N$ 矩阵, 将另一个向量视作 $N \times 1$ 矩阵。那么, 这样的乘法就是“矩阵乘法”。这种运算的实现, 详见“[16.6 将 @ 当作中缀运算符使用<462页>](#)”。

回到标量乘积话题。依然从最简单有效的 `__mul__` 与 `__rmul__` 方法开始:

```

1  # 在 Vector 类中定义
2
3  def __mul__(self, scalar):
4      return Vector(n * scalar for n in self)
5

```

```

6     def __rmul__(self, scalar):
7         return self * scalar

```

这两个方法确实可用,但是为其提供了不兼容的操作数,则会出现问题。参数 scalar 的值必须是数字,与 float 值相乘得到的积是另一个 float 值(因为 Vector 类内部使用 float 数组)。因此,不能为参数 scalar 提供 complex 类型的数值,但可以提供 int、bool(是 int 的子类),甚至可以提供 fractions.Fraction 实例等标量值。示例 16.11 中的 __mul__ 方法未明确检查参数 scalar 的类型,而是直接将其转换为 float 类型。如果转换失败,则返回 NotImplemented。这是一个典型的鸭子类型(Duck Typing)示例。

</> 示例 16.11: vector_v7.py:增加 * 运算符方法

```

1  class Vector:
2      typecode = 'd'
3
4      def __init__(self, components):
5          self._components = array(self.typecode, components)
6
7      # 排版需要, 省略了许多方法, 详见 vector_v7.py
8      # https://github.com/fluentpython/example-code-2e
9
10     def __mul__(self, scalar):
11         try:
12             factor = float(scalar)
13         except TypeError: ❶
14             return NotImplemented ❷
15         return Vector(n * factor for n in self)
16
17     def __rmul__(self, scalar):
18         return self * scalar ❸

```

- ❶ 如果参数 scalar 无法转换为 float 类型 ...
- ❷ ... 不知道如何处理参数 scalar。所以,返回 NotImplemented, 让 Python 尝试在 scalar 操作数上调用 __rmul__ 方法。
- ❸ 在此示例中,__rmul__ 只需执行 self*scalar, 委托 __mul__ 方法, 即可正常工作。

有了示例 16.11 中的代码之后,就可以将 Vector 实例与常规或不太寻常的标量数值类型相乘了。如下所示:

```

1  >>> v1 = Vector([1.0, 2.0, 3.0])
2  >>> 1.4 * v1
3  Vector([14.0, 28.0, 42.0])
4  >>> v1 * True
5  Vector([1.0, 2.0, 3.0])
6  >>> from fractions import Fraction
7  >>> v1 * Fraction(1, 3)
8  Vector([0.3333333333333333, 0.6666666666666666, 1.0])

```

现在,Vector 可以与标量值相乘了。接下来,说明如何实现 Vector 与 Vector 的乘积。



在本书第 1 版中, [示例 16.11](#) 使用的是 [大鹅类型 \(Goose Typing\)](#), 利用 `isinstance(scalar, numbers.Read)` 检查了 `_mul_` 方法的 `scalar` 参数。现在, 我避免使用 `numbers` 抽象基类, 因为 [PEP 484](#) 不支持它们, 而且在运行时使用无法静态检查的类型, 在我看来是个坏主意。

另外, 也可以根据 `typing.SupportsFloat` 协议进行检查()。但是, 我最终选择利用 [鸭子类型 \(Duck Typing\)](#), 因为我认为资深的 Python 程序员应熟练掌握这种编程模式。

当然, 我并没有抛弃 [大鹅类型 \(Goose Typing\)](#), [示例 16.12](#) 中的 `_matmul_` 就是 [大鹅类型 \(Goose Typing\)](#) 的一个很好的示例, 也是第 2 版中的新内容。

16.6 将 @ 当作中缀运算符使用

众所周知, @ 符号是函数装饰器的前缀, 但自 2015 年起, 它还可以用作中缀运算符。多年来, “点积 (dot product)” 在 NumPy 中被写成 `numpy.dot(a,b)`。使得函数调用表示法 “()” 很难将较长的公式, 从数学表示法转换为 Python 代码³。因此, 经数字计算社区的游说, Python 3.5 实现了 “[PEP 465 -A dedicated infix operator for matrix multiplication](#)”。如今, 2 个 NumPy 数组的点积 (dot product) 可以写作 “`a@b`”。

运算符 @ 由特殊方法 `_matmul_`、`_rmatmul_` 和 `_imatmul_` 支持, 它们的名称取自 “矩阵乘法 (matrix multiplication)”。早期, 标准库还没有使用这些方法, 不过自 Python 3.5 起, 解释器就已经能识别了。因此, NumPy 团队以及我们自己可以让用户定义的类型支持 @ 运算符。为了支持这个运算符, Python 解析器也做了更改 (在 Python 3.4 中, `a @ b` 还是一种错误语法)。

这些简单的测试展示了, 如何使用运算符 @ 处理 Vector 实例:

```

1  >>> va = Vector([1, 2, 3])
2  >>> vz = Vector([5, 6, 7])
3  >>> va @ vz == 38.0 # 1*5 + 2*6 + 3*7
4  True
5  >>> [10, 20, 30] @ vz
6  380.0
7  >>> va @ 3
8  Traceback (most recent call last):
9  ...
10 <TypeError: unsupported operand type(s) for @: 'Vector' and 'int'>

```

[示例 16.12](#) 展示了 @ 运算符相关特殊方法 `_matmul_` 与 `_rmatmul_` 的实现。

</> [示例 16.12: vector_v7.py: 矩阵乘法 @ 运算符相关的方法](#)

```

1  class Vector:
2      # 因排版需要, 省略了许多方法
3
4      def __matmul__(self, other):
5          if (isinstance(other, abc.Sized) and

```

³ 关于此问题的详细讨论, 参见 “[16.11 延伸阅读<473页>](#)” 的 “[杂谈](#)”。

```
6     isinstance(other, abc.Iterable)):
7         if len(self) == len(other):          ❷
8             return sum(a * b for a, b in zip(self, other)) ❸
9         else:
10            raise ValueError('@ requires vectors of equal length.')
11     else:
12         return NotImplemented
13
14     def __rmatmul__(self, other):
15         return self @ other
```

- ❶ 2 个操作数都必须实现 `__len__` 与 `__iter__` 方法 ...
- ❷ ... 而且长度相同 ...
- ❸ ... 以便正确应用 `sum`、`zip` 与生成器表达式。



Python 3.10 中 `zip()` 函数的新功能自 Python 3.10 起, 内置函数 `zip` 接受一个可选的关键字参数 `strict`(默认为 `False`)。当 `strict=True` 时, 函数 `zip` 会在可迭代对象长度不同时, 引发 `ValueError` 异常。这种新的严格行为, 符合 Python 的 Fail-Fast(快速失败) 原则。在 [示例 16.12](#) 中, ❷处的内层 `if` 可以替换为 `try/except ValueError`, 并为 `zip` 调用添加 `strict=True`。如下所示。

```
1     try:
2         return sum(a * b for a, b in zip(self, other,
3                                         strict=True))
4     except:
5         raise ValueError('@ requires vectors of equal
6                           length.')
```

[示例 16.12](#) 充分利用了 [大鹅类型\(Goose Typing\)](#)。如果测试操作数 `other` 是否为 `Vector` 对象, 那么 `@` 运算符的操作数就不能是列表或数组类型。这样, 用户将失去一定的灵活性。只要一个操作数是 `Vector`, [示例 16.12](#) 中 `@` 运算符的另一个操作数就可以是 `abc.Sized` 实例或 `abc.Iterable` 实例。这 2 个抽象基类(ABCs)都实现了类方法 `__subclasshook__`(见“[13.5.8 用抽象基类实现结构类型<374页>](#)”)。因此, 凡是提供了 `__len__` 与 `__iter__` 方法的对象都能满足 `isinstance` 测试条件——无需真正子类化这些抽象基类, 也无需注册为这些抽象基类的虚拟子类(详见“[13.5.8 用抽象基类实现结构类型<374页>](#)”)。就 `Vector` 类而言, 它虽未子类化 `abc.Sized` 或 `abc.Iterable`, 但是却通过了针对二者的 `isinstance` 检查, 这是因为 `Vector` 类拥有必要的方法 `__len__` 与 `__iter__`。

接下来, 总结一下 Python 支持的算术运算符。“[16.8 丰富的比较运算符<465页>](#)”将会深入探讨一些特殊的运算符: 比较运算符。

16.7 算术运算符总结

通过实现中缀运算符 `+`、`*`、`@`, 我们了解了中缀运算符最常见的编码模式。这种编码模式适用于表 16.1 中列出的所有运算符(就地运算符将在“[16.9 增量赋值运算符<468页>](#)”中介绍)。

表 16.1: 中缀运算符与方法名

运算符	正向方法	反向方法	就地方法	说明
+	__add__	__radd__	__iadd__	加法或拼接
-	__sub__	__rsub__	__isub__	减法
*	__mul__	__rmul__	__imul__	乘法或重复复制
/	__truediv__	__rtruediv__	__itruediv__	除法
//	__floordiv__	__rfloordiv__	__ifloordiv__	整除(地板除)
%	__mod__	__rmod__	__imod__	求模
divmod()	__divmod__	__rdivmod__	__idivmod__	返回由整除的商与模数组成的元组,即整数商与余数的元组。
**, pow()	__pow__	__rpow__	__ipow__	求幂 ⁴
@	__matmul__	__rmatmul__	__imatmul__	矩阵乘法
&	__and__	__rand__	__iand__	按位与
	__or__	__ror__	__ior__	按位或
^	__xor__	__rxor__	__ixor__	按位异或
«	__lshift__	__rlshift__	__ilshift__	按位左移
»	__rshift__	__rrshift__	__irshift__	按位右移

丰富的比较运算符使用一组不同的规则。

16.8 丰富的比较运算符

Python 解释器对比较运算符 `==`、`!=`、`>`、`<`、`>=` 和 `<=` 的处理与前文类似,但在两个重要方面有所区别:

- 正向调用与反向调用,使用的是同一组方法。表 16.2 总结了这些规则。例如,对 `==` 来说,正向调用与反向调用都用 `__eq__` 方法,只是将参数对调了。而正向的 `__gt__` 方法调用,等价于反向的 `__lt__` 方法调用并对调参数。
- 对于 `==` 与 `!=`,若缺少反向方法或返回 `NotImplemented`。那么,Python 会比较对象 ID,而不是引发 `TypeError`。

表 16.2: 丰富的比较运算符:若正向方法返回 `NotImplemented`,就调用反向方法。

分组	中缀运算符	正向方法调用	反向方法调用	后备机制
相等性	<code>a == b</code>	<code>a.__eq__(b)</code>	<code>b.__eq__(a)</code>	返回 <code>id(a) == id(b)</code>
	<code>a != b</code>	<code>a.__ne__(b)</code>	<code>b.__ne__(a)</code>	返回 <code>not (a==b)</code>
	<code>a > b</code>	<code>a.__gt__(b)</code>	<code>b.__lt__(a)</code>	引发 <code>TypeError</code> 异常

排序

接下页

⁴`pow(a,b,modulo)` 的第 3 个参数 `modulo` 是可选的。直接调用特殊方法时,也支持此参数,如 `a.__pow__(b,modulo)`。

续表 16.2

分组	中缀运算符	正向方法调用	反向方法调用	后备机制
	$a < b$	<code>a.__lt__(b)</code>	<code>b.__gt__(a)</code>	引发 <code>TypeError</code> 异常
	$a \geq b$	<code>a.__ge__(b)</code>	<code>b.__le__(a)</code>	引发 <code>TypeError</code> 异常
	$a \leq b$	<code>a.__le__(b)</code>	<code>b.__ge__(a)</code>	引发 <code>TypeError</code> 异常

了解这些规则之后,下面来回顾并改进 `Vector.__eq__` 方法的行为,该方法在 `vector_v5.py`(见“示例 12.16<337页>”)中的定义如下所示。

```

1 class Vector:
2     # 省略多行
3
4     def __eq__(self, other):
5         return (len(self) == len(other) and
6                 all(a == b for a, b in zip(self, other)))

```

该 `Vector.__eq__` 方法的行为,如示例 16.13 所示。

</> 示例 16.13: 将 `Vector` 实例与 `Vector` 实例、`Vector2d` 实例、元组进行比较

```

1 >>> va = Vector([1.0, 2.0, 3.0])
2 >>> vb = Vector(range(1, 4))
3 >>> va == vb      ❶
4 True
5 >>> vc = Vector([1, 2])
6 >>> from vector2d_v3 import Vector2d
7 >>> v2d = Vector2d(1, 2)
8 >>> vc == v2d    ❷
9 True
10 >>> t3 = (1, 2, 3)
11 >>> va == t3     ❸
12 True

```

- ❶ 具有相同数值分量的 2 个 `Vector` 实例是相等的。
- ❷ 如果 `Vector` 和 `Vector2d` 的分量相等,则它们也相等。
- ❸ 如果 `Vector` 实例的分量与元组或其他任何可迭代对象的数值项相等,那么它们也相等。

示例 16.13 中的结果可能不尽如人意。我们真的希望一个 `Vector` 与包含相同数值项的元组相等吗? 我对此没有硬性规定,这要取决于应用程序的具体场景。《Zen of Python》说过:

面对模棱两可的情况,要拒绝猜测的诱惑。

对操作数过于宽容,可能会导致令人惊讶的结果,而程序员讨厌这种惊讶。

从 Python 自身来看,我们发现 `[1,2] == (1,2)` 的结果是 `False`。因此,我们应该保守一点儿,做一些类型检查。如果第 2 个操作数是 `Vector` 实例(或 `Vector` 子类的实例),则使用与当前 `__eq__` 相同的逻辑。否则,返回 `NotImplemented`,让 Python 去处理。如示例 16.14 所示。

</> 示例 16.14: `vector_v8.py`:`Vector` 类改进的 `__eq__` 方法

```

1  def __eq__(self, other):
2      if isinstance(other, Vector): ❶
3          return (len(self) == len(other) and
4                  all(a == b for a, b in zip(self, other)))
5      else:
6          return NotImplemented ❷

```

- ❶ 如果另一个操作数是 Vector (或 Vector 子类) 的实例, 则按之前的方法进行比较。
 ❷ 否则, 返回 NotImplemented。

如果使用 [示例 16.14](#) 中的新版 Vector.__eq__ 运行 “[示例 16.13<466页>](#)” 中的测试, 得到的结果如 [示例 16.15](#) 所示。

</> [示例 16.15](#): 与 “[示例 16.13<466页>](#)” 一样的测试, 最后一个结果有变化

```

1  >>> va = Vector([1.0, 2.0, 3.0])
2  >>> vb = Vector(range(1, 4))
3  >>> va == vb      ❶
4  True
5  >>> vc = Vector([1, 2])
6  >>> from vector2d_v3 import Vector2d
7  >>> v2d = Vector2d(1, 2)
8  >>> vc == v2d      ❷
9  True
10 >>> t3 = (1, 2, 3)
11 >>> va == t3      ❸
12 False

```

- ❶ 结果与之前 (“[示例 16.13<466页>](#)”) 的相同, 符合预期。
 ❷ 结果与之前 (“[示例 16.13<466页>](#)”) 的相同, 但是为什么? 不应该是 False 么? 稍后解释。
 ❸ 结果与之前 (“[示例 16.13<466页>](#)”) 的不同, 这才是我们想要的。但是为什么会这样? 继续往下读 ...

在 [示例 16.15](#) 的 3 个结果中, 第 1 个没什么说的, 后 2 个是由于 [示例 16.14](#) 中 __eq__ 返回 NotImplemented 而引起的。比较 Vector 实例与 Vector2d 实例 (即 vc==v2d) 的具体步骤如下:

1. 为了求解 vc==v2d, Python 调用了 Vector.__eq__(vc, v2d)。
2. 经 Vector.__eq__(vc, v2d) 确认, v2d 不是 Vector 实例, 因此返回 NotImplemented。
3. Python 得到 NotImplemented 结果, 尝试调用 Vector2d.__eq__(v2d, vc)。Vector2d.__eq__ 的代码见 “[示例 12.16<337页>](#)”。
4. Vector2d.__eq__(v2d, vc) 将 2 个操作数都转换为元组, 并进行比较, 结果为 True。

在 [示例 16.15](#) 中, 比较 Vector 实例与元组 (va==t3) 的具体步骤如下:

1. 为了求解 va==t3, Python 会调用 Vector.__eq__(va, t3)。
2. 经 Vector.__eq__(va, t3) 确认, t3 不是 Vector 实例, 因此返回 NotImplemented。
3. Python 得到 NotImplemented 结果, 尝试调用 tuple.__eq__(t3, va)。
4. tuple.__eq__(t3, va) 不知道 Vector 是什么。因此, 返回 NotImplemented。

5. 对于 `==` 来说, 如果反向调用也返回 `NotImplemented`。Python 将比较对象的 ID, 作为最后一搏。显然 `va` 与 `t3` 两个对象 ID 不同。

无需实现支持`!=` 运算符的 `__ne__` 方法, 因为从 `object` 继承的 `__ne__` 的后备行为即可满足需求: 只要定义了 `__eq__` 方法, 而且不返回 `NotImplemented`, 则 `__ne__` 就会对 `__eq__` 返回的结果取反。

也就是说, 对示例 16.15 中的对象来说, 使用`!=` 运算符比较的结果都是正确的。

```

1  >>> va != vb
2  False
3  >>> vc != v2d
4  False
5  >>> va != (1, 2, 3)
6  True

```

从 `object` 继承而来的 `__ne__` 的工作原理与下面的代码类似, 不过原版代码是用 C 语言实现的⁵:

```

1  def __ne__(self, other):
2      eq_result = self == other
3      if eq_result is NotImplemented:
4          return NotImplemented
5      else:
6          return not eq_result

```

在介绍了中缀运算符重载的要点之后, 让我们来看看另一类运算符: 增量赋值运算符。

16.9 增量赋值运算符

`Vector` 类已经支持增量赋值运算符 `+=` 与 `*=` 了。这是因为当增量赋值运算符的接收者(即调用方法的对象)是可变对象时, 将新建实例, 并将新实例重新绑定到运算符的左侧变量上。

`+=` 与 `*=` 的实际使用如 ?? 所示。

```

</> 示例 16.16: 用 += 与 *= 操作 Vector 实例
1  >>> v1 = Vector([1, 2, 3])
2  >>> v1_alias = v1          ❶
3  >>> id(v1)               ❷
4  4302860128
5  >>> v1 += Vector([4, 5, 6]) ❸
6  >>> v1                   ❹
7  Vector([5.0, 7.0, 9.0])
8  >>> id(v1)               ❺
9  4302859904
10 >>> v1_alias            ❻
11 Vector([1.0, 2.0, 3.0])
12 >>> v1 *= 11             ❼
13 >>> v1                   ❼
14 Vector([55.0, 77.0, 99.0])

```

⁵`object.__eq__` 和 `object.__ne__` 的逻辑, 在 CPython 源代码 `Objects/typeobject.c` (4598 行) 中的 `object_richcompare` 函数中。

```
15 >>> id(v1)
16 4302858336
```

- ❶ 创建一个别名,以便稍后可以检查 `Vector([1, 2, 3])` 对象。
- ❷ 记住初始绑定到 `v1` 的 `Vector` 实例的 ID。
- ❸ 执行增量加运算。
- ❹ 结果符合预期 ...
- ❺ ... 但是,创建了一个新的 `Vector` 实例。
- ❻ 检查 `v1_alias`,以确认原始 `Vector` 未被更改。
- ❼ 执行增量乘法。
- ❽ 同样,结果符合预期,但是创建了新的 `Vector` 实例。

如果一个类未实现“表 16.1<464页>”列出的就地运算符,则增量运算符就是一种语法糖,即 `a+=b` 的计算与 `a=a+b` 完全一致。对于不可变类型,这是预期行为。而且,如果实现了 `__add__` 方法,那么无需编写额外的代码,即可使用 `+=`。

但是,如果确实实现了就地运算符方法(如 `__iadd__`),则会调用该方法来计算 `a += b` 的结果。顾名思义,这种运算符会就地更改左侧操作数,而不是创建新对象作为结果。



不可变类型(例如我们定义的 `Vector` 类)绝不应实现就地更改特殊方法。这一点很明显,但还是值得说一下。

为了展示就地运算符的代码,我们将扩展“示例 13.9<369页>”中的 `BingoCage` 类,实现 `__add__`、`__iadd__` 方法。

将 `BingoCage` 的子类命名为 `AddableBingoCage`。“示例 16.17<469页>”是我们想让运算符 `+` 拥有的行为。

</> 示例 16.17: 用运算符 `+`,新建 `AddableBingoCage` 实例

```
1 >>> vowels = 'AEIOU'
2 >>> globe = AddableBingoCage(vowels)      ❶
3 >>> globe.inspect()
4 ('A', 'E', 'I', 'O', 'U')
5 >>> globe.pick() in vowels                ❷
6 True
7 >>> len(globe.inspect())                  ❸
8 4
9 >>> globe2 = AddableBingoCage('XYZ')       ❹
10 >>> globe3 = globe + globe2
11 >>> len(globe3.inspect())                  ❺
12 7
13 >>> void = globe + [10, 20]                ❻
14 Traceback (most recent call last):
15 ...
16 TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and 'list'
```

- ❶ 创建一个 `globe` 实例, 该实例包含 5 项 (`vowels` 中的各个字母)。
- ❷ 从 `globe` 实例中取出 1 项, 确认该项包含在 `vowels` 中。
- ❸ 确认 `globe` 实例中只剩 4 项。
- ❹ 创建第 2 个实例, 其中包含 3 项。
- ❺ 将前 2 个实例相加, 以创建第 3 个实例。该实例包含 7 项。
- ❻ 尝试将 `AddableBingoCage` 实例添加到列表时, 将引发 `TypeError` 异常。当 `__add__` 方法返回 `NotImplemented` 时, Python 解释器会输出该错误消息。

由于 `AddableBingoCage` 是可变的, 其实现 `__iadd__` 方法之后的行为如 [示例 16.18](#) 所示。

</> [示例 16.18](#): 可以使用 `+=` 运算符载入现有的 `AddableBingoCage` 实例 (接续[示例 16.17](#))

```

1  >>> globe_orig = globe      ❶
2  >>> len(globe.inspect())   ❷
3  4
4  >>> globe += globe2       ❸
5  >>> len(globe.inspect())
6  7
7  >>> globe += ['M', 'N']    ❹
8  >>> len(globe.inspect())
9  9
10 >>> globe is globe_orig   ❺
11 True
12 >>> globe += 1            ❻
13 Traceback (most recent call last):
14 ...
15 TypeError: right operand in += must be 'Tombola' or an iterable

```

- ❶ 创建一个别名, 以便稍后检查对象的标识。
- ❷ `globe` 实例中包含 4 项。
- ❸ `AddableBingoCage` 实例可以从同属一类的其他实例中接收项。
- ❹ `+=` 的右操作数可以是任何可迭代对象。
- ❺ 在整个示例中, `globe` 始终与 `globe_orig` 引用相同的对象。
- ❻ 尝试将非可迭代对象与 `AddableBingoCage` 相加时, 将会失败并返回错误消息。

注意, 与运算符 `+` 相比, 运算符 `+=` 对第 2 个操作数的要求更宽松。运算符 `+` 的 2 个操作数必须是同种类型 (本例为 `AddableBingoCage`), 否则可能会导致结果类型的混乱。而使用 `+=` 时, 因为就地更改左侧操作数, 所以结果的类型是确定的。



通过观察内置类型 `list` 的工作方式, 验证了 `+` 与 `+=` 的行为差异。对于 `my_list + x`, 只能将 2 个 `list` 连接在一起。但对于 `my_list += x`, 则可以使用右侧可迭代对象 `x` 中的项, 来扩展左侧的 `list`——`list.extend()` 的行为也是如此, 它的参数可以是任何可迭代对象。

明确了 `AddableBingoCage` 的预期行为之后, 接下来看看其实现方式 (如 [示例 16.19](#) 所示)。回想一下, “[示例 13.9<369页>](#)” 中的 `BingoCage` 类是 “[示例 13.7<366页>](#)” 中抽象基类 (ABCs) `Tombola` 的具体子类

(Concrete Subclass)。

</> [示例 16.19: bingoaddable.py: AddableBingoCage 扩展了 BingoCage, 以支持 + 与 +=](#)

```

1  from tombola import Tombola
2  from bingo import BingoCage
3
4  class AddableBingoCage(BingoCage):          ❶
5      def __add__(self, other):
6          if isinstance(other, Tombola):        ❷
7              return AddableBingoCage(self.inspect() + other.inspect())
8          else:
9              return NotImplemented
10
11     def __iadd__(self, other):
12         if isinstance(other, Tombola):
13             other_iterable = other.inspect()    ❸
14         else:
15             try:
16                 other_iterable = iter(other)    ❹
17             except TypeError:                ❺
18                 msg = ('right operand in += must be '
19                         "'Tombola' or an iterable")
20                 raise TypeError(msg)
21             self.load(other_iterable)        ❻
22             return self                  ❼

```

- ❶ AddableBingoCage 扩展了 BingoCage。
- ❷ __add__ 方法的第 2 个操作数只能是 Tombola 实例。
- ❸ 在 __iadd__ 中, 如果 other 是 Tombola 实例, 则获取 other 中的项。
- ❹ 否则, 尝试用 other 创建迭代器⁶。
- ❺ 如果创建迭代器失败, 则引发 TypeError 异常, 告知用户应该怎么做。如果可能, 错误消息应明确引导用户找到解决方案。
- ❻ 如果执行到这里, 就将 other_iterable 加载到 self 中。
- ❼ 非常重要: 可变对象的增量赋值特殊方法必须返回 self。此处行为, 符合用户预期。

通过对比 [示例 16.19](#) 中 __add__ 与 __iadd__ 产生结果的 return 语句, 可以总结出就地运算符的整个思想:

- __add__
结果将通过调用 AddableBingoCage 构造函数, 生成一个新实例。
- __iadd__
修改后, 返回 self 即可得到结果。

最后, [示例 16.19](#) 还有一点需要注意: 按照设计, 在 AddableBingoCage 中没有编写 __radd__, 因为根本不需要它。因此, 如果 Python 试图计算 a + b, 其中 a 是 AddableBingoCage, 而 b 不是, 我们将返回

⁶ 内置函数 iter 将在下一章介绍。此处, 也可以使用 tuple(other), 但代价是需要构建一个元组, 会引入不必要的开销。而 [示例 16.19](#) 中❻处的 load(other_iterable) 方法只是需要遍历 other_iterable 中的项。

NotImplemented——此时,或许 b 所属的类可以接手继续处理计算(即 b.`__add__`(a))。但如果表达式是 b + a,而 b 不是 AddableBingoCage,并且返回 NotImplemented。那么,最好让 Python 放弃并引发 TypeError,因为我们无法处理 b。



一般来说,如果中缀运算符的正向方法(如 `__mul__`)只适用于与 `self` 相同类型的操作数,那么实现相应的反向方法(如 `__rmul__`)是没有用的。因为根据定义,只有在处理不同类型的操作数时,才会调用中缀运算符的反向方法。

对 Python 运算符重载的讨论,到此结束。

16.10 本章小结

本章首先回顾了 Python 对运算符重载的一些限制:不能为内置类型重载运算符,而且仅能重载现有的运算符,不过有几个例外(`is`、`and`、`or` 与 `not`)。

随后,讲解了如何重载一元运算符,实现了 `__neg__` 和 `__pos__` 方法。紧接着,讲解如何重载中缀运算符。首先是由 `__add__` 方法支持的中缀运算符 +。我们得知,一元运算符与中缀运算符应通过创建新对象来产生运算结果,而不应该改变它们的操作数。为了支持对其他类型的运算,需要返回 `NotImplemented` 特殊值(而不是异常),以允许 Python 解释器通过对调操作数并调用反向特殊方法(如 `__radd__`)再次运算。Python 用于处理中缀运算符的算法概述,如图 16.1 所示。

如果操作数类型不同,则需要检测出无法处理的操作数。本章使用两种方式处理此问题:第一种,利用 [鸭子类型 \(Duck Typing\)](#),直接尝试执行计算。如果有问题,则引发 `TypeError` 异常。第二种,使用 `isinstance` 执行显式类型检查,如 `__mul__` 与 `__matmul__` 方法就是这么做的。这两种方式各有利弊:[鸭子类型 \(Duck Typing\)](#) 更灵活,显式类型检查则更能预知结果。

一般来说,Python 库应该利用 [鸭子类型 \(Duck Typing\)](#),以为用户打开对象的大门:无论对象类型如何,只要它们支持必要的操作即可。然而,Python 的运算符分派算法与 [鸭子类型 \(Duck Typing\)](#) 结合时,可能会产生误导性的错误消息或意外的结果。因此,在编写用于运算符重载的特殊方法时,通常更有效的方法是使用 `isinstance` 调用对 [抽象基类 \(ABCs\)](#) 进行显式类型检查。这就是 Alex Martelli 称之为 [大鹅类型 \(Goose Typing\)](#) 的技术,详见“[13.5 大鹅类型 \(Goose Typing\) <358页>](#)”。[大鹅类型 \(Goose Typing\)](#) 可很好地平衡灵活性与安全性,因为现有或未来的用户定义类型可以声明为抽象基类的 [具体子类 \(Concrete Subclass\)](#) 或 [虚拟子类 \(Virtual Subclass\)](#)。此外,如果抽象基类 (ABCs) 实现了类方法 `__subclasshook__`,那么对象只要提供所需的方法(无需子类化或注册),即可通过针对抽象基类的 `isinstance` 检查。

接下来的主题是丰富的比较运算符。通过 `__eq__` 方法实现了比较运算符 `==`,并且发现 Python 在继承自 `object` 的 `__ne__` 方法中提供了 `!=` 的简便实现。Python 处理比较运算符的方式略有不同,它们有选择反向方法的特殊逻辑,以及对 `==` 与 `!=` 的回退处理:即不会引发异常,而是会比较对象 ID,以做最后一搏。

最后一节,专注于增量赋值运算符。Python 默认将它们处理为普通运算符与赋值操作的组合,即:`a += b` 被解释为 `a = a + b`。增量赋值运算符总是会创建一个新对象,因此它适用于可变类型或不可变类型。对于可变对象,可以实现就地更改特殊方法,如 `__iadd__` 用于 `+=`,并改变左操作数的值。为了举例说明,我们致力于实现一个 `BingoCage` 子类,以支持 `+=` 将项目添加到随机池中。这与内置类型 `list` 将 `+=` 视作 `list.extend()` 方法的快捷方式类似。在此过程中,了解到:在可接受的类型方面,`+` 通常比 `+=` 更严格。对于序列类型,`+` 通常要求 2 个操作数都是相同类型,而 `+=` 的右操作数通常可以是任何可迭代对象。

16.11 延伸阅读

Guido van Rossum 在 “Why Operators Are Useful”一文中为运算符重载做了很好的辩护。Trey Hunner 在博文 “Tuple ordering and deep comparisons in Python” 中指出, Python 中提供的比较运算符远比其他语言程序员想象的要强大、要灵活。

Python 中的运算符重载, 经常使用 `isinstance` 执行显式类型检查。这种类型检查充分利用了 [大鹅类型 \(Goose Typing\)](#) (详见 “13.5 大鹅类型 (Goose Typing) <358页>”)。若您跳过了该章节, 请务必重新阅读。

运算符特殊方法的主要参考资料是《The Python Language Reference》中的 “3. Data model” 一章。另一篇相关的文章是 “[numbers 模块](#)” 的 “[Implementing the arithmetic operations](#)”。

[pathlib 包](#) (Python 3.4 新增) 提供了一个巧妙的运算符重载示例。其 `Path` 类 重载了运算符 “/”, 以便于根据字符串构建文件系统路径, 如文档中下面这个示例所示:

```

1 >>> p = Path('/etc')
2 >>> q = p / 'init.d' / 'reboot'
3 >>> q
4 PosixPath('/etc/init.d/reboot')

```

在非算术运算符方面, 爬虫框架 [Scapy](#) 也利用了运算符重载。在 [Scapy](#) 中, 运算符 “/” 将来自不同网络层的字段堆叠起来构建分组。详见文档中的 “[Stacking layers](#)” 一节。

如要实现比较运算符, 可以研究一下 `functools.total_ordering`。这是一个类装饰器, 可以自动为具定义了几个比较运算符的类, 生成全部比较运算符。详见 [functools 模块文档](#)。

如果对动态类型语言的运算符方法分派机制感兴趣, 推荐阅读 2 篇具有开创性意义的论文:《[A simple technique for handling multiple polymorphism](#)》(Dan Ingalls⁷) 以及《[Arithmetic and double-dispatching in smalltalk-80](#)》(Kurt J. Hebel 与 Ralph Johnson⁸ 合著)。这 2 篇论文深刻揭示了多态在动态类型语言 (如 Smalltalk、Python 和 Ruby) 中的威力。Python 并未使用这些论文中所述的双分派来处理运算符。Python 使用的正向运算符与反向运算符, 更便于用户定义类型支持双分派 (double dispatching), 但需要 Python 解释器的特殊处理。相比之下, 经典的双分派 (double dispatching) 是一种通用技术, Python 与任何面向对象语言都可以使用, 而且不仅适用于中缀运算符, 例如本段 Ingalls、Hebel 及 Johnson 的 2 篇论文中, 描述双分派的示例。

本章开篇与 “[杂谈](#)” 中引用的几段话, 均出自 “[The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling](#)”⁹ 访谈。如果你对编程语言设计感兴趣, 不妨阅读一下这篇访谈。

杂谈

运算符重载的利弊 如本章开篇引文所述, James Gosling 在设计 Java 语言时, 有意决定放弃支持运算符重载。在那次访谈 (“[The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling](#)”) 中, 他指出:

⁷Dan Ingalls:Smalltalk 语言团队的初始成员之一。

⁸Ralph Johnson:因参与撰写《设计模式》而出名。

⁹刊登于 “Java Report, 5(7), July 2000” 与 “C++ Report, 12(7), July/August 2000”。

约有 20%~30% 的程序员, 认为运算符重载是万恶之源; 有些程序员利用运算符重载, 做了一些令人非常恼火的事情。例如, 用运算符 + 来插入列表, 导致编码一团糟。这些问题很大程度上源于一个事实: 世界上有成千上万个运算符, 但只有少数几个运算符适合重载。因此, 必须进行挑选, 而这些挑选有时会与直觉相冲突。

Guido van Rossum 对运算符重载采取了一种折中方式: 不允许用户随意创建新的运算符 (如 <=> 或:-), 这样可以防止用户对运算符的异想天开, 并且可使 Python 解释器保持简单。Python 也不允许重载内置类型的运算符, 这个限制可以提高代码可读性以及可预知的性能。

Gosling 接着指出:

社区中约有 10% 的程序员确实正确地使用与真正关心运算符重载, 对这些程序员来说, 运算符重载是及其重要的。这类程序员几乎都是从事数字工作的程序员。在这一领域中, 为了符合人类直觉, 表示法特别重要。因为, 他们进入这一领域时, 直觉中已经知晓符号 + 的含义, 他们知道 “ $a+b$ ” 中的 a 与 b 可以是复数、矩阵或其他合理的类型。

当然, 在语言中禁止运算符重载也有好处。我曾看到过这样的论点: 在系统编程方面, C 语言比 C++ 语言更好, 因为 C++ 语言中的运算符重载会让耗费资源的运算显得微不足道。Go 与 Rust 这 2 门现代语言都很成功, 它们能将代码编译成二进制可执行文件。但这 2 门语言却做出了截然不同的选择: Go 不支持运算符重载, 而 Rust 支持。

但是, 如果合理使用运算符重载, 确实可以使代码更易于阅读和编写。这是现代高级语言的一个很棒的特性。

惰性求值一瞥

如果仔细观察 “[示例 16.9<459页>](#)” 中的调用跟踪 (traceback), 您会发现生成器表达式 “惰性计算”的证据。[示例 16.20](#) 再次给出了那个调用跟踪, 不过加上了一些数字标号。

</> [示例 16.20: 与 “示例 16.9<459页>” 一样](#)

```

1  >>> v1 + 'ABC'
2  Traceback (most recent call last):
3      File "<stdin>", line 1, in <module>
4      File "vector_v6.py", line 329, in __add__
5          return Vector(a + b for a, b in pairs) ❶
6      File "vector_v6.py", line 243, in __init__
7          self._components = array(self.typecode, components) ❷
8      File "vector_v6.py", line 329, in <genexpr>
9          return Vector(a + b for a, b in pairs) ❸
10     TypeError: unsupported operand type(s) for +: 'float' and 'str'

```

❶ Vector 调用的 components 参数是一个生成器表达式, 这一步没问题。

❷ 将 components 生成器表达式传给 array 构造函数。在 array 构造函数中, Python 尝试遍历生成器表达式, 从而对第一项 $a+b$ 进行求值。这里引发 TypeError 异常。

❸ 异常向上传播到 Vector 构造函数调用中, 并在调用中报告出来。

这表明，生成器表达式不是在源码中定义它的位置求解，而是在最后时刻才求解。

相反，如果以 `Vector([a + b for a, b in pairs])` 的形式调用 `Vector` 构造函数，那么异常就会在这里发生。因为列表推导式 “[`a + b for a, b in pairs`]” 试图构建一个 `list` 作为参数传递给 `Vector()` 调用。因此，根本不会触及 `Vector.__init__` 方法的主体。

“[十七 迭代器、生成器和经典协程](#)”将详细介绍生成器表达式，但我不想错过 [示例 16.20](#) 中偶然出现的惰性求值特性的机会。

wechat: 119554488

PART IV

第四部分

控制流

- 第 17 章 迭代器、生成器和经典协程
- 第 18 章 `with`、`match` 和 `else` 代码块
- 第 19 章 Python 中的并发模型
- 第 20 章 并发执行器
- 第 21 章 异步编程

wechat: 119554488

迭代器、生成器和经典协程

当我在我的程序中发现了模式 (patterns) 时, 我认为可能是哪里出现了问题。一个程序的结构应该只反映出它需要解决的问题。至少对我来说, 程序中如果有额外的规律性, 表明我对问题的抽象不够深入——这些规律性代码需要编写宏扩展来自动生成。

——Paul Graham, Lisp 黑客和风险投资人 ^a

^a摘自博客文章《Revenge of the Nerds》。

迭代 (Iterator) 是数据处理的基础: 程序通常会将计算应用到数据 (无论是像素数据还是核苷酸数据) 序列上。如果内存空间无法一次性加载所有数据, 我们就需要惰性获取数据项 (即按需一次获取一个数据项)。这就是迭代器 (iterator) 的作用。本章将介绍如何将迭代器设计模式 (Iterator Design Pattern) 内置到 Python 语言中, 这样就无需手工编写这些代码了。Python 中的每个标准集合都是可迭代 (iterable) 的。可迭代对象是一个提供了迭代器 (iterator) 的对象, Python 用它来支持以下操作:

- for 循环
- list、dict 和 set 推导式 (comprehensions)
- 解包赋值
- collection 实例的构建

本章涵盖如下话题:

- Python 如何使用内置函数 `iter()` 处理可迭代 (iterable) 对象;
- 如何在 Python 中实现经典的迭代器模式 (Iterator Pattern);
- 如何用生成器 (generator) 函数或生成器表达式替代经典的迭代器模式;
- 生成器函数的详细工作原理 (逐行描述);
- 利用标准库中的通用生成器函数;
- 使用 `yield from` 表达式合并多个生成器;
- 为何生成器和经典协程看似相同, 实则差别很大, 不能混淆;

17.1 本章新增内容

`yield from` 表达式由第 1 版中的 1 页增加到 6 页, 通过更简单的实验演示了带有“`yield from`”的生成器的行为, 还一步一步地开发了遍历树形数据结构的示例。

新增的章节解释了 `Iterable`、`Iterator` 和 `Generator` 类型的 [类型提示 \(Type Hints\)](#)。

本章最后用 9 页的篇幅介绍了在第 1 版中占了 40 页篇幅的“[17.13 经典协程](#)”主题。我更新了“经典例程”一章, 并将它移到了[配套网站](#)的一个帖子中, 因为它对读者来说是最具挑战性的一章。但在 Python 3.5 引入了“原生协程”之后, “经典例程”就不那么重要了。关于“原生协程”的介绍, 详见“[二十一 异步编程](#)”。

接下来, 先讲解内置函数 `iter()` 如何使序列变得可迭代。

17.2 单词序列

将通过实现一个 `Sentence` 类来开始对可迭代类的探索: 该类可以接受一个包含文本的字符串作为构造参数, 并且可以逐个单词地进行迭代。`Sentence` 类的第 1 版将实现序列协议, 并且因为所有的序列都是可迭代对象, 所以 `Sentence` 也是可迭代对象。此处, 我们要具体说明为什么会这样。

[示例 17.1](#) 定义的 `Sentence` 类通过索引下标从文本中提取单词。

```
</> 示例 17.1: src/part04/sentence.py
1  import re
2  import reprlib
3  from typing import List
4
5  RE_WORD = re.compile(r'\w+')
6
7  class Sentence:
8
9      def __init__(self, text: str) -> None:
10         self.text = text
11         self.words = RE_WORD.findall(text) #(1)
12
13     def __getitem__(self, index: int) -> str:
14         return self.words[index] #(2)
15
16     def __len__(self) -> int: #(3)
17         return len(self.words)
18
19     def __repr__(self) -> str:
20         return 'Sentence(%s)' % reprlib.repr(self.text) #(4)
```

① `.findall` 返回一个列表, 其中包含所有符合正则表达式模式的不重叠的子字符串。

② `self.words` 存储了 `.findall(text)` 返回的结果; `self.words[index]` 返回 `self.words` 中指定索引下标的单词。

③ 为了完善序列协议, 此处实现了 `__len__` 方法。如果只是让对象可迭代, 可以不必实现此方法。

④ `reprlib.repr` 是一个实用程序, 用于生成大型数据结构的缩略字符串表示¹。

默认情况下, 函数 `reprlib.repr()` 将生成的字符串长度限制为 30 个字符。参见示例 17.2 中的控制台会话, 了解如何使用 `Sentence` 类。

</> 示例 17.2: 测试 `Sentence` 示例能否迭代

```
1  >>> from sentence import Sentence
2  >>> s = Sentence('The time has come," the Walrus said,') # (1)
3  >>> s
4  Sentence('The time ha... Walrus said,') # (2)
5  >>> for word in s: # (3)
6  ...     print(word)
7  ...
8  The
9  time
10 has
11 come
12 the
13 Walrus
14 said
15 >>> list(s) # (4)
16 ['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

① 传入一个字符串, 创建一个 `Sentence` 实例。

② 注意, `__repr__` 方法的输出中, `reprlib.repr()` 将过长的字符串缩略为符号 “...”。

③ `Sentence` 实例可以迭代, 稍后说明可以迭代的原因。

④ 因为 `Sentence` 对象可以迭代, 所以可通过 `Sentence` 对象构建 `list` 以及其他可迭代类型。

在接下来的几页中, 将开发能通过示例 17.2 测试的其他版本的 `Sentence` 类。不过, 示例 17.1 中的实现与后续的版本不同, 因为这一版的 `Sentence` 类也是一个序列, 所以可以按索引下标获取单词。

```
1 >>> s[0]
2 'The'
3 >>> s[5]
4 'Walrus'
5 >>> s[-1]
6 'said'
```

Python 程序员都知道序列是可迭代的。现在, 来看看为什么。

17.3 序列可迭代的原因:iter 函数

每当 Python 需要遍历对象 `x` 时, 它都会自动调用 `iter(x)`。内置函数 `iter` 将执行如下操作:

- ① 检查对象是否实现了 `__iter__` 方法。如果实现了, 调用 `__iter__` 获得一个迭代器。
- ② 若未实现 `__iter__`, 但实现了 `__getitem__` 方法, 那么 `iter()` 将创建一个迭代器, 尝试按索引下标 (从 0 开

¹首次使用`reprlib.html`模块, 是在“12.3 Vector 类第 1 版:与 Vector2d 类兼容<322页>”。

始) 获取每个成员元素。

③ 如果获取失败, Python 会引发 `TypeError`, 通常异常信息为 “`C` object is not iterable”, 其中 `C` 是目标对象的类名称。

因为所有 Python 序列都实现了 `__getitem__`, 所以都是可遍历的。事实上, 标准序列除了 `__getitem__` 方法, 还实现了 `__iter__`。您自定义创建的序列也应该实现 `__iter__` 方法, 因为通过 `__getitem__` 进行迭代是为了向后兼容, 将来可能会被移除 (Python 3.10 中还未移除)。

正如章节 “13.4.1 Python 深入理解序列” 中提到的, 这是鸭子类型 (duck types) 的一种极端形式: 实现了特殊方法 `__iter__` 或 `__getitem__` (如下所示) 的对象, 都被视作可迭代对象。

```

1  >>> class Spam:
2      ...     def __getitem__(self, i):
3      ...         print('>', i)
4      ...         raise IndexError()
5      ...
6
7  >>> spam_can = Spam()
8  >>> iter(spam_can)
9  <iterator object at 0x00000214B76C7F40>
10 >>> list(spam_can)
11 -> 0
12 []
13 >>> from collections import abc
14 >>> isinstance(spam_can, abc.Iterable)
15 False

```

如果一个类提供了 `__getitem__` 方法, 那么该类的内置函数 `iter()` 将接受该类的一个实例作为可迭代项 (iterable), 并从该实例构建一个迭代器。Python 的迭代机制将以索引 0 为起始, 调用 `__getitem__` 方法获取迭代项。当没有剩余迭代项时, 将抛出 `IndexError` 异常。

注意, 尽管 `spam_can` 是可迭代的 (其 `__getitem__` 方法可提供迭代项), 但执行 `isinstance(spam_can, abc.Iterable)` 的结果却是 `False`。

在大鹅类型 (Goose Typing) 理论中, 可迭代对象的定义更简单, 但不那么灵活。其定义为: 如果一个对象实现了 `__iter__` 方法, 则它就被认为是可迭代对象。这种方法无需显式的继承或注册, 因为 `abc.Iterable` 类实现了类方法 `__subclasshook__`, 可以动态判断一个类是否是可迭代的 (如 “13.5.8 用抽象基类实现结构类型<374页>” 所述)。示例如下:

```

1  >>> class GooseSpam:
2      ...     def __iter__(self):
3      ...         pass
4      ...
5
6  >>> from collections import abc
7  >>> issubclass(GooseSpam, abc.Iterable)
8  True
9  >>> goose_spam_can = GooseSpam()
10 >>> isinstance(goose_spam_can, abc.Iterable)
11 True

```



从 Python 3.10 开始, 检查对象 `x` 是否可迭代的最准确方法是调用 `iter(x)`。如果不可迭代, 则处理 `TypeError` 异常。这比使用 `isinstance(x, abc.Iterable)` 更准确, 因为 `iter(x)` 还会考虑传统的 `__getitem__` 方法, 而 `abc.Iterable` 抽象基类 则不会考虑传统的 `__getitem__` 方法。

在遍历一个对象之前, 显式检查对象是否可迭代, 或许没有必要, 毕竟尝试遍历不可迭代对象时, Python 抛出的异常信息已经很明确: “`TypeError: 'C' object is not iterable`”。如果除了引发 `TypeError` 异常之外还要做进一步处理, 则可以用 `try/except` 块取代 `isinstance` 显式检查。如果要保留对象以便稍后对其进行迭代, 则 `isinstance` 显式检查可能更有意义。因为, 在此情况下, 尽早捕获错误可以使调试更加容易。

内置函数 `iter()` 更常被 Python 自身使用, 我们自己很少用到。函数 `iter()` 还有另一种鲜为人知的用法, 如下一小节所示。

17.3.1 用 `iter` 处理可调用对象

调用 `iter()` 时, 传入 2 个参数可以为函数或任何可调用对象创建一个迭代器。对于此种用法, 第 1 个参数必须是一个可调用对象——可被重复调用(不传入参数)以产生值; 第 2 个参数是一个 `哨兵 (Sentinel)` 对象——一个标记值, 当可调用对象返回 `哨兵 (Sentinel)` 对象时, 会导致迭代器引发 `StopIteration` 异常, 而不是产生 `哨兵 (Sentinel)` 对象。

下述示例用函数 `iter()` 掷一个骰子, 直到点数为 1 为止。

```

1  >>> def d6():
2      ...     return randint(1, 6)
3
4  >>> d6_iter = iter(d6, 1)
5  >>> d6_iter
6  <callable_iterator object at 0x10a245270>
7  >>> for roll in d6_iter:
8      ...     print(roll)
9
10 4
11 3
12 6
13 3

```

请注意, 示例中的 `iter()` 函数返回一个 `callable_iterator`。`for` 循环可能会运行很长时间, 但永远不会显示 1, 因为 1 是 `哨兵值 (Sentinel Value)`。与迭代器一样, 示例中的 `d6_iter` 对象一旦耗尽就会失去作用。要重新开始, 必须再次调用 `iter()` 来重建迭代器。

`iter()` 文档 包括以下解释和示例代码:

`iter()` 的第二种形式可用于构建一个按块读取工具。例如, 从二进制数据库文件中读取固定宽度的块, 直到文件结束:

```

1 from functools import partial
2
3 with open('mydata.db', 'rb') as f:
4     read64 = partial(f.read, 64) ❶
5     for block in iter(read64, b''):
6         process_block(block)

```

为了表达清晰, 我在原示例的基础上增加了 `read64` 赋值 (❶处)。`functools.partial()` 函数不可省略, 因为传递给 `iter()` 的可调用对象必须是无参数的。在示例中, 哨兵 (Sentinel) 对象是一个空 bytes 对象——无字节可读时, `f.read` 返回的对象。

下一节, 将详细介绍可迭代对象与迭代器之间的关系。

17.4 可迭代对象与迭代器

根据 “[17.3 序列可迭代的原因: iter 函数](#)” 一节的描述, 可以推断出一个定义:

可迭代对象 (Iterable)

使用内置函数 `iter()` 可以从中获取迭代器的对象。若对象实现了能返回迭代器 (Iterator) 的 `__iter__` 方法, 那么该对象就是可迭代对象。序列都是可迭代对象。实现了 `__getitem__` 方法, 并且可接受从 0 开始的索引, 这种对象也是可迭代对象。

明确可迭代对象与迭代器之间的关系很重要: Python 从可迭代对象 (Iterable) 中获取迭代器 (Iterator)。

下面是一个简单的 `for` 循环, 遍历一个字符串。字符串 “ABC” 是这里的可迭代对象。表面上看不出来, 但其背后有一个迭代器:

```

1 >>> s = 'ABC'
2 >>> for char in s:
3 ...     print(char)
4 ...
5 A
6 B
7 C

```

如果没有 `for` 语句, 就只能用 `while` 循环来模拟 `for` 机制, 那我们就得这么写:

```

1 >>> s = 'ABC'
2 >>> it = iter(s) ❶
3 >>> while True:
4 ...     try:
5 ...         print(next(it)) ❷
6 ...     except StopIteration: ❸

```

```
7 ...     del it
8 ...     break
9 ...
10 A
11 B
12 C
```

- ① 根据可迭代对象，构建迭代器 `it`。
 - ② 不断地在迭代器上调用 `next`，以获取下一项。
 - ③ 没有剩余项时，迭代器引发 `StopIteration` 异常。
 - ④ 释放对 `it` 的引用——即迭代器对象被废弃。
 - ⑤ 退出循环。

StopIteration 表明迭代器已耗尽。内置函数 `iter()` 在内部自行处理 for 循环中的 StopIteration 异常, 以及其他迭代上下文(如列表推导式、可迭代对象解包等)中的 StopIteration 异常。

Python 的标准迭代器接口,包含 2 个方法:

- __next__
返回序列中的下一项。如果没有剩余的项，则引发 `StopIteration` 异常。
 - __iter__
返回 `self`，以便在需要可迭代对象的地方（如 `for` 循环中）使用迭代器。

该接口由 `抽象基类 (ABCs)` `collections.abc.Iterator` 确立。该抽象基类声明了抽象方法 (`Abstract Method`) `__next__`, 还从 `Iterable` 类继承了抽象方法 `__iter__`。详见 [??](#)。

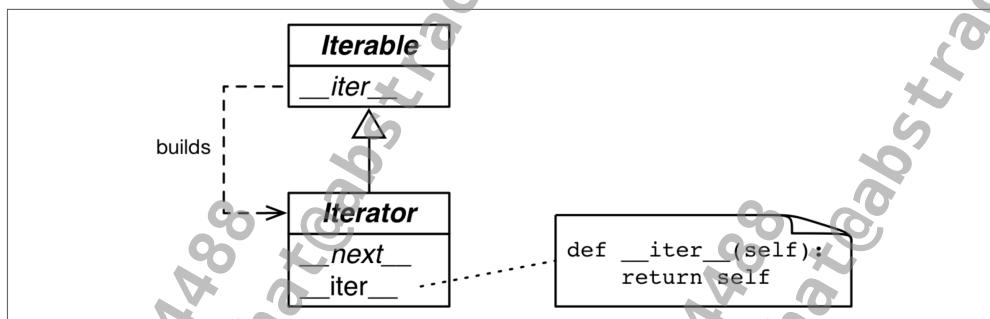


图 17.1: 抽象基类 Iterable 与 Iterator, 斜体显示的是抽象方法。

具体方法 (Concrete Method) `Iterable.__iter__` 应返回一个新的 `Iterator` 实例

具体类 (Concrete Class) Iterator 必须实现 `_next_` 方法

具体方法 (Concrete Method) `Iterator.__iter__` 直接返回实例自身

figure

`collections.abc.Iterator` 源码, 如示例 17.3 所示。

</> 示例 17.3: abc.Iterator 类的源码 (摘自 [Lib/_collections_abc.py](#) 271 行)

```
1 class Iterator(Iterable):  
2  
3     __slots__ = ()
```

```

4
5 @abstractmethod def __next__(self):
6     'Return the next item from the iterator. When exhausted, raise StopIteration'
7     raise StopIteration
8
9 def __iter__(self):
10    return self
11
12 @classmethod
13 def __subclasshook__(cls, C):
14    if cls is Iterator:
15        return __check_methods(C, '__iter__', '__next__') ❶
16    return NotImplemented

```

❶ 类方法 `__subclasshook__` 为 `isinstance` 和 `issubclass` 所做的结构类型检查提供支持, 详见 “13.5.8 用抽象基类实现结构类型<374页>”。

❷ `__check_methods` 遍历类的 `__mro__`, 检查方法是否在其基类中实现。它定义在同一个 `Lib/_collections_abc.py` 模块中。如果方法已实现, `C` 类将被认定为 `Iterator` 的虚拟子类 (Virtual Subclass), 换句话说, `issubclass(C, Iterable)` 将返回 `True`。



在 Python 3 中, 抽象基类 `Iterator` 定义的抽象方法是 `it.__next__()`, 而在 Python 2 中是 `it.next()`。如同往常, 应避免直接调用特殊方法, 使用内置函数 `next(it)` 即可。此函数在 Python 2 与 Python 3 中都能完成既定任务——这对于将代码从 python 2 迁移到 Python 3 很有帮助。

Python 3.9 中的 `Lib/types.py` 模块源代码有一段注释:

```

1 # Iterators in Python aren't a matter of type but of protocol. A large
2 # and changing number of builtin types implement *some* flavor of
3 # iterator. Don't check the type! Use hasattr to check for both
4 # "__iter__" and "__next__" attributes instead.

```

事实上, 这正是抽象基类 `abc.Iterator` 中类方法 `__subclasshook__` 所起的作用。



根据 `Lib/types.py` 中的建议, 以及 `Lib/_collections_abc.py` 中的实现逻辑, 检查对象 `x` 是否为迭代器, 最佳方式是调用 `isinstance(x, abc.Iterator)`。得益于类方法 `Iterator.__subclasshook__`, 即使对象 `x` 所属的类不是 `Iterator` 的真实子类或虚拟子类 (Virtual Subclass), 这个 `isinstance` 检查也能正常工作。

回到 “示例 17.1<480页>” 定义的 `Sentence` 类, 通过 Python 控制台可清楚地看出 `iter()` 是如何构建迭代器的, 以及 `next()` 是如何消耗迭代器的。

```

1 >>> s3 = Sentence('Life of Brian') ❶
2 >>> it = iter(s3) ❷
3 >>> it # doctest: +ELLIPSIS
4 <iterator object at 0x...>
5 >>> next(it) ❸

```

```
6  'Life'
7  >>> next(it)'of'
8  >>> next(it)
9  'Brian'
10 >>> next(it)          ④
11 Traceback (most recent call last):
12 ...
13 StopIteration
14 >>> list(it)          ⑤
15 []
16 >>> list(iterator(s3)) ⑥
17 ['Life', 'of', 'Brian']
```

- ① 创建一个 Sentence 实例 s3, 包含 3 个单词。
- ② 根据 s3, 构建一个迭代器 it。
- ③ 调用 `next(it)`, 获取迭代器 it 中的下一个单词。
- ④ 没有多余的单词, 因此迭代器引发 `StopIteration` 异常。
- ⑤ 迭代器一旦耗尽, 就会引发 `StopIteration` 异常, 使其看起来像是空的。
- ⑥ 如果想再次遍历 s3, 则必须重新构建迭代器。

由于迭代器只需要 `__next__` 与 `__iter__` 这 2 个方法, 因此除了调用 `next()` 与捕获 `StopIteration` 异常之外, 无法检查是否还有剩余项。此外, 也无法“重置”迭代器。如果想要再次迭代, 则必须调用 `iter()`, 并传入最初建立迭代器的可迭代对象。传入迭代器本身也没用, 因为如前所述, `Iterator.__iter__` 方法的实现方式是返回 `self`, 所以无法重置已耗尽的迭代器。

这种简化的接口是合理的, 因为现实中并非所有迭代器都可以重置。例如, 从网络中读取数据包的迭代器就不能重置²。

得益于内置函数 `iter()` 对序列的特殊处理, “[示例 17.1<480页>](#)” 中的第 1 版 Sentence 类是可迭代的。接下来, 将定义另一版 Sentence 类, 实现 `__iter__` 方法, 返回迭代器。

17.5 为 Sentence 类实现 `__iter__` 方法

接下来的几版 Sentence 类将实现标准的可迭代协议。首先, 通过迭代器设计模式实现; 然后, 利用生成器函数实现。

17.5.1 Sentence 类第 2 版: 经典迭代器

接下来的 Sentence 实现, 遵循了《设计模式》一书中的经典迭代器设计模式。需要注意的是, 这并不符合 Python 的惯用方法, 后面重构时会说明原因。但这一版能明确可迭代容器与迭代器之间的区别。

[示例 17.4](#) 中的 Sentence 类是可迭代的, 因为它实现了特殊方法 `__iter__`, 该方法构建并返回一个 `SentenceIterator` 实例。这就是可迭代对象与迭代器之间的关系。

```
</> 示例 17.4: sentence_iter.py: 用迭代器模式实现 Sentence 类
```

²感谢技术审校 Lennardo Rochael 提供的这个精妙的示例。

```

1 import re
2 import reprlib
3
4 RE_WORD = re.compile(r'\w+')
5
6 class Sentence:
7
8     def __init__(self, text):
9         self.text = text
10        self.words = RE_WORD.findall(text)
11
12     def __repr__(self):
13         return f'Sentence({reprlib.repr(self.text)})'
14
15     def __iter__(self):          ❶
16         return SentenceIterator(self.words) ❷
17
18 class SentenceIterator:
19
20     def __init__(self, words):
21         self.words = words ❸
22         self.index = 0 ❹
23
24     def __next__(self):
25         try:
26             word = self.words[self.index] ❺
27         except IndexError:
28             raise StopIteration() ❻
29         self.index += 1 ❻
30         return word ❻
31
32     def __iter__(self): ❾
33         return self

```

- ❶ 与 Sentence 类第 1 版（见“[示例 17.1<480页>](#)”）相比，这里只多了一个 `__iter__` 方法。这一版的 Sentence 类没有 `__getitem__` 方法，其目的是明确表明这个类之所以可以迭代，是因为实现了 `__iter__` 方法。
- ❷ `__iter__` 方法通过实例化并返回一个迭代器，来实现可迭代协议。
- ❸ SentenceIterator 保存了对单词列表的引用。
- ❹ `self.index` 确定下一个要获取的单词。
- ❺ 获取 `self.index` 索引位上的单词。
- ❻ 如果 `self.index` 索引位上没有单词，则引发 `StopIteration` 异常。
- ❼ 递增 `self.index` 的值。
- ❽ 返回单词。
- ❾ 实现 `self.__iter__` 方法。

示例 17.4 中的代码,可以通过示例 17.2 中的测试。

注意,在 `Sentencelerator` 中实现 `__iter__` 方法,并不是这个示例所必需的,但实现 `__iter__` 是正确的做法:迭代器应同时实现 `__next__` 与 `__iter__` 方法,这样做可以使我们的迭代器通过 `issubclass(Sentencelerator, abc.Iterator)` 测试。如果让 `Sentencelerator` 子类化 `abc.Iterator`,那么 `Sentencelerator` 会继承具体方法 `abc.Iterator.__iter__`。

这一版的工作量很大(对习惯了便利的 Python 程序员来说确实如此)。请注意, `Sentencelerator` 中的大部分代码都是用于管理迭代器内部状态的,稍后会说明如何简化。但在此之前,先绕一个弯子,讨论一个看似合理实则错误的实现捷径。

17.5.2 勿让可迭代对象变成其自己的迭代器

构建可迭代对象与迭代器时,常见的一个出错原因是混淆了二者。明确地说:可迭代对象有一个 `__iter__` 方法,每次调用都会实例化一个新的迭代器;而迭代器要实现一个 `__next__` 方法(返回迭代器下一个元素),以及 `__iter__` 方法(返回迭代器自身)。³

因此,迭代器也是可迭代对象,但可迭代对象不是迭代器。

除了实现 `__iter__` 方法之外,您可能还想为 `Sentence` 类实现 `__next__` 方法,以使 `Sentence` 实例既是可迭代对象,又是自身的迭代器,这可能很诱人。但几乎很少是个好主意。Alex Martelli 在 Google 拥有丰富的 Python 代码审查经验,他表示,这也是一种常见的反模式。

《设计模式》一书关于迭代器设计模式的“适用性”部分提到:

迭代器模式可用于:

- 访问聚合对象的内容,而不会暴露其内部表示。
- 支持聚合对象的多次遍历。
- 为遍历不同的聚合结构提供统一的接口(即支持多态迭代)。

为了“支持多次遍历”,就必须能够从同一个可迭代实例中获得多个独立的迭代器,而且每个迭代器都必须保持自己的内部状态,因此该模式的正确实现要求每次调用 `iter(my_iterable)` 都要创建一个新的、独立的迭代器。这就是本例中需要 `Sentencelerator` 类的原因。

至此,已演示了如何正确实现经典迭代器模式,就可以放手去做了。Python 从 Barbara Liskov 发明的 `CLU` 语言中引入了关键字 `yield`。因此,无须自己动手编写实现迭代器的代码。

下一节,将展示如何使用更惯用的方式实现 `Sentence` 类。

17.5.3 Sentence 类第 3 版:生成器函数

使用生成器(generator)来实现相同功能,可以避免实现 `Sentencelerator` 类所需的所有工作,并且更符合 Python 风格。示例 17.5 之后会有一个对生成器的详细解释。

</> 示例 17.5: `sentence_gen.py`:用生成器实现 Sentence 类

³译者注:迭代器(Iterator)和可迭代对象(Iterable)的主要区别在于,可迭代对象是包含多个元素并具有创建迭代器的 `__iter__` 方法,而迭代器则是用于逐个返回可迭代对象中的元素的对象,包含 `__next__` 与 `__iter__` 方法。

```

1 import re
2 import reprlib
3 RE_WORD = re.compile(r'\w+')
4 class Sentence:
5
6     def __init__(self, text):
7         self.text = text
8         self.words = RE_WORD.findall(text)
9
10    def __repr__(self):
11        return 'Sentence(%s)' % reprlib.repr(self.text)
12
13    def __iter__(self):
14        for word in self.words: ❶
15            yield word ❷
16        ❸
17    # done! ❹

```

- ❶ 遍历 self.words。
- ❷ 产生当前的 word。
- ❸ 无需明确的 return 语句。函数可以“穿越 (fall through)”并自动返回。无论有没有 return 语句,生成器函数都不会引发 StopIteration 异常,而是在全部值都生成完毕后直接退出。⁴
- ❹ 无需定义单独的迭代器类。

示例 17.5 用另一种不同的方式实现了 Sentence 类,并且也能通过“示例 17.2<481页>”中的测试。

回顾“示例 17.4<487页>”中的 Sentence 代码, __iter__ 调用了 SentenceIterator 构造函数来构建并返回一个迭代器。现在,示例 17.5 中的迭代器实际上是一个生成器对象,它是在调用 __iter__ 方法时自动构建的,因为这里的 __iter__ 是一个生成器函数。

关于生成器工作原理的完整解释,见下一小节。

17.5.4 生成器工作原理

任何在其主体中包含 yield 关键字的 Python 函数都是生成器函数:调用时返回生成器对象的函数。换句话说,生成器函数就是一个生成器工厂。



区分普通函数和生成器函数的唯一语法是后者在其主体某处有一个 yield 关键字。有些人认为,应该使用一个新的关键字(如 gen)取代 def 来声明生成器函数,但 Guido 并不同意。他的理由详见“PEP 255 –Simple Generators”。

示例 17.6 以一个简单的生成器函数为例,展示其行为。

</> 示例 17.6: 产生 3 个数字的生成器函数

⁴在审查这段代码时, Alex Martelli 建议简化这个方法主体,直接用 return iter(self.words)。当然,他说的也对:毕竟调用 self.words.__iter__() 得到的就是迭代器。不过,此处我用带 yield 关键字的 for 循环,是为了介绍生成器函数的语法(详见“17.5.4 生成器工作原理”)。在审校本书第 2 版时,Leonardo Rochael 为 __iter__ 方法主体提出了另一种捷径:yield from self.words。本章后面还会介绍 yield from。

```
1  >>> def gen_123():
2      ...     yield 1  ❶
3      ...     yield 2
4      ...     yield 3
5      ...
6
7  >>> gen_123 # doctest: +ELLIPSIS
8  <function gen_123 at 0x...> ❷
9
10 >>> gen_123() # doctest: +ELLIPSIS
11 <generator object gen_123 at 0x...> ❸
12 >>> for i in gen_123():
13     ...     print(i) ❹
14
15 1
16 2
17 3
18
19 >>> g = gen_123() ❺
20 >>> next(g) ❻
21 1
22 >>> next(g)
23 2
24 >>> next(g)
25 3
26 >>> next(g) ❼
27 Traceback (most recent call last):
28 ...
29 StopIteration
```

- ❶ 生成器函数的主体通常会在一个循环内设置 `yield`, 但并非必须如此; 在这里, 我只是重复了 3 次 `yield`。
- ❷ 仔细看, `gen_123` 是一个函数对象。
- ❸ 但在调用 `gen_123()` 时, 它会返回一个生成器对象。
- ❹ 生成器对象实现了 `Iterator` 接口, 因此生成器对象也是可迭代的。
- ❺ 将新的生成器对象赋值给 `g`, 以便对其进行实验。
- ❻ 由于 `g` 是一个迭代器, 所以调用 `next(g)` 会获取由 `yield` 产生的下一个项。
- ❼ 当生成器函数返回时, 生成器对象引发 `StopIteration` 异常。

生成器函数构建了一个生成器对象, 该对象封装了函数主体。当在生成器对象上调用 `next()` 时, 执行会前进到函数体中的下一个 `yield`, 并且 `next()` 调用的计算结果是函数体暂停时 `yield` 生成的值。最后, 根据迭代器协议, 当函数体返回时, Python 创建的外层生成器对象将引发 `StopIteration`。



我发现, 在谈论从生成器中获得的值时, 保持严谨是很有帮助的。说生成器“返回 (return)”值会引起混淆。因为, 函数才会返回 (return) 值。调用生成器函数会返回生成器, 而生成器会产生值。生成器不以常规方式“返回”值: 生成器函数主体中的 `return` 语句会导致生成器对象引发 `StopIteration`。如果生成器中包含 `return x`, 则调用方可从 `StopIteration` 异常中获取 `x` 的值, 但通常将这个操作交由 `yield from` 语法自动完成 (详见“17.13.2 让协程返回一个值<521页>”)。

示例 17.7 用 for 循环更清楚地说明了生成器函数主体的执行过程。

</> 示例 17.7: 运行时打印消息的生成器函数

```

1  >>> def gen_AB():
2      ...     print('start')
3      ...     yield 'A'      ❶
4      ...     print('continue')
5      ...     yield 'B'      ❷
6      ...     print('end.')
7      ...
8  >>> for c in gen_AB():
9      ...     print('-->', c)
10     ...
11 start      ❸
12 --> A      ❹
13 continue   ❺
14 --> B      ❻
15 end.
16 >>>      ❼

```

- ❶ for 循环中对 next() 的第 1 次隐式调用 (❶处) 将打印 “start”, 并在第 1 个 yield 处停止, 产生值 “A”。
- ❷ for 循环中对 next() 的第 2 次隐式调用 (❷处) 将打印 “continue”, 并在第 2 个 yield 处停止, 产生值 “B”。
- ❸ for 循环中对 next() 的第 3 次隐式调用 (❸处) 将打印 “end”, 并 “穿越 (fall through) ” 至 gen_AB 函数体末尾, 导致生成器对象引发 StopIteration 异常。
- ❹ 遍历时, for 机制与 g=iter(gen_AB()) 的作用一样, 用于获取生成器对象, 并在每次遍历时调用 next(g)。
- ❺ 循环打印 “->” 与 next(g) 返回的值。但是, 此输出仅在生成器函数 gen_AB 内的 print 之后出现。
- ❻ 文本 start 来自生成器主体中的 print('start')。
- ❼ 生成器主体中的 “yield 'A'” (❶处) 产生 for 循环所消耗的值 A, 并将其赋值给变量 c, 从而输出 “-> A”。
- ❽ 继续遍历, 第 2 次调用 next(g), 将生成器主体从 “yield 'A'” 推进到 “yield 'B'”。文本 continue 由生成器主体中的第 2 次打印输出。
- ❾ 生成器主体中的 “yield 'B'” (❷处) 产生 for 循环所消耗的值 B, 并将其赋值给变量 c, 从而输出 “-> B”。
- ❿ 继续遍历, 第 3 次调用 next(g), 将生成器主体从 “yield 'B'” 推进到函数体的末尾。由于在生成器主体中进行了第 3 次打印, 输出中出现了 “end”。
- ❼ 当生成器函数运行到最后时, 生成器对象会引发 StopIteration。for 循环机制捕获该异常, 从而使循环干净利落地结束。

现在, 希望大家已经清楚 “[示例 17.5<489页>](#)” 中的 Sentence.__iter__ 是如何工作的了: __iter__ 是一个生成器函数, 它被调用时会生成一个实现 Iterator 接口的生成器对象, 因此不再需要 SentenceIterator 类。

此版本的 Sentence 类比前一版 () 更简洁, 但还是不够惰性。如今, 惰性被认为是一个好的特质, 至少在编程语言与 API 中是如此。惰性实现是将生成值尽可能推迟到最后时刻。这样, 可以节省内存, 还可以避免浪费 CPU 周期。

下一节, 将以这种惰性方式构建 Sentence 类。

17.6 惰性版本的 Sentence 类

Sentence 的最后 2 个版本,将使用 `re` 模块 中的惰性函数实现。

17.6.1 Sentence 类第 4 版:惰性生成器

`Iterator` 接口被设计为惰性的:即 `next(my_iterator)` 一次生成一项。惰性求值的反义词是急性求值:惰性 (lazy) 求值与急性 (eager) 求值是编程语言理论中的专业术语。

目前实现的几版 Sentence 类都不是惰性的,因为 `__init__` 会及早地构建好文本中的单词列表,再将其绑定到 `self.words` 属性。这需要处理整个文本,并且列表可能会占用与文本本身一样多的内存(也可能更少,这取决于文本中有多少非单词字符)。若用户仅遍历前几个单词,那么大部分工作都是徒劳的。如果您想知道“Python 中有没有一种惰性方法可以做到这一点?”答案通常是“有”。

`re.finditer` 函数 是 `re.findall` 的惰性版本。`re.finditer` 返回一个生成器(而非列表),该生成器会根据需要生成 `re.MatchObject` 实例。如果匹配项较多,则 `re.finditer` 可以节省大量内存。接下来,将使用 `re.finditer` 函数 实现一个惰性版本的 Sentence 类,即仅在需要时才从文本中读取下一个单词,代码如 [示例 17.8](#) 所示。

</> [示例 17.8: sentence_gen2.py:用 re.finditer 函数实现惰性 Sentence 类](#)

```

1  import re
2  import reprlib
3
4  RE_WORD = re.compile(r'\w+')
5
6  class Sentence:
7
8      def __init__(self, text):
9          self.text = text
10
11     def __repr__(self):
12         return f'Sentence({reprlib.repr(self.text)})'
13
14     def __iter__(self):
15         for match in RE_WORD.finditer(self.text): ①
16             yield match.group() ②

```

- ① 无需构建 `words` 列表。
- ② `re.finditer` 函数 在 `self.text` 的 `RE_WORD` 匹配结果上构建一个迭代器,生成 `MatchObject` 实例。
- ③ `match.group()` 从 `MatchObject` 实例 中提取匹配的文本。

生成器已经使代码很简洁了,但如果使用生成器表达式可以使代码更简洁。

17.6.2 Sentence 类第 5 版:惰性生成器表达式

可以用生成器表达式来取代[示例 17.8](#)中简单的生成器函数。正如列表推导式可以构建列表一样,生成器表达式可以构建生成器对象。二者之间的行为比较,如[示例 17.9](#)所示。

</> 示例 17.9: 生成器函数 gen_AB 先由列表推导式使用,再由生成器表达式使用

```

1  >>> def gen_AB():
2      ...     print('start')
3      ...     yield 'A'
4      ...     print('continue')
5      ...     yield 'B'
6      ...     print('end.')
7
8  >>> res1 = [x*3 for x in gen_AB()] ❷
9  start
10 continue
11 end.
12 >>> for i in res1: ❸
13     ...     print('-->', i)
14 ...
15 --> AAA
16 --> BBB
17 >>> res2 = (x*3 for x in gen_AB()) ❹
18 >>> res2
19 <generator object <genexpr> at 0x10063c240>
20 >>> for i in res2: ❺
21     ...     print('-->', i)
22 ...
23 start
24 --> AAA
25 continue
26 --> BBB
27 end.

```

❶ gen_AB 函数与 “[示例 17.7<492页>](#)” 中的一样。

❷ 列表推导式会急切地遍历由 gen_AB() 返回的生成器对象产生的项:“A” 和 “B”。请注意后面几行的输出是: start、continue、end。

❸ 此 for 循环遍历由列表推导式建立的 res1 列表。

❹ 生成器表达式返回生成器对象 res2。此处不消耗生成器。

❺ 仅当 for 循环遍历 res2 时, 该生成器才会从 gen_AB 中获取项。for 循环的每次遍历都会隐式调用 next(res2), 而 next(res2) 又会调用 gen_AB() 返回的生成器对象的 next(), 将其推进到下一个 yield 值。

❻ 注意, gen_AB() 的输出与 for 循环中的 print 输出交替出现。

可以用生成器表达式进一步精简 Sentence 类的代码, 精简后的 Sentence 类如 [示例 17.10](#)

</> 示例 17.10: sentence_genexp.py: 用生成器表达式实现 Sentence 类

```

1  import re
2  import reprlib
3
4  RE_WORD = re.compile(r'\w+')
5
6  class Sentence:

```

```

7
8     def __init__(self, text):          self.text = text
9
10    def __repr__(self):
11        return f"Sentences({reprlib.repr(self.text)})"
12
13    def __iter__(self):
14        return (match.group() for match in RE_WORD.finditer(self.text))

```

与 [示例 17.8](#) 的唯一区别是 `__iter__` 方法, 这里的 `__iter__` 不是生成器函数 (没有 `yield`), 而是使用生成器表达式构建生成器, 然后返回它。最终结果是相同的: `__iter__` 的调用方将得到一个生成器对象。

生成器表达式是 [语法糖 \(Syntactic Sugar\)](#), 完全可以被替换为生成器函数, 但有时生成器表达式更方便。下一节, 将介绍关于生成器表达式的用法。

17.7 何时用生成器表达式

在 “[示例 12.16<337页>](#)” 中, 为了实现 `Vector` 类, 我使用了多个生成器表达式: `__eq__`、`__hash__`、`__abs__`、`angle`、`angles`、`format`、`__add__` 与 `__mul__` 方法中各有一个。这些特殊方法中的生成器表达式也可以替换为列表推导式, 不过代价是使用更多的内存来存储中间的列表值。

由 [示例 17.10](#) 可知, 生成器表达式是构建生成器的捷径, 无需定义与调用函数。另一方面, 生成器函数更加灵活: 可以用多个语句编写复杂的逻辑, 甚至可以作为协程 (详见 “[17.13 经典协程<518页>](#)”) 使用。

对于较简单的情况, 生成器表达式更容易一目了然, 如 `Vector` 示例 (见 “[示例 12.16<337页>](#)”) 所示。

根据我的经验, 编码时如何选择要使用的语法很简单: 如果生成器表达式的行数超过多行, 为了可读性, 我更倾向于使用生成器函数。



语法提示

当生成器表达式作为单参数传递给函数或构造函数时, 不需要为函数调用写一对括号, 再单独为生成器表达式写一对括号。只写一对括号就可以了, 如 “[示例 12.16<337页>](#)” 中 `__mul__` 方法的 `Vector` 调用, 在此重现如下:

```

1  def __mul__(self, scalar):
2      if isinstance(scalar, numbers.Real):
3          return Vector(n * scalar for n in self)      # 只写
4          一对括号即可, 无需单独为生成器表达式写一对括号
5      else:
6          return NotImplemented

```

但是, 如果在生成器表达式之后有更多的函数参数, 则需要用圆括号括起来, 以避免语法错误。

目前所见的 `Sentence` 示例, 展示了生成器在经典迭代器设计模式中的作用: 即从容器中获取项。不过, 生成器还可用于生成独立于数据源的值。下一节, 将举例说明这一用法。

但首先, 先简短讨论一下 “[迭代器 \(iterator \)](#)” 与 “[生成器 \(generator \)](#)” 这 2 个概念的重叠之处。

迭代器与生成器的对比

在 Python 官方文档和代码库中, 有关迭代器和生成器的术语前后不一且不断变化。本书采用了以下定义:

- **迭代器 (iterator)**

泛指实现了 `__next__` 方法的对象。迭代器用于生成供客户代码消耗的数据, 即通过 `for` 循环或其他遍历方式, 或直接在迭代器上调用 `next(it)` 驱动迭代器。不过, 显式调用 `next()` 并不常见。实际上, 在 Python 中使用的大多数迭代器都是生成器。

- **生成器 (generator)**

由 Python 编译器构建的迭代器。要创建一个生成器, 我们不实现 `__next__`, 而是用 `yield` 关键字创建一个生成器函数, 它是生成器对象的工厂。生成器表达式是构建生成器对象的另一种方法。生成器对象提供了 `__next__` 方法, 所以生成器对象也是迭代器。从 Python 3.5 开始, 可以用 `async def` 声明异步生成器(详见“[二十一 异步编程<619页>](#)”)。

Python 术语表最近引入了“生成器迭代器”一词, 用来指由生成器函数构建的生成器对象, 而“生成器表达式”条目则说它返回一个“迭代器”。

但根据 Python 的说法, 这两种情况返回的都是生成器对象:

```

1  >>> def g():
2      ...
3      yield 0
4
5  >>> g()
6  <generator object g at 0x10e6fb290>
7
8  >>> ge = (c for c in 'XYZ')
9
10 >>> type(g()), type(ge)
11 (<class 'generator'>, <class 'generator'>)

```

17.8 等差数列生成器

经典的迭代器模式都是关于遍历的——遍历某些数据结构。不过, 即便不从容器中获取下一项, 而是从序列中获取动态生成的下一项时, 这种基于方法的标准接口也很有用处。例如, 内置函数 `range()` 可以生成整数的有界等差数列 (AP, arithmetic progression)。如果要生成任意数字 (不仅仅是整数) 类型的 AP, 该怎么办?

[示例 17.11](#) 展示了对 `ArithmeticProgression` 类(见 [示例 17.12](#))的一些控制台测试。[示例 17.11](#) 中构造函数的签名是 `ArithmeticProgression(begin, step[, end])`。内置函数 `range` 的完整签名是 `range(start, stop[, step])`。我选择实现不同的签名, 是因为创建等差数列时必须指定步长 (`step`), 但末项 (`end`) 是可选的。我还把参数名称由“`start/stop`”更改为“`begin/end`”, 以明确表明函数签名不同。[示例 17.11](#) 中的每个测试都对结果调用 `list()`, 以检查生成的值。

</> [示例 17.11](#): 演示 `ArithmeticProgression` 类(见 [示例 17.12](#))的用法

```

1  >>> ap = ArithmeticProgression(0, 1, 3)
2  >>> list(ap)[0, 1, 2]
3  >>> ap = ArithmeticProgression(1, .5, 3)
4  >>> list(ap)
5  [1.0, 1.5, 2.0, 2.5]
6  >>> ap = ArithmeticProgression(0, 1/3, 1)
7  >>> list(ap)
8  [0.0, 0.3333333333333333, 0.6666666666666666]
9  >>> from fractions import Fraction
10 >>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
11 >>> list(ap)
12 [Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
13 >>> from decimal import Decimal
14 >>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
15 >>> list(ap)
16 [Decimal('0'), Decimal('0.1'), Decimal('0.2')]

```

注意,在得到的等差数列中,数字的类型与“begin + step”的类型一致。并且如有需要,等差数列中的数字会根据 Python 算术运算的规则,执行强制类型转换。在示例 17.11 中,我们得到了 int、float、Fraction 和 Decimal 数字构成的列表。ArithmeticProgression 类的实现代码如示例 17.12 所示。

</> 示例 17.12: ArithmeticProgression 类的实现

```

1  class ArithmeticProgression:
2
3      def __init__(self, begin, step, end=None): ❶
4          self.begin = begin
5          self.step = step
6          self.end = end # None -> "infinite" series
7
8      def __iter__(self):
9          result_type = type(self.begin + self.step) ❹
10         result = result_type(self.begin) ❺
11         forever = self.end is None ❻
12         index = 0
13         while forever or result < self.end: ❼
14             yield result ❽
15             index += 1
16             result = self.begin + self.step * index ❾

```

- ❶ `__init__` 需要 2 个必选参数: `begin` 与 `step`; 一个可选参数 `end`。如果 `end=None`, 则生成无穷数列。
- ❷ 获取 `self.begin` 与 `self.step` 相加之和的类型。例如, 若一个是 `int`、另一个是 `float`, 则 `result_type` 将是 `float`。
- ❸ 此行将 `self.begin` 赋值给 `result`。但是, 在赋值之前先将 `self.begin` 强制转换为❹处加法算式得到的类型。⁵

⁵ 在 Python 2 中, 有一个 `coerce()` 内置函数, 但它在 Python 3 中消失了——因为数字强制规则隐含在算术运算符方法中。因此, 我能想到的强制初始值与本数列其余数字具有相同类型的最佳方法是: 执行加法并使用初始值类型来强制转换计算结果的类型。我在 Python 列表中询问了这个问题, 并得到了 Steven D'Aprand 的极好的答复。

- ④ 为了便于阅读,我创建了变量 `forever`。如果 `self.end` 为 `None`,则 `forever` 将为 `True`,从而产生一个无穷数列。
- ⑤ 该循环会一直运行,直至结果匹配或超过 `self.end`。当该循环退出时,该函数也会退出。
- ⑥ 当前的 `result` 已生成。
- ⑦ 计算下一个潜在的 `result`。它可能永远不会产生,因为 `while` 循环可能会终止。

在示例 17.12 的最后一行(⑦处)中,我没有直接用 `self.step` 不断增加 `result`,而是通过“`self.begin + self.step * index`”计算 `result` 的各个新值,以避免连续加法后浮点错误的累积效应。简单做个时间,就能看出两种处理方式的差异。

```

1  >>> 100 * 1.1
2  110.00000000000001
3  >>> sum(1.1 for _ in range(100))
4  109.99999999999982
5  >>> 1000 * 1.1
6  1100.0
7  >>> sum(1.1 for _ in range(1000))
8  1100.0000000000086

```

示例 17.12 中的 `ArithmeticProgression` 类可以按预期那样使用,这也是用生成器函数实现 `__iter__` 特殊方法的另一个示例。然而,如果一个类只是为了构建生成器而去实现 `__iter__` 方法,那还不如直接使用生成器函数来取代这个类。毕竟,生成器函数就是一个制造生成器的工厂。

示例 17.13 展示了一个生成器函数 `aritprog_gen`,它的功能与 `ArithmeticProgression`(见示例 17.12)相同,但代码量更少。如果将 `ArithmeticProgression` 类替换成 `aritprog_gen` 函数,也可以通过“示例 17.11<496页>”中的所有测试。⁶

```

</> 示例 17.13: aritprog_gen 生成器函数
1  def aritprog_gen(begin, step, end=None):
2      result = type(begin + step)(begin)
3      forever = end is None
4      index = 0
5      while forever or result < end:
6          yield result
7          index += 1
8          result = begin + step * index

```

示例 17.13 的代码非常优雅,但请记住:标准库中有大量现成的生成器可用。下一节,将展示用 `itertools` 模块实现一个更简短的版本。

17.8.1 用 `itertools` 模块生成等差数列

Python 3.10 中的 `itertools` 模块有 20 个生成器函数,可以以各种有趣的方式组合在一起。例如, `itertools.count` 函数可返回一个生成数字的生成器。如果不传入参数, `itertools.count` 将生成一系列从 0 开始的整数。不过,可以提供可选的 `start` 与 `step` 值,以实现与 `aritprog_gen` 函数(见“示例 17.13<498页>”)类似的

⁶ 本书代码库中的“17-it-generator/aritprog_runner.py”,可针对 `aritprog*.py` 脚本的所有变体运行测试。

效果。

```
1  >>> import itertools
2  >>> gen = itertools.count(1, .5)
3  >>> next(gen)
4  1
5  >>> next(gen)
6  1.5
7  >>> next(gen)
8  2.0
9  >>> next(gen)
10 2.5
```



`itertools.count` 永不停止。因此,如果调用 `list(count())`,Python 将尝试构建一个能填满所有内存的列表。实际上,在调用失败之前,你的计算机会疯狂地运转。

另外,还有一个 `itertools.takewhile` 函数:它返回一个生成器,该生成器将消耗另一个生成器,并在指定的条件为 `False` 时停止。因此,可将 `itertools.count` 与 `itertools.takewhile` 结合使用,编写如下代码:

```
1  >>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
2  >>> list(gen)
3  [1, 1.5, 2.0, 2.5]
```

示例 17.14 利用 `itertools.count` 与 `itertools.takewhile`,编写出的代码更简短。

</> 示例 17.14: aritprog_v3.py:与“示例 17.13<498页>”的 `aritprog_gen` 函数作用相同

```
1  import itertools
2
3  def aritprog_gen(begin, step, end=None):
4      first = type(begin + step)(begin)
5      ap_gen = itertools.count(first, step)
6      if end is None:
7          return ap_gen
8      return itertools.takewhile(lambda n: n < end, ap_gen)
```

注意,示例 17.14 中的 `aritprog_gen` 不是生成器函数,因为函数主体中没有关键 `yield`。但是,它与生成器函数一样,也返回一个生成器。

然而, `itertools.count` 会不停地累加 `step`。因此,生成的浮点数序列没有“示例 17.13<498页>”那么精确。

示例 17.14 的要点是:在实现生成器之前,要先知道 Python 标准库中有哪些可用的轮子,否则很可能需要重新造轮子。鉴于此,下一节将介绍 Python 标准库中提供的一些现成可用的生成器函数。

17.9 标准库中的生成器函数

Python 标准库提供了许多生成器,有用于逐行遍历的纯文本文件对象,还有出色的 `os.walk` 函数。`os.walk` 函数在遍历目录树时生成文件名,使递归搜索文件系统如同 for 循环一样简单。

`os.walk` 生成器函数的作用令人印象深刻,但本节将重点关注通用函数。这些函数可接受任意可迭代对象作为参数,并返回一个生成器。该生成器可生成选定的、计算出的或重新排列的项。本节余下的几个表格总结了其中的 24 个函数,有些是内置的函数,有些存在于 `itertools` 与 `functools` 模块中。为了方便起见,我按照函数功能统一分组,而不管函数是在哪里定义的。

17.9.1 用于筛选的生成器函数

第一组是用于筛选的生成器函数:从输入的可迭代对象中产出项的子集,而且不更改项本身。与 `itertools.takewhile` 一样,?? 中的多数函数接受一个参数 `predicate`——一个单参数的布尔函数,将被应用到输入中的每一项,以确定该项是否包含在输出中。

表 17.1: 用于筛选的生成器函数)

模块	函数	描述
<code>itertools</code>	<code>compress(it, selector_it)</code>	并行消耗 2 个可迭代对象;若 <code>selector_it</code> 中的项为 <code>True</code> ,则产出 <code>it</code> 中对应的项。
<code>itertools</code>	<code>dropwhile(predicate, it)</code>	消耗 <code>it</code> ,跳过 <code>predicate</code> 计算结果为 <code>True</code> 的项,然后产出剩余的项(不再进一步检查)。
内置	<code>filter(predicate, it)</code>	将 <code>it</code> 中的各个项传给 <code>predicate</code> 。若 <code>predicate(item)</code> 为 <code>True</code> ,则产出对应的项;若 <code>predicate</code> 为 <code>None</code> ,则仅产出真实项。
<code>itertools</code>	<code>filterfalse(predicate, it)</code>	与函数 <code>filter</code> 作用类似,不过 <code>predicate</code> 的逻辑是相反的: <code>predicate</code> 返回 <code>False</code> 时,产出对应的项。
<code>itertools</code>	<code>islice(it, stop) 或 islice(it, start, stop, step=1)</code>	从 <code>it</code> 的切片中生成项,类似于 <code>s[:stop]</code> 或 <code>s[start:stop:step]</code> 。不过 <code>it</code> 可以是任何可迭代对象,并且操作是惰性的。
<code>itertools</code>	<code>takewhile(predicate, it)</code>	<code>predicate</code> 返回 <code>True</code> 时,产出对应的项,然后立即停止,不再继续检查。

示例 17.15 在控制台中演示 表 17.1 中各个函数的用法。

</> 示例 17.15: 演示用于筛选的生成器函数

```

1  >>> def vowel(c):
2      ...     return c.lower() in 'aeiou'
3  ...
4  >>> list(filter(vowel, 'Aardvark'))
5  ['A', 'a', 'a']
6  >>> import itertools
7  >>> list(itertools.filterfalse(vowel, 'Aardvark'))
8  ['r', 'd', 'v', 'r', 'k']
9  >>> list(itertools.dropwhile(vowel, 'Aardvark'))
10 ['r', 'd', 'v', 'a', 'r', 'k']
11 >>> list(itertools.takewhile(vowel, 'Aardvark'))
12 ['A', 'a']
13 >>> list(itertools.compress('Aardvark', (1, 0, 1, 1, 0, 1)))
14 ['A', 'r', 'd', 'a']
15 >>> list(itertools.islice('Aardvark', 4))
16 ['A', 'a', 'r', 'd']
17 >>> list(itertools.islice('Aardvark', 4, 7))
18 ['v', 'a', 'r']

```

```

19 >>> list(itertools.islice('Aardvark', 1, 7, 2))
20 ['a', 'd', 'a']

```

17.9.2 用于映射的生成器函数

第二组是用于映射的生成器函数: 在输入的可迭代对象(函数 `map` 与 `starmap` 可处理多个可迭代对象)中的各项上做计算, 产出计算结果。⁷表 17.2 中的生成器函数为输入可迭代对象中的每一项生成一个结果。若输入来自多个可迭代对象, 则一旦第一个可迭代对象耗尽, 就停止输出。

表 17.2: 用于映射的生成器函数

模块	函数	描述
itertools	<code>accumulate(it,[func])</code>	生成累计求和: 若提供了 <code>func</code> , 则将前 2 项传给 <code>func</code> ; 然后, 将计算结果与下一项传给 <code>func</code> ... 依此类推, 产出最后结果。
内置	<code>enumerate(iterable,start=0)</code>	生成 <code>(index, item)</code> 形式的二元组。其中, <code>index</code> 从 <code>start</code> 计数, 而 <code>item</code> 取自 <code>iterable</code> 。
内置	<code>map(func, it₁,[it₂,..., it_N])</code>	将 <code>func</code> 应用于 <code>it</code> 中的每一项, 产生结果; 若传入 <code>N</code> 个可迭代对象, 那么 <code>func</code> 必须接受 <code>N</code> 个参数, 并且并行处理各个可迭代对象。
itertools	<code>starmap(func, it)</code>	将 <code>func</code> 应用于 <code>it</code> 中的每一项, 产生结果; 输入的可迭代对象应该产出可迭代项 <code>it</code> , 然后以 <code>func(*it)</code> 形式调用 <code>func</code> 。

示例 17.16 演示了 `itertools.accumulate` 的一些用法。

</> 示例 17.16: `itertools.accumulate` 生成器函数示例

```

1 >>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
2 >>> import itertools
3 >>> list(itertools.accumulate(sample)) ❶
4 [5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
5 >>> list(itertools.accumulate(sample, min)) ❷
6 [5, 4, 2, 2, 2, 2, 0, 0, 0]
7 >>> list(itertools.accumulate(sample, max)) ❸
8 [5, 5, 5, 8, 8, 8, 8, 9, 9]
9 >>> import operator
10 >>> list(itertools.accumulate(sample, operator.mul)) ❹
11 [5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
12 >>> list(itertools.accumulate(range(1, 11), operator.mul)) ❺
13 [1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]

```

- ❶ 累计求和。
- ❷ 累计最小值。
- ❸ 累计最大值。
- ❹ 累计乘积。
- ❺ 从 1~10, 计算各个数的阶乘。

表 17.2 的其余函数如示例 17.17 所示。

⁷ 此处的“映射”一词与字典无关, 而是与内置函数 `map()` 有关。

</> 示例 17.17: 演示用于映射的生成器函数

```

1  >>> list(enumerate('albatroz', 1))           ❶
2  [(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z')]
3  >>> import operator
4  >>> list(map(operator.mul, range(11), range(11))) ❷
5  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
6  >>> list(map(operator.mul, range(11), [2, 4, 8])) ❸
7  [0, 4, 16]
8  >>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) ❹
9  [(0, 2), (1, 4), (2, 8)]
10 >>> import itertools
11 >>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) ❺
12 ['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'oooooooo', 'zzzzzzz']
13 >>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
14 >>> list(itertools.starmap(lambda a, b: b / a,
15 ...     enumerate(itertools.accumulate(sample), 1))) ❻
16 [5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
17 5.0, 4.375, 4.888888888888889, 4.5]

```

- ❶ 从 1 开始, 为单词中的字母编号。
- ❷ 从 0 ~10, 计算各个整数的平方。
- ❸ 并行计算 2 个可迭代对象中对应位置上的 2 项乘积, 当最短的可迭代对象耗尽时, 运算结果停止。
- ❹ 此行作用等同于内置函数 `zip`。
- ❺ 根据字母在单词中的位置 (从 1 开始), 将每个字母重复相应的次数。
- ❻ 累计平均值。

17.9.3 用于合并的生成器函数

第三组是用于合并的生成器函数:所有生成项都来自多个输入的可迭代对象。

表 17.3: 用于合并多个可迭代对象的生成器函数)

模块	函数	描述
itertools	<code>chain(it₁, ..., it_N)</code>	依次产出 <code>it₁, it₂, ..., it_N</code> 中的所有项, 无缝衔接。
itertools	<code>chain.from_iterable(it)</code>	依次产出由 <code>it</code> 生成的各个可迭代对象中的项, 无缝衔接。 <code>it</code> 产出的项应该是可迭代对象, 例如元组列表。
itertools	<code>product(it₁, ..., it_N, repeat=1)</code>	计算笛卡尔积: 从输入的各个可迭代对象获取项, 合并成 <code>N</code> 元组, 与嵌套的 <code>for</code> 循环效果一样。 <code>repeat</code> 指明重复处理多少次输入的可迭代对象。

接下页

续表 17.3

模块	函数	描述
内置	<code>zip(it₁, ..., it_N, strict=False)</code>	从输入的各个可迭代对象并行获取项, 产出由此构成的 N 元组。当某个可迭代对象耗尽时, 则默默停止。除非指定参数 strict=True ⁸ 。
itertools	<code>zip_longest(it₁, ..., it_N, fillvalue=None)</code>	从输入的各个可迭代对象并行获取项, 产出由此构成的 N 元组, 直至最长的可迭代对象耗尽才停止。对于空缺的项, 用参数 fillvalue 指定的值填充。

示例 17.18 展示了 `itertools.chain` 与 `zip` 生成器函数及其同类函数的用法。再次提醒, 函数 `zip` 的名字来自 `zip fastener` 或 `zipper` (与 `zip` 压缩无关)。“12.7 Vector 类第 4 版: 哈希与快速等值测试<332页>”中附注栏“出色的 `zip` 函数”介绍过函数 `zip` 与 `zip_longest`。

</> 示例 17.18: 演示用于合并的生成器函数

```

1 >>> list(itertools.chain('ABC', range(2))) ❶
2 ['A', 'B', 'C', 0, 1]
3 >>> list(itertools.chain(enumerate('ABC'))) ❷
4 [(0, 'A'), (1, 'B'), (2, 'C')]
5 >>> list(itertools.chain.from_iterable(enumerate('ABC'))) ❸
6 [0, 'A', 1, 'B', 2, 'C']
7 >>> list(zip('ABC', range(5), [10, 20, 30, 40])) ❹
8 [('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
9 >>> list(itertools.zip_longest('ABC', range(5))) ❺
10 [('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
11 >>> list(itertools.zip_longest('ABC', range(5), fillvalue='?')) ❻
12 [('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]

```

- ❶ 调用 `chain` 时, 通常要传入 2 个或更多个可迭代对象。
- ❷ 只为 `chain` 传入 1 个可迭代对象时, 函数 `chain` 没什么用。
- ❸ `chain.from_iterable(it)` 从可迭代对象 `it` 中获取各项 (各项本身也是可迭代对象); 然后, 按顺序将各项串联。例如, “`list(chain.from_iterable(([1,2,3], ['a','b','c'])))`” 的结果为 “[1, 2, 3, 'a', 'b', 'c']”。
- ❹ 函数 `zip` 可以并行消耗任意数量的可迭代对象, 不过只要有一个可迭代对象耗尽, 生成器就会停止。在 Python ≥ 3.10 中, 可指定关键字参数 `strict=True`, 此时若某个可迭代对象先耗尽, 则引发 `ValueError` 异常。
- ❺ 函数 `zip_longest` 的作用与 `zip` 类似。不过会从头至尾处理输入的所有可迭代对象, 并根据需要用 `None` 填充空缺的项。
- ❻ 可通过函数 `zip_longest` 的关键字参数 `fillvalue` 指定填充空缺项的值。

`itertools.product` 生成器是一种计算笛卡尔积的惰性方法。“2.3.3 笛卡尔积”小节, 在多个 `for` 子句中用列表推导式计算过笛卡尔积 (见“示例 2.4<24页>”)。此外, 也可以用包含多个 `for` 子句的生成器表达式, 以惰性方式计算笛卡尔积。示例 17.19 演示了用函数 `itertools.product` 惰性计算笛卡尔积的用法。

</> 示例 17.19: 演示 `itertools.product` 生成器函数

```

1 >>> list(itertools.product('ABC', range(2))) ❶

```

⁸ 仅限关键字参数 `strict` 是 Python 3.10 新增的。指定 `strict=True` 时, 若输入的可迭代对象长度不同, 则当某个可迭代对象耗尽时, 将引发 `ValueError` 异常。为了向后兼容, `strict` 默认为 `False`。

```

2  [('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
3  >>> suits = 'spades hearts diamonds clubs'.split()>>> list(itertools.product('AK',
4  suits)) ❷
5  [('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
6  ('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
7  >>> list(itertools.product('ABC')) ❸
8  [('A',), ('B',), ('C',)]
9  >>> list(itertools.product('ABC', repeat=2)) ❹
10 [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
11 ('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
12 >>> list(itertools.product(range(2), repeat=3))
13 [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
14 (1, 0, 1), (1, 1, 0), (1, 1, 1)]
15 >>> rows = itertools.product('AB', range(2), repeat=2)
16 >>> for row in rows: print(row)
17 ...
18 ('A', 0, 'A', 0)
19 ('A', 0, 'A', 1)
20 ('A', 0, 'B', 0)
21 ('A', 0, 'B', 1)
22 ('A', 1, 'A', 0)
23 ('A', 1, 'A', 1)
24 ('A', 1, 'B', 0)
25 ('A', 1, 'B', 1)
26 ('B', 0, 'A', 0)
27 ('B', 0, 'A', 1)
28 ('B', 0, 'B', 0)
29 ('B', 0, 'B', 1)
30 ('B', 1, 'A', 0)
31 ('B', 1, 'A', 1)
32 ('B', 1, 'B', 0)
33 ('B', 1, 'B', 1)

```

❶ 包含 3 个字符的字符串与包含 2 个整数的区间, 得到的笛卡尔积是 6 个元组 (即 $3 \times 2 = 6$)。

❷ 2 张牌 ('AK') 与 4 种花色得到的笛卡尔积是 8 个元组。

❸ 传入一个可迭代对象, 函数 `product` 产出一系列元组——不是特别有用。

❹ 关键字参数 `repeat=N` 告诉函数 `product` 重复消耗每个可迭代对象的次数。

17.9.4 用于扩充输入的生成器函数

第四组是用于扩充输入的生成器函数: 这些函数会为每个输入项生成多个值来扩展输入。表 17.4 列出了这些函数。

表 17.4: 将输入的各项扩充为多个输出项的生成器函数

模块	函数	描述
itertools	combinations(it, out_len)	从可迭代对象 it 中选出 out_len 项 (不含重复项), 组合在一起, 然后产出。类似于 $C_{it}^{out_len}$ 。
itertools	combinations_with_replacement(it, out_len)	从可迭代对象 it 中选出 out_len 项 (包含重复项), 组合在一起, 然后产出。类似于 $C_{it}^{out_len}$ 。
itertools	count(start=0, step=1)	从 start 开始, 不断产出数字, 按 step 值为步长。
itertools	cycle(it)	从可迭代对象中获取项, 存储每个项的副本。然后, 无限次地重复生成整个序列。
itertools	pairwise(it)	从可迭代对象 it 中获取连续的重叠对。 ⁹
itertools	permutations(it, out_len=None)	从 it 产生的项目中产生 out_len 项目的排列; 默认情况下, out_len 为 len(list(it))
itertools	repeat(item, [times])	重复不断地产出给定项 item, 除非指定了次数 times。

itertools 模块 中的 count 与 repeat 函数 返回的是“无中生有”生成器, 二者都不接受可迭代对象作为输入。“示例 17.14<499页>”已见过 count 函数。cycle 函数 会对输入的可迭代对象进行备份, 并重复生成其中的项。示例 17.20 count、cycle、pairwise 与 repeat 生成器函数的使用。

```
</> 示例 17.20: 演示 count、cycle、pairwise 与 repeat 生成器函数
1  >>> ct = itertools.count()          ❶
2  >>> next(ct)                   ❷
3  0
4  >>> next(ct), next(ct), next(ct) ❸
5  (1, 2, 3)
6  >>> list(itertools.islice(itertools.count(1, .3), 3)) ❹
7  [1, 1.3, 1.6]
8  >>> cy = itertools.cycle('ABC') ❺
9  >>> next(cy)
10 'A'
11 >>> list(itertools.islice(cy, 7)) ❻
12 ['B', 'C', 'A', 'B', 'C', 'A', 'B']
13 >>> list(itertools.pairwise(range(7))) ❻
14 [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
15 >>> rp = itertools.repeat(7)       ❻
16 >>> next(rp), next(rp)
17 (7, 7)
18 >>> list(itertools.repeat(8, 4)) ❻
19 [8, 8, 8, 8]
20 >>> list(map(operator.mul, range(11), itertools.repeat(5))) ❻
21 [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

❶ 用函数 count 构建生成器 ct。

❷ 获取生成器 ct 中的第 1 项。

❸ 无法用 ct 构建列表, 因为 ct 是无穷的, 所以获取 ct 中的 3 项。

❹ 若生成器函数 count 受到函数 islice 或 takewhile 的限制, 则可从 count 生成器中构建列表。

⁹itertools.pairwise是 Python 3.10 新增的函数。

- ⑤ 从“ABC”构建一个 circle 生成器，并获取其第 1 项“A”。
- ⑥ 只有受到 `islice` 函数的限制，才能构建列表。此处，获取接下来的 7 项。
- ⑦ 对于输入中的每项，若存在下一项，则 `pairwise` 会生成一个包含该项及下一项的二元组。适用于 Python ≥ 3.10 。
- ⑧ 构建一个 `repeat` 生成器，始终生成数字 7。
- ⑨ 为函数 `repeat` 传入参数 `times`，可以限制 `repeat` 生成项的数量。此处，重复生成 4 个 8。
- ⑩ 函数 `repeat` 的常见用法：为函数 `map` 提供固定参数。这里提供的是乘数 5。

在 `itertools` 模块文档中，生成器函数 `combinations`、`combinations_with_replacement`、`permutations`，连同 `product`，统称为“组合生成器（Combinatoric Generator）”。如 [示例 17.21](#) 所示，`itertools.product` 与其余组合函数之间有紧密的联系。

</> [示例 17.21](#): 组合生成器函数为每个输入项生成多个值

```

1  >>> list(itertools.combinations('ABC', 2))           ❶
2  [('A', 'B'), ('A', 'C'), ('B', 'C')]
3  >>> list(itertools.combinations_with_replacement('ABC', 2)) ❷
4  [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
5  >>> list(itertools.permutations('ABC', 2))           ❸
6  [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
7  >>> list(itertools.product('ABC', repeat=2))          ❹
8  [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
9  ('C', 'A'), ('C', 'B'), ('C', 'C')]

```

- ❶ “ABC” 中每 2 项 (`len() == 2`) 的各种组合，不含重复项的组合。在生成的元组中，项的顺序无关紧要（可以视作集合）。
- ❷ “ABC” 中每 2 项 (`len() == 2`) 的各种组合，包含重复项的组合。
- ❸ “ABC” 中每 2 项 (`len() == 2`) 的各种排列。在生成的元组中，项的顺序有重要意义。
- ❹ “ABC” 与 “ABC”的笛卡尔积，类似 `repeat=2` 的效果。

17.9.5 用于产出输入的生成器函数

最后一组是用于产出输入的生成器函数：这些函数用于生成输入的可迭代对象中的全部项，不过会以某种方式重新排列。本组中有 2 个可以返回多个生成器的函数，分别是 `itertools.groupby` 与 `itertools.tee`。还有一个内置函数 `reversed`——唯一一个不接受可迭代对象，仅接受序列为参数的函数。这是合理的，因为函数 `reversed(seq)` 从后向前生成 `seq` 中的项，这仅适用于已知长度的序列类型。但是，`reversed` 按需生成各项，因此无须创建反向的副本。我将 `product` 归类为用于合并的生成器（列在 [表 17.3](#) 中），是因为那组函数都处理多个可迭代对象。而 [表 17.5](#) 中的生成器函数最多只接受一个可迭代对象。

表 17.5: 用于重新排列项(元素)的生成器函数

模块	函数	描述
itertools	groupby(it, key=None)	生成 “(key, group)” 形式的二元组。其中, key 是分组标准, group 是生成组中各项的生成器。
内置	reversed(seq)	从后向前, 倒序生成 seq 中的各项。seq 必须是序列或实现了特殊方法 <code>__reversed__</code> 的对象。
itertools	tee(it, n=2)	生成一个由 n 个生成器组成的元组, 每个生成器单独生成输入的可迭代对象中的各项。

示例 17.22 演示了函数与内置函数 `reversed` 的用法。需要注意的是, 假定输入的可迭代对象 `it` 是按分组标准排序的; 或者即使不排序, 至少也是用指定的标准对 `it` 中的各项进行分组。本书技术审校 Miroslav Šedivý 提供了一个关于 `itertools.groupby` 的用例: 按时间顺序排序 `datetime` 对象; 然后, 用 `groupby` 为工作日分组, 得到一组星期一的数据、一组星期二的数据, …, 再从下一周的星期一开始, 以此类推。

</> 示例 17.22: 演示 `itertools.groupby` 函数的用法

```

1  >>> list(itertools.groupby('LLLLAAGGG'))
2  [('L', <itertools._grouper object at 0x102227cc0>),
3  ('A', <itertools._grouper object at 0x102227b38>),
4  ('G', <itertools._grouper object at 0x102227b70>)]
5  >>> for char, group in itertools.groupby('LLLLAAAGG'):
6      ...     print(char, '->', list(group))
7
8  L-> ['L', 'L', 'L', 'L']
9  A-> ['A', 'A', ]
10 G-> ['G', 'G', 'G']
11 >>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
12 ...             'bat', 'dolphin', 'shark', 'lion']
13 >>> animals.sort(key=len)
14 >>> animals
15 ['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
16 'giraffe', 'dolphin']
17 >>> for length, group in itertools.groupby(animals, len):
18 ...     print(length, '->', list(group))
19 ...
20 3-> ['rat', 'bat']
21 4-> ['duck', 'bear', 'lion']
22 5-> ['eagle', 'shark']
23 7-> ['giraffe', 'dolphin']
24 >>> for length, group in itertools.groupby(reversed(animals), len): ❶
25 ...     print(length, '->', list(group))
26 ...
27 7-> ['dolphin', 'giraffe']
28 5-> ['shark', 'eagle']
29 4-> ['lion', 'bear', 'duck']
30 3-> ['bat', 'rat']
31 >>>

```

❶ 函数 `groupby` 生成 (key, group_generator) 形式的元组。

- ❷ 处理 `groupby` 返回的生成器需要嵌套遍历。这里外层用 `for` 循环，内层用 `list` 构造函数。
- ❸ 按单词长度排序 `animals`。
- ❹ 再次遍历 `key` 与 `group` 值对。将 `key` 显示出来，并将 `group` 展开为列表。
- ❺ 此处用生成器函数 `reverse`，从右向左遍历 `animals`。

本组中的最后一个生成器函数是 `tee(it)`，它有一个独特的行为：从输入的单个可迭代对象 `it` 中产生多个生成器，每个生成器都可以生成可迭代对象 `it` 中的各项。并且，这些生成器都可以独立使用，如示例 17.23 所示。

</> 示例 17.23：`itertools.tee` 生成多个生成器，每个生成器都可生成输入可迭代对象的各项

```

1  >>> list(itertools.tee('ABC'))
2  [<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac08>]
3  >>> g1, g2 = itertools.tee('ABC')
4  >>> next(g1)
5  'A'
6  >>> next(g2)
7  'A'
8  >>> next(g2)
9  'B'
10 >>> list(g1)
11 ['B', 'C']
12 >>> list(g2)
13 ['C']
14 >>> list(zip(*itertools.tee('ABC')))
15 [('A', 'A'), ('B', 'B'), ('C', 'C')]

```

请注意，本节中的几个示例使用了生成器函数的组合。这是这些函数的一个重要特性：因为它们将生成器作为参数并返回生成器，所以可以以多种不同的方式进行组合。

接下来，将回顾一下标准库中另一组善于处理可迭代对象的函数。

17.10 可迭代的规约 (Reducing) 函数

表 17.6 中的函数都接受一个可迭代对象，并返回单个结果。这些函数被称为“归约 (reducing)”函数、“折叠 (folding)”函数或“累积 (accumulating)”函数。我们可以用 `functools.reduce` 实现这里列出的每一个内置函数，但它们之所以作为“内置函数”存在，是因为它们更容易解决一些常见的用例。关于 `functools.reduce` 的更详细解释，请参见“12.7 Vector 类第 4 版：哈希与快速等值测试<332页>”。

函数 `all` 与 `any` 有一项重要的优化措施，是函数 `functools.reduce` 做不到的：`all` 与 `any` 会短路，即一旦确定了结果就立即停止消耗迭代器。详见“示例 17.24<509页>”中对 `any` 的最后一次测试。

表 17.6：读取可迭代对象，并返回单个值的内置函数

模块	函数	描述
----	----	----

接下页

续表 17.6

模块	函数	描述
内置	<code>all(it)</code>	如果可迭代对象 <code>it</code> 中的所有项都为 <code>True</code> , 则结果为 <code>True</code> ; 否则, 结果为 <code>False</code> 。 <code>all([])</code> 结果为 <code>True</code> 。
内置	<code>any(it)</code>	如果可迭代对象 <code>it</code> 中的任一项为 <code>True</code> , 则结果为 <code>True</code> ; 否则, 结果为 <code>False</code> 。 <code>any([])</code> 结果为 <code>False</code> 。
内置	<code>max(it, [key=], [default=])</code>	返回可迭代对象 <code>it</code> 中最大的项; ¹⁰ 参数 <code>key</code> 是一个排序函数, 其作用与函数 <code>sorted</code> 中的 <code>key</code> 一样。如果 <code>it</code> 为空, 则返回 <code>default</code> 指定的值。
内置	<code>min(it, [key=], [default=])</code>	返回可迭代对象 <code>it</code> 中最小的项; ¹¹ 参数 <code>key</code> 是一个排序函数, 其作用与函数 <code>sorted</code> 中的 <code>key</code> 一样。如果 <code>it</code> 为空, 则返回 <code>default</code> 指定的值。
functools	<code>reduce(func, it, [initial])</code>	将可迭代对象 <code>it</code> 中的前 2 项传给 <code>func</code> ; 然后, 将结果与第 3 项传给 <code>func</code> ; ... 依此类推, 返回最后的结果。如果提供了参数 <code>initial</code> , 则将它当作第 1 组值的第 1 项。
内置	<code>sum(it, start=0)</code>	返回可迭代对象 <code>it</code> 中所有项之和。若提供可选的 <code>start</code> 值, 则加上 <code>start</code> 值。计算 <code>float</code> 类型的加法时, 可使用 <code>math.fsum</code> 函数提高计算精度。

函数 `all` 与 `any` 的用法, 如示例 17.24 所示。

</> 示例 17.24: 用 `all` 与 `any` 处理几个序列的结果

```

1  >>> all([1, 2, 3])
2  True
3  >>> all([1, 0, 3])
4  False
5  >>> all([])
6  True
7  >>> any([1, 2, 3])
8  True
9  >>> any([1, 0, 3])
10 True
11 >>> any([0, 0.0])
12 False
13 >>> any([])
14 False
15 >>> g = (n for n in [0, 0.0, 7, 8])
16 >>> any(g) ❶
17 True
18 >>> next(g) ❷
19 8

```

❶ 函数 `any` 遍历对象 `g`, 直至 `g` 中的 7 出现。此时, 函数 `any` 停止, 并返回 `True`。

❷ 因此, `g` 中的下一个值是 8。

另一个接受一个可迭代对象并返回相关结果的内置函数是 `sorted`。与生成器函数 `reversed` 不同, 函数 `sorted` 会构建并返回一个新列表。毕竟, 必须读取输入的可迭代对象中的每一项, 才能对这些项排序。而且

另一个接受可迭代对象并返回其他内容的内置函数是 `sorted`。与生成器函数 `reversed` 不同, 函数 `sorted` 会构建并返回一个新列表。毕竟, 必须读取输入可迭代对象的每个项才能进行排序, 而排序是在列表中进行

¹⁰ 也可以 `max(arg1,arg2,...,[key=?])` 这种形式调用, 返回参数中的最大值。

¹¹ 也可以 `min(arg1,arg2,...,[key=?])` 这种形式调用, 返回参数中的最小值。

的,因此 `sorted` 在完成排序后,才返回排序后的列表。我之所以提到 `sorted`,是因为它可以处理任意一个可迭代对象。

当然,函数 `sorted` 与 `functools.reduce` 仅适用于最终会停止的可迭代对象。否则,这些函数会持续收集可迭代对象中的项,永远无法返回结果。



若您已读到此处,那么您已读完了本章最重要与最有用的内容。余下章节将介绍生成器的高级功能——这些功能不常见,也不常用。例如,“`yield from`”结构与经典协程。

另外,还有几节讨论可迭代对象、迭代器与经典协程的类型提示(Type Hints)。

“`yield from`”语法提供了一种组合生成器的新方法,详见下一节。

17.11 `yield from` 表达式

Python 3.3 引入了“`yield from`”表达式语法,以允许生成器将工作委托给子生成器。

在引入“`yield from`”语法之前,如果一个生成器要根据另一个生成器生成的值来产出值,则需要用 `for` 循环。

```

1  >>> def sub_gen():
2  ...     yield 1.1
3  ...     yield 1.2
4  ...
5  >>> def gen():
6  ...     yield 1
7  ...     for i in sub_gen():
8  ...         yield i
9  ...     yield 2
10 ...
11 >>> for x in gen():
12 ...     print(x)
13 ...
14 1
15 1.1
16 1.2
17 2

```

用“`yield from`”可以得到相同的结果,如示例 17.25 所示。

</> 示例 17.25: 测试 `yield from`

```

1  >>> def sub_gen():
2  ...     yield 1.1
3  ...     yield 1.2
4  ...
5  >>> def gen():
6  ...     yield 1
7  ...     yield from sub_gen()

```

```

8 ...     yield 2
9 ...>>> for x in gen():
10 ...     print(x)
11 ...
12 1
13 1.1
14 1.2
15 2

```

在 [示例 17.25](#) 中, for 循环是客户端代码, 函数 gen 是委托生成器, 函数 sub_gen 是子生成器。注意, “yield from” 会暂停 gen, 并由 sub_gen 接手, 直至 gen 耗尽为止。由 sub_gen 产生的值通过 gen 直接进入到客户端的 for 循环。与此同时, gen 被挂起, 无法看到通过它的值。仅当 sub_gen 完成后, gen 才会恢复。

当子生成器含有 return 语句, 返回一个值时, 可以通过将 “yield from” 作为表达式的一部分, 在委托生成器中捕获该值。如 [示例 17.26](#) 所示。

</> [示例 17.26](#): yield from 获取子生成器的返回值

```

1  >>> def sub_gen():
2 ...     yield 1.1
3 ...     yield 1.2
4 ...     return 'Done!'
5 ...
6 >>> def gen():
7 ...     yield 1
8 ...     result = yield from sub_gen()
9 ...     print('<--', result)
10 ...    yield 2
11 ...
12 >>> for x in gen():
13 ...     print(x)
14 ...
15 1
16 1.1
17 1.2
18 <-- Done!
19 2

```

了解了 “yield from” 基础知识之后, 下面通过几个简单示例说明 “yield from” 的实际用途。

17.11.1 重新实现 chain

如 “[表 17.3<502页>](#)” 可知, `itertools` 模块提供了一个 `chain(it1, ..., itN)` 生成器, 它可从多个可迭代对象中生成项, 先遍历第一个可迭代对象, 再遍历第二个, ..., 直至最后一个可迭代对象。在 Python 中, 可以用嵌套的 for 循环实现自己的 chain 函数,¹⁰ 如下所示:

```

1  >>> def chain(*iterables):
2 ...     for it in iterables: ❶

```

¹⁰chain 与 `itertools` 模块中的多数函数都是用 C 语言编写的。

```

3 ...     for i in it:      ❷
4 ...         yield i      ❸
5 ...
6 >>> s = 'ABC'
7 >>> r = range(3)
8 >>> list(chain(s, r))
9 ['A', 'B', 'C', 0, 1, 2]

```

上述代码中的 `chain` 生成器通过外层 `for` 循环 (❶处) 得到每个可迭代对象 `it`, 在内层 `for` 循环 (❷处) 中驱动可迭代对象 `it` 生成各个项 (❸处)。内层 `for` 循环可以替换为 “`yield from`” 表达式, 如下所示。

```

1 >>> def chain(*iterables):
2 ...     for i in iterables:
3 ...         yield from i
4 ...
5 >>> list(chain(s, t))
6 ['A', 'B', 'C', 0, 1, 2]

```

在此示例中, “`yield from`” 似乎只是一个语法糖 (Syntactic Sugar), 除了提高代码易读性之外, 并无太多用处。接下来, 将介绍一个更复杂的例子。

17.11.2 遍历树状结构

本节在脚本中用 “`yield from`” 遍历树状结构。后续将如 “婴儿学步” 一般, 介绍如何实现此脚本。

此示例处理的树状结构是 Python 异常层次结构。不过, 适当改造后也可用于遍历目录树等其他树状结构。

截至 Python 3.10, 异常层次结构共有 5 层, 最底层 (0 层) 是 `BaseException`。第一步, 将显示第 0 层。

给定一个根类, [示例 17.27](#) 中的 `tree` 生成器将生成根类的名称, 然后停止。

</> [示例 17.27: tree/step0/tree.py](#): 生成根类的名称, 然后停止

```

1 def tree(cls):
2     yield cls.__name__
3
4 def display(cls):
5     for cls_name in tree(cls):
6         print(cls_name)
7
8 if __name__ == '__main__':
9     display(BaseException)

```

?? 的输出只有一行。

1 BaseException

下一步是第 1 层。`tree` 生成器将生成根类的名称以及各个直接子类的名称。子类的名称会以缩进形式显示, 以展示层次结构。预期的输出效果如下所示。

1 \$ python3 tree.py

```
2 BaseException
3     Exception      GeneratorExit
4     SystemExit
5     KeyboardInterrupt
```

示例 17.28 可生成这种格式的输出。

</> 示例 17.28: tree/step1/tree.py:生成根类与直接子类的名称

```
1 def tree(cls):
2     yield cls.__name__, 0
3     for sub_cls in cls.__subclasses__():
4         yield sub_cls.__name__, 1
5
6 def display(cls):
7     for cls_name, level in tree(cls):
8         indent = ' ' * 4 * level
9         print(f'{indent}{cls_name}')
10
11 if __name__ == '__main__':
12     display(BaseException)
```

① 为支持缩进输出,生成类的名称及其在层次结构中的层级。

② 用 `__subclasses__` 特殊方法获取子类列表。

③ 生成子类的名称及层级 1。

④ 用表达式 “`' ' * 4 * level`” 构建缩进字符串。在第 0 层,得到的是一个空字符串。

示例 17.29 中,重构了 `tree` 生成器,将特殊的根类与子类分开,子类在 `sub_tree` 生成器中处理。在 “`yield from`” 处, `tree` 生成器被挂起,由 `sub_tree` 接手,生成值。

</> 示例 17.29: tree/step2/tree.py:`tree` 生成根类的名称,然后委托 `sub_tree`

```
1 def tree(cls):
2     yield cls.__name__, 0
3     yield from sub_tree(cls)           ①
4
5 def sub_tree(cls):
6     for sub_cls in cls.__subclasses__():
7         yield sub_cls.__name__, 1     ②
8
9 def display(cls):
10    for cls_name, level in tree(cls): ③
11        indent = ' ' * 4 * level
12        print(f'{indent}{cls_name}')
13
14 if __name__ == '__main__':
15     display(BaseException)
```

① 委托 `sub_tree`,以生成子类的名称。

- ❷ 生成每个子类的名称与层级 1。由于函数 tree 中包含 “yield from sub_tree(cls)”, 因此这些值会完全绕过生成器函数 tree ...
 ❸ ... 并直接在此处接收。

秉持“婴儿学步”的原则, 将编写最简单的代码以到达(reach)异常层次结构的第 2 层。采用深度优先算法遍历树状结构: 在生成第 1 层中的每个 1 层节点之后, 在恢复第 1 层之前, 我想生成各个 1 层节点在第 2 层中的子节点。¹¹ 此步操作可用嵌套的 for 循环来完成, 如示例 17.30 所示。

</> 示例 17.30: tree/step3/tree.py:sub_tree 深度优先遍历第 1 层与第 2 层

```

1 def tree(cls):
2     yield cls.__name__, 0
3     yield from sub_tree(cls)
4
5 def sub_tree(cls):
6     for sub_cls in cls.__subclasses__():
7         yield sub_cls.__name__, 1
8         for sub_sub_cls in sub_cls.__subclasses__():
9             yield sub_sub_cls.__name__, 2
10
11 def display(cls):
12     for cls_name, level in tree(cls):
13         indent = ' ' * 4 * level
14         print(f'{indent}{cls_name}')
15
16 if __name__ == '__main__':
17     display(BaseException)

```

运行示例 17.30 中的 step3/tree.py 脚本, 得到的结果如下所示。

```

1 $ python3 tree.py
2 BaseException
3     Exception
4         TypeError
5             StopAsyncIteration
6             StopIteration
7             ImportError
8             OSError
9             EOFError
10            RuntimeError
11            NameError
12            AttributeError
13            SyntaxError
14            LookupError
15            ValueError
16            AssertionError
17            ArithmeticError
18            SystemError

```

¹¹ 英文原文: For depth-first tree traversal, after yielding each node in level 1, I want to yield the children of that node in level 2, before resuming level 1。

```

19     ReferenceError
20     MemoryError      BufferError
21     Warning
22     GeneratorExit
23     SystemExit
24     KeyboardInterrupt

```

您或许已知道接下来要做什么,但我还是要遵循“婴儿学步”原则:再嵌套一层 for 循环,以到达异常层次结构的第 3 层。程序的其余部分保持不变,因此 [示例 17.31](#) 仅显示了 sub_tree 生成器。

</> [示例 17.31: tree/step4/tree.py](#) 的 sub_tree 生成器

```

1 def sub_tree(cls):
2     for sub_cls in cls.__subclasses__():
3         yield sub_cls.__name__, 1
4         for sub_sub_cls in sub_cls.__subclasses__():
5             yield sub_sub_cls.__name__, 2
6             for sub_sub_sub_cls in sub_sub_cls.__subclasses__():
7                 yield sub_sub_sub_cls.__name__, 3

```

通过 [示例 17.31](#) 可清晰地看到其处理模式(逻辑):一层 for 循环对应 异常层次结构 的一层子类。若某一层 for 循环生成第 N 层子类,则下一层 for 循环会访问第 N+1 层子类。

如 [小节 17.11.1](#) 所述,可以用“yield from”表达式取代嵌套的内层 for 循环,以生成项。此处也可以应用这一思想:让 sub_tree 函数接受一个 level 参数(指定层级),以传入的当前子类为新的根类,递归地从 sub_tree 中生成项。详见 [示例 17.32](#)。

</> [示例 17.32: tree/step5/tree.py](#): 递归地从 sub_tree 中生成项(只要内存够用)

```

1 def tree(cls):
2     yield cls.__name__, 0
3     yield from sub_tree(cls, 1)
4
5 def sub_tree(cls, level):           \ding{202}
6     for sub_cls in cls.__subclasses__():
7         yield sub_cls.__name__, level
8         yield from sub_tree(sub_cls, level+1) # 指定层级, 以sub\_\_cls为新根类, 递归
9         生成子类
10
11 def display(cls):
12     for cls_name, level in tree(cls):
13         indent = ' ' * 4 * level
14         print(f'{indent}{cls_name}')
15
16 if __name__ == '__main__':
17     display(BaseException)

```

[示例 17.32](#) 可遍历任意深度的树状结构,仅受 Python 递归深度的限制,默认允许有 1000 个带执行的函数。

任何关于递归的优秀教程都会强调基例(base case)对于避免无限递归的重要性。“基例(base case)

”是一种用 if 语句实现的条件分支,一旦满足条件就返回,不再进行递归调用。示例 17.32 中的 sub_tree 没有 if 语句,但在 for 循环(❶处)中有一个隐式条件:若 `cls.__subclasses__()` 返回空列表,则不会执行循环体,因此不会发生递归调用。也就是说,此处的基例是 `cls` 类没有子类。此时, `sub_tree` 不生成任何项,而是直接返回。

示例 17.32 可以按预期运行,不过还可以用示例 17.31 中的处理模式,对代码进一步精简:某层 for 循环生成第 N 层子类,那么下一层 for 循环将生成第 N+1 层子类。在示例 17.32 中,我们用“yield from”表达式取代了下一层循环。现在,可以将 `tree` 与 `sub_tree` 合并成一个生成器函数,如??。这也是本示例的最后一步。

</> 示例 17.33: `tree/step6/tree.py`:递归调用函数 `tree`,传入递增的参数 `level`

```

1 def tree(cls, level=0):
2     yield cls.__name__, level
3     for sub_cls in cls.__subclasses__():
4         yield from tree(sub_cls, level+1)
5
6 def display(cls):
7     for cls_name, level in tree(cls):
8         indent = ' ' * 4 * level
9         print(f'{indent}{cls_name}')
10
11 if __name__ == '__main__':
12     display(BaseException)

```

如“17.11 `yield from` 表达式<510页>”开头所述,“`yield from`”可以绕过委托生成器,直接建立子生成器与客户端代码之间的联系。当将生成器用作协程时,这种联系就变得非常重要,不仅可以生成值,还可以消耗客户端代码中的值,详见“17.13 经典协程<518页>”。

在了解“`yield from`”之后,接下来介绍有关可迭代对象与迭代器的类型提示(Type Hints)。

17.12 泛化可迭代类型

Python 标准库中有许多接受可迭代对象作为参数的函数。在我们编写的代码中,这些函数可以像函数 `zip_replace`(见“示例 8.15<228页>”)那样,用 `collections.abc.Iterable`(如果必须支持 Python ≤ 3.8 ,则用 `typing.Iterable`,详见“8.5.4 泛化容器<220页>”一节的附注栏“向后兼容与弃用的容器类型”)为这些函数添加类型提示(Type Hints)。具体用法如示例 17.34 所示。

</> 示例 17.34: `replacer.py`:返回一个产生字符串元组的迭代器

```

1 from collections.abc import Iterable
2
3 FromTo = tuple[str, str] ❶
4
5 def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷
6     for from_, to in changes:
7         text = text.replace(from_, to)
8     return text

```

- ① 定义类型别名。非必需,不过可以提高后面类型提示的可读性。从 Python 3.10 开始,也应该为 FromTo 添加类型提示 typing.TypeAlias,以明确表明此行代码的意图。例如,“FromTo: TypeAlias = tuple[str, str]”。
- ② 为参数 changes 添加类型提示,表示 change 接受元素类型为 FromTo 的可迭代对象(Iterable)。

Iterator 类型没有 Iterable 类型那么常用,但是编写方式也很简单。示例 17.35 展示了带类型提示的斐波那契生成器。

</> 示例 17.35: fibo_gen.py: fibonacci 返回一个产出整数的生成器

```

1  from collections.abc import Iterator
2
3  def fibonacci() -> Iterator[int]:
4      a, b = 0, 1
5      while True:
6          yield a
7          a, b = b, a + b

```

注意,Iterator 类型用于为含有关键字 yield 的生成器函数添加 类型提示 (Type Hints),也可以为手动编写的含有__next__ 方法的迭代器类添加类型提示。还有一个 collections.abc.Generator (以及弃用的 typing.Generator),可用于注解生成器对象。但是,当将生成器用作迭代器时,则无须多此一举。

当用 Mypy 对示例 17.36 执行静态类型检查时,会发现 Iterator 类型实际上是 Generator 类型的简化特例。

</> 示例 17.36: tergentype.py:注解迭代器的 2 种方式

```

1  from collections.abc import Iterator
2  from keyword import kwlist
3  from typing import TYPE_CHECKING
4
5  short_kw = (k for k in kwlist if len(k) < 5) ❶
6
7  if TYPE_CHECKING:
8      reveal_type(short_kw) ❷
9
10 long_kw: Iterator[str] = (k for k in kwlist if len(k) >= 4) ❸
11
12 if TYPE_CHECKING:
13     reveal_type(long_kw) ❹

```

- ❶ 这个生成器表达式获取字符数少于 5 个的 Python 关键字。
- ❷ MyPy 推断出的类型为 “typing.Generator[builtins.str*, None, None]”。¹²
- ❸ 这个生成器表达式也会生成字符串,但我明确为其添加了类型提示 (Type Hints)。
- ❹ 揭示类型为 “typing.Iterator[builtins.str]”。

“abc.Iterator[str]”与“abc.Generator[str, None, None]”相容 (Consistent-with),因此 MyPy 对示

¹² 截至 0.910 版,MyPy 仍在使用已弃用的类型,如 typing.Generator 等。

例 17.36 执行类型检查时,没有报错。

“Iterator[T]”是“Generator[T, None, None]”的简写形式。这 2 个注解的意思都是“生成器产生 T 类型的项,但不消耗值或不返回值”。能消耗值或能返回值的生成器是“协程 (Coroutine)”,详见下一节。

17.13 经典协程



“PEP 342 –Coroutines via Enhanced Generators”引入了 `send()` 方法与其他功能后,使得将生成器用作协程 (Coroutine) 成为可能。PEP 342 中“协程 (Coroutine)”一词的含义,与这里的含义相同。

遗憾的是,Python 官方文档与标准库在指代用作协程的生成器时,所用的术语不统一。因此,不得不为其加上限定词,用“经典协程 (classic coroutine)”与新出现的“原生协程 (native coroutine)”相互区分。

在 Python 3.5 发布之后,“协程”通常就是指“原生协程 (native coroutine)”。但 PEP 342 还未被废弃,“经典协程”最初的作用未改变,尽管已不受 `asyncio` 支持。

理解 Python 中的经典协程有些令人困惑,因为它们实际上是以不同方式使用的生成器。所以让我们退一步,考虑 Python 中另一个可以以两种方式使用的特性。

如“2.4 无组不仅仅是不可变列表<26页>”所述,元组实例不但可用作记录,还可用作不可变序列。当用作记录时,元组中的项数固定,并且每一项可以是不同类型的值。当用作不可变序列时,元组中的项数随意,并且所有项都应具有相同类型。因此,元组的类型提示有 2 种方式:

```

1 # 一个城市记录, 包括名称、国家 (地区) 与人口:
2 city: tuple[str, str, int]
3
4 # 一个不可变序列, 包含一些列域名
5 domains: tuple[str, ...]

```

生成器也有类似情况。生成器通常被用作迭代器,但也可被用作协程 (Coroutine)。协程 (Coroutine) 其实就是一个生成器函数,即主体中包含关键字 `yield` 的函数。自然,协程对象就是生成器对象。尽管 Python 生成器与协程在 C 语言中有相同的底层实现,但二者的适用场景差异很大,因此其类型提示 (Type Hints) 包含 2 种不同的写法。

```

1 # readings 变量可以绑定到产生 float 项的迭代器或生成器对象
2 readings: Iterator[float]
3
4 # sim_tax 变量可以绑定到一个表示出租车的协程, 模拟离散事件
5 # 该变量产出事件, 接收 float 时间戳, 返回仿真过程中的行程次数
6 sim_taxi: Generator[Event, float, int]

```

更令人不解的是 `typing` 模块 的作者决定将该类型命名为 `Generator`,但实际上该类型描述的是被用作协程 (Coroutine) 的生成器对象的 API,而生成器更常用作简单的迭代器。

`typing` 模块文档 对 `Generator` 的 形式 (Formal) 类型参数 的描述,如下所示:

```
1 Generator[YieldType, SendType, ReturnType]
```

只有将生成器用作协程 (Coroutine) 时,参数 SendType 才有意义。SendType 类型是 gen.send(x) 调用中 x 的类型。在编码为迭代器 (而不是协程) 的生成器上调用.send() 方法将会报错。同样,ReturnType 也仅在注解协程时才有意义,因为迭代器不像常规函数那样拥有 return 语句。对于用作迭代器的生成器来说,唯一合理的选择是 next(it) 调用——可以直接调用,或通过 for 循环等形式间接调用。而 YieldType 是 next(it) 调用的返回值类型。

typing.Coroutine 的 形式 (Formal) 类型参数 与 typing.Generator 相同,如下所示。

```
1 Coroutine[YieldType, SendType, ReturnType]
```

其实,typing.Coroutine 文档也说过:“Coroutine 中类型变量的型变 (variance) 和顺序,与 typing.Generator 一样。”但是,typing.Coroutine (已弃用) 与 collections.abc.Coroutine (自 Python 3.9 起可用的泛型) 仅用于注解原生协程,无法注解经典协程。若想注解经典协程,则只能使用模糊不清的 “Generator[YieldType, SendType, ReturnType]”

David Beazley 创建了一些有关经典协程的最佳演讲和最全面的研讨会。在他的 PyCon 2009 课程讲义中,他有一张名为 “Keeping It Straight” 的幻灯片,内容如下:

- 生成器生产供迭代的数据。
- 协程 (Coroutine) 是数据的消费者。
- 为保持头脑清醒,不要混淆这 2 个概念 (即生成器与协程)。
- 协程 (Coroutine) 与迭代没有关系。
- 注意:虽然在协程中也可以用 yield 生成一个值,但它与迭代无关。¹³

接下来,看看经典协程是如何工作的。

17.13.1 示例:用协程计算累计平均值

在“九装饰器与闭包”中讨论闭包 (closures) 时,分析了可用于计算累计平均值的对象:“示例 9.7<254页>”定义了一个类,而“示例 9.12<256页>”定义了一个高阶函数——生成一个闭包,在多次调用之间记录 total 与 count 变量的值。示例 17.37 展示了如何用协程 (Coroutine) 实现相同的功能。¹⁴

</> 示例 17.37: coroaverager.py: 定义一个计算累计平均值的协程

```
1 from collections.abc import Generator
2
3 def averager() -> Generator[float, float, None]:
4     total = 0.0
5     count = 0
6     average = 0.0
7     while True:          ①
8         term = yield average ②
9         total += term
10        count += 1
11        average = total/count
```

¹³ 详见 “A Curious Course on Coroutines and Concurrency”, 第 33 张幻灯片 “Keeping It Straight”。

¹⁴ 此示例的灵感来自 Jacob Holm 在 Python-ideas 邮件列表中发布的一个代码片段,消息题为 “Yield-From: Finalization guarantees”。在该消息的后续回复中,还有几个不同版本。Holm 在消息 003912 中进一步说明了自己的想法。

- ❶ 该函数返回一个生成器,该生成器产出 float 值,通过.send() 接受 float 值,但不返回有用的数值。¹⁵
- ❷ 这种无限循环意味着,只要客户端代码发送值,协程 (Coroutine) 就会继续产生平均值。
- ❸ 这里的 yield 语句挂起了协程,向客户端产生结果;然后,获取调用者发送给协程 (Coroutine) 的值,开始无限循环的另一次迭代。

使用 协程 (Coroutine) 的好处是,可将 total 与 count 声明为局部变量:当 协程 (Coroutine) 被挂起 (暂停) 并等待下一次.send() 调用时,不需要通过实例属性或闭包来保存上下文。这就是为什么在异步编程中,协程 (Coroutine) 是回调的理想替代品——协程 (Coroutine) 能在两次激活之间保存本地状态。

</> 示例 17.38: coroaverager.py:示例 17.37 中定义的累计平均值协程的 doctests

```

1  >>> coro_avg = averager()    ❶
2  >>> next(coro_avg)         ❷
3  0.0
4  >>> coro_avg.send(10)      ❸
5  10.0
6  >>> coro_avg.send(30)
7  20.0
8  >>> coro_avg.send(5)
9  15.0

```

- ❶ 创建 协程 (Coroutine) 对象。
- ❷ 开始执行 协程 (Coroutine)。这里生成 average 变量的初始值,即 0.0。
- ❸ 开始计算累计平均值:多次调用.send() 方法,生成当前平均值。

在 [示例 17.38](#) 中,调用 next(coro_avg) 使 协程 (Coroutine) 向前执行到 yield 处,生成平均值的初始值 (0.0)。也可以通过调用 coro_avg.send(None) 来启动 协程 (Coroutine)——这实际上是内置函数 next() 的作用。但是,不能发送 None 之外的值,因为 协程 (Coroutine) 只有在 yield 处挂起时,才能接受发送的值。调用 next() 或.send(None) 向前执行到第一个 yield 的过程,被称为“启动 (priming) 协程”。

每次激活后,协程 (Coroutine) 都在 yield 处挂起,并等待发送值。coro_avg.send(10) 一行提供了这个值,从而激活 协程 (Coroutine)。yield 表达式将得到的值 (10) 赋值给变量 term。循环余下的部分将更新 total、count 与 average 这 3 个变量的值。while 循环的下一次迭代将生成变量 average 的值,并且 协程 (Coroutine) 在关键字 yield 处再次挂起。

细心读者可能急于知道如何终止 average 实例 (如 coro_avg) 的执行,因为它的主体包含一个无限循环。我们通常不需要终止生成器,因为一旦没有对生成器的有效引用,生成器就会被当作垃圾回收。如果想显式终止协程,请使用.close() 方法,如 [示例 17.39](#) 所示。

</> 示例 17.39: coroaverager.py:继续 [示例 17.38](#)

```

1  >>> coro_avg.send(20)      ❶
2  16.25
3  >>> coro_avg.close()      ❷
4  >>> coro_avg.close()      ❸
5  >>> coro_avg.send(5)       ❹

```

¹⁵ 事实上,除非有异常打破循环,否则它永远不会返回。Mypy 0.910 同时接受 None 和 typing.NoReturn 作为生成器返回类型参数,但它也接受该位置的 str,因此显然此时它无法完全分析协程代码。

```

6  Traceback (most recent call last):
7    ...StopIteration

```

- ❶ coro_avg 是 [示例 17.38](#) 中创建的实例。
- ❷ .close() 方法在挂起的 yield 表达式处 (见 “[示例 17.37<519页>](#)”❸处) 引发 GeneratorExit 异常。如果 [协程 \(Coroutine\)](#) 函数中没有处理该异常，则该异常会终止协程。GeneratorExit 被包装 [协程 \(Coroutine\)](#) 的生成器对象捕获——这就是我们看不到报错 (GeneratorExit) 的原因。
- ❸ 在先前已关闭的 [协程 \(Coroutine\)](#) 上调用 .close() 没有任何效果。
- ❹ 在先前已关闭的 [协程 \(Coroutine\)](#) 上调用 .send(), 将引发 StopIteration 异常。

除了 .send() 方法之外, “[PEP 342: Coroutines via Enhanced Generators](#)” 还引入了一种让 [协程 \(Coroutine\)](#) 返回值的方法。下一节, 将介绍这种方法。

17.13.2 让协程返回一个值

本节将研究另一个用于计算平均值的 [协程 \(Coroutine\)](#)。此版本不会生成中间结果, 而是返回一个指明项数与平均值的元组。我将代码清单分成两部分, 分别为 [示例 17.40](#) 与 [示例 17.41](#)。

</> [示例 17.40: coroaverager2.py: 文件上半部分](#)

```

1  from collections.abc import Generator
2  from typing import Union, NamedTuple
3
4  class Result(NamedTuple):
5      count: int # type: ignore
6      average: float
7
8  class Sentinel:
9      def __repr__(self):
10         return f'<Sentinel>'
11
12 STOP = Sentinel()
13
14 SendType = Union[float, Sentinel]

```

- ❶ [示例 17.41](#) 中的 averager2 [协程 \(Coroutine\)](#) 将返回一个 Result 实例。
- ❷ Result 实际上是 tuple 的子类。tuple 有一个此处用不到的 .count() 方法。“# type: ignore” 注释是为了防止 Mypy 发出关于 count 字段的警告¹⁶。
- ❸ 创建 [哨兵值 \(Sentinel Value\)](#) 的类。声明 __repr__ 方法是为了提高字符串表示形式的可读性。
- ❹ [哨兵值 \(Sentinel Value\)](#), 用于让 [协程 \(Coroutine\)](#) 停止收集数据并返回结果。
- ❺ [协程 \(Coroutine\)](#) 返回类型, 即 Generator 的第 2 个类型参数 (SendType) 将使用这个类型别名。

此处定义的 SendType 可适用于 Python 3.10。但是, 如果不需要向后兼容, 最好从 [typing 模块](#) 中导入

¹⁶ 我考虑过重新命名此字段, 但 count 是 [协程 \(Coroutine\)](#) 中局部变量的最佳名称, 而且本书中相关示例也都使用这个名称。因此, 在 Result 中, 使用相同的名称也是合理的。如果一味遵从静态类型检查工具, 会导致代码晦涩难懂、异常复杂, 我会毫不犹豫地使用 “# type: ignore” 来避免类型检查工具的限制及烦恼。

TypeAlias 后,像下面这样编写:

```
1 SendType: TypeAlias = float | Sentinel
```

用 `|` 取代 “typing.Union”, 可使代码更简洁易读, 甚至可以无需创建类型别名, 而是直接编写如下所示的 `averager2` 函数签名:

```
1 def averager2(verbose: bool=False) -> Generator[None, float | Sentinel, Result]:
```

接下来,研究一下 `协程 (Coroutine)` 代码本身,如示例 17.41所示。

</> 示例 17.41: coroaverager2.py:文件上半部分

```
1 def averager2(verbose: bool = False) -> Generator[None, SendType, Result]: ❶
2     total = 0.0
3     count = 0
4     average = 0.0
5     while True:
6         term = yield ❷
7         if verbose:
8             print('received:', term)
9         if isinstance(term, Sentinel): ❸
10            break
11         total += term ❹
12         count += 1
13         average = total / count
14     return Result(count, average) ❺
```

❶ 对于此 `协程 (Coroutine)`, 由于不生成数据,因此 `yield` 类型为 `None`。接收 `SendType` 类型的数据,最后返回一个 `Result` 元组。

❷ 像这样使用 `yield` 只有在 `协程 (Coroutine)` 中才有意义, `协程 (Coroutine)` 是为了消耗数据而设计的。此 `yield` 生成 `None`,从`.send(term)`接收 `term`。

❸ 如果 `term` 是一个 `哨兵值 (Sentinel Value)`,则跳出循环。正是因为这个 `isinstance` 检查 ...

❹ ...Mypy 才允许我将 `term` 加到 `total` 上,而不会标记错误:不能将一个 `float` 值加到一个可能为 `float` 值或 `哨兵值 (Sentinel Value)` 上。

❺ 只有将 `哨兵值 (Sentinel Value)` 发送给 `协程 (Coroutine)` 时,才会执行到这一行。

现在,看看如何使用这个协程。首先,从一个实际上不会产生结果的简单示例开始(示例 17.42)。

</> 示例 17.42: coroaverager2.py:关闭协程

```
1 >>> coro_avg = averager2()
2 >>> next(coro_avg)
3 >>> coro_avg.send(10) ❶
4 >>> coro_avg.send(30)
5 >>> coro_avg.send(6.5)
6 >>> coro_avg.close() ❷
```

❶ 注意,`averager2` 不会生成在计算过程中得到的结果。它生成 `None`,被 Python 控制台忽略。

② 调用 `close()` 后, 协程 (Coroutine) 停止, 但不会返回结果, 因为在协程 (Coroutine) 中的 `yield` 处引发了 `GeneratorExit` 异常, 因此永远不会执行到 `return` 语句。

示例 17.43 让这个协程返回结果。

</> 示例 17.43: 捕获 `StopIteration` 异常, 返回一个 `Result`

```
1  >>> coro_avg = averager2()
2  >>> next(coro_avg)
3  >>> coro_avg.send(10)
4  >>> coro_avg.send(30)
5  >>> coro_avg.send(6.5)
6  >>> try:
7  ...     coro_avg.send(STOP) ❶
8  ... except StopIteration as exc:
9  ...     result = exc.value ❷
10 ...
11 >>> result ❸
12 Result(count=3, average=15.5)
```

- ❶ 发送 哨兵值 (Sentinel Value) `STOP`, 使协程 (Coroutine) 退出循环, 并返回一个 `Result`。然后, 封装该协程 (Coroutine) 的生成器对象会引发 `StopIteration` 异常。
- ❷ `StopIteration` 实例有一个 `value` 属性, 该属性被绑定到终止协程 (Coroutine) 的 `return` 语句的值。
- ❸ 信不信由你。

这种从 `StopIteration` 异常中“偷运”协程返回值的想法, 是一种怪异的用法。然而, “PEP 342—Coroutines via Enhanced Generators”中就是这么用的, `StopIteration` 异常文档 与《The Python Language Reference》的“6.2.9. Yield expressions”一节也是这么做的。

委托生成器可以使用“`yield from`”语法直接获取协程 (Coroutine) 的返回值, 如示例 17.44 所示。

</> 示例 17.44: `coroaverager2.py`: 捕获 `StopIteration` 异常, 返回一个 `Result`

```
1  >>> def compute():
2  ...     res = yield from averager2(True) ❶
3  ...     print('computed:', res) ❷
4  ...     return res ❸
5  ...
6  >>> comp = compute() ❹
7  >>> for v in [None, 10, 20, 30, STOP]: ❺
8  ...     try:
9  ...         comp.send(v) ❻
10 ...     except StopIteration as exc:
11 ...         result = exc.value ❼
12 received: 10
13 received: 20
14 received: 30
15 received: <Sentinel>
16 computed: Result(count=3, average=20.0)
17 >>> result ❽
```

```
18 | Result(count=3, average=20.0)
```

- ❶ res 将收集 averager2 的返回值; “yield from” 机制在处理表示协程 (Coroutine) 终止的 StopIteration 异常时, 会获取协程 (Coroutine) 的返回值。当参数 verbose=True 时, 可使协程 (Coroutine) 打印接收到的值, 方便观察操作过程。
- ❷ 运行此生成器时, 留意该行的输出。
- ❸ 返回结果。此结果也被包含在 StopIteration 异常中。
- ❹ 创建委托协程 (Coroutine) 对象。
- ❺ 该循环将驱动委托的协程。
- ❻ 发送的第一个值是 None, 用于启动协程; 最后一个值是 哨兵值 (Sentinel Value), 用于停止协程。
- ❼ 捕获 StopIteration, 以获取 compute 的返回值。
- ❽ averager2 与 compute 输出完毕后, 再查看 Result 实例。

示例 17.44 虽然未做太多事情, 但代码却很难理解。用.send() 方法驱动协程, 并获取结果可能会使代码难以理解。虽然可通过 “yield from” 语法简化操作, 但只能在委托生成器或委托协程中使用 “yield from” 语法。而且这个委托生成器或委托协程最终也需要通过一些复杂的代码来驱动, 如**示例 17.44** 所示。

从这些示例可看出, 直接使用协程 (Coroutine) 既繁琐又复杂。如果再加上异常处理与协程的.throw() 方法, 示例会更复杂。本书不会介绍.throw() 方法, 因为它与.send() 方法一样, 仅在手动驱动协程时才有用。但是, 我不建议手动驱动协程, 除非您正在从头开发一个基于协程 (Coroutine) 的框架。



如果对深入理解 “经典协程” (包括 throw() 方法) 更感兴趣, 请访问本书配套网站 text 上的 “”。该文章提供了类 Python 伪代码, 详细介绍了如何用 yield from 驱动生成器与驱动协程, 以及通过一个小型的离线事件方针, 演示如何用协程实现一种并发形式 (而无需异步编程框架)。

在实践中, 用协程进行高效工作, 需要专门框架的支持。早在 Python 3.3 中, asyncio 就为经典协程提供了这种支持。随着 Python 3.5 中原生协程的出现, Python 核心开发人员正在逐步淘汰 asyncio 对经典协程的支持。但是, 底层机制十分相似。async def 语法使得代码中的原生协程更容易被发现, 这是一大进步。在内部, 原生协程使用 await (而不是 yield from) 来委托其他协程。原生协程详见 “[二十一 异步编程](#)”。

本章即将结束, 最后讨论一个令人费解的话题: 关于协程类型提示中的协变 (covariance) 与逆变 (contravariance)。

17.13.3 经典协程的泛型注解

“[15.7.4.3 逆变类型<443页>](#)” 所述, typing.Generator 是 Python 标准库 中少数具有逆变 (contravariance) 类型参数的类型之一。在学习经典协程之后, 现在可以介绍 typing.Generator 这种泛型 (Generic Type) 了。

在 Python 3.6 中¹⁷, typing.py 模块对 typing.Generator 的声明 (见 Lib/typing.py 2060 行), 如下所示:

```
1 | T_co = TypeVar('T_co', covariant=True)
```

¹⁷ 从 Python 3.7 开始, typing.Generator 及与 collections.abc 中抽象基类对应的其他类型, 都用相应抽象基类的包装器进行了重构。因此, 在 typing.py 源文件中看不到它们的泛化参数。正因如此, 这里引用了 Python 3.6 的源码。

```

2 V_co = TypeVar('V_co', covariant=True)
3 T_contra = TypeVar('T_contra', contravariant=True)
4 # 省略许多行
5
6 class Generator(Iterator[T_co], Generic[T_co, T_contra, V_co],
7     extra=_G_base):

```

根据上述声明,可知 Generator 类型提示包含 3 个类型参数(如“17.13 经典协程<518页>”所见),示例如下所示:

```
1 my_coro : Generator[YieldType, SendType, ReturnType]
```

从形式 (Formal) 类型参数的类型变量可以看出, YieldType 与 ReturnType 可以 协变 (covariance), SendType 可以 逆变 (contravariance)。为理解其中原因,可将 YieldType 与 ReturnType 视作“输出”类型,二者都描述了来自 协程 (Coroutine) 对象(即用作协程对象的生成器对象)的数据。

YieldType 与 ReturnType 可以 协变 (covariance),是有道理的。因为任何期望生成 float 值的协程的代码,都可以使用生成 int 值的协程。这就是 Generator 在其 YieldType 参数上是 协变 (covariance) 的原因。同样的推理过程,也适用于 协变 (covariance) 的 ReturnType 参数。

根据“15.7.4.2 协变类型<443页>”引入的符号,第 1 个与第 3 个参数的 协变 (covariance),可以用指向同一方向的:> 符号表示:

```

1     float :> int
2 Generator[float, Any, float] :> Generator[int, Any, int]

```

YieldType 和 ReturnType 体现了“15.7.4.4 型变经验法则”中的第 1 条规则:

- 如果形式 (Formal) 类型参数定义了从对象中产生的数据类型,那么它就可以是 协变 (covariance) 的。

另一方面,SendType 是“输入”参数,是协程对象.send(value) 方法中参数 value 的类型。如果客户端代码需要向协程发送 float 值,那么就不能使用 SendType 为 int 的协程,因为 float 不是 int 的子类型。换句话说, float 与 int 不相容 (Consistent-with)。但是客户端可以使用以 complex 作为 SendType 的协程,因为 float 是 complex 的子类型,即 float 与 complex 相容 (Consistent-with)。

可 逆变 (contravariance) 的第 2 个参数,用:> 符号描述更形象:

```

1     float :> int
2 Generator[Any, float, Any] <: Generator[Any, int, Any]

```

这体现了“15.7.4.4 型变经验法则”中的第 2 条规则:

- 如果形式 (Formal) 类型参数定义了在初始构造后进入对象的数据的类型,则它可以是 逆变 (contravariance) 的。

对型变 (variance) 的讨论,到此结束。至此,本书最长的一章也就此落下帷幕。

17.14 本章小结

Python 语言对迭代的支持如此深入,以至于我经常会说“Python 已经 (grok) 了迭代器”。¹⁸ Python 从语义上集成迭代器模式,很好地例证了“设计模式并非同样适用于所有编程语言。”在 Python 中,自己“手工”实现的经典迭代器(“[示例 17.4<487 页>](#)”),除了用于教学之外,并没有什么实际用处。

本章构建了多个版本的 Sentence 类,用于遍历文本文件(内容可能很长)中的各个单词。首先,介绍了如何用内置参数 `iter()` 将类似序列的对象创建为迭代器;然后,定义了一个含有 `__next__()` 方法的经典迭代器类;最后,用生成器多次重构 Sentence 类,使代码更简洁、更易读。

接下来,编写了一个用于生成等差数列的生成器,并展示了如何用 `itertools` 模块来简化代码。随后,概述了 Python 标准库中多数通用的生成器函数。

然后,通过 `chain` 与 `tree` 示例,以简单的生成器为背景,学习了“`yield from`”表达式的用法。

最后一节,重点介绍了经典协程。Python 3.5 增加了原生协程之后,经典协程的重要性逐渐减弱。尽管,经典协程在实践中很难使用,但经典协程却是原生协程的基础,而且“`yield from`”表达式是 `await` 关键字的直接前身。

此外,还介绍了 `Iterable`、`Iterator` 与 `Generator` 类型的[类型提示 \(Type Hints\)](#)。`Generator` 带有可逆变(`contravariance`)的类型参数,这是关于逆变(`contravariance`)的罕见示例。

17.15 延伸阅读

关于生成器的详细技术解释,请参阅《[The Python Standard Library](#)》的“[6.2.9. Yield expressions](#)”一节。关于定义生成器函数的 PEP 是“[PEP 255 -Simple Generators](#)”。

[itertools 模块文档](#)非常出色,因为其中包含了所有示例。虽然该模块内的函数都是用 C 语言编写的,但是文档展示了如何用 Python 实现其中的部分函数——这通常要借助于模块内其他函数来实现。用法示例也很出色,例如,有一个代码片段展示了如何使用 `accumulate` 函数,计算带利息的分期还款,还得出了每次应还多少钱。文档中还有 `Itertools Recipes` 一节,说明如何用 `itertools` 模块中的现有函数实现其他高性能函数。

在《[The Python Standard Library](#)》之外,我推荐您了解 `More Itertools` 包,它遵循 `itertools` 的优良传统,提供了功能强大的生成器、大量示例以及一些实用秘诀。

《[Python Cookbook, 3rd ed](#)》(David Beazley、Brian K. Jones 著)中的“第 4 章, Iterators and Generators”包含 16 个秘诀,从多个不同角度,以实际应用为出发点,涵盖了生成器与迭代器这一话题。其中,还包括一些关于“`yield from`”的示例,很具有启发性。

Sebastian Rittau([typeshed](#) 顶级贡献者)在文章“[Java: Iterators are not Iterable](#)”(2006 年)中,解释了迭代器为什么应该是可迭代的。

在“[What's New In Python 3.3](#)”的“[PEP 380-Syntax for Delegating to a Subgenerator](#)”一节,通过示例介绍了“`yield from`”的语法。我在 [fluentpython.com](#) 上发表的“[Classic Coroutines](#)”一文对“`yield from`”进行了深入介绍,包括其在 C 语言中实现的 Python 伪代码。

David Beazley 是 Python 生成器与协程方面的终极权威。他与 Brian K. Jones 合著的《[Python Cookbook, 3rd ed](#)》中有许多关于[协程 \(Coroutine\)](#)的示例。Beazley 的 PyCon 教程以其深度和广度而闻名。首先是 PyCon US 2008 上的“[Generator Tricks for Systems Programmers, v3.0](#)”。在 PyCon US 2009,他带来

¹⁸根据 [Jargon File](#) 的定义,“gork”是指不仅仅学会了某种知识,而且还要充分吸收它,做到“人剑合一”。

了传奇性的“*A Curious Course on Coroutines and Concurrency*”(3 个部分的视频链接都很难找到:第 1 部分、第 2 部分、第 3 部分)。他在 PyCon 2014 蒙特利尔 (Montréal) 会议上的教程是“*Generators: The Final Frontier*”,他在其中讨论了更多并发示例,因此实际上更多是关于本书“[二十一 异步编程](#)”的主体。Dave 在课堂上总是忍不住让人脑洞大开,甚至在“*Generators: The Final Frontier*”的最后一部分,用协程 (Coroutine) 取代了经典的 Visitor 模式,实现了一个算术表达式求值器。

协程 (Coroutine) 允许以新的方式组织代码,不过与递归或多态 (动态分派) 一样,需要花点时间才能适应这种方式。James Powell 在文章“*Greedy algorithm with coroutines*”中用协程 (Coroutine) 重写了经典的算法。

《*Effective Python (第 1 版)*》(Brett Slatkin 著)一书中,有一条标题为“考虑用协程来并发地运行多个函数”,虽然篇幅不长,但内容很精彩。该书第 2 版删掉了这一条,但网上有样章,可[在线阅读](#)。Slatkin 给出的用“`yield from`”驱动协程的示例,是我见过最棒的。该示例实现了 John Conway 发明的“生命游戏 (Game of Life)”,用协程 (Coroutine) 管理游戏运行过程中各个单元的状态。我重构了该游戏示例的代码,将实现游戏的函数和类与 Slatkin 原始代码中使用的测试片段分开。我还编写了 doctest 形式的测试,因此无需运行脚本即可看到各个协程与类的输出。重构后的示例发布在 [GitHub Gist](#) 上。

杂谈

Python 中极简的迭代器接口

《*设计模式*》一书讲解迭代器模式时,在“实现”一节写道:^a

迭代器的最简接口由 `First`、`Next`、`IsDone` 和 `CurrentItem` 操作组成。

不过,这句话包含一个脚注,内容如下:

可以将 `Next`、`IsDone` 与 `CurrentItem` 合并为单个操作来进一步减小接口的规模。这个新的操作会前进到下一个对象并返回该对象。如果遍历已经完成,那么这个操作会返回一个特殊值(例如 0),用以标记迭代的结束。

这与我们在 Python 中使用的方法很相似:只用一个方法 `__next__` 即可完成工作。不过,为了表明迭代结束,此方法未使用可能会被意外忽略的 [哨兵值 \(Sentinel Value\)](#),而是使用 `StopIteration` 异常来表示迭代的结束。简单而正确:这就是 Python 的方式。

插入式的生成器

任何管理大型数据集的程序员,都可以在实践中找到机会使用生成器。如下是我首次围绕生成器构建实用解决方案的真实故事。

多年前,我在 BIREME 工作,这是 PAHO/WHO (Pan-American Health Organization/World Health Organization, 泛美卫生组织/世界卫生组织) 在巴西圣保罗运营的数字图书馆。BIREME 创建的书目数据集包括 LILACS (Latin American and Caribbean Health Sciences index, 拉丁美洲和加勒比地区健康科学索引) 与 SciELO (Scientific Electronic Library Online, 科学电子在线图书馆), 这 2 个综合数据库完整地索引了该地区发表的健康科学的研究文献。

自 20 世纪 80 年代末以来,用于管理 LILACS 的数据库系统是 CDS/ISIS,这是一个由 UNESCO (United Nations Educational, Scientific and Cultural Organization, 联合国教科文组织) 创建的非关系型

文档数据库。我的工作之一是研究替代方案, 将 LILACS (以及最终将规模更大的 SciELO) 迁移到现代开源文档数据库(如 CouchDB 或 MongoDB)。当时, 我写了一篇题为 “From ISIS to CouchDB: Databases and Data Models for Bibliographic Records” 的论文, 文中介绍了半结构化数据模型以及用类 JSON 的记录表示 CDS/ISIS 数据的不同方式。

在研究过程中, 我编写了一个 `isis2json.py` 脚本, 用于读取 CDS/ISIS 文件并写入适合导入 CouchDB 或 MongoDB 的 JSON 文件。最初, 该脚本读取由 CDS/ISIS 导出的 ISO-2709 格式的文件。由于完整的数据集比主内层大得多, 因此读写必须以增量方式进行。这很简单: 主 `for` 循环每次迭代都会从`.iso` 文件中读取一条记录, 对其进行处理, 然后写入`.json` 输出文件。

然而, 在实际操作中, `isis2json.py` 脚本有必要支持另一种 CDS/ISIS 数据格式——BIREME 生产中使用的二进制`.mst` 文件, 以避免导出 ISO-2709 格式时消耗过多资源。当时, 我遇到了一个问题: 用于读取 ISO-2709 与`.mst` 文件的库提供的 API 差异很大。而 JSON 写入的循环已经很复杂了, 因为脚本要接受多种命令行选项来重组每条输出记录的结构。在生成 JSON 的同一 `for` 循环中使用两个不同的 API 来读取数据会非常麻烦。

我的解决方案是将读取逻辑隔离到一对生成器函数中: 每个生成器函数对应一种受支持的输入格式。最后, 我将 `isis2json.py` 脚本拆分为 4 个函数。您可以在 GitHub 上的 [fluentpython/isis2json](#) 存储库中查看具有依赖项的 Python 2 源码。^b

以下是关于此脚本的结构概述:

- `main`

`main` 函数用 `argparse` 模块读取命令行选项, 配置输出记录的结构。根据输入文件的扩展名, 选择合适的生成器函数来读取数据, 并逐条生成记录。

- `iter_iso_records`

该生成器函数用于读取`.iso` 文件(假定为 ISO-2709 格式)。它接受 2 个参数: 一个是文件名, 另一个是 `isis_json_type`——与记录结构相关的选项之一。在此函数的 `for` 循环中, 每次迭代都会读取一条记录, 创建一个空 `dict`, 将字段数据填充其中, 并生成 `dict`。

- `iter_mst_records`

这是另一个生成器函数, 用于读取`.mst` 文件。^c 在查看 `isis2json.py` 源码后, 您会发现此函数不像 `iter_iso_records` 那么简单, 不过接口与整体结构是相同的: 接受 2 个参数, 一个是文件名, 另一个是 `isis_json_type`, `for` 循环每次迭代都会构建并生成一个 `dict`, 表示一条记录。

- `write_json`

该函数每次执行一条 JSON 记录的真实写入。它接受多个参数, 但第 1 个参数 `input_gen` 是对生成器函数(`iter_iso_records` 或 `iter_mst_records`)的引用。`write_json` 中的主 `for` 循环会遍历所选 `input_gen` 生成器生成的 `dict`, 根据命令行选项以不同方式对其重组, 然后将 JSON 记录追加到输出文件中。

通过利用生成器函数, 将读取逻辑与写入逻辑解耦。当然, 最简单的解耦方法是将所有记录读入内存, 然后再写入磁盘。但由于数据集过大, 这种方法不可行。使用生成器, 读取与写入可以交错进行, 因此脚本可以处理任意大小的文件。此外, 读取不同输入格式记录的特殊逻辑与调整记录结构的逻辑也是分开的。

现在, 如果需要 `isis2json.py` 脚本支持额外的输入格式(如 MARCXML), 则只需再添加一个生成器函数即可轻松实现, 而复杂的 `write_json` 函数无需任何改动。

这并非什么尖端科技,而是一个真实的案例。在这个案例中,生成器提供了一个高效灵活的解决方案,以记录流的形式处理数据库,无论数据集的大小如何,都能保持较低的内存使用率。

^a《设计模式:可复用面向对象软件的基础》(第 174 页)

^b代码之所以使用 Python 2,是因为它的一个可选依赖项是一个名为 Bruma 的 Java 库,可以在使用 Jython 运行脚本时导入该库,而 Jython 还不支持 Python 3。

^c用于读取复杂.mst 二进制文件的库实际上是用 Java 编写的,因此仅当使用 Jython 解释器(版本 2.5 或更高版本)执行 isis2json.py 时,此功能才可用。更多详情,请参阅存储库中的 README.rst 文件。因为依赖项被导入到需要它的生成器函数中,因此即使只有一个外部库可用,脚本也可以运行。

wechat: 119554488

with、match 和 else 代码块

上下文管理器可能最终会变得几乎与子程序 (subroutine) 本身一样重要。目前, 我们只是了解了上下文管理器的一些皮毛。在 Basic 等许多语言中都有一个 with 语句, 但各语言中 with 语句的作用各不相同, 并且它们都只是做了一些表面工作。虽然 with 语句可以帮你避免使用重复的点运算符(.) 查找属性, 但它并不会执行事前准备与事后清理。不要因为它们有相同的名称, 就认为它们的作用也一样。with 语句是一项非常重要的功能。^a

——Raymond Hettinger, Python 布道者

^a节选自 PyCon US 2013 主题演讲 “What Makes Python Awesome” 中关于 with 语句的部分 (从 23:00 开始到 26:15 结束)。

本章所介绍的控制流特性在其它语言中并不常见。因此, 在 Python 中往往被忽视或使用不够充分。它们是:

- with 语句与上下文管理器
- 使用 match/case 进行模式匹配
- for、while、try 语句的 else 子句

with 语句会设置一个临时上下文, 并在上下文管理器对象的控制下可靠地将临时上下文删除。这样可以防止错误并减少模板代码, 同时使 API 更安全、更易用。除了自动关闭文件外, with 语句还有许多与代码块相关的用途。

在前几章中, 已经介绍过模式匹配 (Pattern Match)。在本章, 将探讨如何用 序列模式 (Sequence Pattern) 来表达语言的语法。这也解释了为什么 match/case 是创建易于理解和扩展的语言处理器的有效工具。我们将研制一个完整的解释器, 实现 Scheme 语言的一小部分功能。您可以照猫画虎地使用类似的技术来开发模板语言或 DSL (Domain Specific Language, 特定领域语言), 以便在大型系统中编码业务逻辑。

else 子句并不是很重要, 但如果与 for、while 和 try 结合使用, 更有助于表达意图。

18.1 本章新增内容

“18.3 案例分析: lis.py 中的模式匹配”是第 2 版中新增的内容。

“18.2.1 contextlib 包中的实用工具”中的内容有更新,涵盖了自 Python 3.6 以来增加的上下文管理模块的一些特性,以及 Python 3.10 中引入的带括号的上下文管理器语法。

接下来,先从强大的 with 语句讲起。

18.2 上下文管理器与 with 块

with 语句旨在简化 try/finally 的一些常见用法,它能保证即使代码块被 return、异常或 sys.exit() 调用终止,也能执行某些指定的操作。finally 子句中的代码通常用于释放关键资源或恢复一些暂时更改的先前状态。

Python 社区正在为上下文管理器寻找新的创造性用途。标准库中的一些例子如下

- 在 sqlite3 模块中管理事务,详见 [sqlite3 模块文档](#) 的“How to use the connection context manager”一节。
- 安全地处理锁、条件与信号量,详见 [threading 模块文档](#)。
- 为十进制对象的算术运算设置自定义环境,详见 [decimal.localcontext 文档](#)。
- 为了测试,给对象打补丁,详见 [unittest.mock.patch 函数文档](#)。

上下文管理器接口由 `__enter__` 和 `__exit__` 方法组成。在 with 块的顶部,Python 会调用上下文管理器对象的 `__enter__` 方法,做某些事前准备工作。当 with 代码块完成或因任何原因终止时,Python 会调用上下文管理器对象的 `__exit__` 方法,做某些事后清理工作。

最常见的例子是确保关闭文件对象。[示例 18.1](#) 详细演示了如何使用 with 语句关闭文件。

</> [示例 18.1: 将文件对象作为上下文管理器的演示](#)

```

1  >>> with open('mirror.py') as fp: ❶
2      ...     src = fp.read(60) ❷
3
4  >>> len(src)
5  60
6  >>> fp
7  <_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
8  >>> fp.closed, fp.encoding ❸
9  (True, 'UTF-8')
10 >>> fp.read(60) ❹
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   ValueError: I/O operation on closed file.

```

❶ fp 与打开的文本文件绑定,因为文件的 `__enter__` 方法返回 `self`(表示打开的文件)。

❷ 从 fp 中读取 60 个 Unicode 字符。

❸ fp 变量仍然可用,with 块不会像函数那样定义新的作用域。

❹ with 块结束后,仍可读取 fp 对象的属性。

❺ 但无法从 fp 中读取更多文本,因为在 with 块结束时,TextIOWrapper.`__exit__` 方法被调用,并关闭了文件。

[示例 18.1](#) 中❶处的调用提出了一个微妙而关键的问题:对 with 后面的表达式进行求解的结果是一个上

下文管理器对象；但绑定到目标变量（as子句中的fp）的值是上下文管理器对象调用`__enter__`方法返回的结果。

碰巧，`open()`函数返回一个`TextIOWrapper`的实例，其`__enter__`方法返回`self`。但在其他类中，`__enter__`方法也可能返回其他对象，而不是上下文管理器实例。

无论控制流以哪种方式退出with代码块，都将在上下文管理器对象上调用`__exit__`方法，而不是在`__enter__`返回的对象上调用。

with语句中的as子句是可选的。对于`open()`函数来说，我们需要通过as子句来获取文件的引用，以便调用文件的方法。不过，有些上下文管理器会返回`None`，因为它们没有有用的对象可返回给用户。

示例18.2展示了无意义的上下文管理器操作，目的是强调上下文管理器与其`__enter__`方法返回的对象之间的区别。

</> 示例18.2：测试`LookingGlass`上下文管理器类

```

1  >>> from mirror import LookingGlass
2  >>> with LookingGlass() as what:          ❶
3  ...     print('Alice, Kitty and Snowdrop') ❷
4  ...     print(what)
5  ...
6  pordwonS dna yttiK ,ecila
7  YKCOWREBBAJ
8  >>> what
9  'JABBERWOCKY'
10 >>> print('Back to normal.')           ❸
11 Back to normal.

```

- ❶ 上下文管理器是一个`LookingGlass`实例；Python在上下文管理器上调用`__enter__`后，将结果会绑定到变量`what`上。
- ❷ 打印字符串，然后打印目标变量`what`的值。每次打印的内容都是反向的。
- ❸ 现在，with块执行完毕。可以看到`__enter__`返回的值（即存储在`what`中的值）是字符串‘JABBERWOCKY’。
- ❹ 程序打印的内容不再是反向的了。

示例18.3是`LookingGlass`类的实现。

</> 示例18.3：mirror.py: `LookingGlass`上下文管理器类的实现代码

```

1  import sys
2
3  class LookingGlass:
4      def __enter__(self):          ❶
5          self.original_write = sys.stdout.write ❷
6          sys.stdout.write = self.reverse_write ❸
7          return 'JABBERWOCKY'                 ❹
8
9      def reverse_write(self, text):      ❺
10         self.original_write(text[::-1])
11

```

```

12     def __exit__(self, exc_type, exc_value, traceback): ❶
13         sys.stdout.write = self.original_write ❷
14         if exc_type is ZeroDivisionError: ❸
15             print('Please DO NOT divide by zero!')
16             return True ❹
17     ❺

```

- ❶ Python 在调用 `__enter__` 时,除了 `self` 之外不带任何参数。
- ❷ 保留原始的 `sys.stdout.write` 方法,以便以后恢复。
- ❸ 为 `sys.stdout.write` 打猴子补丁 (Monkey-patch),将 `sys.stdout.write` 替换为自己编写的方法。
- ❹ 返回字符串“JABBERWOCKY”,该字符串内容会被存入目标变量 `what`。
- ❺ 这是取代 `sys.stdout.write` 的方法,将会反转 `text` 参数的内容,然后调用了原始实现。
- ❻ 如果一切顺利,Python 会用参数组合 `(None, None, None)` 调用 `__exit__` 方法;如果出现异常, `__exit__` 方法的 3 个参数 `(exc_type, exc_value, traceback)` 会得到异常数据,如本示例后所述。
- ❼ 还原 `sys.stdout.write` 的原始方法。
- ❽ 如果发生异常,且异常类型为 `ZeroDivisionError`,则打印一条信息。
- ❾ 并返回 `True`,告诉解释器异常已被处理。
- ❿ 如果 `__exit__` 返回 `None` 或其他 `False` 值,在 `with` 代码块中印发的任何异常都会被传播。



当实际应用程序接管标准输出时,它们往往希望用另一个类文件对象暂时替代 `sys.stdout`,然后再切换回原来的对象。`contextlib.redirect_stdout` 上下文管理器正是用于此目的:只需将替代 `sys.stdout` 的类文件对象传给它即可。

解释器调用 `__enter__` 方法时,除了隐式 `self` 之外,不带任何参数。但调用 `__exit__` 时,传入的 3 个参数是:

- `exc_type`
异常类(如 `ZeroDivisionError`)。
- `exc_value`
异常实例。有时,传递给异常构造函数的参数(如错误信息)可以在 `exc_value.args` 中找到。
- `traceback`
`traceback` 对象。

要详细了解上下文管理器的工作原理,请参见 [示例 18.4](#),其中 `LookingGlass` 在 `with` 代码块之外使用,因此我们可以手动调用它的 `__enter__` 和 `__exit__` 方法。

</> [示例 18.4: 在没有 with 代码块的情况下使用 LookingGlass](#)

```

1  >>> from mirror import LookingGlass
2  >>> manager = LookingGlass() ❶
3  >>> manager  # doctest: +ELLIPSIS
4  <mirror.LookingGlass object at 0x...>
5  >>> monster = manager.__enter__() ❷

```

¹如果在 `try/finally` 语句的 `finally` 代码块中调用 `sys.exc_info()`,`self` 接收到的就会与 `__exit__` 的 3 个参数完全相同。考虑到 `with` 语句旨在取代 `try/finally` 语句的大多数用法,而通常需要调用 `sys.exc_info()` 来确定需要进行哪些清理操作,因此这样做合理的。

```
6  >>> monster == 'JABBERWOCKY'          ❸
7  eurT>>> monster
8  'YKCOWREBBAJ'
9  >>> manager      # doctest: +ELLIPSIS
10 >... ta tcejbo ssalGgnikooL.rorrim<
11 >>> manager.__exit__(None, None, None) ❹
12 >>> monster
13 'JABBERWOCKY'
```

- ❶ 实例化并查看 LookingGlass 实例。
- ❷ 调用 manager 器的 `__enter__` 方法，并将结果存储在 `monster` 中。
- ❸ `monster` 是字符串 “JABBERWOCKY”。打印出的 True 标识符是反向的，因为所有通过 `stdout` 的输出都要先通过在 `__enter__` 中打了补丁的 `write` 方法进行处理（详见“示例 18.3<533页>”）。
- ❹ 调用 `manager.__exit__`，恢复以前的 `stdout.write`。



Python 3.10 中的带括号的上下文管理器

Python 3.10 采用了一种新的解析器，允许使用新语法，比旧的 LL(1) 解析器功能更强大。其中一个语法改进是允许使用括号的上下文管理器，就像下面这样：

```
1  with (
2      CtxManager1() as example1,
3      CtxManager2() as example2,
4      CtxManager3() as example3,
5  ):
6      ...
```

在 Python 3.10 版之前，必须用嵌套的 `with` 块才能达到同样效果。

标准库中的 `contextlib` 包 提供了一些函数、类和装饰器，可方便用于构建、组合和使用上下文管理器。

18.2.1 contextlib 包中的实用工具

在创建自己的上下文管理器类之前，先看一下 Python 文档中的“[contextlib — Utilities for with-statement contexts](#)”。也许你需要的功能已经存在，或者可能有一些类或可调用对象可用于简化上下文管理器类的实现过程。

除了 [示例 18.3](#) 后面提到的 `redirect_stdout` 上下文管理器之外，Python 3.5 还增加了 `redirect_stderr`，其作用与前者相同，但将输出指向 `stderr`。

`contextlib` 包 中包括以下函数：

- `closing(thing)`

若对象提供了 `close()` 方法，但未实现 `__enter__` 与 `__exit__` 接口，则可以使用此函数根据该对象构建上下文管理器。

- `suppress(*exceptions)`

用于构建暂时忽略给定异常的上下文管理器。

- `nullcontext(enter_result=None)` (Python 3.7 新增) 返回一个从 `__enter__` 返回 `enter_result` 的上下文管理器, 除此之外不执行任何操作。它可以用作 `with` 语句的占位符, 以满足语法要求, 从而以简化某些情况下的代码逻辑。当不确定 `with` 块之前的条件代码有没有为 `with` 语句提供上下文管理器时, 就可以用 `nullcontext` 代替。

`contextlib` 模块 提供的几个类与一个装饰器比上述函数的实用范围更广。

- `@contextmanager`

这个装饰器可以将一个简单的生成器函数转变成上下文管理器，而无需创建一个类并实现上下文管理接口。详见“[18.2.2 使用 @contextmanager](#)”。

- `AbstractContextManager` (Python 3.6 新增)

一个正式定义上下文管理器接口的抽象基类 (ABCs)，并且通过子类化此抽象基类 (ABCs)，可简化上下文管理器类的构建。

- ContextDecorator

这是一个基类，用于定义基于类的上下文管理器。这种基于类的上下文管理器也可用作函数装饰器，在受管理的上下文中运行整个函数。

- `ExitStack`

是一个上下文管理器，它可以让您进入多个上下文管理器。当 with 块结束时，ExitStack 会以 LIFO 顺序（后进先出）依次调用堆叠的上下文管理器的 `__exit__` 方法。当事先不知道在 with 块中需要进入多少个上下文管理器时，请使用这个类；例如，同时打开任意文件列表中的所有文件。

Python 3.7 为 `contextlib` 增加了 `AbstractAsyncContextManager`、`@asynccontextmanager` 和 `AsyncExitStack`。它们与名称中没有 `async` 部分的对应实用程序类似,但都是为了与新的 `async with` 语句一起使用而设计的,这将在“[二十一 异步编程](#)”中介绍。

这些实用程序中使用最广泛的是 `@contextmanager` 装饰器，因此值得更多关注。这个装饰器也很有趣，因为它展示了与迭代无关的 `yield` 语句的用法。

18.2.2 使用 @contextmanager

上下文管理器装饰器 `@contextmanager` 是一个优雅而实用的工具, 它将 3 个独特的 Python 特性 (函数装饰器、生成器和 with 语句) 结合到了一起。

使用 `@contextmanager` 能减少创建上下文管理器的样板代码: 你无需编写包含 `_enter_` 与 `_exit_` 方法的整个类, 而只需实现一个生成器, 生成器中的单个 `yield` 即可产生想让 `_enter_` 方法返回的值。

在使用 `@contextmanager` 装饰的生成器中, `yield` 将函数体分成两部分: `yield` 之前的代码将在 `with` 块开始时(在解释器调用 `enter` 时)执行; `yield` 之后的代码将在 `with` 块结束时(调用 `exit` 时)执行。

示例 18.5 用一个生成器函数替换了“示例 18.3<533页>”中的 LookingGlass 类。

</> 示例 18.5: mirror_gen.py:用生成器函数实现的上下文管理器

```
1 import contextlib
2 import sys
3
4 @contextlib.contextmanager
5 def looking_glass():
6     pass
```

```

6     original_write = sys.stdout.write    ❷
7     def reverse_write(text):            ❸
8         original_write(text[::-1])
9
10    sys.stdout.write = reverse_write    ❹
11    yield 'JABBERWOCKY'               ❺
12    sys.stdout.write = original_write  ❻

```

- ❶ 应用 `contextmanager` 装饰器。
- ❷ 保留原来的 `sys.stdout.write` 方法。
- ❸ `reverse_write` 可以稍后调用 `original_write`, 因为 `original_write` 在 `reverse_write` 的闭包中是可用的。
- ❹ 用 `reverse_write` 替换 `sys.stdout.write`。
- ❺ 产生一个值, 该值将绑定到 `with` 语句 `as` 子句的目标变量中。在执行 `with` 块的主体时, 生成器会暂停。
- ❻ 当控制流退出 `with` 块时, 生成器函数将继续执行 `yield` 之后的代码; 这里将恢复原来的 `sys.stdout.write`。

示例 18.6 演示了 `looking_glass` 函数的运行情况。

</> 示例 18.6: 测试 `looking_glass` 上下文管理器

```

1  >>> from mirror_gen import looking_glass
2  >>> with looking_glass() as what:    ❶
3  ...     print('Alice, Kitty and Snowdrop')
4  ...     print(what)
5  ...
6  pordwonS dna yttiK ,ecila
7  YKCOWREBBAJ
8  >>> what
9  'JABBERWOCKY'
10 >>> print('back to normal')
11 back to normal

```

- ❶ 与“示例 18.2<533页>”唯一不同的是上下文管理器的名称: `LookingGlass` 变成了 `looking_glass` 而不是。

`contextlib.contextmanager` 装饰器将该迭代器函数封装成一个实现了 `_enter__` 和 `_exit__` 方法的类²。

该类的 `_enter__` 方法具有以下作用:

1. 调用生成器函数, 获取生成器对象——姑且称之为“`gen`”。
2. 调用 `next(gen)`, 使其执行到 `yield` 关键字。
3. 返回 `next(gen)` 产生的值, 以便用户将其绑定到 `with/as` 语句中的目标变量。

`with` 块终止时, `_exit__` 方法做以下几件事:

1. 检查是否已将异常传递给 `exc_type`: 如果是, 则调用 `gen.throw(exception)`, 在生成器函数体内的 `yield` 行处产生异常。
2. 否则, 将调用 `next(gen)`, 在 `yield` 之后继续执行生成器函数余下的代码。

² 实际的类名为 `_GeneratorContextManager`。如果您想了解它的具体工作原理, 请阅读 Python 3.10 中的 `Lib/contextlib.py` 源码。

“示例 18.5<536页>”有一个缺陷：如果在 with 块的主体中引发异常，Python 解释器会捕获它，然后在 looking_glass 内部的 yield 表达式处再次引发该异常。但 yield 处没有错误处理，所以 looking_glass 生成器将终止执行，而永远不会恢复原来的 sys.stdout.write 方法，导致系统处于无效状态。

示例 18.7 添加了对 ZeroDivisionError 异常的特殊处理，使其在功能上等同于基于类的“示例 18.3<533页>”。

</> 示例 18.7：mirror_gen_exc.py：基于生成器的上下文管理器执行异常处理——外部行为同示例 18.3

```

1 import contextlib
2 import sys
3
4 @contextlib.contextmanager
5 def looking_glass():
6     original_write = sys.stdout.write
7
8     def reverse_write(text):
9         original_write(text[::-1])
10
11     sys.stdout.write = reverse_write
12     msg = ''          ❶
13     try:
14         yield 'JABBERWOCKY'
15     except ZeroDivisionError: ❷
16         msg = 'Please DO NOT divide by zero!'
17     finally:
18         sys.stdout.write = original_write ❸
19         if msg:
20             print(msg)          ❹

```

❶ 创建一个变量，保存可能出现的错误消息；与“示例 18.5<536页>”相比，这是第一处改动。

❷ 通过设置错误信息，来处理 ZeroDivisionError 异常。

❸ 撤销对 sys.stdout.write 的猴子补丁（Monkey-patch）。

❹ 如果设置了错误信息，则显示错误信息。

回想一下，`__exit__` 方法通过返回一个真实值告诉解释器它已经处理了异常；此时，解释器会抑制（`suppress`）异常。相反，如果 `__exit__` 没有显式地返回一个值，解释器就会得到通常的 `None` 值，并继续传播异常。使用了 `@contextmanager` 装饰器之后，默认行为就反过来了：装饰器提供的 `__exit__` 方法会假定任何发送到生成器的异常都已被处理，因此应该被抑制（`suppress`）。



在 `yield` 周围添加 `try/finally`（或 `with` 块）是使用 `@contextmanager` 装饰器无法避免的代价，因为你永远不知道上下文管理器的用户会在 `with` 块中做什么。^a

^a本提示引自本书技术审校 Leonardo Rochael 的评论。说得好，噢耶！

`@contextmanager` 还有一个鲜为人知的特性——用 `@contextmanager` 装饰的生成器，也是一个装饰器³。这是因为 `@contextmanager` 是通过 `contextlib.ContextDecorator` 类实现的。

³至少我和其他技术审校都不知道，是 Caleb Hattingh 告诉我们的。谢谢你，Caleb！

示例 18.8 将“示例 18.5<536页>”中的 looking_glass 上下文管理器当作装饰器使用。

</> 示例 18.8: looking_glass 上下文管理器也可当作装饰器使用

```
1  >>> @looking_glass()
2  ... def verse():
3  ...     print('The time has come')
4  ...
5  >>> verse()          ❶
6  emoc sah emit ehT
7  >>> print('back to normal') ❷
8  back to normal
```

❶ looking_glass 在 verse 主体运行之前和运行之后完成了自己的任务。

❷ 确认原来的 sys.write 已经恢复。

请对比一下示例 18.8 与“示例 18.6<537页>”。在示例 18.6 中,looking_glass 被用作上下文管理器。

在标准库之外, Martijn Pieters 的一个有趣示例“Easy in-place file rewriting”也使用了 @contextmanager, 如示例 18.9 所示。

</> 示例 18.9: 一个就地重写文件的上下文管理器

```
1  import csv
2
3  with inplace(csvfilename, 'r', newline='\n') as (infh, outfh):
4      reader = csv.reader(infh)
5      writer = csv.writer(outfh)
6
7      for row in reader:
8          row += ['new', 'columns']
9          writer.writerow(row)
```

inplace 函数是一个上下文管理器, 在示例中为同一个文件提供了两个句柄 (infh 与 outfh), 以便同时读写同一个文件。相比于标准库中的 fileinput.input 函数 (顺便说一句, 此函数也提供了一个上下文管理器), inplace 函数更好用。

如果您想研究 Martijn 编写的 inplace 源码 (在他写的文章“Easy in-place file rewriting”中给出了), 找到 yield 关键字, 在此之前的所有代码都用于设置上下文: 这包括先创建一个备份文件; 然后, 打开并产出 __enter__ 方法返回的可读文件句柄与可写文件句柄的引用。yield 关键字之后的代码相当于 __exit__ 方法, 负责关闭文件句柄, 并在出现问题时从备份中恢复文件。

关于 with 语句和上下文管理器的概述到此为止。下面我们通过一个完整的示例来了解 match/case 模式匹配。

18.3 案例分析:lis.py 中的模式匹配

“2.6.1 用模式匹配序列实现一个解释器<37页>”中以 Peter Norvig 的 lis.py 解释器 (移植到 Python 3.10) 为例, 分析了函数 evaluate 中用到的序列模式 (Sequence Pattern)。本节将更广泛地介绍 lis.py 的工

作原理,并探讨 evaluate 的所有 case 子句,不仅说明各个模式 (Pattern),还将分析解释器在各个 case 子句中都做了什么。

除了进一步讲解模式匹配 (Pattern Match) 之外,编写本节还有 3 个原因:

1. Norvig 的 list.py 是惯用 Python 代码的典范。
2. Scheme 的简洁性堪称语言设计的大师级作品。
3. 通过学习解释器的工作原理,加深了我对 Python 与一般编程语言 (解释型或编译型) 的理解。

在分析 Python 代码之前,先了解一下 Scheme 语言,为案例分析奠定基础——说不定您以前未听说过 Scheme 或 Lisp。

18.3.1 Schema 语法

在 Scheme 中,表达式与语句之间没有区别,这与 Python 一样。但是,Scheme 语言没有中缀运算符,所有表达式都用前缀表示法,例如 $(+ x 13)$ 而不是 $x + 13$ 。同样的前缀符号也用于函数调用,例如 $(gcd x 13)$ 。 $(define x 13)$ 是一种特殊形式,相当于 Python 中的赋值语句 $x = 13$ 。Scheme 与大多数 Lisp 方言使用的表达法被称为“S 表达式”⁴。

示例 18.10 是一段简单的 Scheme 示例。

</> 示例 18.10: 用 Scheme 计算最大公约数

```

1 (define (mod m n)
2   (- m (* n (quotient m n))))
3
4 (define (gcd m n)
5   (if (= n 0)
6       m
7       (gcd n (mod m n)))) # (递归) 尾调用
8
9 (display (gcd 18 45))

```

示例 18.10 展示了 3 个 Scheme 表达式,其中 2 个是函数定义,即 mod 与 gcd;还有一个 display 调用,将输出 9——即 $(gcd 18 45)$ 的结果。示例 18.11 用 Python 改写这段代码,代码简洁了不少。

</> 示例 18.11: 作用同示例 18.10,用 Python 编写

```

1 def mod(m, n):
2     return m - (m // n * n)
3
4 def gcd(m, n):
5     if n == 0:
6         return m
7     else:
8         return gcd(n, mod(m, n)) # (递归) 尾调用
9
10 print(gcd(18, 45))

```

⁴人们总是抱怨 Lisp 中有太多的括号,但是合理的缩进与优秀的编辑器基本上可以解决这个问题。主要的可读性问题是函数调用使用相同的 $(f \dots)$ 表示法,以及像 $(define \dots)$ 、 $(if \dots)$ 、 $(quote \dots)$ 这样的特殊形式,它们的行为完全不像函数调用。

在 Python 的惯用法中,应使用% 运算符而不是重新发明 mod;使用 while 循环(而不是递归)会更有效率。此处定义 2 个函数,是为了使示例尽可能相似,以帮助你阅读前面的 Scheme 代码。

Scheme 没有像 while 或 for 这样的迭代控制流命令。Scheme 中的迭代是通过递归完成的。请注意,在 Scheme 与 Python 的示例中都没有赋值。大量使用递归和尽少使用赋值是函数式编程的标志性特征。⁵

现在,一起回顾一下 Python 3.10 版本的 lis.py 代码。带有测试的完整源代码位于 GitHub 仓库 fluent-python/example-code-2e 的 18-with-match/lisp/Py3.10/ 目录中。

18.3.2 导入与类型

示例 18.12 展示了 lis.py 文件的前几行。Python 3.10 才能使用 TypeAlias 与类型联合运算符 |。

</> 示例 18.12: lis.py:文件前几行

```

1 import math
2 import operator as op
3 from collections import ChainMap
4 from itertools import chain
5 from typing import Any, TypeAlias, NoReturn
6
7 Symbol: TypeAlias = str
8 Atom: TypeAlias = float | int | Symbol
9 Expression: TypeAlias = Atom | list

```

此处定义了如下几个类型:

- Symbol

只是 str 的别名。在 lis.py 中,Symbol 用于表示标识符。它不具备字符串数据类型的操作,比如切片、分割等。⁶

- Atom

一种简单的语法元素。例如,数字或 Symbol,与由不同部分构成的复合结构(如列表)相反。

- Expression

Scheme 程序的构建块(基本单元),是由 Atom 和 list(可以嵌套)组成的表达式。

18.3.3 解析器

Norvig 的解析器只有 36 行代码(如示例 18.13 所示),展现了 Python 在处理 S 表达式这种简单递归语法时的强大功能。不过,lis.py 未考虑字符串数据、注释、宏等 Scheme 标准功能,以便增加复杂度。

</> 示例 18.13: lis.py:负责解析的函数

```

1 def parse(program: str) -> Expression:
2     "从字符串中读取Scheme表达式"

```

⁵为了使递归迭代实用且高效,Scheme 和其他函数式语言都实现了适当的尾调用。有关此内容的更多信息,请参阅“杂谈”。

⁶但是,Norvig 的第 2 版 lispy.py 支持字符串作为数据类型,还引入了诸如语法宏、延续和适当的尾调用等高级特性。尽管功能很强大,但 lispy.py 的代码量几乎是 lis.py 的 3 倍,并且更加难以理解。

```

3     return read_from_tokens(tokenize(program))
4 def tokenize(s: str) -> list[str]:      "将字符串转换为标记 (token) 列表"
5     return s.replace('(', ' ( ').replace(')', ') ').split()
6
7 def read_from_tokens(tokens: list[str]) -> Expression:
8     "从标记 (token) 序列中读取表达式"
9     # 排版需要, 省略部分解析代码

```

此部分的主要函数是 `parse`, 它读取一个字符串形式的 S 表达式, 并返回一个 `Expression` 对象。根据 [示例 18.12](#) 中的定义, `Expression` 可以是一个 `Atom`, 也可以是一个包含更多 `Atom` 或嵌套列表的 `list`。

Norvig 在 `tokenize` 中使用了一个聪明的技巧: 它在输入中的每个圆括号前后添加空格; 然后, 将其拆分, 得到一个以 ‘(’ 和 ‘)’ 作为独立标记的语法标记列表。这种快捷方式之所以有效, 是因为 `lis.py` 实现的 Scheme 子集中没有字符串类型; 因此, 每个 ‘(’ 或 ‘)’ 都是表达式的定界符。递归解析代码位于 `read_from_tokens` 函数中, 该函数包含 14 行代码。这里我们将跳过此函数, 重点关注解释器的其他部分。

以下是从 `18-with-match/lispy/py3.10/examples_test.py` 中提取的一些 doctests。

```

1  >>> from lis import parse
2  >>> parse('1.5')
3  1.5
4  >>> parse('ni!')
5  'ni!'
6  >>> parse('(gcd 18 45)')
7  ['gcd', 18, 45]
8  >>> parse('''
9  ... (define double
10 ...   (lambda (n
11 ...     (* n 2)))
12 ...   ''')
13 ['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]

```

这个 Scheme 子集的解析规则很简单:

1. 看起来像数字的标记被解析为 `float` 或 `int`。
2. 任何不是 ‘(’ 或 ‘)’ 的内容都会被解析为一个符号 (`Symbol`)——一个字符串, 用于表示标识符。这包括像 `+`、`set!` 和 `make-counter` 这样的源文本, 它们在 Scheme 中是有效的标识符, 但在 Python 中不是。
3. ‘(’ 与 ‘)’ 之间的表达式 (`Expression`) 被递归解析为包含 `Atom` 的列表或嵌套列表, 嵌套列表可能还包含 `Atom` 与更多嵌套列表。

使用 Python 解释器的术语, `parse` 的输出是一个 AST (抽象语法树): 这是 Scheme 程序的一种方便表示形式, 是由嵌套列表构成的树状结构, 其中最外层列表是树干, 内层列表是树枝, 而原子 (`Atom`) 则是树叶 (如图 18.1 所示)。

18.3.4 环境

如 [示例 18.14](#) 所示, `Environment` 类扩展了 `collections.ChainMap` 类, 并增加了 `change` 方法, 用于更新链式字典中的值, 这些字典由 `ChainMap` 实例以映射列表的形式保存在 `self.maps` 属性中。`change` 方法为 Scheme 中的 `(set!...)` 形式提供支持 (见 “[18.3.6.7 \(set!...\)<550页>](#)”)。

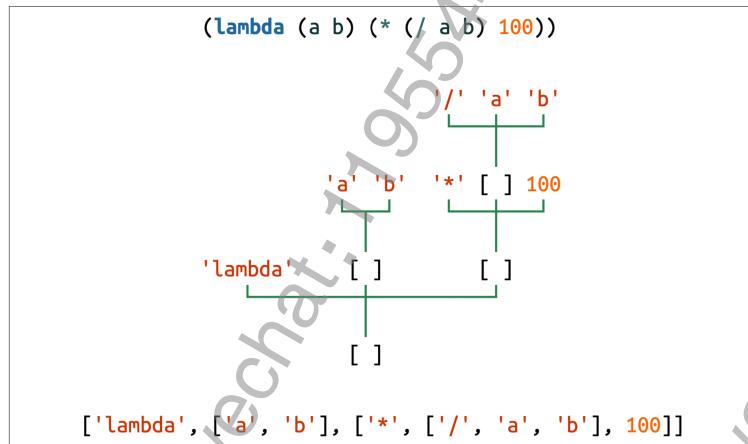


图 18.1: 一个以源码(具体语法)、树、Python 对象序列(抽象语法)表示的 Scheme lambda 表达式

</> 示例 18.14: lis.py:Environment 类

```

1 class Environment(ChainMap[Symbol, Any]):
2     "ChainMap 的子类, 允许就地更改映射中的项"
3
4     def change(self, key: Symbol, value: Any) -> None:
5         "找到 key 的定义位置, 并更新对应的值"
6         for map in self.maps:
7             if key in map:
8                 map[key] = value # type: ignore[index]
9                 return
10            raise KeyError(key)

```

注意, change 方法只更新现有的“键 (Key)”,⁷尝试更新未找到的“键 (Key)”会引发 KeyError 异常。

Environment 类的作用如以下 doctest 所示。

```

1 >>> from lis import Environment
2 >>> inner_env = {'a': 2}
3 >>> outer_env = {'a': 0, 'b': 1}
4 >>> env = Environment(inner_env, outer_env)
5 >>> env['a']         ❶
6 2
7 >>> env['a'] = 111   ❷
8 >>> env['c'] = 222
9 >>> env
10 Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 1})
11 >>> env.change('b', 333) ❸
12 >>> env
13 Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 333})

```

❶ 读取值时, Environment 会像 ChainMap 一样工作: 在嵌套映射中从左至右搜索“键 (Key)”。因此,

⁷之所以存在 “# type: ignore[index]” 这样的注解, 是因为我编写本章时, [typeshed issue #6042](#) 还未解决。ChainMap 被注解为 MutableMapping, 但是 maps 属性中的类型提示表明, maps 的值是一个 Mapping 列表。这间接导致 Mypy 将整个 ChainMap 视作不可变对象。

outer_env 中 a 的值被 inner_env 中的值遮盖了。

- ② 用 [] 进行赋值会覆盖或插入新项, 但始终是在第一个映射内进行操作, 例如本例中的 inner_env。
- ③ env.change('b', 333) 在 outer_env 中查找 "b" 键并为其就地赋值。

接下来是 standard_env() 函数, 它构建并返回一个加载了预定义函数的环境, 类似于始终可用的 Python 模块 `__builtins__` (如示例 18.15 所示)。

</> 示例 18.15: lis.py:standard_env() 构建并返回全局环境

```

1  def standard_env() -> Environment:
2      "An environment with some Scheme standard procedures."
3      env = Environment()
4      env.update(vars(math)) # sin, cos, sqrt, pi, ...
5      env.update({
6          '+': op.add,
7          '-': op.sub,
8          '*': op.mul,
9          '/': op.truediv,
10         # omitted here: more operator definitions
11         'abs': abs,
12         'append': lambda *args: list(chain(*args)),
13         'apply': lambda proc, args: proc(*args),
14         'begin': lambda *x: x[-1],
15         'car': lambda x: x[0],
16         'cdr': lambda x: x[1:],
17         # omitted here: more function definitions
18         'number?': lambda x: isinstance(x, (int, float)),
19         'procedure?': callable,
20         'round': round,
21         'symbol?': lambda x: isinstance(x, Symbol),
22     })
23     return env

```

概括地说, env 映射加载了以下内容:

- Python `math` 模块中的所有函数。
- Python `op` 模块中的部分运算符。
- 用 Python `lambda` 表达式构建的简单而强大的函数。
- Python 内置函数, 有些名称有变, 如 `callable` 变成了 `procedure`; 有些直接映射, 如 `round`。

18.3.5 REPL

Norvig 的 REPL (read-eval-print-loop, “读取-求值-输出-循环”) 易于理解, 但对用户并不友好 (如示例 18.16 所示)。如果未给 `lis.py` 提供命令行参数, 则函数 `repl()` 将由定义在模块尾部的 `main()` 来调用。在 “`lis.py>`” 提示符下, 必须输入正确且完整的表达式。如果忘记输入闭合 (右) 括号, `lis.py` 将会直接崩溃。⁸

⁸ 在研究 Norvig 的 `lis.py` 与 `lisp.py` 过程中, 我新做了 `mylis` 分支。该分支增加了一些功能, 包括一个可接受部分 S 表达式并提示继续的 REPL——类似于 Python 的 REPL 知道我们尚未完成并显示辅助提示符 (...), 直到我们输入一个可以求值的完整表达式或语句。`mylis` 还能优雅地处理一些错误, 不过还是容易崩溃, 远没有 Python 的 REPL 那样健壮。

```

</> 示例 18.16: lis.py:REPL 函数

1 def repl(prompt: str = 'lis.py> ') -> NoReturn:
2     "A prompt-read-eval-print loop."
3     global_env = Environment({}, standard_env())
4     while True:
5         ast = parse(input(prompt))
6         val = evaluate(ast, global_env)
7         if val is not None:
8             print(lispstr(val))
9
10    def lispstr(exp: object) -> str:
11        "Convert a Python object back into a Lisp-readable string."
12        if isinstance(exp, list):
13            return '(' + ' '.join(map(lispstr, exp)) + ')'
14        else:
15            return str(exp)

```

这 2 个函数的作用简述如下:

- repl(prompt: str = 'lis.py> ') -> NoReturn

调用 standard_env() 为全局环境提供内置函数;然后,进入无限循环,读取并解析输入的每一行,在全局环境中对其求解,并显示结果(除非结果为 None)。global_env 可能被 evaluate() 修改。例如,当用户定义一个新的全局变量或命名函数时,它将被存储在环境的第一个映射中,即 repl 第一行 Environment 构造函数调用的空 dict 中。

- lispstr(exp: object) -> str

与函数 parse 的作用相反。给定一个代表表达式的 Python 对象, lispstr 将返回该表达式的 Scheme 源码。例如,给定 “[+, 2, 3]”,结果为 “(+ 2 3)”。

18.3.6 求解函数

现在,我们可以欣赏 Norvig 实现的表达式求值器的美妙之处了——我用 match/case 语法重新实现的版本更加优美(如示例 18.17 所示)。示例 18.17 中的 evaluate 函数接受由 parse 构建的 Expression 和一个 Environment。

evaluate 函数的主体是一个单独的 match 语句,以表达式 exp 为匹配对象(Subject)。case 子句中的模式(Pattern)表达了 Scheme 的语法与语义,一目了然。

```

</> 示例 18.17: lispy/py3.10/lis.py:evaluate 函数接受一个表达式,并对其求值

```

```

1 KEYWORDS = ['quote', 'if', 'lambda', 'define', 'set!']
2
3 def evaluate(exp: Expression, env: Environment) -> Any:
4     "Evaluate an expression in an environment."
5     match exp:
6         case int(x) | float(x):
7             return x
8         case Symbol(var):

```

```

9     return env[var]
10    case ['quote', x]:           return x
11    case ['if', test, consequence, alternative]:
12        if evaluate(test, env):
13            return evaluate(consequence, env)
14        else:
15            return evaluate(alternative, env)
16    case ['lambda', [*parms], *body] if body:
17        return Procedure(parms, body, env)
18    case ['define', Symbol(name), value_exp]:
19        env[name] = evaluate(value_exp, env)
20    case ['define', [Symbol(name), *parms], *body] if body:
21        env[name] = Procedure(parms, body, env)
22    case ['set!', Symbol(name), value_exp]:
23        env.change(name, evaluate(value_exp, env))
24    case [func_exp, *args] if func_exp not in KEYWORDS:
25        proc = evaluate(func_exp, env)
26        values = [evaluate(arg, env) for arg in args]
27        return proc(*values)
28    case _:
29        raise SyntaxError(lispstr(exp))

```

接下来,依次分析每个 case 子句及其作用。有时,我会添加一些注释,注释内容为解析成 Python 列表时匹配模式的“S 表达式”(见“[subsubsection 18.3.6.3<547页>](#)”)。doctest 摘自 [02-array-seq/lispy/py3.10/examples_test.py](#),用于演示各个 case 子句。

18.3.6.1 求解数字

```

1 case int(x) | float(x):
2     return x

```

- **模式 (Pattern)**

int 或 float 实例。

- **操作结果**

原封不动地返回原值。

- **示例演示**

```

1 >>> from lis import parse, evaluate, standard_env
2 >>> evaluate(parse('1.5'), {})
3 1.5

```

18.3.6.2 求解符号

```

1 case Symbol(var):
2     return env[var]

```

- 模式 (Pattern)

Symbol 的实例,即用作标识符的 str。

- 操作结果

在 env 中查找 var,返回找到的值。

- 示例演示

```

1  >>> evaluate(parse('+'), standard_env())
2  <built-in function add>
3  >>> evaluate(parse('ni!'), standard_env())
4  Traceback (most recent call last):
5      ...
6  KeyError: 'ni!'

```

18.3.6.3 (quote...)

特殊形式 quote 将原子 (Atom) 与列表视作数据,而不是需要求解的表达式。

```

1  # (quote (99 bottles of beer))      # 将被解析为列表的 S 表达式
2  case ['quote', x]:
3      return x

```

- 模式 (Pattern)

以符号 'quote' 开始、后跟一个表达式 x 的列表。

- 操作结果

不求值,直接返回 x。

- 示例演示

```

1  >>> evaluate(parse('(quote no-such-name)'), standard_env())
2  'no-such-name'
3  >>> evaluate(parse('(quote (99 bottles of beer))'), standard_env()) ❶
4  [99, 'bottles', 'of', 'beer']
5  >>> evaluate(parse('(quote (/ 10 0))'), standard_env())           ❷
6  ['/ ', 10, 0]                                                       ❸

```

若表达式中没有保留关键字 quotes,则上述测试中的每个表达式都会引发异常。

❶ 将在环境中查找 no-such-name,从而引发 KeyError

❷ (99 bottles of beer) 无法求解,因为数字 99 不是命名特殊形式、运算符或函数的符号 (Symbol)。

❸ (/ 10 0) 将引发 ZeroDivisionError 异常。

为何编程语言有保留关键字

quotes 的作用虽然简单,但是不能实现为 Scheme 中的函数。它的特殊功能是阻止解释器对表达式 (quote (f 10)) 中的 (f 10) 进行求解:结果只是一个包含 Symbol 与 int 的列表。相反,在像 (abs (f 10)) 这样的函数调用中,解释器在调用 abs 之前,先计算 (f 10)。这是因为 quote 是一个必须以特殊形式处理的保留关键字。

一般来说,保留关键字的作用如下:

- 引入特殊的求解规则,如 quote 和 lambda——它们不求解任何子表达式。
- 改变控制流,如 if 与函数调用,它们也有特殊的求解规则。
- 管理环境,例如 define 和 set。

这也是 Python 和一般编程语言需要保留关键词的原因。想想 Python 的 def、if、yield、import、del 以及它们的作用。

18.3.6.4 (if...)

```

1  # (if (< x 0) 0 x)
2  case ['if', test, consequence, alternative]:
3      if evaluate(test, env):
4          return evaluate(consequence, env)
5      else:
6          return evaluate(alternative, env)

```

- **模式 (Pattern)**

以“if”开头的列表,后跟3个表达式: test、consequence、alternative。

- **操作结果**

求解 test 表达式:

- 为 True 时,求解 consequence,并返回求解结果。
- 为 False 时,求解 alternative,并返回求解结果。

- **示例演示**

```

1  >>> evaluate(parse('if (= 3 3) 1 0)'), standard_env())
2  1
3  >>> evaluate(parse('if (= 3 4) 1 0)'), standard_env())
4  0

```

consequence 与 alternative 分支必须是单个表达式。如果一个分支需要多个表达式,可通过(begin exp1 exp2 ...) 将它们组合起来。在 lis.py 中,begin 是一个函数,详见“[示例 18.15<544页>](#)”。

18.3.6.5 (lambda...)

Scheme 用 lambda 形式定义的匿名函数。它没有 Python lambda 表达式那么多的限制:在 Scheme 中,任何函数都可以用 (lambda...) 语法编写。

```

1  # (lambda (a b) (/ (+ a b) 2))
2  case ['lambda' [*parms], *body] if body:
3      return Procedure(parms, body, env)

```

- **模式 (Pattern)**

一个列表,以“lambda”开头,其后紧跟:

- 包含零个或多个参数名称的列表。

- body 中收集了一个或多个表达式, [卫 \(guard\) 语句](#) 确保了 body 不为空。

- **操作结果**

用参数名称、表达式列表 (作为函数主体) 以及当前环境, 创建并返回一个新 Procedure 实例。

- **示例演示**

```

1  >>> expr = '(lambda (a b) (* (/ a b) 100))'
2  >>> f = evaluate(parse(expr), standard_env())
3  >>> f # doctest: +ELLIPSIS
4  <lis.Procedure object at 0x...>
5  >>> f(15, 20)
6  75.0

```

Procedure 类实现了闭包的概念: 一个可调用对象, 其中包含参数名、函数体和函数定义环境的引用。稍后将分析 Procedure 类的代码。

18.3.6.6 (define...)

define 关键字有 2 种不同的语法形式。最简单的一种是:

```

1  # (define half (/ 1 2))
2  case ['define', Symbol(name), value_exp]:
3      env[name] = evaluate(value_exp, env)

```

- **模式 (Pattern)**

一个列表, 以 “define” 开头, 其后紧跟一个 Symbol 与一个表达式。

- **操作结果**

求解表达式, 以 name 为 “键 (Key)”, 并将表达式的结果放入 env 中。

- **示例演示**

```

1  >>> global_env = standard_env()
2  >>> evaluate(parse('(define answer (* 7 6))'), global_env)
3  >>> global_env['answer']
4  42

```

本例的 doctest 创建了一个 global_env, 以便于验证 evaluate 函数是否将 “answer” 放入了 Environment。

用这种简单的 define 形式可以创建变量, 或者以 (define...) 为 value_exp, 为匿名函数绑定名称。

标准 Scheme 为定义具名函数提供了简洁方式, 这就是第 2 种 define 形式:

```

1  # (define (average a b) (/ (+ a b) 2))
2  case ['define', [Symbol(name), *parms], *body] if body:
3      env[name] = Procedure(parms, body, env)

```

- **模式 (Pattern)**

一个列表, 以 “define” 开头, 其后紧跟:

- 一个列表, 以 Symbol(name) 开头, 后跟一个名为 params 的列表, 列表中包含 0 个或多个项。

- body 中收集了一个或多个表达式, “if body” 确保了 body 不为空。

- 操作结果

- 用参数名称、表达式列表 (作为函数主体) 以及当前环境, 创建并返回一个新 Procedure 实例。

- 以 name 为“键 (Key)”, 并将 Procedure 实例放入 env 中。

- 示例演示

示例 18.18 中的 doctest 定义了一个名为% 的函数 (用于计算百分比), 并将该函数添加到 global_env 中。

</> 示例 18.18: 定义一个名为% 的函数, 用于计算百分比

```

1 >>> global_env = standard_env()
2 >>> percent = '(define (% a b) (* (/ a b) 100))'
3 >>> evaluate(parse(percent), global_env)
4 >>> global_env['%'] # doctest: +ELLIPSIS
5 <lis.Procedure object at 0x...>
6 >>> global_env['%'](170, 200)
7 85.0

```

调用 evaluate 函数后, 检查% 是否绑定到了一个 Procedure 实例, 该实例接收 2 个数字参数, 并返回 1 个百分比。

匹配第 2 种 define 的模式 (Pattern), 不强制要求 parms 中的项都是 Symbol 实例。必须在构建 Procedure 实例之前检查这一点, 不过我没那么做——为了保持代码与 Norvig 的版本一样简单易懂。

18.3.6.7 (set!...)

set! 形式可更改先前定义的变量的值。⁹

```

1 # (set! n (+ n 1))
2 case ['set!', Symbol(name), value_exp]:
3     env.change(name, evaluate(value_exp, env))

```

- 模式 (Pattern)

一个列表, 以“set!” 开头, 后跟一个 Symbol 与一个表达式。

- 操作结果

用表达式的求解结果, 更新 env 中 name 的值。

Environment.change 方法从局部环境到全局环境依次遍历, 并用新值更新第一个找到的 name。若不是为了实现“set!” 关键字, 那么在这个解释器中, 使用 Environment 类型的地方, 都可以用 Python 的 ChainMap。

⁹许多编程教程都是一开始就教授如何赋值, 但在经典的 Scheme 书籍《Structure and Interpretation of Computer Programs, 2nd ed》(Abelson 等人著, 即人们常说的 SICP 或“魔法书”)中, 直到 220 页才出现 set!。函数式编程与命令式或面向对象编程不同, 不更改状态也可做很多事情。

Python 的 nonlocal 与 Scheme 的 set! 可解决同样的问题

set! 与 Python 中的 nonlocal 关键字作用类似: 声明 nonlocal x 允许 $x = 10$ 更新之前在局部作用域之外定义的 x 变量。如果没有声明 nonlocal x, 则 $x = 10$ 将始终在 Python 中创建一个局部变量(详见“9.7 nonlocal 声明<256页>”。

类似地, (set! x 10) 更新可能在函数局部环境之外定义的 x。相比之下, (define x 10) 中的变量 x 始终是一个局部变量, 在局部环境中创建或更新。

nonlocal 与 (set! ...) 都是用来更新闭包内变量中保存的程序状态。“示例 9.13<257页>”演示了如何用 nonlocal 来实现计算累计平均值的函数, 并将 count 与 total 保存在闭包中。下面用 lis.py 定义的 Scheme 子集实现同样的功能。

```

1 (define (make-averager)
2   (define count 0)
3   (define total 0)
4   (lambda (new-value)
5     (set! count (+ count 1))
6     (set! total (+ total new-value)))
7     (/ total count)
8   )
9 )
10 (define avg (make-averager)) ①
11 (avg 10) ②
12 (avg 11) ③
13 (avg 15) ④

```

- 用 lambda 定义的内部函数, 创建一个闭包, 将变量 count 与 total 的值初始化为 0。将得到的闭包绑定到 avg。
- 返回 10.0。
- 返回 10.5。
- 返回 12.0。

前面的代码是 lispy/py3.10/examples_test.py 中的一个测试。

接下来, 分析函数调用。

18.3.6.8 函数调用

```

1 # (gcd (* 2 105) 84)
2 case [func_exp, *args] if func_exp not in KEYWORDS:
3   proc = evaluate(func_exp, env)
4   values = [evaluate(arg, env) for arg in args]
5   return proc(*values)

```

• 模式 (Pattern)

一个包含 1 项或多项的列表。

卫 (guard) 语句确保 func_exp 不在 ['quote', 'if', 'define', 'lambda', 'set!'] 中——“示例 18.17<545页

>”中,位于 evaluate 函数之前。

此模式可匹配任何包含一个或多个表达式的列表,将第一个表达式绑定到 func_exp,余下的表达式将作为列表(可能为空)绑定到 args。

- 操作结果

- 求解 func_exp,以获得函数 proc。
- 求解 args 中的各项,以构建参数值列表。
- 用参数值列表中的不同参数调用 proc,并返回结果。

- 示例演示

```
1 >>> evaluate(parse('(% (* 12 14) (- 500 100))'), global_env)
2 42.0
```

此 doctest 接续“[示例 18.18<550页>](#)”:假定 global_env 有一个名为%的函数。传给%函数的是算术表达式,以强调在调用函数之前已对参数进行了求解。

在此 case 子句中,需要卫(guard)语句,因为 [func_exp, *args] 可以匹配任何包含一项或多项的序列。但是,如果 func_exp 是某个关键字,而且匹配对象与之前的任何 case 子句都不匹配,则说明这确实是一个语法错误。

18.3.6.9 捕获语法错误

如果匹配对象 exp 与之前的任何 case 子句都不匹配,则兜底的 case 子句将引发 SyntaxError 异常。

```
1 case _:
2     raise SyntaxError(lispstr(exp))
```

如下所示,在匹配格式错误的 (lambda ...) 时,将引发 SyntaxError 异常。

```
1 >>> evaluate(parse('(lambda is not like this)'), standard_env())
2 Traceback (most recent call last):
3 ...
4 SyntaxError: (lambda is not like this)
```

如果匹配函数调用的 case 子句中不包含拒绝关键字的卫(guard)语句,则 (lambda is not like this) 表达式会被当作函数调用处理,将引发 KeyError 异常。因为,‘lambda’不在环境中——就像 lambda 不是 Python 内置函数一样。

18.3.7 实现闭包的 Procedure 类

Procedure 类完全可以命名为 Closure,因为它代表的就是闭包(closures):一个函数定义与一个环境的结合体。函数定义包括参数名称和构成函数体的表达式。当函数被调用时,环境被用来为自由变量(Free Variable)提供值。自由变量(Free Variable)指函数体中出现的除了参数、局部变量或全局变量以外的变量。闭包(closures)与自由变量(Free Variable)的概念,详见“[9.6 闭包<254页>](#)”。

我们已经学过在 Python 中如何使用闭包,下面将深入研究 lis.py 是如何实现闭包的。

```
1 class Procedure:
```

```

2     "A user-defined Scheme procedure."
3     def __init__(①
4         self, parms: list[Symbol], body: list[Expression], env: Environment
5     ):
6         self.parms = parms
7         self.body = body
8         self.env = env
9
10    def __call__(self, *args: Expression) -> Any: ②
11        local_env = dict(zip(self.parms, args)) ③
12        env = Environment(local_env, self.env) ④
13        for exp in self.body: ⑤
14            result = evaluate(exp, env) ⑥
15        return result ⑦

```

- ① 当用 lambda 或 define 关键字定义函数时,会调用此构造方法。
- ② 保存函数名称、body 表达式、环境,以供后续使用。
- ③ 由 case [func_exp, *args] 子句中,最后一行内的 proc(*values) 调用(见“示例 18.17<545页>”)。
- ④ 以 self.parms 内容为局部变量的名称,以 args 内容为变量值,构建 local_env 映射。
- ⑤ 构建一个新的组合 env,将 local_env 放在前面, self.env(定义函数时保存的环境)放在后面。
- ⑥ 遍历 self.body 中的每个表达式,在组合的 env 中对表达式进行求解。
- ⑦ 返回最后一个表达式的求解结果。

在 lis.py 的 evaluate 后面有几个简单的函数: run 读取并执行完整的 Scheme 程序, main 根据命令行参数调用 run 或 repl——类似于 Python 的做法。我不会详述这 2 个函数,因为它们并没什么新知识点。我的目的是与您分享 Norvig 这个小型解释器的精妙之处,让大家更深入地了解闭包(closures)的工作原理,并展示 match/case 如何成为 Python 的功能补充。

本节介绍了许多模式匹配(Pattern Match)的内容。最后,要正式确立 OR 模式的概念。

18.3.8 使用 OR 模式

用符号“|”分隔的模式(Pattern),称为“OR 模式”:若其中任一个模式(Pattern)匹配成功,则该模式就匹配成功。“18.3.6.1 求解数字<546页>”中的模式就是一个 OR 模式(如下所示):

```

1     case int(x) | float(x):
2         return x

```

OR 模式中的所有子模式必须使用相同的变量。这个限制是必要的,以确保无论匹配哪一个子模式,变量在卫(guard)语句表达式和 case 主体中都可用。



在 case 子句的上下文中,运算符“|”有特殊含义。它不会触发特殊方法 `__or__`,该方法用于处理其他上下文中类似“a | b”的表达式。在其他上下文中,运算符“|”可能根据操作数的不同,被重载为执行集合并集或整数位或等操作。

OR 模式不限于出现在模式(Pattern)的顶层。也可以在子模式中使用运算符“|”。例如,如果希望 liy.py

接受希腊字母 λ ¹⁰ 与 lambda 关键字, 这可以这样重写模式 (Pattern) :

```

1 # (λ (a b) (/ (+ a b) 2) )
2 case ['lambda' | 'λ', [*parms], *body] if body:
3     return Procedure(parms, body, env)

```

接下来, 进入本章第 3 个, 也是最后一个主题: Python 中的 else 子句可出现在哪些不寻常的位置。

18.4 先这样, 再那样: if 之外的 else 块

这并不是什么秘密, 但它是一个未被充分重视的语言特性: else 子句不仅可以在 if 语句中使用, 还可以在 for、while 和 try 语句中使用。

for/else、while/else 和 try/else 的语义密切相关, 不过与 if/else 差异很大。起初, else 一词甚至阻碍了我对这些特性的理解, 但最终还是习惯了。

else 子句的规则如下:

- for

仅当 for 循环运行完成时, else 代码块才会运行 (也就是说, 如果 for 被 break 中断, 则 else 代码块不会运行)。

- while

仅当 while 循环因条件为 False 而退出时, else 代码块才会运行 (也就是说, 如果 while 被 break 中断, 则 else 代码块不会运行)。

- try

仅当 try 代码块中未出现异常时, else 代码块才会运行。官方文档还指出: “else 子句中的异常不会被前面的 except 子句处理。”

在所有情况下, 如果异常或 return、break 或 continue 语句导致控制权跳出了复合语句的主代码块, else 子句也会被跳过。



我认为除了 if 之外的其他语句中, 使用 else 关键字是个非常糟糕的选择。关键字 “else” 蕴含着“排他性”的意思。例如, “要么运行这个循环, 否则做那个操作。” 但循环中 else 的语义恰恰相反: “运行这个循环, 然后做那个操作。” 这表明, 用 then 作为关键字或许更好。并且, then 用在 try 上下文中也说得通: “尝试运行这个, 然后做那个操作。” 但是, 添加新关键字属于语言的重大变更, 这不是一个容易做出的决定。

在这些语句中使用 else 通常会使代码更易于阅读, 并省去设置控制标志或编写额外 if 语句的麻烦。

在循环中使用 else 时, 通常会遵循如下代码片段的模式:

```

1 for item in my_list:
2     if item.flavor == 'banana':
3         break

```

¹⁰ (U+03BB) 的官方 Unicode 名称是“GREEK SMALL LETTER LAMDA”。这不是拼写错误: 在 Unicode 数据库中, 该字符被命名为“lambda”, 没有字母 “b”。根据维基百科词条“Lambda”所述, Unicode 联盟采用这种拼写是因为“preferences expressed by the Greek National Body (希腊国家机构表达的偏好)”。

```
4     else:  
5         raise ValueError('No banana flavor found!')
```

在 try/except 块中,else 一开始可能显得多余。毕竟,只有当 `dangerous_call()` 没有引发异常时,以下代码片段中的 `after_call()` 才会运行,对吗?

```
1 try:  
2     dangerous_call()  
3     after_call()  
4 except OSError:  
5     log('OSError...')
```

然而,这样做会毫无理由地将 `after_call()` 放在 try 代码块中。为了清晰与正确起见,try 代码块的主体应只包含可能产生预期异常的语句。因此,像下面这样编码更好:

```
1 try:  
2     dangerous_call()  
3 except OSError:  
4     log('OSError...')  
5 else:           # 只有try块未引发异常,才会执行 alter\call  
6     after_call()
```

这样编写很明确,try 块防守的是 `dangerous_call()` 可能出现的错误,而不是 `after_call()` 中可能出现的错误。同样明确的是,只有 try 块未引发异常,`after_call()` 才会执行。

在 Python 中,try/except 不仅用于处理异常,还常用于控制流程。在 [Python 官方术语表](#) 中甚至还定义了这样一个缩略词/口号:

EAAP 原则

“Easier to Ask for Forgiveness than Permission.”(请求原谅比请求许可更容易)。这是一种常见的 Python 编程风格,先假定存在有效的键或属性,并在假设错误时捕获异常。这种风格简洁、快速,特点是在代码中存在大量的 try/except 语句。此种技术与许多其他语言(如 C 语言)中常见的 LBYL 风格形成鲜明对比。

然后,[Python 术语表](#)对 LBYL 的定义如下:

LBYL

三思而后行(Look-before you leap)。这种编程风格在调用函数或查找属性或键之前,先显式测试前提条件。与 [EAAP 原则](#)相比,这种风格的特点是代码中存在大量的 if 语句。在多线程环境中,LBYL 风格可能会在“检查”与“行动”之间引入条件竞争。例如,对“`if key in mapping: return mapping[key]`”这段代码来说,如果在测试之后(查找之前),另一个线程从映射中删除了 key,那么这段代码就会失败。此问题可以用锁或者 [EAAP 原则](#)风格解决。

如果使用 EAAP 风格,就要深入了解 else 子句,并要在 try/except 语句中合理使用 else 子句。



在讨论 match 语句时,有些人(包括我)认为它也应该有一个 else 子句。最后决定不需要 else 子句,是因为 case _: 也能完成同样的工作。^a

^a通过观看 python-dev 邮件列表中的讨论,我认为 else 被否决的原因之一是:对如何在 match 中缩进缺乏共识“else 应与 match 缩进在同一层次,还是与 case 缩进在同一层次?”

下面,对本章内容做一个总结。

18.5 本章小结

本章首先介绍了上下文管理器与 with 语句的作用。很快我们就知道,with 语句除了可自动关闭打开的文件之外,还有许多其他用途。我们还实现了一个自定义上下文管理器:带有 `__enter__`/`__exit__` 方法的 `LookingGlass` 类,并了解了说和在 `__exit__` 方法中处理异常。Raymond Hettinger 在 PyCon US 2013 主题演讲中,传达了一个重要观点:with 不仅可用于资源管理,它还是一种工具——可用于将常见的设置和清理代码,或者任何需要在另一个过程之前和之后执行的一对操作,提取出来。¹

随后,回顾了标准库中 `contextlib` 模块提供的函数。其中,`@contextmanager` 装饰器 可用一个包含 `yield` 语句的简单生成器来实现上下文管理器——这比编写至少包含 2 个方法的类更为简洁。之后,用生成器函数 `looking_glass` 重新实现了 `LookingGlass` 类的功能,并讨论了在使用 `@contextmanager` 时如何处理异常。

接下来,研究了 Peter Norvig 编写的 `lis.py`,这是一个用 Python 编写的 Scheme 解释器,我用 match/case 重构了 `evaluate` 函数——此函数是解释器的核心。若要理解 `evaluate` 工作原理,需要稍微了解 Scheme 语法、S 表达式解析器、一个简单的 REPL,以及如何通过 `collection.ChainMap` 的子类 `Environment` 构建嵌套作用域。最后,透过 `lis.py` 不仅可进一步探讨模式匹配的运行逻辑,还能学到更多其他知识。例如, `lis.py` 展示了解释器的不同部分如何协同工作,借此对 Python 自身的核心特性也有了深入的理解:为什么要有保留关键字、作用域规则的工作原理、如何构建和使用闭包。

18.6 延伸阅读

《The Python Language Reference》的“8. Compound statements”全面介绍了 `if`、`for`、`while` 与 `try` 语句中 `else` 子句的作用。关于 `try/except` 语句(不管有没有 `else` 子句)是否符合 Python 风格,Raymond Hettinger 在 Stack Overflow 的“Is it a good practice to use try-except-else in Python?”中,给出了精彩的回答。《Python in a Nutshell, 3rd Ed》(Martelli 等)中,有一章内容是关于异常的,那一章对 EAPP 原则风格进行了精彩讨论,并赞扬了计算机先驱 Grace Hopper 创造了这句话“请求原谅比请求许可更容易”。

《The Python Standard Library》的“Built-in Types”有一节专门介绍“上下文管理器类型”。《The Python Language Reference》的“3.3.9. With Statement Context Managers”一节,专门介绍了特殊方法 `__enter__`/`__exit__`。上下文管理器是在“PEP 343 -The “with” Statement”中引入的。

Raymond Hettinger 在 PyCon US 2013 主题演讲中强调:“with 语句是一种成功的语言特性”。在该次会议上,他还在题为“Transforming Code into Beautiful, Idiomatic Python”的演讲中,展示了上下文管理器的一些有趣应用。

¹¹“Python is Awesome”,第 21 张幻灯片。

Jeff Presning 的博文 “[The Python with Statement by Example](#)” 以 pycairo 图形库为例,列举了一些关于上下文管理器的有趣示例。

Nikolaus Rath 基于他最初的想法,构建了 `contextlib.ExitStack` 类。Nikolaus Rath 在 “[On the Beauty of Python's ExitStack](#)” 一文中,解释了需要 `contextlib.ExitStack` 类 的原因。在该文中,Rath 指出: `ExitStack` 与 Go 中的 `defer` 语句(我认为这是 Go 最优秀的想法之一)作用类似,但更灵活。

《[Python Cookbook, 3rd Ed](#)》(Beazley 与 Jones)一书中,创建了一些功能独特的上下文管理器。“8.3. Making Objects Support the ContextManagement Protocol”一节,实现了一个 `LazyConnection` 类,其实例是一个上下文管理器——在 `with` 块中可自动打开和关闭网络连接。“9.22. Defining Context Managers the Easy Way”一节,实现了一个用于统计代码运行时间的上下文管理器。该节还实现了一个用事务修改 `list` 对象的上下文管理器:在 `with` 块中创建 `list` 实例的副本,所有改动都应用于副本,仅当 `with` 块在未引发异常的情况下完成时,才用副本取代原始列表。这样做,既简单又巧妙。

Peter Norvig 在他的博文《[\(How to Write a \(Lisp\) Interpreter \(in Python\)\)](#)》和《[\(An \(\(Even Better\) Lisp\) Interpreter \(in Python\)\)](#)》中分析了他实现的小型 Scheme 解释器。`lis.py` 和 `lisp.py` 的代码位于 `norvig/pytudes` 存储库中。我的存储库 `fluentpython/lisp` 是对 `lis.py` 的 `mylis` 分支,已更新到 Python 3.10,具有完善的 REPL、命令行集成、示例、测试,还包括了学习 Scheme 的更多参考资料。目前, `Racket` 是最好的 Scheme 方言,也是学习和实验的最佳环境。

杂谈

分离面包

Raymond Hettinger 在 PyCon US 2013 的演讲 “[What Makes Python Awesome](#)” 中指出,当他首次看到关于 `with` 语句的提案时,觉得“有些晦涩难懂”。这与我最初的反应类似。PEP 通常都难以理解,PEP343 尤甚。

然后, Hettinger 告诉我们,他认识到:子程序是计算机语言史上最重要的发明。如果您有 `A;B;C`、`P;B;Q` 这样的操作序列,您可以用子程序将 `B` 分离出来。这就像分离三明治的馅:用金枪鱼馅搭配不同的面包。但是,如果您想将面包分离出来,用小麦面包制作三明治,每次使用不同的馅,该怎么办呢?这就是 `with` 语句提供的功能,它是对子程序的补充。Hettinger 接着说道:

with 语句是个非常重要的概念。我建议大家在实践中深挖这个功能的用途。用 with 语句或许能做一些意义深远的事情。with 语句的最佳用途尚未被发掘。我预计,如果您能充分利用它,它可能会被复制到其他语言或者未来的语言中。或许,您正在参与的事情几乎与子程序的发明一样意义深远。

Hettinger 承认他过分夸大了 `with` 语句的作用。尽管如此, `with` 语句仍是一个非常有用的功能。他用三明治做类比,来解释 `with` 语句是对子程序的补充。那一刻,我脑海中浮现了许多可能性。

如果您想让别人相信 Python 非常棒,则一定要看看 Hettinger 的主题演讲。关于上下文管理器的部分是从 23:00 到 26:15。但整个主题演讲都非常精彩。

真尾调用实现高效递归

标准的 Scheme 实现需要提供“真尾调用(`proper tail calls, PTC`)”,通过递归迭代替代命令式语言中的 `while` 循环。一些作者将 `PTC` 称作“尾部调用优化(`tail call optimization, TCO`)”。而对于另

一些作者来说, TCO 有其他含义。更多详情, 参见维基百科上的 “[Tail call](#)” 与 “[Tail call optimization in ECMAScript 6](#)”。

“尾调用”是指函数返回一个函数(可能是同一个函数, 也可能不是)调用的结果。“[示例 18.10<540页>](#)”与“[示例 18.11<540页>](#)”的 gcd 函数在 if 语句的 False 分支中进行了(递归)尾调用。

但是, 下面的 factorial 函数未做尾调用。

```
1 def factorial(n):
2     if n < 2:
3         return 1
4     return n * factorial(n - 1) ❶
```

最后一行(❶处)对 factorial 的调用不是尾调用, 因为返回值不是递归调用的结果——而是将调用结果与 n 相乘之后, 才返回。

如下代码, 是一个使用尾调用的替代方案, 因此是“尾递归 (tail recursive)”:

```
1 def factorial_tc(n, product=1):
2     if n < 1:
3         return product
4     return factorial_tc(n - 1, product * n)
```

Python 不支持真尾调用(PTC), 因此编写尾递归函数没什么益处。就上述 2 个版本的 factorial 函数而言, 我认为第 1 版更简洁易读。而在实际使用中, 别忘了 Python 有现成可用的 `math.factorial`, 它是用 C 语言编写的, 没有递归。关键是, 即便在实现了真尾调用(PTC)的语言中, 也不是所有递归函数都可从真尾调用(PTC)中受益, 只有那些精心编写以进行真尾调用(PTC)的函数才能从中受益。

对于支持真尾调用(PTC)的语言, 当解释器遇到真尾调用(PTC)时, 会跳转到被调用函数的主体, 而不会创建新的堆栈帧, 从而节省内存。有些编译型语言也实现了真尾调用(PTC), 有时是一种可以禁用的优化措施。

对于那些起初就不是为了函数式编程设计的语言(如 Python 或 JavaScript)中, 对于真尾调用(PTC)的定义或带来的价值并没有达成普遍共识。在函数式语言中, 真尾调用(PTC)只是一种预期特性, 而不仅仅是一种锦上添花的优化措施。如果一种语言除了递归之外, 没有其他迭代机制, 那么对于实际使用, 真尾调用(PTC)是必需的。Norvig 的 `lis.py` 并未实现真尾调用(PTC), 但更复杂的 `lispy.py` 解释器实现了真尾调用(PTC)。

Python 与 JavaScript 反对真尾调用(PTC)的理由

CPython 没有实现真尾调用(PTC), 而且可能永远都不会实现。Guido van Rossum 在“[text](#)”一文中解释了不支持真尾调用(PTC)的原因。如下是该文的关键段落, 概括了文章主旨:

我个人认为, 对于某些语言来说, 真尾调用(PTC)是一个很好的特性, 但我认为它不适合 Python: 仅消除部分调用的堆栈跟踪, 而不消除其他调用的堆栈跟踪, 肯定会让许多用户感到困惑。因为这些用户未受到真尾调用(PTC)信仰的熏陶, 而是通过在调试器中跟踪一些调用了解调用语义。

2015 年, 真尾调用(PTC)被纳入 JavaScript 的 ECMAScript 6 标准中。截至 2021 年 10 月, WebKit

中的 JavaScript 解释器已支持了真尾调用 (PTC)。WebKit 是 Safari 浏览器使用的引擎。其他主要浏览器中的 JavaScript 解释器均未支持真尾调用 (PTC)。依赖于 V8 (Google 为 Chrome 开发的) 引擎的 Node.js 也不支持真尾调用 (PTC)。根据 “[ECMAScript 6 compatibility table](#)”, 针对 JS (例如 TypeScript、ClojureScript 和 Babel) 的转译器 (Transpiler) 和填充程序 (Polyfill) 也不支持真尾调用 (PTC)。

拒绝支持真尾调用 (PTC) 的原因很多, 但多数都与 Guido van Rossum 所言类似: 真尾调用 (PTC) 增加了代码调试难度, 只有少数坚持使用递归进行迭代的人能从中受益。详情请参阅 Graham Marlow 撰写的 “[What happened to proper tail calls in JavaScript?](#)”。

某些时候, 递归是最佳解决方案, 即使在不支持真尾调用 (PTC) 的 Python 中也是如此。早前, Guido 在关于此话题的一篇文章中写道:

...典型的 Python 实现最多允许 1000 次递归, 这对于未使用递归的代码与依赖遍历的代码 (如, 典型的解析树) 来说已经足够了。但对于用递归编写的大列表循环来说还不够。

我认同 Guido 与大多数 JS 实现者的观点: 真尾调用 (PTC) 并不适合 Python 或 JavaScript。与受限的 lambda 语法相比, 缺少真尾调用 (PTC) 是阻碍以函数式风格编写 Python 程序的主要限制因素。

如果想了解真尾调用 (PTC) 如何在一个比 Norvig 的 lispy.py 功能更少 (代码更少) 的解释器中运行, 请查看 [mylis_2/lis.py](#)。与真尾调用 (PTC) 相关的代码位于 evaluate 函数中那个无线循环内匹配函数调用的 case 子句中 (见示例 18.19 的❶处), 同时满足 2 个条件时, 开始真尾调用 (PTC): 解释器跳转到下一个 Procedure 的主体, 而不会在尾部调用时递归调用 evaluate。这些小型解释器展示了抽象的强大: 尽管 Python 没有实现真尾调用 (PTC), 但用 Python 编写一个实现真尾调用 (PTC) 的解释器是可行的, 而且不是很难。我的实现方式借鉴了 Peter Norvig 的代码, 感谢教授分享!

</> 示例 18.19: [mylis_2/lis.py](#): 实现真尾调用 (PTC)

```

1 TCO_ENABLED = True
2 # ... 排版需要, 省略多行代码
3
4 def evaluate(exp: Expression, env: Environment) -> Any:
5     "Evaluate an expression in an environment."
6     while True:
7         match exp:
8             case int(x) | float(x):
9                 return x
10            # ... 排版需要, 省略多行代码
11            case [op, *args] if op not in KEYWORDS:           # (proc exp*)
12                proc = evaluate(op, env)
13                values = [evaluate(arg, env) for arg in args]
14                if TCO_ENABLED and isinstance(proc, Procedure): ❶ 实现真尾调用
(PTC)
15                    exp = ['begin', *proc.body]
16                    env = proc.application_env(values)
17                else:
18                    # ... 排版需要, 省略多行代码
19            case _:

```

20

```
raise InvalidSyntax(lispstr(exp))
```

Norvig 对 evaluate() 中模式匹配的看法

我与 Peter Norvig 分享了 Python 3.10 版本的 lis.py 代码。他很喜欢这个用模式匹配实现的版本, 不过提出了一个不同的解决方案: 他未采用我编写的 卫 (guard) 语句, 而是为每个关键字都设置一个 case 子句, 并在 case 块中执行测试, 以提供更明确的 SyntaxError 消息——例如, 指明 body 为空。如此一来, “case [func_exp, *args] if func_exp not in KEYWORDS:” 中的 卫 (guard) 语句 (见示例 18.20 的❶处) 就没有存在的必要了。因为每个关键字都会在函数调用的 case 子句之前得到处理。

</> 示例 18.20: lispy/py3.10/lis.py; 摘自 “示例 18.17<545页>”

```

1 KEYWORDS = ['quote', 'if', 'lambda', 'define', 'set!']

2

3 def evaluate(exp: Expression, env: Environment) -> Any:
4     "Evaluate an expression in an environment."
5     match exp:
6         case int(x) | float(x):
7             return x
8             # ... 排版需要, 省略多行代码
9         case [func_exp, *args] if func_exp not in KEYWORDS: ❶
10            proc = evaluate(func_exp, env)
11            values = [evaluate(arg, env) for arg in args]
12            return proc(*values)
13
14         case _:
15             raise SyntaxError(lispstr(exp))

```

当为 mylis 添加新功能时, 我可能会遵循 Norvig 的建议。不过, “示例 18.17<545页>” 中 evaluate 的组织方式更多是为了本章的教学: 需要与 “示例 2.11<38页>” 中的 if/elif/... 块保持对应, 借助 case 子句讲解模式匹配 (Pattern Match) 的更多特性, 并且表明用模式匹配 (Pattern Match) 编写的代码也更简洁。

Python 中的并发模型

并发 (Concurrency) 是指多个任务在同一时间段, 交替运行。通常通过利用时间片轮转或事件驱动等方式来实现。

并行 (Parallelism) 是指多个任务在同一时刻, 真正同时运行。通常通过多线程、多进程或分布式计算等方式来实现。

二者不同, 但有联系。

并发是关于结构, 并行是关于执行。

并发提供了一种结构化的解决方案, 以解决可能(但不一定)并行的问题。

——Rob Pike, Go 语言联合创始人^a

^a摘自 “Concurrency is not parallelism” 演讲的第 8 张幻灯片。

本章将讲述如何让 Python “同时处理多件事”。这可能涉及并发或并行编程——即使是热衷于术语的学者, 也会在如何使用这些术语上产生分歧。本章开篇引用了 Rob Pike 关于并发与并行的非正式定义。但是请注意, 我发现某些论文与图书所声称的并行计算, 实际上主要是并发。¹

在 Pike 看来, 并行 (Parallelism) 是并发 (Concurrency) 的一种特例。所有并行系统都是并发的, 但并非所有并发系统都是并行的。在 21 世纪初, 我们用单核设备在 GNU Linux 上并发处理 100 个进程。在正常情况下, 一台配备 4 个 CPU 核心的现代笔记本电脑, 随随便便都会运行 200 多个进程。如果要并行运行 200 个任务, 就需要 200 个 CPU 核心。因此, 实际上大多数计算都是并发 (Concurrency) 的, 而不是并行 (Parallelism) 的。操作系统管理着数百个进程, 确保每个进程都有机会取得进展, 即使 CPU 本身同时做的事情不能超过 4 件。

本章假定您事先不了解并发或并行编程。在简单的概念介绍之后, 我们将通过学习简单的示例, 来介绍与比较 Python 并发编程的 3 个核心包:`threading`、`multiprocessing`、`asyncio`。

本章的最后 30% 内容概述了可以增强 Python 应用性能和可伸缩性的第三方工具、库、应用服务器和分布式任务队列。这些都是重要的话题, 但超出了本书的范围——专注于 Python 语言核心特性。即便如此, 在

¹我曾与 Imre Simon 教授一起学习与工作, 他说科学界有两大过错: 用不同的词表示同一事物, 以及用同一词表示不同事务。Imre Simon (1943-2009) 是巴西计算机科学的先驱, 他对自动机理论 (Automata Theory) 做出了开创性贡献, 并开创了热带数学 (Tropical Mathematics) 领域。他还是自由软件与自由文化的倡导者。

书第2版中,我认为有必要介绍这些主题。因为Python之所以适用于并发与并行计算,除了标准库提供了相关支持以外,还有第三方库的功劳。这就是为什么YouTube、DropBox、Instagram、Reddit等公司在初期都使用Python作为主要语言实现可伸缩Web的原因,尽管有人一直声称“Python不具备伸缩性”。

19.1 本章新增内容

本章是第2版新增的。“19.4 并发的HelloWorld<565页>”中的spinner示例之前位于有关[asyncio](#)的章节中。本版中对那些示例做了些改进,并首次展示了Python的3种并发方法:线程、进程、原生协程。

只有少数几个段落与第1版介绍[concurrent.futures](#)与[asyncio](#)相同,余下内容都是全新的。

“19.7 多核世界的Python<583页>”与本书其他部分不同,没有代码示例。其目的是提及一些值得研究的重要工具,以利用标准库之外的工具实现高性能的并发与并行。

19.2 全局概览

导致并发编程困难的因素有很多,但我想谈谈最基本的因素:启动线程或进程很容易,但如何跟踪它们呢?

2

当调用函数时,调用代码会被阻塞,直到函数返回。这样你就知道函数何时完成,并能轻松获取函数的返回值。如果函数引发异常,则可将函数调用放在try/except块中,以捕获异常。

这些熟悉的功能在您启动线程或进程后,就都不能用了:您无法轻松地知道线程或进程何时完成,若想获取结果或捕获异常,则需要设置某种通信信道,如消息队列。

此外,启动线程或进程的成本并不低,所以你不希望启动一个线程或进程只进行一次计算就退出。通常,应该将每个线程或进程变成一个“worker”,循环等待要处理的输入,以此来分摊启动成本。但是,这将进一步增加通信难度,还会引入更多问题。当不再需要worker时,如何让其自动退出?怎样退出才能不中断作业,避免留下未处理完毕的数据与未释放的资源(如已打开的文件)呢?通常,解决这些问题要涉及消息与队列。

协程的启动成本很低。如果用await关键字启动协程,很容易获得协程的返回值。也可以安全地取消已启动的协程,而且捕获异常的位置也很明确。但是,协程通常由异步框架启动,因此协程的监控难度与线程或进程差不多。

最后,正如本章后文所述,Python协程与线程不适合CPU密集型任务。

鉴于以上原因,并发编程需要学习一些新的术语和编码模式。首先,对核心术语确立统一认识。

19.3 术语定义

以下是我在这章余下部分和接下来的2章中将使用的一些术语:

- **并发(Concurrency)**

并发(Concurrency)是指处理多个待处理任务的能力,逐个或并行地进行处理,以便最终每个任务都成功或失败。在单核CPU上实现并发性的关键是运行一个操作系统调度程序,交替执行待处理任务。并发也称为多任务处理。

²本节内容是我朋友BruceEckel提议编写的,他是Kotlin、Scala、Java和C++相关书籍的作者。

- **并发 (Concurrency)**

并行 (Parallelism) 则是指同时执行多个计算任务的能力, 需要多核 CPU、多个 CPU、GPU 或者集群中多台计算机的支持。与并发不同的是, 并行是在同一时刻处理多个计算任务, 而不是依次逐个处理单个任务。

- **执行单元 (Execution Unit)**

并发执行代码的对象的统称, 每个对象都有独立的状态与调用栈。Python 原生支持 3 种执行单元: 进程、线程、协程。

- **进程 (Process)**

计算机程序在运行时的一个实例, 会消耗内层与一部分 CPU 时间。现代桌面操作系统通常会同时管理数百个进程, 现代桌面操作系统通常会同时管理数百个进程, 并且每个进程都独立于其他进程, 在自己私有的内存空间中运行。进程之间可以通过管道、套接字或内存映射文件进行通信, 这些通信方式只能传递原始字节数据。为了在进程之间传递 Python 对象, 需要将对象序列化 (转换) 为原始字节, 而这种操作的成本较高, 并且并非所有的 Python 对象都是可序列化的。进程还可以派生子进程, 子进程之间以及与父进程之间也是相互隔离的。进程实现了抢占式多任务处理机制: 操作系统调度程序会定期暂停 (挂起) 每个运行中的进程, 以便让其他进程运行。这意味着一个冻结的进程不会冻结整个系统, 至少在理论上是这样的。

- **线程 (Thread)**

单个进程内部的执行单元。当一个进程启动时, 会使用单个线程, 即主线程。进程通过调用操作系统 API 可创建更多线程, 以便并发地执行操作。一个进程内的线程共享相同的内存空间 (用于存储活动的 Python 对象)。这种共享内层空间的方式, 使得线程之间可以轻松地共享数据, 但也可能导致多个线程同时更新同一对象时出现数据损坏的情况。与进程类似, 线程也能在操作系统调度程序的监督下开启抢占式多任务处理。对于同样的处理任务, 线程消耗的资源比进程更少。

- **协程 (Coroutine)**

可以在执行过程中暂停 (挂起) 自身, 并在稍后恢复的函数。在 Python 中, 经典协程是通过生成器函数构建的, 而原生协程则是通过 `async def` 定义的。“[17.13 经典协程<518页>](#)” 已介绍过经典协程的概念, 原生协程的用法将在 “[二十、异步编程<619页>](#)” 中介绍。Python 协程通常在 “事件循环” 的监督下, 在单个线程中运行。事件循环与协程运行在同一线程中。`asyncio`、`Curio` 或 `Trio` 等异步编程框架为基于协程的非阻塞 I/O, 提供了事件循环与支持库。协程支持协作式多任务处理机制: 每个协程必须用 `yield` 或 `await` 关键字明确放弃控制权, 以便其他协程可并发 (不是并行) 开展工作。这意味着, 协程中的任何阻塞代码都会阻塞事件循环和所有其他协程的执行——这与进程和线程支持的抢占式多任务处理机制形成了鲜明对比。另外, 对于同样的处理任务, 协程消耗的资源比线程或进程更少。

- **队列 (Queue)**

一种数据结构, 通常以先入先出 (FIFO, first in first out) 的顺序放入项和取出项。不同的执行单元可通过队列交换应用数据和控制消息, 例如错误代码、终止信号。队列的实现因底层并发模型而异: Python 标准库中的 `queue` 包提供了支持线程的队列类, 而 `multiprocessing` 和 `asyncio` 包则实现了它们自己的队列类。`queue` 和 `asyncio` 包还提供了非 FIFO 的队列: `LifoQueue` 和 `PriorityQueue`。

- **锁 (Lock)**

执行单元可用于同步其操作并避免损坏数据的对象。在更新共享数据结构时, 运行的代码应持有相关的锁。这将向程序的其他部分发出信号, 要求它们在访问同一数据结构之前等待锁被释放。最简单的锁被称为互斥锁 (mutex)。锁的实现取决于底层并发模型。

- 争用 (Contention)

对有限资源的争夺。当多个执行单元试图访问共享资源 (如锁或存储) 时, 就会发生资源争用。当计算密集型进程或线程必须等待操作系统调度程序为其分配 CPU 时间时, 也会发生 CPU 争用。

接下来, 用这些术语分析 Python 对并发的支持。

19.3.1 进程、线程、GIL 锁

下面用 10 点说明这些术语在 Python 编程中的应用:

1. Python 解释器的每个实例都是一个进程。可以用 `multiprocessing` 库 或 `concurrent.futures` 库 启动额外的 Python 进程。Python 的 `subprocess` 库 旨在启动进程来运行外部程序——无论外部程序用何种语言编写。
2. Python 解释器仅使用单个线程运行用户程序与内层垃圾回收程序。可以用 `threading` 库 或 `concurrent.futures` 库 启动额外的 Python 线程。
3. 对对象引用计数和其他内部解释器状态的访问由锁控制, 即“全局解释器锁 (GIL)”。在任意时刻, 只有一个 Python 线程可以持有 GIL。这意味着, 无论 CPU 内核数量多少, 任意时刻都只能有一个线程执行 Python 代码。
4. 为了防止 Python 线程无限期地持有 GIL, Python 的字节码解释器默认每 5ms 挂起一次当前的 Python 线程³, 释放 GIL。然后, 该线程可以尝试重新获取 GIL, 但如果有其他线程在等待, 操作系统调度程序可能会从中选择一个线程继续工作。
5. 我们编写的 Python 代码无法控制 GIL。但是, 用 C 语言 (以及任何可通过 Python/C API 与 Python 交互的语言) 编写的内置函数或扩展, 在运行耗时的任务时可释放 GIL。
6. Python 标准库中发起系统调用⁴的函数均可释放 GIL。包括所有执行磁盘 I/O、网络 I/O 的函数, 以及 `time.sleep()`。NumPy/SciPy 库中许多 CPU 密集型函数, 以及 `zlib`、`bz2` 模块中执行压缩和解压操作的函数, 也都会释放 GIL。⁵
7. 在 Python/C API 层集成的扩展也可以启动不受 GIL 影响的其他非 Python 线程。这些不受 GIL 影响的线程通常无法更改 Python 对象, 但可以读写支持缓冲区协议的内层底层对象, 如 `bytearray`、`array.array` 和 NumPy 数组。
8. GIL 对用 Python 线程进行网络编程的影响相对较小, 因为 I/O 函数会释放 GIL, 而且与内存读写相比, 网络读写的延迟总是很高。各个单独的线程无论如何都要花费大量时间去等待, 所以线程可以交错执行, 不会对整体吞吐量产生重大影响。正如 David Beazley 所言:“Python 线程非常擅长什么都不做。”⁶
9. 对 GIL 的争用会降低计算密集型 Python 线程的运行速度。对于此类任务, 在单线程中顺序执行的代码更简单, 也更快速。
10. 要在多个 CPU 核心上运行 CPU 密集型 Python 代码, 必须使用多个 Python 进程。

`threading` 模块文档 对此做了精彩总结⁷, 如下所示:

³ 调用 `sys.getswitchinterval()` 获取时间间隔; 使用 `sys.setswitchinterval(s)` 更改时间间隔。

⁴ 系统调用是用户代码对操作系统内核功能的调用。I/O、定时器和锁是通过系统调用提供的一些内核服务。要了解更多信息, 请阅读维基百科“System call”。

⁵ Antoine Pitrou (为 Python 3.2 贡献了时间片 GIL 逻辑) 在 `python-dev` 邮件列表 中, 特别提到了 `zlib` 与 `bz2` 模块。

⁶ 摘自“Generators: The Final Frontier”教程第 106 张幻灯片。

⁷ 摘自“Thread Objects”。

CPython 实现细节: 在 CPython 中, 由于全局解释器锁(Global Interpreter Lock)的存在, 一次只能有一个线程执行 Python 代码(尽管某些面向性能的库可能会克服这一限制)。如果想让应用程序充分利用多核机器的计算资源, 建议使用multiprocessing或concurrent.futures.ProcessPoolExecutor。不过, 如果想同时运行多个 I/O 密集型任务, 线程仍是一个合适的模型。

引文以“CPython 实现细节”开头, 因为 GIL 并不是 Python 语言规定的机制。Jython 与 IronPython 中没有 GIL。但遗憾的是, 二者较落后, 还停留在 Python 2.7。性能卓越的 PyPy 解释器在其 2.7 与 3.7(截至 2021 年 6 月的最新版)版本中也有 GIL。



本节未提到协程, 因为默认情况下, 协程之间以及由异步框架提供的监督事件循环之间共享同一个 Python 线程, 因此不受 GIL 的影响。在异步程序中也可以使用多线程, 但是最佳实践是一个线程运行事件循环与所有协程, 其他线程负责执行特定的任务。详见“21.8 将任务委托给执行器<635页>”。

概念先讲到这里。接下来, 分析一些代码。

19.4 并发的 Hello World

在一次关于线程和如何避免 GIL 的讨论中, Python 贡献者 Michele Simionato 发布了一个类似于并发“Hello World”示例。这是一个展示 Python “一心二用”最简单的程序。

Simionato 的程序使用了multiprocessing, 但我对它进行了改造, 分别实现了threading版本与asyncio版本。下面先从threading版本开始。如果您学习过 Java 或 C 语言中的线程, 那么这个版本对您而言或许并不陌生。

19.4.1 用线程实现 Spinner

接下来几个示例的想法很简单: 启动一个函数, 在终端显示字符动画的同时阻塞 3 秒钟, 让用户知道程序正在“思考”, 而不是停滞不前。

该脚本制作一个动画旋转器(Spinner), 在屏幕同一位置轮转显示“\|-”中的每个字符。⁸当慢速计算完成时, 旋转器所在的行被清除, 并显示结果:“Answer: 42”。

图 19.1 展示了 2 个版本旋转器示例的输出结果: 首先是线程版本, 然后是协程版本。若您不在电脑前, 可以想象一下最后一行的字符“-”正在旋转。

首先回顾一下 spinner_thread.py 脚本。示例 19.1 列出了脚本中的前 2 个函数, 示例 19.2 列出了其余函数。

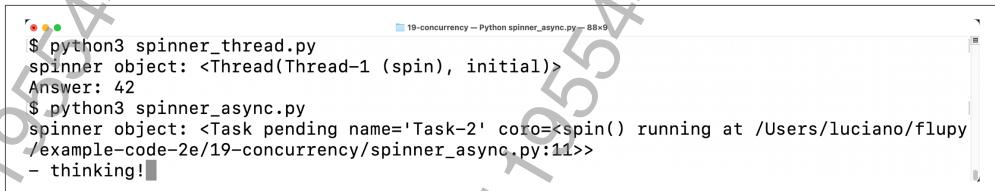
</> 示例 19.1: spinner_thread.py: spin 与 slow 函数

```

1 import itertools
2 import time
3 from threading import Thread, Event

```

⁸Unicode 中有许多对简单动画有用的字符, 例如盲文图案。为简单起见, 我使用的是 ASCII 字符“\|-”。



```
19-concurrency - Python spinner_async.py 88x1
$ python3 spinner_thread.py
spinner object: <Thread(Thread-1 (spin), initial)>
Answer: 42
$ python3 spinner_async.py
spinner object: <Task pending name='Task-2' coro=<spin() running at /Users/luciano/flupy/example-code-2e/19-concurrency/spinner_async.py:11>>
- thinking!
```

图 19.1: `spinner_thread.py` 与 `spinner_async.py` 产生类似的输出: Spinner 对象的字符串表示与文本 “Answer: 42”
截图中, `spinner_async.py` 仍在运行, 显示的是动画消息 “- thinking!”。3 秒后, 该行被替换为 “Answer: 42”

```
4
5  def spin(msg: str, done: Event) -> None: ❶
6      for char in itertools.cycle(r'\|/-'):
7          status = f'\r{char} {msg}' ❸
8          print(status, end='', flush=True)
9          if done.wait(.1): ❹
10             break ❺
11      blanks = ' ' * len(status)
12      print(f'\r{blanks}\r', end='') ❻
13
14  def slow() -> int:
15      time.sleep(3) ❼
16      return 42
```

- ❶ 函数 `spin` 将在单独的线程中运行。参数 `done` 是一个 `threading.Event` 实例, 这是一个用于同步线程的简单对象。
- ❷ 这是一个无限循环, 因为 `itertools.cycle` 每次只能生成一个字符, 一直循环遍历字符串中的字符。
- ❸ 文本动画的诀窍: 用回车 ASCII 控制字符 (`\r`) 将光标移回到行的起始位置。
- ❹ 当事件被其他线程设置时, `Event.wait(s)` 方法返回 `True`; 如果超时, 则返回 `False`。`.1` 秒超时会将动画的“帧率”设置为 10 FPS。如果希望旋转器运行得更快, 请使用更小的超时时间。
- ❺ 退出无限循环。
- ❻ 用空格覆盖并将光标移回行的起始位置, 以清除状态行。
- ❼ 函数 `slow` 将由主线程调用。想象一下, 这是一个通过网络进行的慢速 API 调用。调用 `sleep` 会阻塞主线程, 但 GIL 会被释放, 因此 `spinner` 线程可以继续运行, 即旋转器动画继续运行。



通过此示例, 了解到的最重要一点是: `time.sleep()` 会阻塞调用线程, 但会释放 GIL, 从而允许其他 Python 线程继续运行。

函数 `spin` 与 `slow` 将同时执行。主线程 (程序启动时唯一的线程) 将启动一个新线程运行 `spin`, 然后调用 `slow`。根据设计, Python 未提供终止线程的 API。若想终止线程, 则必须向线程发送一条关闭消息。

`threading.Event` 类是 Python 用来协调线程的最简单的信号机制。`Event` 实例有一个内部布尔标志, 初始值为 `False`。调用 `Event.set()` 会将该标志设置为 `True`。当标志为 `False` 时, 如果一个线程调用 `Event.wait(s)`, 该线程将被阻塞, 直至其他线程调用 `Event.set()` 致使 `Event.wait(s)` 返回 `True`。如果为 `Event.wait(s)` 设置

了一个超时时间(单位:秒),则超时后,Event.wait(s)将返回False;或者如果其他线程调用Event.set(),则Event.wait(s)立即返回True。

示例 19.2 中列出的函数 supervisor,用一个 Event 实例向函数 spin 发送退出信号(见示例中的⑦处)。

</>示例 19.2: spinner_thread.py:supervisor 与 main 函数

```
1 def supervisor() -> int:
2     done = Event()
3     spinner = Thread(target=spin, args=('thinking!', done)) ❶
4     print(f'spinner object: {spinner}') ❷
5     spinner.start() ❸
6     result = slow() ❹
7     done.set() ❺ # 设置 done 事件, 发送线程退出信号
8     spinner.join() ❻
9     return result
10
11 def main() -> None:
12     result = supervisor() ❻
13     print(f'Answer: {result}')
14
15 if __name__ == '__main__':
16     main()
```

- ❶ 函数 supervisor 将返回 slow() 的调用结果。
- ❷ threading.Event 实例 是协调 main 线程与 spinner 子线程活动的关键,详见以下的分析。
- ❸ 创建一个 Thread 实例,将函数 spin 传给关键字参数 target,并为参数 args 提供一个元组——作为传给函数 spin 的位置参数。
- ❹ 打印 spinner 对象。输出结果是“<Thread(Thread-1, initial)>”,其中 initial 是线程状态,表示线程尚未启动。
- ❺ 启动 spinner 子线程。
- ❻ 调用函数 slow,阻塞 main 线程。与此同时,辅助线程正在运行旋转器动画(即函数 spin 中的 for 循环,见“示例 19.1<565页>”)。
- ❼ 将 threading.Event 实例 内部的布尔标志设置为 True;这将终止函数 spin 内的 for 循环。
- ❽ 等待 spinner 子线程结束。
- ❾ 运行函数 supervisor。我单独编写了函数 main 与 supervisor,以使这个示例看起来更像“示例 19.4<569页>”中的 asyncio 版本。

main 线程设置 done 事件(示例 19.2 的⑦处)后,spinner 子线程会收到此信号,并干净地退出。

现在,看一下使用 multiprocessing 包 实现的版本。

19.4.2 用进程实现 Spinner

multiprocessing 包 支持在单独的 Python 进程(而非线程)中运行并发任务。创建 multiprocessing.Process 实例 后,一个全新的 Python 解释器会以子进程的形式在后台启动。因为每个 Python 进程都有自己的 GIL,所以程序可以使用所有可用的 CPU 核心,但这最终取决于操作系统的调度程序。实际影响见“19.6 自研进程

池<576页>”，但对于这个简单程序来说，没什么实质差别。

本节重点是介绍 `multiprocessing` 包，并说明其 API 与 `threading` API 的对应关系，从而将线程版的简称程序改写为进程版。用 `multiprocessing` 包实现的版本如 [示例 19.3](#) 所示。

```
</> 示例 19.3: spinner_proc.py: 仅展示改动部分, 其他代码与 spinner_thread.py 一致

1 import itertools
2 import time
3 from multiprocessing import Process, Event
4 from multiprocessing import synchronize
5
6 def spin(msg: str, done: synchronize.Event) -> None:
7
8 # 函数 spin 的余下部分及函数 slow 均没有改动, 与 spinner_thread.py 一样
9
10 def supervisor() -> int:
11     done = Event()
12     spinner = Process(target=spin,
13                         args=(msg, done))
14     print(f'spinner object: {spinner}')
15     spinner.start()
16     result = slow()
17     done.set()
18     spinner.join()
19     return result
20
21 # 函数 main 也无改动
```

- ❶ `multiprocessing` API 基本都是模仿的 `threading` API。但是类型提示与 Mypy 工具揭示了 `multiprocessing` API 与 `threading` API 之间的差异：`multiprocessing.Event` 是函数（而 `threading.Event` 是类），返回 `synchronize.Event` 实例 ...
- ❷ ... 因此，必须导入 `multiprocessing.synchronize` ...
- ❸ ... 才能编写此处的类型提示（Type Hints）——“`done: synchronize.Event`”。
- ❹ `Process` 类的基本用法与 `Thread` 类 相似。
- ❺ `spinner` 对象显示为 “<Process name='Process-1' parent=14868 initial>”，其中 14868 是运行 `spinner_proc.py` 的 Python 实例的进程 ID。

`threading` 与 `multiprocessing` 的 API 基本相似，但实现方式却截然不同，而且为了处理多进程编程增加的复杂度，`multiprocessing` 拥有更多的 API。例如，将线程转换为进程所面临的一个挑战是，如何在（被操作系统隔离且无法共享 python 对象的）进程之间进行通信。这意味着必须对跨进程传输的 Python 对象进行序列化与反序列化，但这样会带来开销。在 [示例 19.3](#) 中，跨进程传输的数据只有 Event 状态。在 `multiprocessing` 模块的底层 C 语言实现中，Event 状态是通过操作系统底层信号量实现的。⁹

⁹信号量是一种基础构件，可用于实现其他同步机制。Python 为线程、进程与协程提供了不同的信号量类。[“21.7.1 使用 asyncio.as_completed 和线程<629页>”](#) 将用到 `asyncio.Semaphore` 类。



从 Python 3.8 开始, 标准库提供了一个 `multiprocessing.shared_memory` 包, 但它不支持用户定义类的实例。除了原始字节外, 该包还允许进程共享 `ShareableList`, 这是一种可变序列类型, 可保存固定数量的项, 项的类型可以为 `int`、`float`、`bool` 与 `None`, 以及单项不超过 10MB 的 `str` 和 `bytes`。详见 [ShareableList 文档](#)。

现在, 看一下如何用 [协程 \(Coroutine\)](#) 取代线程或进程, 实现相同的行为。

19.4.3 用协程实现 Spinner



“[二十一 异步编程](#)”将专门讲解如何用 [协程 \(Coroutine\)](#) 实现异步编程。本节只是简单介绍, 为了与线程与进程 2 个并发模型进行对比。因此, 本节将忽略许多细节。

操作系统调度程序的工作是分配 CPU 时间以驱动线程和进程。与之相反, [协程 \(Coroutine\)](#) 是由应用级事件循环驱动的, 该事件循环管理一个待处理协程的队列, 逐个驱动它们, 监视由协程启动的 I/O 操作引发的事件, 并在每个事件发生时将控制权传递给相应的协程。事件循环、库协程和用户协程都在同一个线程中执行。因此, 协程中耗费的任何时间都会降低事件循环的速度, 以及其他所有协程的速度。简而言之, 协程的执行会影响事件循环和其他协程的性能表现。

如果从函数 `main` 入手, 再分析函数 `supervisor`, 更易于理解协程版本的 `spinner` 程序。详见 [示例 19.4](#)。

</> [示例 19.4: spinner_async.py](#):main 函数与 supervisor 协程

```
1  def main() -> None:
2      result = asyncio.run(supervisor()) ❶
3      print(f'Answer: {result}')
4
5  async def supervisor() -> int: ❷
6      spinner = asyncio.create_task(spin('thinking!')) ❸
7      print(f'spinner object: {spinner}') ❹
8      result = await slow() ❺
9      spinner.cancel() ❻
10     return result
11
12 if __name__ == '__main__':
13     main()
```

- ❶ 在此程序中, `main` 是唯一的常规函数, 其他函数都是 [协程 \(Coroutine\)](#)。
- ❷ 函数 `asyncio.run` 启动事件循环, 以驱动一个协程, 该协程最终会启动其他协程。函数 `main` 将保持阻塞状态, 直到函数 `supervisor` 返回。`supervisor` 的返回值将成为 `asyncio.run` 的返回值。
- ❸ 原生协程用 `async def` 定义。
- ❹ `asyncio.create_task` 调度 `spin` 的最终执行, 并立即返回 `asyncio.Task` 实例。
- ❺ `spinner` 对象的字符串表示形式看起来像 “`<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:11>`”。

❶ `await` 关键字会调用 `slow`, 阻塞 `supervisor`, 直至 `slow` 返回。`slow` 的返回值将被赋值给 `result`。

❷ `Task.cancel` 方法会在 `spin` 协程内引发 `asyncio.CancelledError` 异常, 如 [示例 19.5](#) 所示。

[示例 19.4](#) 展示了运行协程的 3 种主要方式:

- `asyncio.run(coro())`

在常规函数内调用 (见 [示例 19.4 ❷](#) 处), 以驱动协程对象 `coro`, 该对象通常是程序中所有异步代码的入口点, 例如 [示例 19.4](#) 中的 `supervisor`。这个 `run()` 调用会保持阻塞, 直到协程 `coro` 的主体返回。`run()` 调用的返回值就是协程 `coro` 主体的返回值。

- `asyncio.create_task(coro())`

在协程内调用 (见 [示例 19.4 ❸](#) 处), 调度另一个协程的最终执行。这个 `create_task()` 调用不会挂起当前协程, 并且会返回一个 `asyncio.Task` 实例。该实例封装了协程对象, 并提供了方法来控制和查询协程对象的状态。

- `await coro()`

在协程内部调用 (见 [示例 19.4 ❹](#) 处), 以将控制权转移给 `coro()` 返回的协程对象。`await` 表达式会挂起当前协程, 直到协程 `coro` 的主体返回。`await` 表达式的值就是协程 `coro` 主体的返回值。



请记住: 以 `coro()` 方式调用协程会立即返回一个协程对象, 但不会运行 `coro` 函数的主体。驱动协程的主体是事件循环的工作。

下面分析一下协程 `spin` 与 `slow`, 如 [示例 19.5](#) 所示。

</> [示例 19.5: spinner_async.py](#): `spin` 与 `slow` 协程

```

1 import asyncio
2 import itertools
3
4 async def spin(msg: str) -> None:          ❶
5     for char in itertools.cycle(r'\|/-'):
6         status = f'\r{char} {msg}'
7         print(status, flush=True, end='')
8         try:
9             await asyncio.sleep(.1)          ❷
10        except asyncio.CancelledError: ❸
11            break
12     blanks = ' ' * len(status)
13     print(f'\r{blanks}\r', end='')
14
15 async def slow() -> int:                  ❹
16     await asyncio.sleep(3)
17     return 42

```

❶ 与 `spinner_thread.py` (见 “[示例 19.1](#)”⁵⁶⁵) 不同, 此处无需通过 `Event` 信号指明 `slow` 的工作已完成。

❷ 用 “`await asyncio.sleep(.1)`” 代替 “`time.sleep(.1)`”, 以便挂起时不阻塞其他协程。请见此示例之后的实

验。

- ③ 当在控制此协程的 Task 实例上调用 cancel 方法时, 将引发 `asyncio.CancelledError` 异常。捕获到此异常后, 退出 for 循环。
- ④ 协程 slow 也用 “`await asyncio.sleep(3)`” 代替 “`time.sleep(3)`”。

19.4.3.1 实验: 故意破坏以加深理解

这是我推荐的一个实验, 有助于理解 `spinner_async.py` 的工作原理。导入 `time` 模块后, 用 “`time.sleep(3)`” 替换协程 `slow` 中的 “`await asyncio.sleep(3)`”, 如 [示例 19.6](#) 所示。

```
</> 示例 19.6: spinner_async_experiment.py: 用 "time.sleep(3)" 替换 "await asyncio.sleep(3)"
```

```
1  async def slow() -> int:  
2      time.sleep(3)  # 用 ``time.sleep(3)`` 替换 ``await asyncio.sleep(3)``  
3      return 42
```

执行代码并观察其行为, 比阅读源码更能加深记忆。请动手试一下, 我等你。

当运行此实验时, 您会得到如下输出:

1. 显示 `spinner` 对象, 类似于 “`<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:12>`”。
2. `spinner` 动画一直未出现。程序挂起 3 秒。
3. 显示 “`Answer: 42`”, 然后, 程序终止。

要理解上述行为的关键是要知道, 使用 `asyncio` 的 Python 代码只有一个执行流, 除非显式启动了额外的线程或进程。这意味着, 在任意时刻都只有一个 [协程 \(Coroutine\)](#) 在执行。并发是通过将控制权从一个协程传递给另一个协程来实现的。对于这个实验, 我们应将注意力集中在协程 `supervisor` 与 `slow` 上 (如 [示例 19.7](#) 所示)。

```
</> 示例 19.7: spinner_async_experiment.py: 协程 supervisor 与 slow
```

```
1  async def slow() -> int:  
2      time.sleep(3)          ④  
3      return 42  
4  
5  async def supervisor() -> int:  
6      spinner = asyncio.create_task(spin('thinking!')) ①  
7      print(f'spinner object: {spinner}') ②  
8      result = await slow()          ③  
9      spinner.cancel()          ⑤  
10     return result
```

- ① 创建 `spinner` 任务, 驱动协程 `spin` 的最终执行, 并立即返回 `asyncio.Task` 实例。
- ② 此输出表明 `Task` 处于 “待定 (pending)” 状态。
- ③ 当前协程 (`supervisor`) 通过 `await` 表达式, 将控制权转移至协程 `slow`。
- ④ `time.sleep(3)` 会阻塞 3 秒。在此期间, 程序什么也做不了, 因为主线程被阻塞了, 而主线程是唯一的线程。操作系统将继续进行其他活动。3 秒过后, `sleep` 解除阻塞, 协程 `slow` 返回。

❸ 协程 slow 返回后, spinner 任务立即被取消。控制流从未到达协程 spin 的主体, 所以 spinner 动画一直未出现。

spinner_async_experiment.py 给我们上了重要一课, 详见下方警告栏:



在 asyncio 协程中永远不要使用 `time.sleep()`, 除非你想暂停整个程序。如果一个协程需要空闲一段时间什么也不做, 应使用 `await asyncio.sleep(DELAY)`, 将控制权交还给 asyncio 事件循环, 从而可以驱动其他待处理 (pending) 的协程。

greenlet 与 gevent

在讨论协程并发时, 就不得不提 `greenlet` 包, 这个包已存在多年, 并被广泛使用。该包通过轻量级协程 (被称为 “greenlets”) 支持协作式多任务处理, 无需任何特殊的语法 (如 `yield` 或 `await`), 因此更容易集成到现有的顺序执行代码库中。SQLAlchemy 1.4 ORM 内部使用 greenlets 实现了其新的异步 API, 与 `asyncio` 兼容。

网络库 `gevent` 猴子补丁 (Monkey-patch) 对 Python 标准库 `socket` 模块 进行了修补, 将其中的部分代码替换成了 greenlets, 以防止阻塞。在很大程度上, gevent 对周围的代码是透明的, 从而使顺序应用程序和库 (如数据库驱动) 无须大改, 就能执行并发网络 I/O。许多开源项目 都使用了 gevent, 包括 “19.7.4 WSGI 应用服务器<586页>” 中提到的被广泛部署的 `Gunicorn`。

19.4.4 对比几版 supervisor

`spinner_thread.py` 与 `spinner_async.py` 的行数相当。函数 `supervisor` 是这些示例的核心。本节将详细比较几个版本中的 `supervisor` 函数。示例 19.8 只列出了 “示例 19.2<567页>” 中的 `supervisor` 函数。

</> 示例 19.8: `spinner_thread.py`: 线程版 supervisor 函数

```

1 def supervisor() -> int:
2     done = Event()
3     spinner = Thread(target=spin,
4                       args=('thinking!', done))
5     print('spinner object:', spinner)
6     spinner.start()
7     result = slow()
8     done.set()
9     spinner.join()
10    return result

```

为了便于比较,示例 19.9 列出了 “示例 19.4<569页>” 中的 supervisor 协程。

</> 示例 19.9: `spinner_async.py`: 异步版 supervisor 协程

```

1 async def supervisor() -> int:
2     spinner = asyncio.create_task(spin('thinking!'))
3     print('spinner object:', spinner)
4     result = await slow()
5     spinner.cancel()

```

```
6 |     return result
```

以下两版 supervisor 的异同概述：

- `threading.Thread` 的作用基本等同于 `asyncio.Task`。
- `threading.Thread` 调用一个可调用对象, 而 `asyncio.Task` 驱动一个协程对象。
- 协程 (Coroutine) 通过 `await` 关键字可以显式让出控制权。
- 我们无需手动实例化 `asyncio.Task`, 而是通过向 `asyncio.create_task(...)` 传递一个协程对象来获取 `Task` 实例。
- 当 `asyncio.create_task(...)` 返回一个 `Task` 实例时, 该实例已被调度运行; 而 `Thread` 实例必须通过调用其 `start` 方法才能运行。
- 在线程版 `supervisor` 中, `slow` 是常规函数, 由主线程直接调用。在异步版 `supervisor` 中, `slow` 是一个协程, 由 `await` 关键字驱动。
- 没有从外部终止线程的 API, 必须发送一个信号, 例如设置 `Event` 对象 `done`。而 `Task` 有一个实例方法 `Task.cancel()`, 它会在挂起协程主体的 `await` 表达式处引发 `asyncio.CancelledError` 异常。
- `supervisor` 协程必须在 `main` 函数中, 用 `asyncio.run()` 来启动。详见 “[示例 19.4<569页>](#)” 的 ② 处。

通过对比, 应该可以帮助您理解如何使用 `asyncio` 调度并发作业。相比之下, 您可能更熟悉 `Thread` 模块的运作机制。

关于线程与协程还有最后一点需要注意: 如果您用线程进行过复杂的编程, 您就会知道推断程序状态有多么困难, 因为调度器可以随时中断线程。您必须牢记要通过持有锁来保护程序的关键部分, 以避免在多步操作的中途被中断——这可能会导致数据处于无效状态。

使用协程时, 您编写的代码默认会受中断保护。您必须显式使用 `await` 关键字让出控制权, 程序的其他部分才能运行。与用锁来同步多个线程的操作不同, 协程 (Coroutine) 本身就是“同步的”——即在任意时刻都仅有一个协程在运行。当想要放弃控制权时, 可使用 `await` 关键字将控制权交还给调度器。这也是为什么可以安全地取消一个协程的原因: 根据定义, 协程只能在挂起它的 `await` 表达式处被取消, 因此可以通过处理 `asyncio.CancelledError` 异常来执行清理工作。

`time.sleep()` 会阻塞线程, 但程序什么也做不了。接下来, 我们将试验一个 CPU 密集型调用, 以便深入地理解 GIL, 以及异步代码中 CPU 密集型函数的效果。

19.5 GIL 的真实影响

对于 “[示例 19.1<565页>](#)” 中的线程版 `Spinner` 程序, 将函数 `slow` 中的 `time.sleep(3)` 替换成某个网络库中的 HTTP 客户端请求后, 旋转器动画仍然可继续运行。这是因为一个设计精良的网络库, 会在等待网络响应的过程中, 释放 GIL。

同样, 也可以用某个网络库中的 HTTP 客户端请求替换掉 `slow` 协程中的 `asyncio.sleep(3)` 表达式, 让 `await` 等待异步网络库返回响应。这是因为一个设计精良的网络库, 会在等待网络响应期间, 将控制权交还给事件循环。与此同时, 旋转器动画将继续运行。

对于 CPU 密集型代码, 情况则所有不同。以 [示例 19.10](#) 中的函数 `is_prime` 为例, 如果参数为素数, 则返回 `True`, 否则返回 `False`。

`</> 示例 19.10: primes.py: 易读的素数检查函数 (摘自 Python 文档 “ProcessPoolExecutor 示例”)`

```

1 def is_prime(n: int) -> bool:
2     if n < 2:
3         return False
4     if n == 2:
5         return True
6     if n % 2 == 0:
7         return False
8
9     root = math.sqrt(n)
10    for i in range(3, root + 1, 2):
11        if n % i == 0:
12            return False
13    return True

```

在我使用的公司笔记本电脑¹⁰上,调用 is_prime(5_000_111_000_222_021) 大约耗时 3.3 秒。

19.5.1 小测验

请根据目前所讲的内容,回答如下 3 个问题。其中,有一个问题很难回答(至少对我来说是这样)。

假设 $n=5\text{,}000\text{,}111\text{,}000\text{,}222\text{,}021$ (我的电脑需要 3.3 秒才能验证该素数),如果做如下更改,旋转器(spinner)动画会发生什么变化?

1. 在 spinner_proc.py 中,用 is_prime(n) 调用取代 time.sleep(3)?
2. 在 spinner_thread.py 中,用 is_prime(n) 调用取代 time.sleep(3)?
3. 在 spinner_async.py 中,用 is_prime(n) 调用取代 await asyncio.sleep(3)?

在运行代码或继续阅读之前,建议您先自己找出答案。然后,根据建议复制并修改 spinner*.py 示例。

如下按从简到难给出答案:

1. multiprocessing 版答案

旋转器(spinner)动画由子进程控制。因此,在父进程进行素数检测(执行 is_prime(n))的过程中,旋转器(spinner)动画会继续运行。¹¹

2. threading 版答案

旋转器(spinner)动画由辅助现场控制。因此,在主线程进行素数检测(执行 is_prime(n))的过程中,旋转器(spinner)动画会继续运行。

我一开始没有答对:我原以为旋转器(spinner)会被冻结,因为我高估了 GIL 的影响。

对此示例来说,旋转器(spinner)保持旋转是因为 Python 每 5 毫秒(默认值)暂停正在运行的线程,使全局解释器锁(GIL)可供其他等待的线程使用。因此,主线程运行 is_prime 函数时每 5 毫秒被中断一次,以允许辅助线程唤醒并在 for 循环中迭代一次,直到它调用 done 事件的 wait 方法(见“示例 19.1<565 页>”的④处),释放 GIL。然后,主线程将抢夺 GIL, is_prime 接着计算 5 毫秒。

¹⁰ 这是一款 15 寸 MacBook Pro 2018,配置 Intel Core i7 CPU,6 核,2.2GHz。

¹¹ 今天之所以如此,是因为你使用的可能是具有抢占式多任务处理功能的现代操作系统。NT 时代之前的 Windows 和 OSX 时代之前的 macOS 都不支持“抢占式”,因此任何进程都可能占用 100% 的 CPU,导致整个系统假死。今天,我们还没有完全摆脱这个问题,但请相信我这个白胡子老头:在 20 世纪 90 年代,这种问题困扰着每一位用户,而硬重启是唯一的解决办法。

这对本示例没有明显影响,因为 spin 函数迭代一次的速度很快,并在收到 done 事件后就释放 GIL。所以,对 GIL 的争用并不多。大多数时间,GIL 都由运行 is_prime 的主线程持有。

在这个简单的实验中,因为只涉及 2 个线程,所以我才用线程来处理计算密集型任务。在这 2 个线程中,一个独占 CPU,另一个每秒仅唤醒 10 次来更新旋转器(spinner)。

但是,如果有 2 个或更多线程都想争夺大量 CPU 时间,那么并发执行的运行速度要比顺序执行的代码更慢。

3. asyncio 版答案

对于 `spinner_async.py` 示例,在协程 slow 中调用 `is_prime(5_000_111_000_222_021)`,旋转器(spinner)动画永远都不会出现。效果与在“[示例 19.6<571页>](#)”中用 `time.sleep(3)` 替换“`await asyncio.sleep(3)`”的情况相同,指针根本不旋转。控制权将由 supervisor 传递给 slow,然后传递给 `is_prime`。当 `is_prime` 返回时,slow 也会返回,而 supervisor 会恢复运行,并在 `spinner` 任务执行之前立即将其取消。程序会冻结约 3 秒,然后显示答案。

用 `sleep(0)` 小憩一下

保持旋转器(spinner)指针旋转的一种方法是,将 `is_prime` 重写为 [协程\(Coroutine\)](#),并在 `await` 表达式中定期调用 `asyncio.sleep(0)`,以将控制权交还给事件循环,如[示例 19.11](#)所示。

</> [示例 19.11: spinner_prime_async_nap.py](#):协程版的 `is_prime`

```

1  async def is_prime(n):
2      if n < 2:
3              return False
4      if n == 2:
5              return True
6      if n % 2 == 0:
7              return False
8
9      root = math.sqrt(n)
10     for i in range(3, root + 1, 2):
11         if n % i == 0:
12                 return False
13         if i % 100_000 == 1:
14                 await asyncio.sleep(0) ❶
15     return True

```

❶ 每 50,000 次(因为 `range` 内的步长是 2)迭代,就休眠一次。

[asyncio 存储库](#) 的“[Issue #284](#)”详细介绍了 `asyncio.sleep(0)` 的用法。

但是,请注意这会减慢 `is_prime` 的运行速度,而且更重要的是,还会减慢事件循环与整个程序的运行速度。换成每 100,000 次迭代执行 1 次“`await asyncio.sleep(0)`”时,旋转器(spinner)动画很平衡,但程序在我的笔记本电脑上运行了 4.9s,几乎比原始的 `is_prime` 函数的执行时间长了 50%(参数同为 `5_000_111_000_222_021`)。

使用“`await asyncio.sleep(0)`”只是权宜之计,更稳妥的做法是重构异构代码,将 CPU 密集型计算委托给另一个进程。“[二十一 异步编程](#)”中将介绍一种用 `asyncio.loop.run_in_executor` 实现

的方案。另外,还可以使用任务队列(详见“[19.7.5 分布式任务队列](#)”^{588页})。

到目前为止,我们所做的实验都只调用了一个CPU密集型函数。下一节,将介绍如何并发执行多个CPU密集型调用。

19.6 自研进程池



编写本节的目的是为了展示如何使用多个进程来处理CPU密集型任务,以及使用队列来分配任务和收集结果的常见模式。“[二十并发执行器](#)”将展示一种将任务分配给进程的更简单的方法:`concurrent.futures`包中的`ProcessPoolExecutor`,它在内部使用队列。

本节将编写一个程序,计算 $2 \sim 9,999,999,999,999,999$ (即 $10^{16}-1$)或大于 2^{53} 的20个整数样本是否为素数¹²。整数样本中包括大素数、小素数,以及可分解为大、小素数的合数¹³。

以`sequential.py`程序的性能作为基准。下面是某次运行的结果:

```

1 $ python3 sequential.py
2      2 P 0.000001s
3 142702110479723 P 0.568328s
4 299593572317531 P 0.796773s
5 3333333333333301 P 2.648625s
6 3333333333333333 0.000007s
7 3333335652092209 2.672323s
8 4444444444444423 P 3.052667s
9 4444444444444444 0.000001s
10 4444444488888889 3.061083s
11 5555553133149889 3.451833s
12 5555555555555503 P 3.556867s
13 5555555555555555 0.000007s
14 6666666666666666 0.000001s
15 66666666666666719 P 3.781064s
16 6666667141414921 3.778166s
17 7777777536340681 4.120069s
18 7777777777777753 P 4.141530s
19 7777777777777777 0.000007s
20 9999999999999917 P 4.678164s
21 9999999999999999 0.000007s
22 Total time: 40.31

```

显示结果包括如下3列:

- 第1列,是待检查的整数。
- 第2列,在整数是素数时,显示为P,否则显示空白。

¹²素数(prime):指只能被1和自身整除的正整数。

¹³合数(composite):指除了1和自身之外,还有其他因子的正整数。

- 第 3 列, 执行素数检查时, 所使用的耗时。

在本示例中, 总耗时单独计算, 近似于每次检查的耗时之和。如示例 19.12 所示。

</> 示例 19.12: `sequential.py`: 对一个小型数据集做素数检查 (顺序执行)

```
#!/usr/bin/env python3
"""
sequential.py: baseline for comparing sequential, multiprocessing,
and threading code for CPU-intensive work.
"""

from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class Result(NamedTuple):          ❶
    prime: bool
    elapsed: float

def check(n: int) -> Result:      ❷
    t0 = perf_counter()
    prime = is_prime(n)
    return Result(prime, perf_counter() - t0)

def main() -> None:
    print(f'Checking {len(NUMBERS)} numbers sequentially:')
    t0 = perf_counter()
    for n in NUMBERS:              ❸
        prime, elapsed = check(n)
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

    elapsed = perf_counter() - t0  ❹
    print(f'Total time: {elapsed:.2f}s')

if __name__ == '__main__':
    main()
```

- ❶ `check` 函数 (❷处) 返回一个 `Result` 元组, 其中包含 `is_prime` 调用返回的布尔值与耗时。
- ❷ `check(n)` 调用 `is_prime(n)`, 并计算耗时, 返回一个 `Result`。
- ❸ 对于样本中的每个整数 `n`, 都会调用 `check(n)`, 并显示调用结果。
- ❹ 计算并显示总耗时。

19.6.1 基于进程的方案

下一个示例 (“示例 19.13<579页>” 中的 `procs.py`) 展示了如何用多个进程将素数检查分配给多个 CPU 核心。我运行 `procs.py` 的耗时如下:

```

1 $ python3 procs.py
2 Checking 20 numbers with 12 processes: 2 P 0.000002s
3 3333333333333333 0.000021s
4 4444444444444444 0.000002s
5 5555555555555555 0.000018s
6 6666666666666666 0.000002s
7 142702110479723 P 1.350982s
8 7777777777777777 0.000009s
9 299593572317531 P 1.981411s
10 9999999999999999 0.000008s
11 333333333333301 P 6.328173s
12 333335652092209 6.419249s
13 4444444888888889 7.051267s
14 444444444444423 P 7.122004s
15 555553133149889 7.412735s
16 555555555555503 P 7.603327s
17 6666666666666719 P 7.934670s
18 666667141414921 8.017599s
19 777777536340681 8.339623s
20 777777777777753 P 8.388859s
21 999999999999917 P 8.117313s
22 20 checks in 9.58s

```

输出的最后一行表明, procs.py 的耗时是 sequential.py 的约 23.77%。

19.6.2 理解耗时

请注意, 第 1 列中的耗时是检查特定数字的耗时。例如, `is_prime(77777777777753)` 用了近 8.4 秒才返回 `True`。与此同时, 其他进程也在并行检查其他数字。

有 20 个数字需要检查。我编写了 `procs.py`, 以启动多个工作 (worker) 进程, 具体的进程数量取决于 `multiprocessing.cpu_count()` 获取的 CPU 核数。

在此情况下, 总耗时要远小于各个检查所耗时的总和。在进程启动与进程间通信方面会产生一些开销, 因此最终结果是多进程版本仅比顺序版本快 4.2 倍。这已经很不错了, 但考虑到代码会启动 12 个进程来使用这台笔记本电脑上的所有 CPU 核心, 这个总耗时还是有些令人失望。



在我用来撰写本章内容的 Macbook Pro 笔记本电脑中, `multiprocessing.cpu_count()` 返回 12。这台笔记本电脑的 CPU 实际上是一个 6 核 CPU 的 Core-i7 处理器, 但操作系统却显示 12 核 CPU, 这是因为采用了超线程技术, 这是英特尔的一项技术, 每个核心可执行 2 个线程。不过, 当其中一个线程的工作强度不如同核心的另一个线程时, 超线程的效果会更好——也许第一个线程在缓存缺失后停滞不前, 等待数据, 而另一个线程正在忙于处理数字。总之, 天下没有免费的午餐: 对于不占用大量内存的计算密集型工作, 比如简单的素数测试, 这台笔记本电脑就像一台 6 核 CPU 一样。

19.6.3 多核素数检测器代码

当将计算任务委托给线程或进程时,我们的代码不能直接调用承担工作的 worker 函数,因此无法轻易地获取 worker 的返回值。worker 有线程库或进程库驱动,最终产生的结果需要存储在某处。在并发编程(以及分布式系统)中,经常用队列来协调 worker 以及收集 worker 的结果。

procs.py 中的大部分新代码都与队列的设置与使用有关。文件顶部内容,如示例 19.13 所示。



`multiprocessing.SimpleQueue` 是 Python 3.9 新增的。若使用的 Python 版本较老,可将 `SimpleQueue` 替换为 `multiprocessing.Queue`。

</> 示例 19.13: procs.py: 多进程素数检测; 导入、类型和函数

```
1 import sys
2 from time import perf_counter
3 from typing import NamedTuple
4 from multiprocessing import Process, SimpleQueue, cpu_count ❶
5 from multiprocessing import queues
6
7 from primes import is_prime, NUMBERS
8
9 class PrimeResult(NamedTuple):
10     n: int
11     prime: bool
12     elapsed: float
13
14 JobQueue = queues.SimpleQueue[int] ❷
15 ResultQueue = queues.SimpleQueue[PrimeResult]
16
17 def check(n: int) -> PrimeResult:
18     t0 = perf_counter()
19     res = is_prime(n)
20     return PrimeResult(n, res, perf_counter() - t0)
21
22 def worker(jobs: JobQueue, results: ResultQueue) -> None: ❸
23     while n := jobs.get():
24         results.put(check(n)) ❹
25         results.put(PrimeResult(0, False, 0.0)) ❺
26
27 def start_jobs(
28     procs: int, jobs: JobQueue, results: ResultQueue ❻
29 ) -> None:
30     for n in NUMBERS:
31         jobs.put(n) ❽
32     for _ in range(procs):
33         proc = Process(target=worker, args=(jobs, results)) ❾
34         proc.start() ❿
```

```
35     jobs.put(0) ⑯
```

- ❶ 为了模拟线程, `multiprocessing` 提供了 `multiprocessing.SimpleQueue`, 但这是一个方法, 被绑定到底层 `BaseContext` 类的一个预定义的实例上。`SimpleQueue` 仅用于构建队列, 而不能用于类型提示 (Type Hints)。
- ❷ 类型提示所需的 `SimpleQueue` 类, 位于 `multiprocessing.queues` 中。
- ❸ `PrimeResult` 包含要进行素数检测的数字 `n`。将数字 `n` 与其他结果字段放在一起, 以简化稍后的结果显示。
- ❹ 这是 `queues.SimpleQueue` 的类型别名。`main` 函数 (见示例 19.14) 将使用此队列类型向执行工作的进程发送数字。
- ❺ 为另一个 `queues.SimpleQueue` 类型创建类型别名, 该队列类型用于收集 `main` 函数的结果。队列中的值是一个元组, 由待检测的数字与一个 `Result` 元组构成。
- ❻ 此函数与 `sequential.py` (见“示例 19.12<577页>”) 中的类似, 用于执行素数检测以及计算检测耗时。
- ❼ `worker` 的参数是 2 个队列: 一个队列用于存放待检测的数字, 另一个队列用于存放检测结果。
- ❽ 这段代码中, 我用数字 0 作为“毒丸 (poison pill)”——是 `worker` 完成工作的信号。如果 `n` 不是 0, 则继续循环。¹⁴
- ❾ 调用素数检测函数 `check`, 并将检测结果放入队列 `ResultQueue`。
- ❿ 发回一个 `PrimeResult(0, False, 0.0)`, 让主循环知道此 `worker` 已完成工作。
- ❾ procs 是并行检测素数的进程数。
- ❿ 将待检查的数字放入 `jobs` 队列中。
- ⓫ 为每个 `worker` 派生一个子进程。每个子进程都会在各自的 `worker` 函数中运行循环, 直到从 `jobs` 队列中获取 0。
- ⓬ 启动子进程。
- ⓭ 在各个进程中, 将 0 放入队列, 以终止进程。

循环、哨兵值 (Sentinel Value) 和毒丸 (poison pill)

示例 19.13 中的 `worker` 函数遵循了并发编程中的常见模式: 在无限循环中获取队列中的项, 并用执行实际工作的函数处理每个项。当队列生成一个 `哨兵值 (Sentinel Value)` 时, 循环结束。在此种模式中, 用于关闭 `worker` 进程的 `哨兵值 (Sentinel Value)` 通常被称为“毒丸 (poison pill)”。

`None` 经常被用作 `哨兵值 (Sentinel Value)`, 但如果出现在数据流中, 就可能不合适了。为了获得用于 `哨兵值 (Sentinel Value)` 的唯一值, 通常会调用 `object()`。但是, 这在跨进程时不起作用, 因为 Python 对象必须被序列化才能用于进程间通信。当对一个 `object` 的实例执行 `pickle.dump` 与 `pickle.load` 后, 未 `pickle` 的实例会与原始实例不同——它们不相等。`None` 有一个很好的替代品, 即内置对象 `Ellipsis` (即 `...`), 它在序列化时不会丢失其原有身份。^a

Python 标准库中使用许多不同的值作为 `哨兵值 (Sentinel Value)`。“PEP 661 –Sentinel Values”提出了标准 `哨兵 (Sentinel)` 类型。截至 2021 年 9 月, 此 PEP 还只是一个草案。

^a 在序列化的过程中能生存下来, 同时又不丧失其原有身份, 这是一个相当不错的人生目标。

¹⁴ 在本例中, 0 是一个很方便的 `哨兵值 (Sentinel Value)`。另一个常用的 `哨兵值` 是 `None`。但这里使用 0, 可以简化 `PrimeResult` 的类型提示与 `worker` 的代码。

接下来,分析 `procs.py` 中的 `main` 函数,如 [示例 19.14](#) 所示。

```
</> 示例 19.14: procs.py:多进程版素数检测器中的 main 函数

1  def main() -> None:
2      if len(sys.argv) < 2:
3          procs = cpu_count() ❶
4      else:
5          procs = int(sys.argv[1])
6
7      print(f'Checking {len(NUMBERS)} numbers with {procs} processes.')
8      t0 = perf_counter()
9      jobs: JobQueue = SimpleQueue() ❷
10     results: ResultQueue = SimpleQueue()
11     start_jobs(procs, jobs, results) ❸
12     checked = report(procs, results) ❹
13     elapsed = perf_counter() - t0
14     print(f'{checked} checks in {elapsed:.2f}s') ❺
15
16     def report(procs: int, results: ResultQueue) -> int: ❻
17         checked = 0
18         procs_done = 0
19         while procs_done < procs: ❼
20             n, prime, elapsed = results.get() ❼
21             if n == 0: ❼
22                 procs_done += 1
23             else:
24                 checked += 1
25                 label = 'P' if prime else ' '
26                 print(f'{n:16} {label} {elapsed:9.6f}s') ❼
27         return checked
28
29     if __name__ == '__main__':
30         main()
```

- ❶ 如若未提供命令行参数,则将进程数设为 CPU 核数;否则,创建的进程数与第一个参数值相同。
- ❷ `jobs` 与 `results` 是 [示例 19.13](#) 定义的队列。
- ❸ 启动 `proc` 进程,处理 `jobs` 队列中的作业,将结果放入 `results` 队列。
- ❹ 获取并显示 `results` 队列。`report` 函数在 ❻ 处定义。
- ❺ 显示检测了多少个数字,以及总耗时。
- ❻ 参数是进程的数量 `procs` 与发布结果的队列 `results`。
- ❼ 一直循环,直至所有进程都结束。
- ❼ 在队列 `results` 上调用 `get()`,以获取一个 `PrimeResult`。在 `results` 队列中有项之前,`get()` 方法会一直处于阻塞状态;也可将此操作设置为非阻塞,或为其设置超时,详见 [SimpleQueue.get](#) 文档。
- ❼ 如果 `n` 等于 0,则表示有一个进程退出;增加 `procs_done` 的计数。
- ❼ 否则,增加 `checked` 计数(记录检测了多少个数字),并显示检测结果。

结果的返回顺序与提交作业的顺序不同,这也是我为每个 `PrimeResult` 元组加入 `n` 的原因。如若不然,无

法知道每个结果属于哪个数字。

如果主进程在所有子进程完成之前退出,则可能会引发由 `multiprocessing` 内部锁导致的 `FileNotFoundError` 异常,其 traceback 令人困惑。并发代码的调试本就很困难,而调试 `multiprocessing` 更是难上加难,因为所有复杂的细节都隐藏在看似线程的表象之下。幸运的是,“[二十一 异步编程](#)”中所介绍的 `concurrent.futures.ProcessPoolExecutor` 更易于使用,也更健壮。



感谢读者 Michael Albert 在阅读本书试读版时,为我指出本书 [示例 19.14](#) 中存在“条件竞争”(见 `primes/procs_race_condition.py`)。条件竞争是一种 bug,它出现与否取决于并发执行单元执行操作的顺序。若 A 在 B 之前发生,则一切正常;一旦 B 先发生,就会出现问题。

若您对此 bug 感兴趣,可以查看 [example-code-2e/commit/2c123057](#),以了解代码差异以及修正方案。不过请注意,后来我重构了该示例,将 `main` 的部分职责委托给了 `start_jobs` 与 `report` 函数。该目录中的 `README.md` 文件,对此问题及解决方案做了说明。

19.6.4 实验:进程数多一些或少一些

可以运行 `procs.py`,通过命令行参数为其设置 worker 进程的数量,如下所示:

```
1 $ python3 procs.py 2
```

上述命令将启动 2 个 worker 进程,得到计算结果的耗时大概是 `sequential.py` (见“[示例 19.12](#)”577 页”)的一半——如果您的电脑至少为双核 CPU,且未忙于运行其他程序。

我分别用 1~20 个进程,运行了 12 次 `procs.py` 脚本,共计 240 次。然后,对于每个进程数,计算了 12 次运行的中位数耗时,基于此绘制了 [图 19.2](#)。

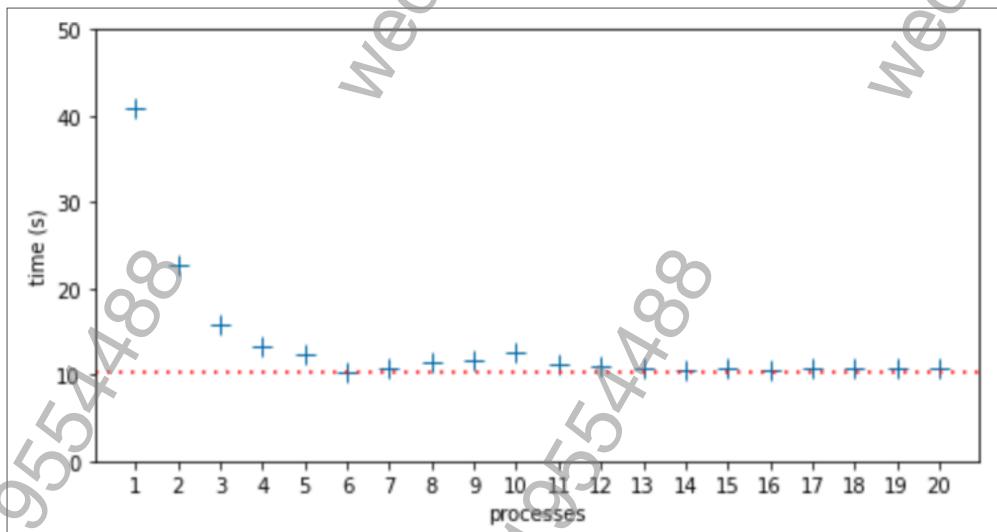


图 19.2: 从 1~20 个进程的运行耗时中位数。使用 1 个进程的中位数耗时最高,为 40.81 秒;使用 6 个进程的中位数耗时最低,未 10.39 秒(图中虚线)。

在我这台 6 核笔记本电脑中,6 个进程的中位数耗时最短——10.39 秒,如图 19.2 中的虚线所示。我原以为在超过 6 个进程时,会因为 CPU 争用而使运行耗时增加,在进程数为 10 个时,耗时达到最大值 12.51 秒。

但我没想到,也无法解释为何性能在 11 个进程时有所提高,而在 13~20 个进程时性能几乎持平,并且中位数耗时仅略高于 6 个进程时的最低中位数耗时。

19.6.5 基于线程的方案不可靠

我还编写了 `threads.py`,将 `procs.py` 中的 `multiprocessing` 替换为 `threading`。这两个版本代码非常相似——对于简单的示例,在这 2 个 API 之间转换,大都如此。由于存在 GIL,而且 `is_prime` 是计算密集型函数,线程版代码比 `sequential.py` (见“示例 19.12<577 页>”)中的顺序执行版更慢,而且随着线程数的增加,速度还会更慢。这是因为涉及了 CPU 争用,以及上下文切换所带来的开销。为了切换到新线程,操作系统需要保存 CPU 寄存器,并更新程序计数器与栈指针,这一过程所触发的副作用(如使 CPU 缓存失效、交换内存叶等)开销较大。接下来的 2 章,将继续介绍 Python 并发编程:“二十 并发执行器”使用高级 `concurrent.futures` 库来管理线程和进程;“二十一 异步编程”使用 `asyncio` 库进行异步编程。

本章余下部分旨在回答以下问题:

在目前讨论的诸多限制条件下,Python 如何在多核世界中繁荣发展?

19.7 多核世界的 Python

如下内容摘自“*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*”(Herb Sutter)一文:

从 Intel 和 AMD 到 Sparc 和 PowerPC,主流的处理器制造商与架构为了提高 CPU 性能,已用尽了大多数传统方法。他们不再追求更高的时钟速度与线性指令吞吐量,而是大量转向超线程与多核架构。

Sutter 所说的“免费的午餐 (free lunch)”是指无需开发人员额外干预,即可提高软件运行速度的趋势,因为 CPU 执行顺序代码的速度一年比一年快。从 2004 年开始,这个趋势就已不再存在:CPU 时钟速度与执行优化均达到了顶峰,如今任何显著的性能提升都必须通过利用多核或超线程技术来实现,而只有并发执行的代码才能从中受益。

Python 的故事始于 20 世纪 90 年代初,当时 CPU 的顺序代码执行速度仍然呈指数级增长。那时,除了超级计算机之外,几乎没人讨论多核 CPU。当时,引入 GIL 的决定是理所当然的。得益于 GIL,Python 解释器在单核 CPU 上运行的速度更快,而且实现也更简单。¹⁵借助于 GIL,通过 Python/C API 编写简单扩展也更容易。



我之所以强调“简单扩展”,是因为扩展无需处理 GIL。用 C 或 Fortran 编写的函数可能比 Python 中的等效函数快数百倍。^a因此,在许多情况下可能不需要为了利用多核 CPU,而大费周章地释放 GIL。因此,要感谢 GIL 为 Python 提供了许多可用的扩展,这无疑是该语言如今如此流行的关键因素之一。

^a读大学时,有一项作业要求我必须用 C 语言实现 LZW 压缩算法。但我先用 Python 实现了该算法,以检验我对规范的理解。用 C 语言实现的版本,速度快了大约 900 倍。

¹⁵也许正是这些原因促使 Ruby 语言的创造者 Yukihiro Matsumoto,在他的解释器中也使用了 GIL。

尽管存在 GIL 的限制,但是 Python 在需要并发或并行执行的应用领域中仍然能蓬勃发展,这要归功于绕过 CPython 限制的库和软件架构。

接下来,将探讨在这个多核 CPU 与分布式计算的年代,Python 在系统管理、数据科学与服务端应用开发等领域如何发挥作用。

19.7.1 系统管理

Python 广泛用于管理大型服务器集群、路由器、负载平衡器和 NAS 存储。Python 也是软件定义网络 (SDN) 和黑客攻击的首选语言。主流的云服务提供商都提供了 Python 库与相关教程,包括通过其自己开发的,以及 Python 用户社区编写的。

在这个领域,Python 脚本向远程设备发送命令,由远程设备执行,以实现配置任务自动化。由于很少涉及 CPU 密集型操作,因此非常适合使用线程或协程。特别是,“[二十一 异步编程](#)”中介绍的 `concurrent.futures` 包,可用于同时在多台远程设备中执行相同的操作,而且不会引入太多的复杂性。

除了标准库之外,还有一些基于 Python 的流行项目,可用于管理服务器集群,如 `Ansible`、`Salt` 等工具,以及 `Fabric` 库等。

支持协程与 `asyncio` 的系统管理库也在不断增加。2016 年,Facebook 的[生产工程师团队](#)报告称“我们越来越依赖于 Python 3.4 引入的 `AsyncIO`,在将代码库从 Python 2 迁移至 Python 3 之后,性能得到了巨大的提升。”

19.7.2 数据科学

数据科学(包括人工智能)和科学计算是 Python 的强项。这些领域的应用程序都属于计算密集型,但 Python 用户可以从 C、C++、Fortran、Cython 等语言编写的大量数值计算库生态系统中获益,其中许多库都可以利用多核设备、GPU 和(或)异构集群中的分布式并行计算。

截至 2021 年,Python 的数据科学生态系统包括了大量令人印象深刻的工具,例如:

- **Jupyter 项目**

提供了 2 个基于浏览器的界面(`Jupyter Notebook`与 `JupyterLab`),这 2 个界面允许用户在远程设备上运行和记录分析代码,可能跨网络运行。二者都是 Python/JavaScript 混合型应用程序,支持用不同语言编写的计算内核,然后通过 ZeroMQ(一个用于分布式应用程序的异步消息传递库)集成。`Jupyter`这个名字实际上来自于 `Notebook` 最先支持的 3 种语言:Julia、Python 和 R。建立在 `Jupyter` 工具之上的丰富生态系统丰富多样,其中包括 `Bokeh`,这是一个功能强大的交互式可视化库。`Bokeh` 凭借现代 JavaScript 引擎和浏览器的性能,允许用户浏览大型数据集或持续更新的流数据,并与之进行交互。

- **TensorFlow 与 PyTorch**

根据 O'Reilly 2021 年 1 月发布的“[2020 年学习资源使用情况报告](#)”,二者是排名前 2 位的深度学习框架。这 2 个项目都是用 C++ 编写的,并且能充分利用多核 CPU、GPU 与集群资源。它们也支持其他语言,但重点支持的是 Python 语言,这也是大多数用户使用的语言。`TensorFlow` 由 Google 创建并在内部使用;`PyTorch` 由 Facebook 创建并在内部使用。

- **Dask**

`Dask` 是一个并行计算库,可将工作分派给本地进程或设备集群——正如其[主页](#)所言“已在世界上最大的超级计算机中进行了测试”。`Dask` 提供的 API 非常接近于 `NumPy`、`pandas` 和 `scikit-learn`,这些都

是当今数据科学领域和机器学习领域最流行的库。Dask 可以在 JupyterLab 或 Jupyter Notebook 中使用, 它利用 Bokeh 不仅可以实现数据可视化, 还可以实现交互式仪表盘——以近乎实时的方式显示数据流, 以及跨进程/跨设备的计算。Dask 令人印象深刻, 因此我建议大家观看一段由 Matthew Rocklin (Dask 维护者之一) 录制的 [15 分钟视频](#), 该视频演示了: 在分布于 8 台 AWS EC2 设备的 64 个 CPU 核心上, 用 Dask 进行数据运算的过程。

以上只是一些示例, 旨在说明数据科学领域如何利用 Python 优点, 并克服 CPython 运行时的限制, 来构建解决方案的。

19.7.3 服务端 Web 与移动开发

Python 被广泛应用于 Web 应用程序开发和为移动应用程序提供支持的后端 API 开发。Google、YouTube、Dropbox、Instagram、Quora 和 Reddit 等公司是如何成功构建 Python 服务器端应用程序, 为数亿用户提供 7x24 小时服务的? 同样, 答案远远超出了 Python 提供的“开箱即用”范围。

在讨论 Python 支持弹性伸缩的工具之前, 首先引用 Thoughtworks Technology Radar 的一段话:

对高性能的渴望、对 Web-Scale 的渴望

我们看到许多团队因为“可能需要伸缩”而选择了复杂的工具、框架或架构, 从而陷入困境。Twitter 和 Netflix 等公司需要支持极高的负载, 因此需要这些架构, 但他们拥有技术精湛的开发团队, 能够应对由此带来的复杂性。而大多数情况下, 并不需要这类技术; 开发团队应该克制对 Web-Scale 的渴望, 选择更简单可行的解决方案。^a

^a资料来源: Thoughtworks 技术顾问委员会, [Technology Radar](#), 2015 年 11 月刊

Web-Scale 的关键是要有一个可水平伸缩的架构。在该架构中, 所有系统都是分布式系统, 并且没有任何一种编程语言能够包揽全部解决方案。

分布式系统是一个学术研究领域, 但幸运的是, 一些从业者已经根据扎实的研究和实践经验, 写出了多本通俗易懂的书籍。例如,《Designing Data-Intensive Applications》(Martin Kleppmann) 就是其中一本。

如 图 19.3 所示的架构图, 就是摘自《Designing Data-Intensive Applications》。图中有一部分组件与 Python 相关, 有些我用过, 有些我只是了解。

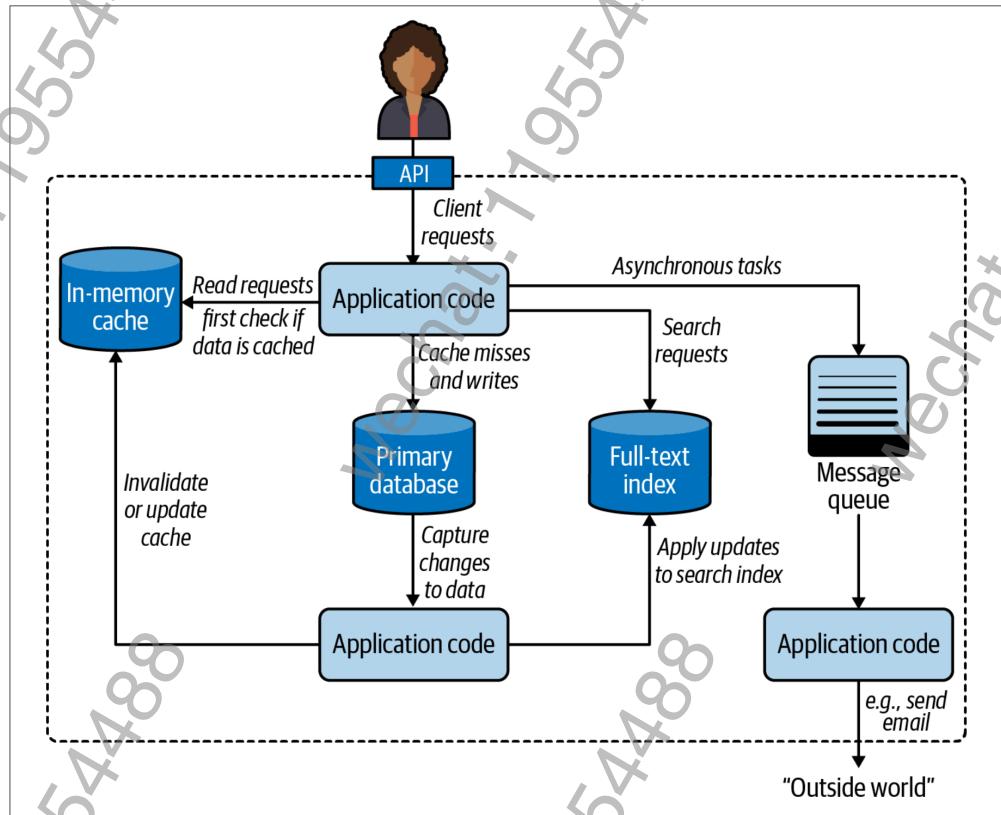
- 应用程序缓存:¹⁶ memcached、Redis、Varnish。
- 关系型数据库: PostgreSQL、MySQL。
- 文档数据库: Apache CouchDB、MongoDB。
- 全文索引: Elasticsearch、Apache Solr。
- 消息队列: RabbitMQ、Redis。

在上述每个类别都有与其对应的工业级开源产品, 主流云供应商也提供了自己的专属替代产品。

Kleppmann 给出的图都是通用的, 与特定语言无关, 他的书也是如此。对于 Python 服务器端应用程序, 通常会部署如下 2 个特定组件:

¹⁶ 对比应用程序缓存 (由应用程序代码直接使用) 与 HTTP 缓存, 后者被放置在 图 19.3 的顶部边缘, 用于提供静态资产 (如图像、CSS、JS 文件)。内容分发网络 (Content Delivery Network, CDN) 提供另一种类型的 HTTP 缓存, 部署在靠近终端用户的数据中心内。

¹⁷ 此图改编自《Designing Data-Intensive Applications》(Martin Kleppmann) 中的“图 1-1”。

图 19.3: 由多个组件构成的系统可能采用的架构¹⁷

- 一个应用程序服务器, 用于在 Python 应用程序的多个实例之间分配负载。应用程序服务器应放在图 19.3 靠近顶部的位置, 在客户端请求到达应用程序代码之前对请求进行处理。
- 一个任务队列, 围绕图 19.3 右侧消息队列构建的任务队列, 旨在提供更易用的高级 API, 以便为其他设备中运行的进程分配任务。

接下来的 2 节将深入探讨应用程序服务器与任务队列, 二者是 Python 服务器端部署中推荐的最佳实践。

19.7.4 WSGI 应用服务器

WSGI (Web Server Gateway Interface, Web 服务器网关接口) 是一种标准 API, 用于 Python 框架或应用程序从 HTTP 服务器接收请求并向其发送响应。¹⁸ WSGI 应用程序服务器管理运行应用程序的一个或多个进程, 从而最大限度地利用 CPU。

“图 19.4<587 页>”展示了典型的 WSGI 部署。



Python Web 项目中最著名的应用程序服务器包括:

- mod_wsgi

¹⁸ 有些人逐字母的拼读 WSGI, 而有些人则将其视作一个单词来拼读, 音似 “whisky”。

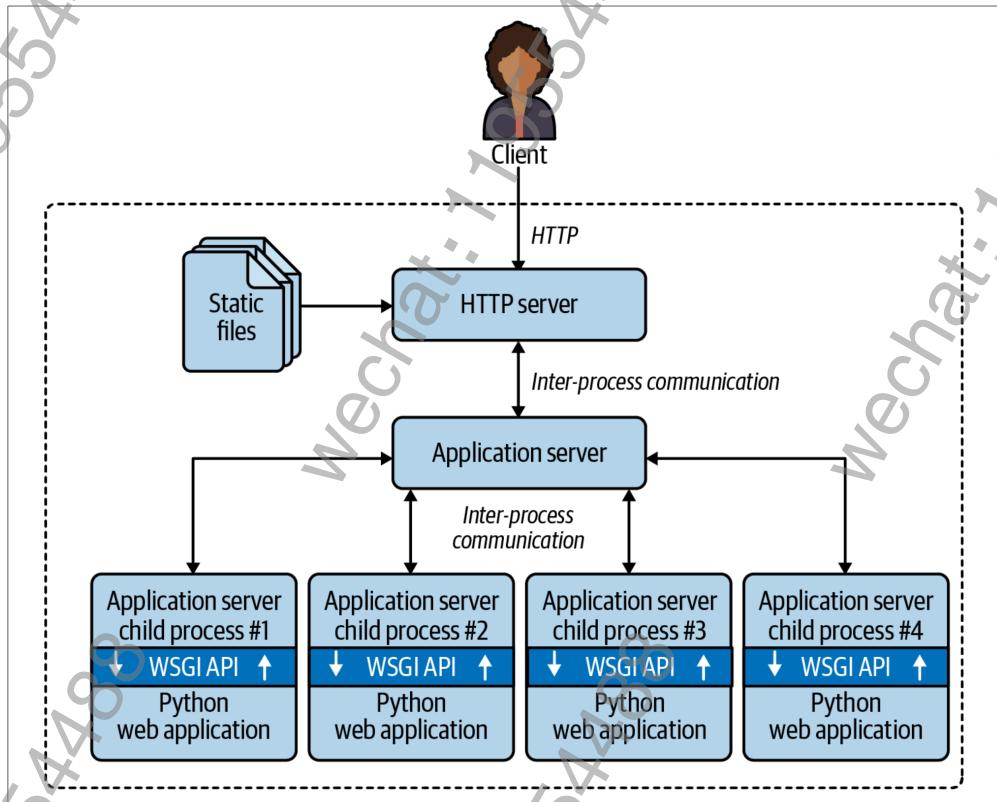


图 19.4: 客户端连接 HTTP 服务器, 后者负责分发静态文件并将其他请求路由到应用服务器; 应用程序服务器派生子进程, 利用多个 CPU 核心运行应用程序代码。WSGI API 是应用服务器与 Python 应用程序代码之间的粘合剂

- [uWSGI¹⁹](#)
- [Gunicorn](#)
- [NGINX Unit](#)

对于 Apache HTTP 服务器的用户来说, `mod_wsgi` 是最佳选择。它与 WSGI 本身一样古老, 但一直在积极维护。现在, 还提供了一个名为 `mod_wsgi-express` 的命令行启动器, 使其更易于配置, 也更适合在 Docker 容器中使用。

据我所知, 新开发的项目大都首选 [uWSGI](#) 与 [Gunicorn](#)。二者经常与 NGINX HTTP 服务器搭配使用。除了本职工作之外, [uWSGI](#) 还提供了许多额外功能, 包括应用程序缓存、任务队列、类似于 cron 的定期任务等。另一方面, [uWSGI](#) 的配置难度要比 [Gunicorn](#) 大得多。²⁰

[NGINX Unit](#) 发布于 2018 年, 是知名 NGINX HTTP 服务器和反向代理制造商推出的一款新产品。

`mod_wsgi` 与 [Gunicorn](#) 仅支持 Python Web 应用程序, 而 [uWSGI](#) 与 [NGINX Unit](#) 还支持其他编程语言。详见各自文档。

要点: 所有这些应用程序服务器都可以通过派生多个 Python 进程来, 运行使用 Django、Flask、Pyramid 等老式顺序代码编写的传统 Web 应用程序, 从而充分利用服务器的所有 CPU 核心。这也解释了为何 Python Web 开发人员无须学习 `threading`、`multiprocessing`、`asyncio` 模块, 也能找到工作——应用程序服务器可以透明地处理并发。

¹⁹uWSGI 拼写时, 开头字母为小写字母 “u”, 但发音未希腊字母 “μ”。所以, 整个名称听起来像 “micro-whisky”, “k” 发音 “g”。

²⁰Bloomberg 的工程师 Peter Sperl 与 Ben Green 撰写的 “[Configuring uWSGI for Production Deployment](#)” 一文, 揭露了 [uWSGI](#) 的许多默认设置并不适合许多常见的部署场景。Sperl 在 [EuroPython 2019](#) 的演讲总结了推荐设置, 强烈推荐给 [uWSGI](#) 用户。



ASGI——异步服务器网关接口

WSGI 是一个同步 API, 它不支持用 `async/await` 创建的协程。而在 Python 中, [协程 \(Coroutine\)](#) 是实现 WebSocket 或 HTTP 长轮询最有效的方式。ASGI 规范是 WSGI 的继任者, 专为异步 Python Web 框架(如 `aiohttp`、`Sanic`、`FastAPI` 等)设计, 而 Django 与 Flask 等传统框架也正在逐步增加异步功能。

接下来, 介绍可绕过 GIL, 以提高服务器端 Python 应用程序性能的另一种方法。

19.7.5 分布式任务队列

当应用服务器将请求分发给运行代码的某个 Python 进程时, 应用程序需要快速响应: 您希望进程尽快可用以处理下一个请求。然而, 某些请求可能需要花费更长时间来执行某些操作, 例如发送电子邮件或生成 PDF。这就是分布式任务队列旨在解决的问题。

在提供 Python API 的开源任务队列中, [Celery](#) 与 [RQ](#) 最为著名。云供应商也会提供其专属的任务队列。

这些产品封装了一个消息队列, 并提供了高级 API, 用于将任务委派给可能在不同服务器上运行的 worker。



在任务队列中, 使用“生产者”与“消费者”这 2 个词代替传统的“客户端/服务器”术语。例如, Django 视图处理程序会生产作业请求, 这些请求会被放入队列中, 以供一个或多个 PDF 渲染进程消费。

下面直接引用了 [Celery FAQ](#) 中的一些典型用例:

- 在后台运行某些任务。例如, 尽快完成 Web 请求, 然后增量更新用户页面。这会给用户留下性能良好与“即时响应”的印象, 尽管实际工作可能还需要一些时间。
- 在 Web 请求完成后, 执行某些操作。
- 通过异步执行某些操作, 并在失败后重试, 以确保操作一定完成。
- 调度定期工作。

除了解决这些显而易见的问题之外, 任务队列还支持横向扩展。生产者与消费者是解耦的: 生产者不会调用消费者, 而是将请求放入队列。消费者不需要了解有关生产者的任何信息(但如果需要确认, 请求可能包括有关生产者的信息)。最重要的是, 随着需求的增长, 您可以很容易地添加更多的 worker 来处理(消费)任务。这也是 [Celery](#) 与 [RQ](#) 被称为“分布式任务队列”的原因。

回想一下, 前面那个简单的 `procs.py` (见“[示例 19.13](#)”[579 页](#)”) 使用了 2 个队列: 一个用于作业请求, 另一个用于收集结果。[Celery](#) 与 [RQ](#) 分布式架构也使用了类似的模式。二者都支持用 NoSQL 数据库 [Redis](#) 作为消息队列与结果存储器。[Celery](#) 还支持使用其他消息队列 (RabbitMQ 或 Amazon SQS), 以及用于存储结果的其他数据库。

对 Python 并发性的介绍到此结束。接下来的 2 章将继续这一主题, 但重点介绍标准库中的 `concurrent.futures` 和 `asyncio` 包。

19.8 本章小结

本章在介绍了一些理论之后,用 Python 的 3 种原生并发编程模型实现了 `spinner_*` 脚本:

- 用 `threading` 包实现的线程版 (`spinner_thread.py`);
- 用 `multiprocessing` 包实现的进程版 (`spinner_proc.py`);
- 用 `asyncio` 包 实现的异步协程版 (`spinner_async.py`);

然后,通过一个实验探讨了 GIL 的真实影响:将旋转器示例中的 `sleep(s)` 调用替换为 `is_prime` 素数检测调用,并观察由此产生的行为影响。经过比较后发现, `asyncio` 不善于处理 CPU 密集型函数,因为它们会阻塞事件循环。尽管存在 GIL,但 `threading` 版仍能正常工作,因为 Python 会周期性地中断线程,而且该示例仅使用了 2 个线程:一个用于执行计算密集型工作,另一个每秒仅更新指针动画 10 次。`multiprocessing` 版可绕过 GIL,启动一个新进程用于处理指针动画,而主进程则用于执行素数检测 (`is_prime`)。

随后是一个关于素数检测的示例,该示例强调了 `multiprocessing` 与 `threading` 之间的区别,也证实了只有多进程才能让 Python 从多核 CPU 中受益。对于计算繁重的任务,受 Python GIL 的限制,线程版代码比顺序执行版表现的要更糟。

在关于 Python 并发与并行的讨论中,一定绕不开 GIL,但也不要过于高估 GIL 的影响(如“[19.7 多核世界的 Python](#)”所述)。例如,在系统管理中使用 Python,很多时候不受 GIL 的影响。另外,数据科学领域与服务器端开发社区已经根据他们的特定需求,为绕过 GIL 定制了工业级的解决方案。最后两节提到了支持 Python 服务器端应用程序水平扩展的 2 个常见组件:WSGI 应用程序服务器、分布式任务队列。

19.9 延伸阅读

本章延伸阅读材料较多,因此将其分为多节。

19.9.1 用线程和进程实现并发

“[二十 并发执行器](#)”中介绍的 `concurrent.futures` 库,背后涉及了线程、进程、锁与队列,但这些细节由高度抽象的 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 负责管理,对用户是隐藏的。如果想深入了解有关使用这些底层对象进行并发编程的实践,则建议先阅读《[An Intro to Threading in Python](#)》(Jim Anderson)一文。《[Python 3 Standard Library by Example, 1st ed](#)》(Doug Hellmann)一书中的“Concurrency with Processes, Threads, and Coroutines”一章,也提供了并发编程相关内容。

《[Effective Python, 2nd ed](#)》(Brett Slatkin 著)、《[Python Essential Reference, 4th ed](#)》(David Beazley 著)、《[Python in a Nutshell, 3rd Ed](#)》(Martelli 等) 虽然是通用性的 Python 参考资料,但也都重点介绍了 `threading` 模块与 `multiprocessing` 模块。`multiprocessing` 官方文档内容丰富,在 [Programming guidelines](#) 一节给出了许多实践建议。

`multiprocessing` 包 由 Jesse Noller 与 Richard Oudkerk 贡献, [PEP 371 -Addition of the multiprocessing package to the standard library](#) 对此包进行了介绍。`multiprocessing` 官方文档规模达 63 页,是 Python 标准库中篇幅最长的章节之一。

《[High Performance Python, 2nd Ed](#)》(Micha Gorelick 与 Ian Ozsvald 著)一书关于 `multiprocessing` 的章节中,作者提供了一个素数检测的示例,与本书 `procs.py` 脚本(见“[示例 19.13](#)”)使用的策略不

同。对于每个数字 n , 将 $2 \sim \sqrt{n}$ ²¹ 之间可能的因数划分成多个分组; 然后, 每个 worker 分别遍历其中的一个分组。这种“分而治之”的方式, 在科学计算应用程序中非常常见, 因为这种应用程序中涉及的数据集非常庞大, 而工作站(或集群)的 CPU 核心比较多。在处理众多用户请求的服务器端系统中, 让每个进程从头到尾都只处理一个计算会更简单高效——减少了进程之间通信和协调的开销。除了 `multiprocessing`, 作者还介绍了利用多核 CPU、GPU、集群、性能分析器和编译器(如 Cython 和 Numba)等, 开发部署高性能数据科学应用程序的许多其他方法。最后一章“实践经验”收集了由其他 Python 高性能计算从业者贡献的简短案例, 非常有价值。

《Advanced Python Development》(Matthew Wilkes 著)中包含了一些术语介绍的简短示例, 同时展示了如何构建一个准备投入生产使用的真实应用程序: 一个数据聚合器, 类似于 DevOps 监控系统或用于分布式传感器的 IoT 数据收集器。该书有 2 章涵盖了 `threading` 与 `asyncio` 并发编程。

《Parallel Programming with Python》(Jan Palach 著, 2014)一书解读了并发与并行背后的核心概念, 涵盖了 Python 标准库以及 Celery。

《Using Asyncio in Python》(Caleb Hattingh 著)第 2 章“The Truth About Threads”介绍了线程的优缺点(引用了权威资料, 令人信服), 还明确指出线程的挑战与 Python 或 GIL 无关。以下内容摘自该书的第 14 页:

我想强调的是:

- 使用线程的代码, 难以理解。
- 对于大规模并发(数千个并发任务)而言, 线程是一种低效模型。

要想学习如何在不丢工作的情况下, 了解线程与锁的推断是多么困难, 则可以尝试阅读《The Little Book of Semaphores》(Allen Downey 著)一书, 并完成其中的练习。该书中的练习从简到难, 甚至有些练习可能无解, 但即使是简单的练习也能让你大开眼界。

19.9.2 GIL

如果您对 GIL(全局解释器锁)感到好奇, 请记住一点: Python 代码无法控制 GIL。所以, 有关 GIL 的权威参考资料都在 C-API 文档中——Thread State and the Global Interpreter Lock。Python 标准库与扩展 FAQ 中回答了 [Can't we get rid of the Global Interpreter Lock?](#) 这一问题。此外, 文章 ["It isn't Easy to Remove the GIL"](#) (Guido van Rossum) 与 ["Python Threads and the Global Interpreter Lock"](#) (Jesse Noller) 也值得一读。

《CPython Internals》(Anthony Shaw)一书从 C 语言编程层面, 解读了 CPython 3 解释器的实现。最长一章“并行性和并发性”: 深入探讨了 Python 对线程和进程的原生支持, 包括在扩展中如何用 C/Python API 管理 GIL。

最后, David Beazley 在演讲“text”中对 GIL 做了详细探讨。²² 在第 54 张幻灯片中, Beazley 报告称: 使用 Python 3.2 中引入的新 GIL 算法, 特定基准测试的处理时间有所增加。实现 GIL 新算法的 Antoine Pitrou 在 [Python issue#7946](#) (由 Beazley 提交) 的评论中指出, Beazley 发现的这个问题在真实工作负载中并不明显。

²¹ \sqrt{n} 是一个数学函数, 用于计算一个数的平方根。它返回一个浮点数, 表示输入 n 的平方根。

²² 感谢 Lucas Brunialti 给我发来这个讲座的链接。

19.9.3 标准库之外的并发

本书侧重于 Python 语言的核心特性与标准库的核心部分。《Full Stack Python》是对本书一个很好的补充,涉及整个 Python 生态系统,包括开发环境、数据处理、Web 开发、DevOps 等。

前面提到的《High Performance Python, 2nd Ed》与《Parallel Programming with Python》,不仅涵盖如何用 Python 标准库进行并发编程,还用大量篇幅介绍了并发编程的第三方库与工具。《Distributed Computing with Python》(Francesco Pierfederici 著)一书,介绍了 Python 标准库、云供应商,以及 HPC(高性能计算)集群的使用。

Matthew Rocklin 于 2019 年 6 月发布的“Python, Performance, and GPUs”一文,介绍了“Python 使用 GPU 加速器的新情况”。

“Instagram 目前拥有全球最大规模的 Django Web 框架部署,该框架完全由 Python 编写。”这是 Min Ni (Instagram 软件工程师)撰写的博客文章“Web Service Efficiency at Instagram with Python”的开篇句。该文章描述了 Instagram 用于优化其 Python 代码库效率的指标和工具,以及在“每天 30-50 次”部署后端时检测和诊断性能回归的指标和工具。

《Architecture Patterns with Python》(Harry Percival、Bob Gregory 合著)介绍了 Python 服务器端应用程序的架构模式。作者还在 cosmicpython.com 免费在线提供了这本书。

`lelo` 库(作者 Joao S. O. Bueno)与 `python-parallelize` 库(作者 Nat Pryce)是 2 个可通过多个进程处理并行任务的库,二者易于使用且方式优雅。`lelo` 库定义的 `@parallel` 装饰器,可用于装饰任何函数。被 `@parallel` 装饰过的函数,像变魔术一样变成非阻塞函数,运行在单独的 Python 进程中。`python-parallelize` 库中定义了一个 `parallelize` 生成器,可将 `for` 循环的执行分到多个 CPU 上。这 2 个库都是以 `multiprocessing` 库为基础。

Eric Snow (Python 核心开发人员)维护了一个维基页面——“Multicore Python”,其中记录了他和其他人为改进 Python 对并行执行的支持所做的努力。Snow 创建的“PEP 554 –Multiple Interpreters in the Stdlib”如果被获准并实施,将为未来功能增强奠定基础,最终可使 Python 无需借助额外开销的 `multiprocessing` 模块,即可使用多个 CPU 核心。目前,最大的障碍之一是,如何解决多个活动子解释器与假定单解释器的扩展之间的复杂交互。

Mark Shannon (也是 Python 维护者)创建了一个比较 Python 中并发模型的表格。他在 Python-dev 邮件列表上与 Eric Snow 和其他开发者讨论子解释器时,引用了该表格。表格中的“Ideal CSP”列指的是 Tony Hoare 于 1978 年提出的理论上的通信顺序进程模型。Go 也允许共享对象,这违反了 CSP 的基本约束:执行单元应通过通道进行消息传递通信。

Stackless Python(简称 Stackless)是 CPython 的一个实现了微线程的分支。微线程是应用程序级的轻量级线程,而不是操作系统线程。大型多人在线游戏“EVE Online”就是基于 Stackless 开发的,游戏公司 CCP 的工程师曾是 Stackless 的维护者。Stackless 的一些功能在 Pypy 解释器和 greenlet 包(`gevent` 网络库的核心技术)中得到了重新实现,而 greenlet 包又是 Gunicorn 应用服务器的基础。

并发编程的“参与者模型(Actor Model)”是高度可伸缩的 Erlang 和 Elixir 语言的核心,也是 Scala 和 Java 的 Akka 框架的模型。如果想在 Python 中尝试参与者模型,可了解一下 Thespian 库和 Pykka 库。

下面推荐的资料很少或根本不会提及 Python。但如果您对本章主题感兴趣,这些资料也是有一定相关性的。

19.9.4 Python 之外的并发与水平扩展

《RabbitMQ in Action》(Alvaro Videla、Jason J. W. Williams 合著) 一书对 RabbitMQ 与高级消息队列协议 (Advanced Message Queuing Protocol, AMQP) 的介绍绘彩纷呈, 还提供了用 Python、PHP、Ruby 编写的示例。无论您用什么技术栈, 即便在底层使用含有 RabbitMQ 的 Celery, 我也推荐这本书。因为该书涵盖了分布式消息队列相关的概念、动机和模式, 以及如何操作与调校 RabbitMQ, 以实现弹性伸缩。

我从《Seven Concurrency Models in Seven Weeks》(Paul Butcher 著) 一书中学到了很多。这本书的副标题富有表现力——理不顺的线程。书中的第 1 章介绍了 Java 使用线程和锁进行编程的核心概念与面对的挑战²³。余下 6 章专门探讨了作者认为更好的并发和并行编程替代方案, 涉及不同的语言、工具和库。书中的示例涵盖了 Java、Clojure、Elixir 和 C (使用 OpenCL 框架进行并行编程的章节)。CSP 模型的示例使用了 Clojure, 不过对于这种方法的普及, Go 语言功不可没。Actor 模型的示例使用了 Elixir。作者对 Actor 模型示例做了修订, 换成了 Scala 和 Akka 框架。除非您已了解 Scala, 否则 Elixir 是更易学的语言, 也是更适合尝试 Actor 模型以及 Erlang/OTP 分布式系统平台的语言。

Thoughtworks 公司的 Unmesh Joshi 在 [Martin Fowler 博客](#) 上发表了几篇关于“分布式系统模式”的文章。开篇是对这一主题的精彩介绍, 并提供了各个模式的链接。Joshi 正在逐步增加模式, 但其中已经包含了多年来在关键任务系统中积累的宝贵经验。

《Designing Data-Intensive Applications》(Martin Kleppmann 著) 是一本难得的好书, 其作者是拥有深厚行业经验与先进学术背景的从业者。在成为剑桥大学分布式系统研究员之前, 作者曾在 LinkedIn 和两家初创公司从事大规模数据基础设施的工作。Kleppmann 在书中的每一章末尾都给出了大量参考文献, 包括最新的研究成果。书中还包括大量富有启发性的图表与精美的概念图。

我很荣幸能在 OSCON 2016 上参加了 Francesco Cesarini 关于可靠分布式系统架构的研讨会: “Designing and architecting for scalability with Erlang/OTP” ([视频](#))。Cesarini 在视频的 9:35 处解释道:

我要讲的内容很少是针对 Erlang 的 [...]。但是您要知道, Erlang 可消除许多意外的困难, 使系统具有弹性、永不失效、可扩展。因此, 如果您使用 Erlang 语言, 或者使用在 Erlang 虚拟机上运行的其他语言, 那么您就会更容易理解我要讲的内容。

该研讨会基于《Designing for Scalability with Erlang/OTP》(Francesco Cesarini、Steve Vinoski 合著) 一书的最后 4 章。

分布式系统编程既具有挑战性, 又令人兴奋。不过, 如前所述, 不要一味追求 Web-Scale。KISS 原则始终是可靠的工程建议。

读一读由 Frank McSherry、Michael Isard 和 Derek G. Murray 发表的论文 “Scalability! But at what COST?”。作者指出, 学术研讨会上提出的并行图形处理系统需要数百个 CPU 核心才能胜过“合格的单线程实现”。他们还发现, 有些系统“在其报告的所有配置中, 性能可能还不如单个线程”。

这些发现让我想起了黑客经常说的一句嘲讽话:

我的 Perl 脚本比你的 Hadoop 集群都快。

²³ Python 的线程和并发 API 深受 Java 标准库的影响。

杂谈

画地为牢, 规行矩止

我学会了在 TI-58 计算器上编程。TI-58 计算器使用的“语言”类似于汇编语言。在这个层次上, 所有“变量”都是全局变量, 而且没有奢侈的结构化控制流语句。您需要根据条件来跳转: 根据 CPU 寄存器或标志的值, 使用跳转指令直接跳到某处(当前位置之前或之后)执行。

基本上, 你在汇编语言中可以做任何事情, 而这正是挑战所在: 无拘无束, 随时都可能犯错, 并且在修改代码后, 还需要帮助维护者理解代码。

我学会的第二种编程语言是 8 位计算机附带的非结构化 BASIC——与后来出现的 Visual Basic 完全不同。它有 FOR、GOSUB 和 RETURN 语句, 但仍然没有局部变量的概念。GOSUB 不支持参数传递: 它只是一个花哨的 GOTO, 它将一个返回的行号放入堆栈, 以便为 RETURN 语句提供跳转目标。子程序可以自由使用全局数据, 并将结果放在那里。其他形式的控制流只能用 IF 和 GOTO 的组合临时实现, 同样也允许你跳转到程序的任一行。

在使用跳转和全局变量编程数年之后, 我开始学习 Pascal。我还记得, 当时我努力转变思路, 重建“结构化编程”的概念。那时, 我不得不在只有一个入口点的代码块周围使用控制流语句。我无法随意跳转到我喜欢的指令。全局变量在 BASIC 中是不可避免的, 但现在却成了禁忌。我需要重新思考数据流, 并显式地将参数传递给函数。

我的下一个挑战是学习面向对象编程。面向对象编程的核心是具有更多约束和多态性的结构化编程。信息隐藏迫使我不得不重新思考数据存放位置。我不止一次感到沮丧, 因为我必须重构我的代码, 以便我重新编写的方法可以获得封装在对象中的信息——重构前的方法居然无法直接访问对象中的信息。

函数式编程语言又增加了其他约束, 但经过数十年的命令式编程和面向对象编程的洗礼, “不变性”是最难接受的概念。然而, 一旦习惯这些约束之后, 就会发现这是一种幸事。因为它们使写出的代码更容易理解。

缺乏约束是基于线程和锁模型进行并发编程的主要问题。Paul Butcher 在总结《Seven Concurrency Models in Seven Weeks》第 1 章时, 写道:

然而, 这种方法的最大弱点在线程和锁的编程难度很大。对于语言设计者来说, 将其添加到语言中可能很容易, 但对于这些可怜的程序员来说, 编程语言层面却没有提供什么帮助。

该模型中不受约束行为的一些示例:

- 线程可以共享访问任意可变的数据结构。
- 调度程序几乎可以在任何时刻中断线程, 包括在执行 $a += 1$ 这种简单操作的中间时刻。在源代码表达式级别上, 原子操作很少见。
- 锁往往是“建议性的 (advisory)”。这是一个专业术语, 意思是在更新共享数据结构之前, 你必须记住显式加锁。如果您忘记获取锁, 那么当另一个线程尽职地持有锁并更新相同的数据时, 你的代码将不可避免地破坏数据。

相比之下, 参与者 (Actor) 模型可以强制执行一些约束。其中的执行单元被称为“参与者”^a:

- 参与者 (Actor) 可以拥有内部状态, 但不能与其他参与者共享状态。
- 参与者 (Actor) 只能通过发送和接收消息来进行通信。

- 消息只能保存数据副本, 而不能保存对可变数据的引用。
- 一个参与者(Actor)一次只能处理一条消息。单个参与者内部不存在并发执行。

当然, 只要遵循这些规则, 您可以在任何语言中采用参与者编码风格。你也可以在 C 语言中使用面向对象编程习语, 甚至在汇编语言中使用结构化编程模式。但要做到这一点, 需要每个接触代码的人都达成一致并遵守约定。

在 Erlang 和 Elixir 实现的参与者(Actor)模型中, 所有数据类型都是不可变的, 因此无需管理锁。

线程和锁不会消失。我只是认为, 在编写应用程序(而不是内核模块或数据库)时, 不应将时间浪费在这种底层实体的处理上。

我保留随时改变想法的权利。但现在, 我确信参与者(Actor)模型是最合理的通用并发编程模型。CSP(通信顺序进程)也很合理, 但它在 Go 中的实现遗漏了一些限制。CSP 的理念是使用队列(在 Go 中称为“通道”)来交换数据和同步协程(在 Go 中称为“goroutines”)。但 Go 也支持内存共享和锁。我看到过一本关于 Go 的书, 提倡使用共享内存和锁来代替通道, 还美其名曰“为了性能”。真是旧习难改。

^aErlang 社区使用术语“进程”表示参与者。在 Erlang 中, 每个进程都是其自身循环中的一个函数, 因此 Erlang 中的进程是非常轻量级的, 并且可以在一台机器上同时激活数百万个进程——与我们在本章其他地方讨论的重量级操作系统进程无关。因此, 这也体现了 Simon 教授说的 2 个重要过错: 使用不同的词来表示同一事物, 以及使用同一个词来表示不同的事物。

并发执行器

抨击线程的通常是系统开发程序员，他们所考虑的使用场景可能是应用开发程序员一辈子都不会遇到的。... 应用程序开发程序员可能遇到的 99% 的场景中，只需知道如何“派生一堆独立线程，然后用队列收集结果”即可。

——Michele Simionato, Python 的深度思考者^a

^a摘自 Michele Simionato 的文章“Threads, processes and concurrency in Python: some thoughts”。

本章重点介绍实现了 `concurrent.futures.Executor` 接口的类，这些类对 Michele Simionato 所提的“派生一堆独立线程，然后用队列收集结果”模式进行了封装，使用起来更加容易，不仅可适用于线程，而且还适用于进程——对于计算密集型任务非常有用。

本章还介绍了“futures”这一概念——代表异步执行操作的对象，类似于 JavaScript 中的 `promise`。`futures` 是 `concurrent.futures` 模块与 `asyncio` 包（见“[二十一 异步编程](#)”）的基础。

20.1 本章新增内容

本章标题从原来的“Concurrency with Futures”更改为“Concurrent Executors”，因为 `Executor` 是本章最重要的高级特性。`futures` 是底层对象，主要在“[20.2.3 future 对象在何处](#)”中重点介绍，本章余下部分几乎不会涉及。

现在，所有 HTTP 客户端示例都换用了新的 `HTTPX` 库，该库同时提供同步 API 与异步 API。

得益于 Python 3.7 中的 `http.server` 添加了多线程服务器，因此“[20.5 显示下载进度并处理错误](#)”中的实验设置变得更加简单。在此之前，Python 标准库中只提供了单线程的 `BaseHttpServer`，这不利于并发客户端的实验。因此，本书第 1 版中，不得不借助于外部工具。

“[20.3 用 concurrent.futures 启动进程](#)”展示了如何用 `Executor` 简化“[19.6.3 多核素数检测器代码](#)”中的示例代码。

最后，将大部分理论移到了新增的“[十九 Python 中的并发模型](#)”中。

20.2 并发网络下载

并发对于高效处理网络 I/O 至关重要:应用程序不应无所事事地等待远程设备,而应在接收到远程设备响应之前,先做些其他事情。¹

为了用代码演示这一点,我编写了 3 个简单脚本,从网上下载 20 个国家的国旗图片。第 1 个脚本 `flags.py` 按序下载:只有上一个图片下载完毕并保存本地后,才会请求下一个图片;而另外 2 个脚本则是并发下载:同时请求多张图片,并在图片到达时将其保存。`flags_threadpool.py` 脚本使用 `concurrent.futures` 包,而 `flags_asyncio.py` 则使用 `asyncio` 包。

示例 20.1 显示了运行这 3 个脚本的结果,每个脚本运行 3 次。这些脚本是从 fluentpython.com 下载图片的,该网站已建立 CDN 缓存,因此首次运行时可能会看到较慢的结果。示例 20.1 是多次运行后的结果,因此 CDN 缓存是有效的。

</> 示例 20.1: 3 个脚本 `flags.py`、`flags_threadpool.py` 和 `flags_asyncio.py` 的运行结果

```

1 $ python3 flags.py
2 BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ❶
3 20 flags downloaded in 7.26s ❷
4 $ python3 flags.py
5 BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
6 20 flags downloaded in 7.20s
7 $ python3 flags.py
8 BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
9 20 flags downloaded in 7.09s
10 $ python3 flags_threadpool.py
11 DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
12 20 flags downloaded in 1.37s ❸
13 $ python3 flags_threadpool.py
14 EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
15 20 flags downloaded in 1.60s
16 $ python3 flags_threadpool.py
17 BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
18 20 flags downloaded in 1.22s
19 $ python3 flags_asyncio.py ❹
20 BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
21 20 flags downloaded in 1.36s
22 $ python3 flags_asyncio.py
23 RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
24 20 flags downloaded in 1.27s
25 $ python3 flags_asyncio.py
26 RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR ❺
27 20 flags downloaded in 1.42s

```

❶ 每次运行脚本后,首先按序显示下载完毕的国家代码,最后显示一条消息,说明总耗时。

❷ 运行 3 次 `flags.py` 脚本,平均耗时 $(7.26 + 7.20 + 7.09)/3 = 7.18\text{s}$ 。

❸ 运行 3 次 `flags_threadpool.py` 脚本,平均耗时 $(1.37 + 1.60 + 1.22)/3 = 1.40\text{s}$ 。

¹ 尤其是某些云供应商按设备使用时长收费,与 CPU 使用量无关。

④ 运行 3 次 `flags asyncio.py` 脚本, 平均耗时 $(1.36 + 1.27 + 1.42)/3 = 1.35\text{s}$ 。

⑤ 请注意国家代码的顺序: 使用并发脚本时, 每次下载的顺序都不同。

2 并发下载脚本之间的性能差异并不大, 但都比顺序下载脚本快 5 倍以上——这还只是下载 20 个几千字节文件的小任务。若将下载的文件数量增加到几百个, 并发脚本的速度会比顺序下载快 20 倍或更多。



在互联网中测试并发 HTTP 客户端时, 可能会在无意中发起 DoS (Denial-of-Service) 攻击, 或者被怀疑发起了这种攻击。示例 20.1 不受此影响, 毕竟只发起了 20 个请求。本章稍后将使用 Python 的 `http.server` 包 来运行测试。

接下来, 分析示例 20.1 中脚本 `flags.py` 与 `flags_threadpool.py` 的实现。脚本 `flags asyncio.py` 将留到“[二十一 异步编程](#)”中进行介绍。本节将 3 个脚本放到一起演示, 是为了说明两点:

- 对于网络 I/O 操作, 无论使用哪种并发模型 (线程或协程), 只要代码编写正确, 吞吐量都要比顺序执行的代码高很多。
- 对于可以控制请求次数的 HTTP 客户端来说, 线程与协程之间的性能差异不明显。²

继续分析源码。

20.2.1 顺序下载脚本

`flags.py` 脚本的实现如示例 20.2 所示。此脚本并无特别之处, 只是实现并发下载的脚本会复用 `flags.py` 中的部分代码与设置, 所以还是有必要分析一下。



为简单起见, 示例 20.2 暂时未做异常处理。此处重点关注代码的基本结构, 以便更容易将此脚本与并发脚本进行对比。

</> 示例 20.2: `flags.py`: 顺序下载脚本; 某些函数将被其他脚本复用

```

1 import time
2 from pathlib import Path
3 from typing import Callable
4
5 import httpx      ❶
6
7 POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
8             'MX PH VN ET EG DE IR TR CD FR').split() ❷
9
10 BASE_URL = 'https://www.fluentpython.com/data/flags' ❸
11 DEST_DIR = Path('downloaded') ❹
12
13 def save_flag(img: bytes, filename: str) -> None: ❺

```

²对于同时接收很多客户端访问的服务器来说, 差异还是有的: 协程的伸缩性更好, 因为协程使用的内存比线程少很多, 而且没有上下文切换的开销 (详见“[19.6.5 基于线程的方案不可靠](#)”).

```

14     (DEST_DIR / filename).write_bytes(img)
15
16     def get_flag(cc: str) -> bytes:
17         url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
18         resp = httpx.get(url, timeout=6.1,
19                           follow_redirects=True)
20         resp.raise_for_status()
21         return resp.content
22
23     def download_many(cc_list: list[str]) -> int:
24         for cc in sorted(cc_list):
25             image = get_flag(cc)
26             save_flag(image, f'{cc}.gif')
27             print(cc, end=' ', flush=True)
28         return len(cc_list)
29
30     def main(downloader: Callable[[list[str]], int]) -> None:
31         DEST_DIR.mkdir(exist_ok=True)
32         t0 = time.perf_counter()
33         count = downloader(POP20_CC)
34         elapsed = time.perf_counter() - t0
35         print(f'\n{count} downloads in {elapsed:.2f}s')
36
37     if __name__ == '__main__':
38         main(download_many)

```

① 导入 `httpx` 库。该库不在 Python 标准库中,因此按约定,在标准库模块之后导入,并与标准库之间空一行。

② 列出人口最多的 20 个国家的 ISO 3166 国家代码,按人口数量降序排列。

③ 存放国旗图片的 Web 目录。

④ 保存下载图片的本地目录。

⑤ 将字节序列 `img` 保存到目录 `DEST_DIR` 中,命名为 `filename`。

⑥ 根据给定的国家代码,构建 URL 并下载图片,返回响应中的二进制内容。

⑦ 为网络操作添加合适的超时,是一种很好的做法。这样可以避免莫名其妙的阻塞几分钟。

⑧ HTTPX 默认不跟踪重定向。

⑨ 此脚本未做异常处理,但是如果 HTTP 状态码不为 2XX,此方法会引发异常。强烈建议调用此方法,以避免异常后的悄无声息。

⑩ `download_many` 是与并发实现进行比较的关键函数。

⑪ 按字母顺序,循环遍历国家代码列表,以便在输出中保留顺序,返回已下载的国家代码数量。

⑫ 在同一行中,每次显示一个国家代码,以展示下载进度。`end=" "` 将常规的换行符替换为一个空格,以便在同一行依次显示各个国家代码。参数 `flush=True` 不可缺少,因为默认情况下,Python 输出是行缓冲的,即 Python 只在换行符后显示可打印字符。

⑬ 调用 `main` 函数时,必须传入执行下载任务的函数。这样就可以将 `main` 视为库函数,在 `threadpool` 和 `ascyncio` 示例中,为其传入其他 `download_many` 实现。

⑭ 如果需要,则创建 `DEST_DIR`;如果目录已存在,则不要引发异常。

⑯ 记录并报告运行 downloader 函数后的耗时。

⑰ 调用 main 函数,为其传入 download_many 函数。



HTTPX 库 的灵感来源于 requests 包,不过底层构建更符合现代化思想。更重要的是,HTTPX 同时提供了同步 API 与异步 API。因此,本章和下一章的所有 HTTP 客户端示例都可以使用 HTTPX 库提供的 API。Python 标准库提供了 `urllib.request` 模块,但它只提供了同步 API,对用户不够友好。

`flags.py` 脚本并无特别之处,只是作为与其他脚本对比的基线。而且,我将它当作一个库来使用,以避免实现其他脚本时重复编写代码。下面分析用 `concurrent.futures` 模块 实现的 `flags_threadpool.py`。

20.2.2 用 `concurrent.futures` 实现下载

`concurrent.futures` 包的功能主要由 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 类 提供,二者实现的 API 能将可调用对象(如函数)提交给不同的线程或进程来执行。这 2 个类在内部透明地管理着一个 worker 线程或进程,以及分配任务与收集结果的队列。不过,这些接口抽象程度较高,像本例中这种简单脚本,无须关心这些实现细节。

示例 20.3 展示了如何用 `ThreadPoolExecutor.map()` 方法,以最简单的方式实现并发下载。

</> 示例 20.3: `flags_threadpool.py`: 使用 `futures`,`ThreadPoolExecutor` 的线程版下载脚本

```
1  from concurrent import futures
2
3  from flags import save_flag, get_flag, main
4
5  def download_one(cc: str):
6      image = get_flag(cc)
7      save_flag(image, f'{cc}.gif')
8      print(cc, end=' ', flush=True)
9
10     return cc
11
12 def download_many(cc_list: list[str]) -> int:
13     with futures.ThreadPoolExecutor() as executor:
14         res = executor.map(download_one, sorted(cc_list))
15
16     return len(list(res))
17
18 if __name__ == '__main__':
19     main(download_many)
```

① 复用 flags 模块(见“[示例 20.2<597页>](#)”)中的一些函数。

② 下载单个图片的函数;这是每个 worker 将执行的内容。

③ 将 `ThreadPoolExecutor` 实例化为上下文管理器;`executor.__exit__` 方法将调用 `executor.shutdown(wait=True)`,它会阻塞直到所有线程都结束。

④ `map` 方法与内置函数 `map` 类似,不同之处在于:`download_one` 函数会被多个线程并发调用;它会返回

一个生成器,你可以遍历生成器来获取每次函数调用返回的值——在本例中,每次调用 `download_one` 都会返回一个国家代码。

- ⑤ 返回获取的结果数量。如果有线程中的函数调用引发了异常,则该异常会在 `list` 构造函数内的隐式 `next()` 尝试从 `executor.map` 返回的迭代器中获取返回值时,引发该异常。
- ⑥ 调用 `flags` 模块的 `main` 函数,为其传入并发版的 `download_many` 函数。

请注意,示例 20.3 中的 `download_one` 函数本质上就是“示例 20.2<597页>”中 `download_many` 函数内 `for` 循环的主体。这是编写并发代码时常见的重构方法:将顺序执行的 `for` 循环主体改写为并发调用的函数。



示例 20.3 的代码非常简短,因为我复用了 `flags.py` 脚本中的多个函数。`concurrent.futures` 的最大优势是:可以更方便地在原有顺序执行代码之上添加并发执行逻辑。

`ThreadPoolExecutor` 构造函数可接受多个参数,本例中未全部用到。其中,第 1 个参数 `max_workers` 最为重要,用于设置最多执行多少个 worker 线程。当 `max_workers` 为 `None` (默认值) 时,`ThreadPoolExecutor` 会使用下面的表达式来决定 `max_workers` 参数值 (自 Python 3.8 起):

```
1 max_workers = min(32, os.cpu_count() + 4)
```

`ThreadPoolExecutor` 文档解释了其中的原理:

该默认值可以为 I/O 密集型任务保留至少 5 个 worker 线程。对于释放了 GIL 的 CPU 密集型任务,最多可使用 32 个 CPU 核心。这样可以避免在多核设备上隐式使用大量资源。

现在, `ThreadPoolExecutor` 在启动 `max_workers` 个 worker 线程之前,也会重用空闲的 worker 线程。

总之,计算得到的 `max_workers` 默认值是合理的,`ThreadPoolExecutor` 可以避免不必要的启动新 worker。理解 `max_workers` 背后的逻辑,有助于您决定何时以及如何自行设置 `max_workers` 参数。

我用的库是 `concurrent.futures`,但在“示例 20.3<599页>”中却没有见到 `future` 对象,因此您可能想知道它们在哪里。下一节,将解释此问题。

20.2.3 future 对象在何处

`futures` 对象是 `concurrent.futures` 与 `asyncio` 的核心组件,但作为这些库的用户,我们有时却见不到 `futures` 对象。“示例 20.3<599页>”在背后用到了 `future` 对象,但是我编写的代码并没有直接使用它。本节将概述 `future` 对象,并通过一个示例展示 `future` 对象的实际应用。

从 Python 3.4 开始,标准库中有 2 个名为 `Future` 的类:`concurrent.futures.Future` 与 `asyncio.Future`。二者作用相同,这 2 个类的实例都表示可能已完成或可能未完成的延迟计算。这有点类似于 Twisted 中的 `Deferred` 类、Tornado 中的 `Future` 类以及现代 JavaScript 中的 `Promise`。

`future` 对象封装了待处理的操作,这样就可以将它们放入队列中,检查它们是否已完成,并在得到结果(或引发异常)时检索结果(或异常)。

但要记住,不应手动创建 future 对象,只应由并发框架(如 `concurrent.futures` 或 `asyncio`)实例化。原因很简单:future 对象表示最终将执行的操作,必须排期运行,而这正是并发框架的职责。具体而言,只有将可调用对象(如函数)提交给某个 `concurrent.futures.Executor` 子类执行时,才会创建 `concurrent.futures.Future` 实例。例如, `Executor.submit()` 方法接受一个可调用对象,排期执行,并返回一个 Future 实例。

应用代码不应改变 future 对象的状态,并发框架会在 future 对象代表的延迟计算结束之后,改变 future 对象的状态,而我们无法掌控计算何时结束。

两种类型的 future 对象都提供了 `.done()` 方法,该方法是非阻塞的,并返回一个布尔值,指明 future 对象内封装的可调用对象是否已执行。而客户端代码通常无须反复询问 future 是否完成,只须等待通知即可。这就是为何 2 种 future 对象都提供 `.add_done_callback()` 方法。为 `.add_done_callback()` 提供一个可调用对象,当 future 对象执行完毕时,该可调用对象将以 future 对象作为唯一参数被调用。请注意,回调的可调用对象与封装在 future 中的函数,将在同一 worker 线程(或进程)中运行。

此外,还有一个 `.result()` 方法,该方法在 2 种 future 对象中的行为,取决于 future 对象的完成状态:

- 当 future 对象执行完毕时

此时 `.result()` 方法在 2 种 future 对象中的行为是一样的,该方法返回可调用对象(如函数)的执行结果,或者重新引发在可调用对象执行期间可能引发的异常。

- 当 future 对象未执行完毕时

此时 `.result()` 方法在 2 种 future 对象中的行为差异较大:

- 在 `concurrency.futures.Future` 实例中,调用 `f.result()` 会阻塞调用者的线程,直到结果准备就绪。

可以为 `result()` 指定一个可选的超时参数,如果在指定时间内 future 未完成, `result()` 方法会引发 `TimeoutError` 异常。

- `asyncio.Future.result()` 方法不支持超时, `await` 是从 `asyncio` 中获取 future 结果的首选方式,但 `await` 不能与 `concurrency.futures.Future` 实例一起使用。

这 2 个库中有多个函数都返回 future 对象,还有一些函数以对用户透明的方式在实现中使用 future 对象。“[示例 20.3<599页>](#)”中的 `executor.map` 属于后者,它返回一个迭代器,其中 `__next__` 调用了每个 future 对象的 `result()` 方法,因此得到的是 future 对象的结果,而不是 future 对象本身。

为了从实用角度理解 future 对象,可以用 `concurrent.futures.as_completed` 函数重写“[示例 20.3<599页>](#)”。该函数的参数是一个由 future 对象构成的可迭代对象,返回值是一个迭代器——在 future 对象执行结束后,生成 future 对象。

若要使用 `concurrent.futures.as_completed` 函数,需要更改 `download_many` 函数,将 `executor.map` 调用换成 2 个 for 循环:一个用于创建并排期 future 对象,另一个用于获取 future 对象的执行结果。同时,添加了几个 `print` 调用,以显示运行结束前后的 future 对象。重构的 `download_many` 函数如[示例 20.4](#)所示,代码量由 5 行增加到 17 行。不过,现在能一窥神秘的 future 对象了。其他函数保持不变,与“[示例 20.3<599页>](#)”中的一样。

</> **示例 20.4: `flags_threadpool_futures.py`:用 `futures.as_completed` 重构的 `download_many` 函数**

```

1  def download_many(cc_list: list[str]) -> int:
2      cc_list = cc_list[:5]
3      with futures.ThreadPoolExecutor(max_workers=3) as executor: ❶
4          to_do: list[futures.Future] = []
5          for cc in sorted(cc_list): ❷

```

```

6     future = executor.submit(download_one, cc)           ④
7     to_do.append(future)          ⑤
8     print(f'Scheduled for {cc}: {future}')           ⑥
9
10    for count, future in enumerate(futures.as_completed(to_do), 1): ⑦
11        res: str = future.result()  ⑧
12        print(f'{future} result: {res!r}')           ⑨
13
14    return count

```

- ① 本次演示仅使用人口最多的 5 个国家。
- ② 将 max_workers 设置为 3,以便在输出中观察待处理的 future 对象。
- ③ 按字母顺序遍历国家代码,以强调返回的结果是无序的。
- ④ executor.submit() 排期要执行的可调用对象,并返回一个 future 对象,表示待执行的操作。
- ⑤ 存储各个 future 对象,以便稍后可用 as_completed() 提取。
- ⑥ 显示一条消息,包含国家代码与对应的 future 对象。
- ⑦ 当 future 对象执行完毕时,as_completed() 函数会生成 (yield) 该 future 对象。
- ⑧ 获取 future 对象的结果。
- ⑨ 显示 future 对象及其结果。

请注意,在此示例中,future.result() 调用永远不会阻塞,因为 future 对象由 as_completed() 函数生成。运行 [示例 20.4](#) 得到的结果,如[示例 20.5](#)所示。

```

</>示例 20.5: flags_threadpool_futures.py 的输出

$ python3 flags_threadpool_futures.py
Scheduled for BR: <Future at 0x100791518 state=running> ①
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending> ②
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN' ③
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR' ④
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'
5 downloads in 0.70s

```

- ① 按国家代码的字母顺序,排序 future 对象。future 对象的 repr() 方法显示了 future 对象的状态:前 3 个 future 对象的状态是 running,因为有 3 个 worker 线程。
- ② 后 2 个 future 对象的状态是 pending,等待可用的 worker 线程。
- ③ 此行的第 1 个 CN 是由运行在 worker 线程中的 download_one 函数输出的,余下内容是由 download_many 函数输出的。
- ④ 此处有 2 个线程输出国家代码 (BR、ID),然后主线程中的 download_many 函数可以显示第 1 个线程的结果。



建议运行几次脚本 `flags_threadpool_futures.py` 试试。多次运行, 您会发现每次结果的顺序都会不同。若将参数 `max_workers` 的值增大到 5, 那么结果的顺序变化更大。将参数 `max_workers` 的值设为 1, 将使脚本按序执行, 结果的顺序始终与调用 `submit` 的顺序一致。

我们分析了 2 个用 `concurrent.futures` 实现的下载脚本: 使用 `ThreadPoolExecutor.map()` 方法的示例 20.3 与使用 `futures.as_completed()` 函数的示例 20.4。如果对 `flags asyncio.py` 脚本的源码感到好奇, 可以看一下“二十一 异步编程”的“示例 21.3<625页>”。

接下来, 简单说明在 CPU 密集型作业中, 如何用 `concurrent.futures` 轻松绕过 GIL。

20.3 用 concurrent.futures 启动进程

`concurrent.futures` 文档的副标题是“启动并行任务”, 该模块支持在多核设备上进行并行计算, 因为它支持用 `ProcessPoolExecutor` 类在多个 Python 进程之间分配工作。

`ProcessPoolExecutor` 类与 `ThreadPoolExecutor` 类都实现了 `Executor` 接口, 因此可以用 `concurrent.futures` 模块 轻松地将基于线程的方案转换成基于进程的方案。

下载国旗图片的示例等 I/O 密集型作业, 使用 `ProcessPoolExecutor` 类 并得不到什么优势提升。要验证这一结论也很容易, 只需将“示例 20.3<599页>”的这几行:

```
1 def download_many(cc_list: list[str]) -> int:
2     with futures.ThreadPoolExecutor() as executor:
```

更改成:

```
1 def download_many(cc_list: list[str]) -> int:
2     with futures.ProcessPoolExecutor() as executor:
```

`ProcessPoolExecutor` 类 的构造函数也有一个 `max_workers` 参数, 默认为 `None`。此时, 执行器会将 Worker 的数量限制为 `os.cpu_count()` 返回的数值。

与线程相比, 进程耗费的内存更多, 启动时间更长。因此, `ProcessPoolExecutor` 的价值在 CPU 密集型作业中才能体现出来。接下来, 用 `concurrent.futures` 模块 重构“19.6 自研进程池”中的素数检测器。

20.3.1 终极版的多核素数检测器

“19.6.3 多核素数检测器代码”中, 我们分析了 `procs.py` (见“示例 19.13<579页>”), 这是一个用 `multiprocessing` 做素数检测的脚本。示例 20.6 中的 `proc_pool.py` 脚本使用 `ProcessPoolExecutor` 解决了同样的问题。`procs.py` 共包含 43 行代码 (不含空行), 而 `proc_pool.py` 只有 31 行, 减少约 28%。

<> 示例 20.6: `proc_pool.py`; 用 `ProcessPoolExecutor` 重写 `procs.py`

```
1 import sys
2 from concurrent import futures
3 from time import perf_counter
4 from typing import NamedTuple
```

```

5
6     from primes import is_prime, NUMBERS
7     class PrimeResult(NamedTuple):      ❷
8         n: int
9         flag: bool
10        elapsed: float
11
12    def check(n: int) -> PrimeResult:
13        t0 = perf_counter()
14        res = is_prime(n)
15        return PrimeResult(n, res, perf_counter() - t0)
16
17    def main() -> None:
18        if len(sys.argv) < 2:
19            workers = None          ❸
20        else:
21            workers = int(sys.argv[1])
22
23    executor = futures.ProcessPoolExecutor(workers)          ❹
24    actual_workers = executor._max_workers # type: ignore ❺
25
26    print(f'Checking {len(NUMBERS)} numbers with {actual_workers} processes:')
27
28    t0 = perf_counter()
29
30    numbers = sorted(NUMBERS, reverse=True)                  ❻
31    with executor:                                         ❼
32        for n, prime, elapsed in executor.map(check, numbers): ❽
33            label = 'P' if prime else ' '
34            print(f'{n:16} {label} {elapsed:9.6f}s')
35
36    time = perf_counter() - t0
37    print(f'Total time: {time:.2f}s')
38
39    if __name__ == '__main__':
40        main()

```

- ❶ 无需导入 multiprocessing、SimpleQueue 等,由 concurrent.futures 模块 在背后完成导入。
- ❷ PrimeResult 元组与 check 函数,同 procs.py 脚本(见“示例 19.13<579页>”中的一样。但现在不再需要那些队列与 worker 函数了。
- ❸ 未提供命令行参数时,将 workers 设置为 None,由 ProcessPoolExecutor 决定 worker 进程数。
- ❹ 在 with 块(❽处)之前构建 ProcessPoolExecutor 实例,以便可在下一行(❺处)显示实际的 worker 进程数量。
- ❺ _max_workers 是 ProcessPoolExecutor 的实例属性(无文档记载)。我决定用它来显示(workers 变量为 None 时) worker 进程数量。不出所料,Mypy 报错了。因此,我加上了“# type: ignore”注释,用于消除 Mypy 报错。
- ❻ 降序排列要检查的数字。这将暴露 prod_pool.py 与 procs.py 的行为差异。详见示例后文的说明。

⑦ 用 ④ 处构建的 executor 对象, 作为上下文管理器。

⑧ executor.map() 会以与参数 numbers 相同的顺序, 返回由函数 check() 返回的 PrimeResult 实例。

运行示例 20.6 的 proc_pool.py, 得到的结果将严格按 (待检数字) 降序排列, 如示例 20.7 所示。相比之下, procs.py 的输出顺序 (见“19.6.1 基于进程的方案<577页>”) 则取决于各个数字的素数检测难度。例如, procs.py 现在靠近顶部位置显示了 7, 777, 777, 777, 777, 777 的检测结果, 因为它有一个较小的因子 7, 所以 is_prime 函数很快就能确定它不是素数。

相比之下, 7, 777, 777, 536, 340, 681 是 $88,191,709^2$, 因此 is_prime 需要耗费更长时间才能确定它是一个合数, 而确定 77, 777, 777, 777, 753 是素数则需要更长的时间。因此, 这 2 个数字都出现在 procs.py 输出的末尾。

运行 proc_pool.py, 不仅会看到输出结果按待检数字降序排列, 还会看到脚本在显示 9, 999, 999, 999, 999, 999 的结果后似乎卡住了。

</> 示例 20.7: proc_pool.py 的输出

```
1 $ ./proc_pool.py
2 Checking 20 numbers with 12 processes:
3 9999999999999999  0.000024s ①
4 9999999999999917  P 9.500677s ②
5 7777777777777777  0.000022s ③
6 7777777777777753  P 8.976933s
7 777777536340681   8.896149s
8 6666667141414921  8.537621s
9 6666666666666719  P 8.548641s
10 6666666666666666  0.000002s
11 5555555555555555  0.000017s
12 555555555555503   P 8.214086s
13 555553133149889  8.067247s
14 4444444488888889  7.546234s
15 4444444444444444  0.000002s
16 444444444444423   P 7.622370s
17 3333335652092209  6.724649s
18 3333333333333333  0.000018s
19 333333333333301   P 6.655039s
20 299593572317531   P 2.072723s
21 142702110479723   P 1.461840s
22                      2 P 0.000001s
23 Total time: 9.65s
```

① 这一行很快就显示。

② 这一行耗时 9.5s 才显示。

③ 余下各行几乎都立即显示。

如下是 proc_pool.py 出现这种行为的原因:

- 如前所述, executor.map(check, numbers) 会按 numbers 中待检数字的顺序, 返回结果。
- proc_pool.py 默认使用的 worker 进程数与 CPU 核数相等——这是 max_workers 为 None 时, ProcessPoolExecutor 的行为。在我的笔记本电脑中, 是 12 个进程。

- 由于 numbers 中的数字是降序提交的,所以检测的第一个数字是 9,999,999,999,999,999,999。该数字的因子是 9,得到检测结果的速度很快。
- 第 2 个数字是 99,999,999,999,917,是待检数字中最大的素数。此数字的检测耗时最长。
- 与此同时,其余 11 个进程将检测其他数字——可能是素数、因子较大的合数、因子较小的合数。
- 当负责 99,999,999,999,917 的 worker 进程最终确定这是素数时,所有其他 worker 进程都已完成其最后一项工作,因此检测结果会立即显示出来。



尽管 proc_pool.py 的处理过程不像 procs.py 那样明显,但对于相同数量的 worker 与 CPU 核心,总体耗时基本与“图 19.2<582 页>”所示的相同。

要了解并发程序的行为不容易,下面是第 2 个实验,或许能帮助你直观理解 Executor.map 的运行。

20.4 实验:Executor.map 方法

本节将研究 Executor.map()方法:用包含 3 个 worker 线程的 ThreadPoolExecutor 实例,运行 5 个可调用对象,输出带有时间戳的消息。代码如示例 20.8 所示,输出如示例 20.4 所示。

</> 示例 20.8: demo_executor_map.py:简单演示 ThreadPoolExecutor 类的 map 方法

```

1  from time import sleep, strftime
2  from concurrent import futures
3
4  def display(*args):      ❶
5      print(strftime('[%H:%M:%S]'), end=' ')
6      print(*args)
7
8  def loiter(n):          ❷
9      msg = '{}loiter({}): doing nothing for {}s...'
10     display(msg.format('\t'*n, n, n))
11     sleep(n)
12     msg = '{}loiter({}): done.' ❸
13     display(msg.format('\t'*n, n))
14     return n * 10
15
16 def main():
17     display('Script starting.')
18     executor = futures.ThreadPoolExecutor(max_workers=3) ❹
19     results = executor.map(loiter, range(5)) ❺
20     display('results:', results)
21     display('Waiting for individual results.')
22     for i, result in enumerate(results): ❻
23         display(f'result {i}: {result}')
24
25 if __name__ == '__main__':
26     main()

```

- ❶ 函数 `display` 只需打印所接收的参数,并在参数前加上格式为 [HH:MM:SS] 的时间戳。
- ❷ 函数 `loiter` 只在脚本开始时显示一条消息,然后休眠 n 秒,最后在脚本结束时再显示一条消息。消息用制表符进行缩进,缩进量由参数 n 确定。
- ❸ 函数 `loiter` 返回 $n \times 10$,以便让我们了解收集结果的方式。
- ❹ 创建一个包含 3 个 worker 线程的 `ThreadPoolExecutor` 实例
- ❺ 向 `executor` 提交 5 个任务。因为只有 3 个 worker 线程,所以只有 3 个任务会立即开始: `loiter(0)`、`loiter(1)`、`loiter(2)`。`executor.map()` 是一个非阻塞调用。
- ❻ 立即显示调用 `executor.map()` 的结果——一个生成器,如示例 20.4 中❻处的输出所示。
- ❼ `for` 循环中的 `enumerate()` 将隐式调用 `next(results)`,而 `next(results)` 又在(内部)代表第 1 个任务(`loiter(0)`)的 `future` 对象 `_f` 上调用 `_f.result()`。`result()` 方法会阻塞,直至 `future` 对象运行结束,因此这个 `for` 循环每次遍历都必须等待下一个 `result` 就绪。

我建议您运行一下示例 20.8 中的 `demo_executor_map.py` 脚本,观察任务结果是不是逐个更新。此外,还可以修改 `ThreadPoolExecutor` 的 `max_workers` 参数,以及 `executor.map()` 方法中 `range` 函数的参数值,或将 `range` 替换为精心挑选的值列表,以得到不同的延迟。

运行示例 20.8 的某次输出结果,如示例 20.4 所示:

```

1 $ python3 demo_executor_map.py
2 [15:56:50] Script starting.          ❶
3 [15:56:50] loiter(0): doing nothing for 0s...  ❷
4 [15:56:50] loiter(0): done.
5 [15:56:50]     loiter(1): doing nothing for 1s...  ❸
6 [15:56:50]             loiter(2): doing nothing for 2s...
7 [15:56:50] results: <generator object result_iterator at 0x106517168> ❹
8 [15:56:50]                 loiter(3): doing nothing for 3s...  ❺
9 [15:56:50] Waiting for individual results:
10 [15:56:50] result 0: 0               ❻
11 [15:56:51]     loiter(1): done.    ❼
12 [15:56:51]                 loiter(4): doing nothing for 4s...
13 [15:56:51] result 1: 10            ❽
14 [15:56:52]             loiter(2): done.  ❾
15 [15:56:52] result 2: 20
16 [15:56:53]                 loiter(3): done.
17 [15:56:53] result 3: 30
18 [15:56:55]                 loiter(4): done.  ❿
19 [15:56:55] result 4: 40

```

- ❶ 本次运行,从 15:56:50 开始。
- ❷ 第 1 个线程执行 `loiter(0)`,因此休眠 0 秒,甚至可能在第 2 个线程开始前就结束,但视情况而定³。
- ❸ `loiter(1)` 与 `loiter(2)` 会立即启动(因为线程池中有 3 个 worker 线程,所以可以同时运行 3 个函数)
- ❹ 这说明 `executor.map()` 返回的结果是一个生成器;到目前为止,无论任务数与 `max_workers` 是多少, `executor.map()` 都不会阻塞。

³ 视情况而定(Your mileage may vary, YMMV):对于线程来说,您永远无法知道本应几乎同时发生的事件的实际顺序;在另一台设备上, `loiter(1)` 有可能在 `loiter(0)` 结束之前就开始了,这是因为 `sleep` 函数总是会释放 GIL。所以,即使 `sleep(0)`,Python 也可能会切换到另一个线程。

- ⑤ 由于 `loiter(0)` 已完成, 所以第 1 个 worker 现在可以为运行 `loiter(3)`, 而启动第 4 个线程。
- ⑥ 这是执行可能会阻塞的地方, 具体取决于传给 `loiter(n)` 的参数: `results` 生成器的 `__next__` 方法必须等待, 直至第一个 `future` 运行结束。此时不会阻塞, 因为 `loiter(0)` 在 `for` 循环开始前就已运行结束。请注意, 到目前为止的所有操作都发生于同一时刻: 15:56:50。
- ⑦ 1 秒后(即 15:56:51), `loiter(1)` 运行结束。线程可以被释放, 以启动 `loiter(4)`。
- ⑧ 显示 `loiter(1)` 的结果: 10。现在, `for` 循环将阻塞, 等待 `loiter(2)` 的运行结果。
- ⑨ 同上: `loiter(2)` 于 15:56:52 运行结束, 显示其结果: 20。`loiter(3)` 也是如此, 于 15:56:53 运行结束。
- ⑩ 2 秒后(即 15:56:55), `loiter(4)` 运行结束。因为 `loiter(4)` 在 15:56:51 开始, 并空等了 4 秒。

虽然函数 `Executor.map()` 很易用, 但是通常更希望在任务执行结束时, 就获取其运行结果, 而不管任务以何种顺序提交。为此, 要将 `Executor.submit()` 方法与 `futures.as_completed()` 函数结合使用, 如“[示例 20.4<601页>](#)”所示。“[20.5.2 使用 `futures.as_completed` 函数<614页>](#)”将再次回顾这一技术。



`Executor.submit()` 方法与 `futures.as_completed()` 函数的组合, 比 `Executor.map()` 更灵活。因为 `Executor.submit()` 方法可提交不同的可调用对象与参数, 而 `Executor.map()` 旨在用不同的参数调用同一个可调用对象。此外, 传给 `futures.as_completed()` 函数的一系列 `future` 对象可以来自多个 `Executor` 实例——可能某些 `future` 对象由 `ThreadPoolExecutor` 实例创建, 而另一些由 `ProcessPoolExecutor` 实例创建。

下一节, 将根据新需求继续实现下载国旗图片的示例。新需求将迫使我们不能用 `Executor.map()` 方法, 而是必须遍历 `futures.as_completed()` 函数的结果。

20.5 显示下载进度并处理错误

如“[20.2 并发网络下载](#)”所述, 为了便于阅读和结构对比, 3 种下载方案(顺序执行、多线程并发、异步并发)的脚本均未包含异常处理。

为了测试对各种错误的处理, 我创建了 `flags2` 示例:

- `flags2_common.py`

该模块包含所有 `flags2` 示例使用的通用函数与设置, 如 `main` 函数, 负责命令行解析、计时、报告结果。

该脚本中的代码用于提供支持, 与本章主题无直接关系, 因此书中未列其源码。但可在[随书源码库](#)中阅读([20-executors/getflags/flags2_common.py](#))。

- `flags2_sequential.py`

顺序执行版 HTTP 下载客户端, 能正确处理错误与显示下载进度条。`flags2_threadpool.py` 脚本复用了本模块中的 `download_one` 函数。

- `flags2_threadpool.py`

并发执行版 HTTP 下载客户端, 能正确处理错误与显示下载进度条。基于 `futures.ThreadPoolExecutor` 类实现。

- `flags2_asyncio.py`

异步执行版 HTTP 下载客户端, 能正确处理错误与显示下载进度条。基于 `asyncio` 和 `httpx` 实现。此脚本将在“[21.7 增强 `asyncio` 下载器<628页>](#)”中进行分析。

测试并发客户端要谨慎



在互联网上测试并发 HTTP 客户端要谨慎, 因为每秒可能发起许多请求, 这相当于 Dos 攻击。在访问公共服务器时, 要对 HTTP 客户端限流。在测试 HTTP 客户端时, 应架设本地 HTTP 服务器, 方法详见后文标记栏 “[搭建测试服务器](#)”。

flags2 系列示例最明显的特征是, 增加了用 `tqdm` 包实现的文本动画下载进度条。我在 YouTube 上发布了一段 108 秒的视频, 以显示进度条并对比三个 flags2 脚本的速度。在视频中, 我从顺序版脚本 (`flags2_sequential.py`) 开始, 但在 32 秒后中断, 因为访问 676 个 URL 并下载 194 个国旗图片需要耗时 5 分钟以上。然后, 运行并发执行版 (`flags2_threadpool.py`) 与异步执行版 (`flags2_asyncio.py`) 各 3 次, 每次都能在 6 秒内完成下载任务 (即速度快 60 倍以上)。图 20.1 显示了 `flags2_threadpool.py` 运行中与运行后的 2 张屏幕截图。

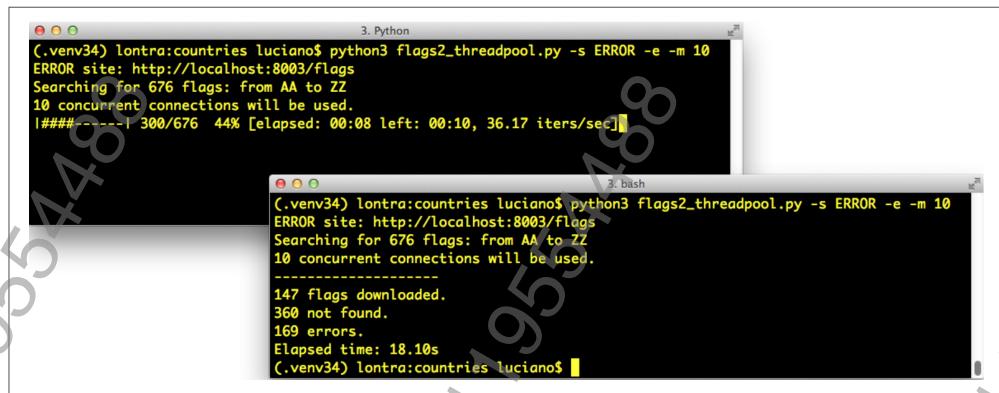


图 20.1: 左上: `flags2_threadpool.py` 运行中, 显示 `tqdm` 包生成的进度条
右下: 同一个终端窗口, 脚本运行完毕后

`tqdm` 项目的 `README.md` 文件中有一个.gif 动画, 演示了 `tqdm` 的最简单用法。安装 `tqdm` 之后, 在 Python 控制台中输入以下代码, 将在行末 (❶处) 看到进度条动画:

```

1 >>> import time
2 >>> from tqdm import tqdm
3 >>> for i in tqdm(range(1000)):
4     ... time.sleep(.01)
5 ...
6 33% |██████████| 330/1000 [00:04<00:06, 94.60it/s]
❶

```

除了效果精妙之外, `tqdm.tqdm()` 函数 的实现方式很独特: 它消耗任意可迭代对象, 并生成一个迭代器, 在消耗迭代器的同时, 显示进度条并预估完成所有迭代所需的剩余时间。为了计算剩余时间, `tqdm.tqdm()` 函数 需要接收一个能用 `len()` 确定其长度的可迭代对象, 或通过参数 `total` 指定可迭代对象中预期的项数。将 `tqdm` 与 flags2 示例集成后, 可以更深入地了解并发脚本的工作原理。集成 `tqdm` 后, 必须使用 `future.as_completed()` 和 `asyncio.as_completed()` 函数, 才能使 `tqdm.tqdm()` 函数 在每个 `future` 对象运行结束后, 更新脚本执行进度。

flags2 系列示例的另一个特征是提供了命令行接口, 3 个脚本都接受相同的选项, 运行任意脚本并指定 `-h` 选项, 即可列出接受的所有选项。如示例 20.9 所示。

</> 示例 20.9: flags2 系列脚本的帮助界面

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
                            [-v]
                            [CC [CC ...]]

Download flags for country codes. Default: top 20 countries by population.

positional arguments:
  CC                  country code or 1st letter (eg. B for BA...BZ)

optional arguments:
  -h, --help           show this help message and exit
  -a, --all            get all available flags (AD to ZW)
  -e, --every          get flags for every possible code (AA...ZZ)
  -l N, --limit N     limit to N first codes
  -m CONCURRENT, --max_req CONCURRENT
                      maximum concurrent requests (default=30)
  -s LABEL, --server LABEL
                      Server to hit; one of DELAY, ERROR, LOCAL, REMOTE(default=
  LOCAL)
  -v, --verbose        output detailed progress info
```

所有选项都是可选的。但选项 `-s`/`--server` 对于测试至关重要, 可指定测试时使用的 HTTP 服务器与端口。

该选项可设置为如下四者之一 (不区分大小写), 用于确定脚本从何处下载国旗图片。

- LOCAL

使用 `http://localhost:8000/flags`, 这是默认设置。需要搭建一个本地 HTTP 服务器, 响应 8000 端口的请求。服务器搭建说明见下方标记栏 “[搭建测试服务器](#)”。

- REMOTE

使用 `http://fluentpython.com/data/flags`, 这是我托管在共享服务器上的网站。请不要用太多并发请求访问这个网站。该网站域名由 [Cloudflare CDN](#) 提供加速, 因此首次下载可能稍慢, 但一旦 CDN 有了缓存, 速度就会变快。

- DELAY

使用 `http://localhost:8000/flags`, 这个服务器会延迟 HTTP 响应, 监听端口为 8001。为了便于实验, 我编写了脚本 `slow_server.py`, 放在随书代码库的 `20-executors/getflags` 目录中。服务器搭建说明见下方附注栏 “[搭建测试服务器](#)”。

- ERROR

使用 `http://localhost:8002/flags`, 这个服务器监听 8002 端口, 返回一些 HTTP 错误。服务器搭建说明见下方标记栏 “[搭建测试服务器](#)”。

搭建测试服务器

若您没有供测试使用的本地 HTTP 服务器, 可以参照我编写的“搭建说明”([20-executors/getflags/README.adoc](#))来搭建。该说明仅用到了 Python ≥ 3.9 标准库, 未用到外部库。简而言之, [README.adoc](#) 描述了如何使用如下 3 个 HTTP 服务器:

- `python3 -m http.server`
监听 8000 端口的 LOCAL 服务器。
- `python3 slow_server.py`
监听 8001 端口的 DELAY 服务器, 会在每次响应客户端之前随机延迟 0.5~5 秒。
- `python3 slow_server.py 8002 --error-rate .25`
监听 8002 端口的 ERROR 服务器, 除了随机延迟之外, 还有 25% 的概率返回“418 I'm a teapot”错误响应。

默认情况下, 每个 `flags2*.py` 脚本都将使用默认的并发连接数(各脚本有所不同)从本地服务器(`http://localhost:8000/flags`)获取人口最多的 20 个国家的国旗图片。[示例 20.10](#) 显示了使用全部默认值运行 `flags2_sequential.py` 脚本得到的输出。运行此脚本前, 需按标记栏“**测试并发客户端要谨慎**”的说明, 搭建本地 HTTP 服务器。

</> [示例 20.10](#): 用默认值(LOCAL 服务器、20 个国旗、1 个并发)运行 `flags2_sequential.py` 脚本

```
1 $ python3 flags2_sequential.py
2 LOCAL site: http://localhost:8000/flags
3 Searching for 20 flags: from BD to VN
4 1 concurrent connection will be used.
5 -----
6 20 flags downloaded.
7 Elapsed time: 0.10s
```

可通过多种方式选择要下载的国旗。[示例 20.11](#) 展示了如何下载国家代码以 A、B 或 C 开头的国旗图片。

</> [示例 20.11](#): 运行 `flags2_sequential.py`: 从 DELAY 服务器获取国家代码前缀为 A、B 或 C 的国旗

```
1 $ python3 flags2_threadpool.py -s DELAY a b c
2 DELAY site: http://localhost:8001/flags
3 Searching for 78 flags: from AA to CZ
4 30 concurrent connections will be used.
5 -----
6 43 flags downloaded.
7 35 not found.
8 Elapsed time: 1.72s
```

无论如何选择国家代码, 都可以用 `-l/-limit` 选项限制下载国旗图片的数量。[示例 20.12](#) 展示了如何发起 100 个请求, 结合 `-a` 与 `-l` 选项下载 100 个国旗图片。

</> [示例 20.12](#): 运行 `flags2_sequential.py`: 并发 100 (-m 100) 从 ERROR 服务器下载 100 个图片 (-al 100)

```

1 $ python3 flags2_asyncio.py -s ERROR -al 100 -m 100ERROR site: http://localhost
2 :8002/flags
3 Searching for 100 flags: from AD to LK
4 100 concurrent connections will be used.
5 -----
6 73 flags downloaded.
7 27 errors.
8 Elapsed time: 0.64s

```

以上是 flags2 系列示例的用户界面。接下来,分析这些脚本的实现方式。

20.5.1 flags2 示例中的错误处理

3 个示例使用相同的策略处理 HTTP 错误: 404 错误 (未找到) 由负责下载单个文件的函数 (download_one) 处理,而其他错误都会向上传播 (冒泡),交给 download_many 函数或 supervisor 协程 (asyncio 版) 处理。

如前所述,还是从顺序执行版脚本 (flags2_sequential.py) 开始分析,该版本的代码更易于理解,并且并发线程执行版 (flags2_threadpool.py) 复用了这里的大部分代码。示例 20.5.1 展示了 flags2_sequential.py 与 flags2_threadpool.py 脚本中真正执行下载的函数。

```

1 from collections import Counter
2 from http import HTTPStatus
3
4 import httpx
5 import tqdm # type: ignore ❶
6
7 from flags2_common import main, save_flag, DownloadStatus ❷
8
9 DEFAULT_CONCUR_REQ = 1
10 MAX_CONCUR_REQ = 1
11
12 def get_flag(base_url: str, cc: str) -> bytes:
13     url = f'{base_url}/{cc}/{cc}.gif'.lower()
14     resp = httpx.get(url, timeout=3.1, follow_redirects=True)
15     resp.raise_for_status() ❸
16     return resp.content
17
18 def download_one(cc: str, base_url: str, verbose: bool = False) -> DownloadStatus:
19     try:
20         image = get_flag(base_url, cc)
21     except httpx.HTTPStatusError as exc: ❹
22         res = exc.response
23         if res.status_code == HTTPStatus.NOT_FOUND:
24             status = DownloadStatus.NOT_FOUND ❺
25             msg = f'not found: {res.url}'
26         else:
27             raise ❻

```

```

28     else:
29         save_flag(image, f'{cc}.gif')      status = DownloadStatus.OK
30         msg = 'OK'
31
32     if verbose:                      ⑦
33         print(cc, msg)
34
35     return status

```

- ① 导入进度条显示库 `tqdm`, 告知 Mypy 跳过类型检查。⁴
- ② 从 `flags2_common` 模块导入 2 个函数和 1 个 `Enum`。
- ③ 若 HTTP 状态码不在 200~300 范围内, 则引发 `HTTPStatusError` 异常。
- ④ `download_one` 捕获 `HTTPStatusError` 异常, 专门处理 HTTP 404 错误 ...
- ⑤ ... 方法是, 将局部变量 `status` 设置为 `DownloadStatus.NOT_FOUND`; `DownloadStatus` 是从 `flags2_common` 模块 导入的一个 `Enum`。
- ⑥ 任何其他 `HTTPStatusError` 异常, 都会被重新引发, 向上传播(冒泡)给调用方。
- ⑦ 若命令行中设置了 `-v/-verbose` 选项, 则显示国家代码与状态信息, 即详细模式中看到的进度信息。

示例 20.13 展示了顺序执行版 `download_many()` 函数。这段代码虽然简洁, 但有必要研究一下, 以便稍后与并发版 `download_many()` 进行对比。关注重点是 `download_many()` 函数如何报告进度、处理错误和统计下载量。

</> 示例 20.13: `flags2_sequential.py`: 顺序执行版 `download_many()` 函数的实现

```

1  def download_many(cc_list: list[str],
2                     base_url: str,
3                     verbose: bool,
4                     _unused_concur_req: int) -> Counter[DownloadStatus]:
5     counter: Counter[DownloadStatus] = Counter()           ①
6     cc_iter = sorted(cc_list)                            ②
7     if not verbose:
8         cc_iter = tqdm.tqdm(cc_iter)          ③
9     for cc in cc_iter:
10        try:
11            status = download_one(cc, base_url, verbose)    ④
12        except httpx.HTTPStatusError as exc:                ⑤
13            error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
14            error_msg = error_msg.format(resp=exc.response)
15        except httpx.RequestError as exc:                  ⑥
16            error_msg = f'{exc} {type(exc)}' strip()
17        except KeyboardInterrupt:                         ⑦
18            break
19        else:
20            error_msg = ''                                ⑧
21
22        if error_msg:

```

⁴ 截至 2021 年 9 月, `tqdm` 包还未提供类型提示。不过没关系, 世界不会因此而毁灭。Python 中的类型是可选的, 感谢 Guido 提供这一功能!

```

23     status = DownloadStatus.ERROR          ⑨
24     counter[status] += 1                  ⑩
25     if verbose and error_msg:
26         print(f'{cc} error: {error_msg}')  ⑪
27
28     return counter                      ⑫

```

- ❶ 这个 Counter 实例用于统计不同的下载状态: DownloadStatus.OK、DownloadStatus.NOT_FOUND 或 DownloadStatus.ERROR。
- ❷ cc_list 保存通过参数传入的国家代码列表, 按字母升序排列。
- ❸ 若未运行在详细模式下, 将 cc_iter 传给 tqdm.tqdm() 函数, 返回一个迭代器, 生成 cc_iter 中的项, 同时显示进度条动画。
- ❹ 不断调用 download_one() 函数。
- ❺ 由 get_flag() (见示例 20.5.1) 引发, 但未被 download_one() 处理的 HTTPStatusError 异常, 将在此处处理。
- ❻ 其他与网络相关的 RequestError 异常, 也在此处处理。除此之外的异常会终止脚本, 因为调用 download_many() 的 flags2_common.main() 函数中, 没有 try/except 块。
- ❼ 若用户按下 Ctrl-C 键, 则退出循环。
- ❽ 若 download_one() 函数未用引发异常, 则清空错误消息。
- ❾ 若有错误, 则将局部变量 status 设置为相应状态。
- ❿ 递增相应状态 (status) 的计数器。
- ❾ 如果在详细模式下, 并且有错误, 则显示带有当前国家代码的错误消息。
- ❿ 返回 counter 对象, 以便 main 函数能在最终的报告中显示统计数据。

接下来, 分析重构后的线程池版国旗下载脚本——flags2_threadpool.py。

20.5.2 使用 futures.as_completed 函数

为了集成 tqdm 进度条, 并处理每个请求中的错误, flags2_threadpool.py 脚本用到了 ThreadPoolExecutor 类 与 asyncio.as_completed() 函数。flags2_threadpool.py 的完整代码如示例 20.14 所示, 该脚本仅实现了 download_many 函数, 其他函数复用于 flags2_common.py 与 flags2_sequential.py。

</> 示例 20.14: flags2_threadpool.py: 完整代码清单

```

1  from collections import Counter
2  from concurrent.futures import ThreadPoolExecutor, as_completed
3
4  import httpx
5  import tqdm # type: ignore
6
7  from flags2_common import main, DownloadStatus
8  from flags2_sequential import download_one ❶
9
10 DEFAULT_CONCUR_REQ = 30                      ❷
11 MAX_CONCUR_REQ = 1000                         ❸

```

```
12
13 def download_many(cc_list: list[str],                                base_url: str,
14                     verbose: bool,
15                     concur_req: int) -> Counter[DownloadStatus]:
16     counter: Counter[DownloadStatus] = Counter()
17     with ThreadPoolExecutor(max_workers=concur_req) as executor: ❸
18         to_do_map = {} ❹
19         for cc in sorted(cc_list): ❺
20             future = executor.submit(download_one, cc,
21                                       base_url, verbose)
22             to_do_map[future] = cc ❻
23         done_iter = as_completed(to_do_map) ❻
24         if not verbose:
25             done_iter = tqdm.tqdm(done_iter, total=len(cc_list)) ❻
26         for future in done_iter: ❻
27             try:
28                 status = future.result() ❻
29             except httpx.HTTPStatusError as exc: ❻
30                 error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}' ❻
31                 error_msg = error_msg.format(resp=exc.response)
32             except httpx.RequestError as exc:
33                 error_msg = f'{exc} {type(exc)}'.strip()
34             except KeyboardInterrupt:
35                 break
36             else:
37                 error_msg = ''
38
39             if error_msg:
40                 status = DownloadStatus.ERROR
41             counter[status] += 1
42             if verbose and error_msg:
43                 cc = to_do_map[future] ❻
44                 print(f'{cc} error: {error_msg}') ❻
45
46     return counter
47
48 if __name__ == '__main__':
49     main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

- ❶ 复用 flags2_sequential.py (见“[示例 20.5.1<612页>](#)”) 中的 download_one 函数。
- ❷ 若命令行未指定 -m/-max_req 选项，则用此值作为并发请求数的最大值 (即线程池大小)。如果要下载的国旗图片较少，则实际的数量可能会小一些。
- ❸ MAX_CONCUR_REQ 用于设置最大并发请求数上限。无论下载多少图片，也无论-m/-max_req 选项的值是多少，最大并发请求数都不会超过此限制。这是一项安全措施，可避免启动过多线程，耗费太多内存。
- ❹ 创建 executor，并设置 max_workers 为 concur_req。main 函数会从 MAX_CONCUR_REQ、len(cc_list)、-m/-max_req 选项中，选取最小值赋值给 concur_req，以避免创建不必要的线程。

- ⑤ 此 dict 会将各个 Future 实例 (代表一次下载) 映射到相应的国家代码,以便在处理错误时使用。
- ⑥ 按字母顺序遍历国家代码列表。返回结果的顺序主要由 HTTP 响应时间长短决定。不过,若线程池 (由 `concur_req` 设置) 比 `len(cc_list)` 小很多,则可能会按字母顺序分批下载。
- ⑦ 每次调用 `executor.submit()` 都会排期执行一个可调用对象,并返回一个 Future 实例。第 1 个参数是可调用对象,余下的参数是传给可调用对象的参数。
- ⑧ 将返回的 future 对象与相应的国家代码存入 dict 中。
- ⑨ `concurrent.futures.as_completed` 函数 返回一个迭代器 `done_iter`,该迭代器在每个任务运行结束时生成 (产出) future 对象。
- ⑩ 如果不是详细模式,则用 `tqdm.tqdm()` 函数 包装 `futures.as_completed` 返回的结果,以便显示任务进度条。因为迭代器 `done_iter` 没有长度,所以必须通过参数 `total` 为 `tqdm.tqdm()` 函数明确指定 `done_iter` 中预期的项数,以便于 `tqdm.tqdm()` 预估剩余的工作量。
- ⑪ 遍历迭代器 `done_iter`,获取运行结束的 future 对象。
- ⑫ 调用 `future.result()` 方法,获取可调用对象的返回值 (或可调用对象引发的异常)。此方法可能会阻塞,等待确定可调用对象的执行结果。但本示例不会阻塞,因为 `concurrent.futures.as_completed` 只返回已运行结束的 future 对象。
- ⑬ 处理可能出现的异常。此函数余下代码与 “示例 20.13<613页>” 中的 `download_many()` 函数一样,唯有一点 (⑭处) 除外。
- ⑭ 为了给错误消息提供上下文,以当前 future 对象为键,从字典 `to_do_map` 中检索国家代码。在顺序版脚本 `flags2_sequential.py` (见 “示例 20.5.1<612页>”) 中,无须这么做。因为顺序版脚本遍历的是国家代码,所以知道当前国家代码是什么,而本示例遍历的是 future 对象。



示例 20.14 用到了一个非常适合 `futures.as_completed()` 函数 的惯用法: 构建一个 dict, 将每个 future 对象映射到 future 运行结束时可能用到的其他数据上。例如, [示例 20.14](#) 在字典 `to_do_map` 中, 将各个 future 对象映射到对应的国家代码。这样, 尽管 future 对象结果的生成是无序的, 仍然可轻松地对 future 结果进行后续处理。

Python 线程非常适合处理 I/O 密集型任务,并且 `concurrent.futures` 包大大简化了 Python 线程在某些场景下的用法。另外,针对“**高度并行**”计算的场景,还可以利用 `ProcessPoolExecutor` 在多核设备上处理 CPU 密集型任务。我们对 `concurrent.futures` 基本用法的介绍,至此结束。

20.6 本章小结

本章开头将 2 个并发版 HTTP 下载客户端与 1 个顺序版客户端进行比较,证明了并发方案比顺序执行方案具有显著的性能提升。

在研究了第 1 个基于 `concurrent.futures` 的示例之后,深入研究了 future 对象——即 `concurrent.futures.Future` 实例或 `asyncio.Future` 实例,着重介绍了二者的相同点;而关于二者的不同点,将在“[二十一 异步编程](#)”中详述。随后,介绍了如何用 `Executor.submit()` 方法 创建 future 对象,以及如何用 `concurrent.futures.as_completed` 函数遍历运行结束的 future 对象。

然后,讨论了如何借助 `ProcessPoolExecutor` 类来使用多个进程,以绕过 GIL 并利用多核 CPU 来简化

“示例 19.13<579页>”中的多核素数检测程序。

随后,通过一个教学示例,介绍了 `concurrent.futures.ThreadPoolExecutor` 的工作原理。该教学示例启动多个任务,每个任务 `sleep(n)` 不同的时长,然后显示包含时间戳的任务状态,除此之外,什么都不做。

最后,回到国旗图片下载示例,为这些示例增加了进度条显示与错误处理,并进一步探讨了 `concurrent.futures.as_completed` 函数。示例“示例 20.14<614页>”展示了 `concurrent.futures.as_completed` 的一种惯用法:将 `future` 对象存储在一个字典中,以便在执行 `Executor.submit` 提交时,可为 `future` 对象关联更多信息;这样,当 `futures.as_completed` 返回的迭代器生成 `future` 对象后,就可以利用字典中关联的那些信息。

20.7 延伸阅读

`concurrent.futures` 包的贡献者——Brian Quinlan,在 PyCon Australia 2010 的演讲 “*The future is soon!*” 中介绍了 `concurrent.futures` 包。Quinlan 在演讲时未准备幻灯片,而是直接在 Python 控制台中输入代码以展示 `concurrent.futures` 包的功能。作为一个激励人心的例子,演讲中展示了一段简短的视频,其中 XKCD 漫画家/程序员 Randall Munroe 对 Google 地图进行了无意的 DoS 攻击,以构建其所在城市周边驾驶时间的彩色地图。该库的正式介绍文件是 “[PEP 3148 -futures - execute computations asynchronously](#)”,该 PEP 中提到,`concurrent.futures` 库 “深受 Java `java.util.concurrent` 包的影响”。

与 `concurrent.futures` 模块 有关的其他资料,请参阅 “[十一 异步编程](#)”。“[19.9.3 标准库之外的并发](#)”中给出的与 `threading` 和 `multiprocessing` 模块有关的资料,同样适用于 `concurrent.futures`。

杂谈

远离线程

并发 (Concurrency): 计算机科学中最困难的主题之一(最好别去招惹)。

——David Beazley, Python 讲师与科学狂人^a

^a摘自 PyCon 2009 上 “[A Curious Course on Coroutines and Concurrency](#)” 教程的第 9 张幻灯片。

上述引文与本章开篇的引文,虽然相互矛盾,但我都认同。

我在大学时学过一门并发课程,利用 `POSIX 线程` 编程。通过该课程,我得出一个结论:不要自己管理线程与锁,其原因与不要自己管理内存分配与释放一样。这些工作最好由资深的系统开发程序员来处理,他们拥有专业知识、意愿与时间来完成这些工作——但愿如此。我拿薪水是为了开发应用程序,而不是开发操作系统。所以,不需要对线程、锁、`malloc` 与 `free` 进行精细控制,详见 “[C dynamic memory allocation](#)”。

`concurrent.futures` 包很棒,因为它将线程、进程与队列视作可提供服务的基础设施,不必自己动手直接处理。当然,此包是针对简单问题而设计的,即所谓的 [易并行 \(Embarrassingly Parallel\)^a](#) 问题。但正如本章开篇 Simionato 所述,开发应用程序(而非操作系统或数据库服务器)时,面临的大部分并发问题都属于这一种。

对于 [非易并发 \(Nonembarrassing Concurrency\)](#) 问题,线程和锁也不是万全之策。在操作系统层面,线程永远不会消失。在过去几年,我发现每种令人眼前一亮的编程语言都对并发提供了更高层次的抽象,这些抽象更易于使用,正如《[Seven Concurrency Models in Seven Weeks](#)》(Paul Butcher 著) 中的示例

所示。这些示例包括 Go、Elixir 和 Clojure 等语言。Erlang (Elixir 的实现语言) 是一个典型示例，从设计之初就考虑到了并发。但 Erlang 并没有吸引我，原因很简单：我觉得它的语法很丑陋。Python 在这方面宠坏了我。

José Valim 曾是 Ruby on Rails 的核心贡献者，他用令人愉悦的现代语法设计了 Elixir。与 Lisp 和 Clojure 一样，Elixir 也实现了语法宏，这是一把双刃剑。语法宏可以实现强大的 DSL (Domain-Specific Language, 领域特定语言)，但衍生语言的激增可能会导致代码库不兼容和社区分裂。大量涌现的宏导致 Lisp 没落，因为各种 Lisp 实现都使用独特而难懂的方言。以 Common Lisp 为核心的标准化导致了 Lisp 语言的臃肿。我希望 José Valim 能够引领 Elixir 社区，避免重蹈覆辙。到目前为止，情况看起来还不错。Ecto (数据库封装器与查询器) 使用起来令人心情愉悦。这是使用宏创建灵活且用户友好的 DSL 的一个典范，用于与关系和非关系数据库进行交互。

与 Elixir 类似，Go 也是一种拥有全新理念的现代编程语言。但在某些方面与 Elixir 相比，Go 是一种保守的语言。Go 中没有宏，语法也比 Python 简单。Go 不支持继承和运算符重载，而且提供的元编程也不如 Python。这些限制被认为是 Go 语言的特点，因为这些限制带来了可预测的行为与性能。关于希望用 Go 取代 C++、Java 和 Python 高并发的关键任务来说，这种限制是一大优势。

虽然 Elixir 与 Go 在高并发领域是直接竞争对手，但它们的设计理念却吸引了不同的人群。两者都可能蓬勃发展。但在编程语言的历史上，保守的语言往往会吸引更多的程序员。

^a译者注：Embarrassingly Parallel 直译为“尴尬并行”，应该译为“易并行”，是指如果一个并行计算问题可被分解成多个完全独立的子任务，那么这个并行计算问题就被称作“易并行问题”。

异步编程

常规异步编程方法的问题在于它们都是非此即彼的命题。你应该重写所有代码，以便彻底去除阻塞，否则你只是在浪费时间。

——Alvaro Videla、Jason J. W. Williams,《RabbitMQ in Action》

本章将讨论密切相关的 3 大主题：

- Python 的 `async def`、`await`、`async with` 和 `async for` 结构；
- 支持这些结构的对象：原生协程、异步上下文管理器、可迭代对象、生成器及推导式。
- `asyncio` 及其他异步库。

本章以可迭代对象、生成器（“[十七 迭代器、生成器和经典协程](#)”，尤其是“[17.13 经典协程](#)”）、上下文管理器（“[十八 with、match 和 else 代码块](#)”）以及并发编程的一般概念（“[十九 Python 中的并发模型](#)”）为基础。

本章将研究与“[二十 并发执行器](#)”中类似的并发 HTTP 客户端，用原生协程与异步上下文管理器重写，使用与之前相同的 `HTTPX` 库，但利用 `HTTPX` 库的异步 API。本章还将说明如何将速度慢的操作委托给线程或进程执行器，以避免阻塞事件循环。

最后，简要说明了异步变成的优缺点。

本章涵盖内容较多，但因篇幅有限只能提供些基本示例。但是，这些基本示例都能帮助您理解每个概念的核心思想和用法。



经过 Yury Selivanov^a 的重新编排后，`asyncio` 文档 有很大改进，将几个适用于应用程序开发的函数与面向 Web 框架、数据库驱动程序等包创建者的底层 API 区分开。

若想全面了解有关 `asyncio` 的内容，推荐阅读《[Using Asyncio in Python](#)》^b (Caleb Hattingh)，Caleb 是本书的技术审校之一。

^aSelivanov 在 Python 中实现了 `async/await`，并编写了相关的 PEP:492、525 和 530。

^b中文版书名为：《[Python 异步编程](#)》（译者：汪阳）。

21.1 本章新增内容

在我编写本书第 1 版时, `asyncio` 库是临时性的, 而 `async/await` 关键词还不存在。因此, 本书第 2 版时, 我不得不更新所有示例。此外, 我还创建了一些新示例, 如探测域名的脚本、FastAPI Web 服务, 以及用 Python 控制台新的异步模式所做的几个实验。

本章还新增了几节, 用于介绍本书第 1 版中不存在的语言特性, 如原生协程、`async with`、`async for`, 以及支持这些结构的对象。

“21.13 异步工作原理与陷阱<655页>”中的观点反映了我在使用异步编程过程中获得的经验, 我认为这些经验是所有人的必读内容。无论您使用的是 Python 还是 Node.js, 它们都能为您省去很多麻烦。

最后, 我删除了几段关于 `asyncio.Futures` 的内容, 现在它被视为 `asyncio` 底层 API。

21.2 术语定义

如节 17.13 所述, Python 3.5 及以上版本提供了如下 3 种协程:

- 原生协程 (Native Coroutine)

又称“异步函数”, 是指用 `async def` 定义的 `协程 (Coroutine)` 函数。可以用 `await` 关键字——类似于经典协程中的“`yield from`”, 将一个原生协程委托给另一个原生协程。`async def` 语句定义的始终是原生协程, 即使协程主体中没有使用 `await` 关键字。¹ 在原生协程之外, 不能使用 `await` 关键字。¹

- 经典协程 (Classic Coroutine)

一个生成器函数, 通过 `my_coro.send(data)` 调用消耗发送给它的数据, 并用 `yield` 读取该数据。一个经典协程可以用 `yield from` 委派给其他经典协程。经典协程无法由 `await` 驱动, 并且不受 `asyncio` 支持。

- 基于生成器的协程 (Generator-based Coroutine)

用装饰器 `@types.coroutine` (在 Python 3.5 中引入) 装饰的生成器函数。该装饰器使生成器与新的 `await` 关键字兼容。

本章将重点介绍原生协程与异步生成器:

- 异步生成器 (Asynchronous Generator)

用 `async def` 定义, 并在其主体中使用 `yield` 的生成器函数。它返回一个异步生成器对象, 该对象提供了 `_anext_` 方法——一个用于检索下一项的协程方法。



`@types.coroutine` 装饰器没有前途^a

根据 issue#43216, 用于经典协程与基于生成器协程的 `@asyncio.coroutine` 装饰器在 Python 3.8 中已被弃用, 并计划在 Python 3.11 中移除。相比之下, 根据 issue#36921, `@types.coroutine` 仍然得以保留。虽然 `asyncio` 不再支持它, 但 Curio 和 Trio 异步框架的底层代码中仍在使用。

^a请原谅我的直白。

¹此规则有一个例外: 若用 `-m asyncio` 选项运行 Python, 则可以直接在 `>>>` 提示符处使用 `await` 驱动原生协程。详见“21.10.1.1 在 Python 异步控制台中实验<646页>”。

21.3 asyncio 示例:探测域名

假设您想创建一个关于 Python 的博客,并计划使用 Python 关键字注册后缀为.DEV 的域名,例如:AWAIT.DEV。示例 21.1 是一个使用 asyncio 同时检查多个域名的脚本。它产生的输出如下所示:

```
1 $ python3 blogdom.py
2   with.dev
3 + elif.dev
4 + def.dev
5   from.dev
6   else.dev
7   or.dev
8   if.dev
9   del.dev
10 + as.dev
11   none.dev
12   pass.dev
13   true.dev
14 + in.dev
15 + for.dev
16 + is.dev
17 + and.dev
18 + try.dev
19 + not.dev
```

请注意,域名的显示没有特定顺序。运行该脚本,您会发现各个域名以不同的延迟逐个显示出来。+ 号表示您的设备能通过 DNS 域名解析。而不带 + 号的域名无法解析,说明该域名或许可以注册。²

在 `blogdom.py` 中,DNS 探测是通过原生协程对象完成的。由于异步操作是交错进行的,因此检查 18 个域名所需的时间要比顺序检查少得多。事实上,总时间基本与单个最慢的 DNS 响应时间相当,而不是所有响应时间的总和。

脚本 `blogdom.py` 的代码,如示例 21.1 所示。

</> 示例 21.1: `blogdom.py`: 为一个 Python 博客搜索域名

```
1 #!/usr/bin/env python3
2 import asyncio
3 import socket
4 from keyword import kwlist
5
6 MAX_KEYWORD_LEN = 4      ❶
7
8 async def probe(domain: str) -> tuple[str, bool]: ❷
9     loop = asyncio.get_running_loop() ❸
10    try:
11        await loop.getaddrinfo(domain, None) ❹
12    except socket.gaierror:
13        return (domain, False)
```

²在我写这篇文章时, true.dev 域名的价格是 360 美元/年。我看到 for.dev 已经注册,但没有配置 DNS。

```

14     return (domain, True)
15
16     def main() -> None: ❶
17         names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN) ❶
18         domains = (f'{name}.dev'.lower() for name in names) ❷
19         coros = [probe(domain) for domain in domains] ❸
20         for coro in asyncio.as_completed(coros): ❹
21             domain, found = await coro ❺
22             mark = '+' if found else '-'
23             print(f'{mark} {domain}')
24
25     if __name__ == '__main__':
26         asyncio.run(main()) ❻

```

- ① 设置域名关键词的最大长度,因为域名越短越好。
- ② 原生协程 `probe` 返回一个包含域名和布尔值的元组; `True` 表示域名已解析。返回域名以方便显示结果。
- ③ 获取对 `asyncio 事件循环` 的引用,以便下一步使用。
- ④ `loop.getaddrinfo(..)` 协程方法返回一个五元组³,是套接字所连接到的地址。在本例中,我们不需要 `getaddrinfo()` 返回的结果。如果得到了返回结果,说明域名可解析,否则不可解析。
- ⑤ `main` 函数必须是一个协程,这样我们才能在其中使用 `await` 关键字(⑩处)。
- ⑥ 一个生成器,可生成长度不超过 `MAX_KEYWORD_LEN` 的 Python 关键字。
- ⑦ 一个生成器,可生成后缀为`.dev` 的域名。
- ⑧ 调用原生协程 `probe`,并依次传入各个 `domain`,构建一个协程对象列表。
- ⑨ `asyncio.as_completed` 是一个生成器,产出的协程会按传入协程的完成顺序(而非提交顺序),返回协程运行结果,作用类似于“[示例 20.4<601页>](#)”(⑦处)的 `concurrent.futures.as_completed` 函数。
- ⑩ 此时,我们知道协程已运行结束,因为 `concurrent.futures.as_completed` 就是这样工作的。因此, `await` 表达式不会阻塞,但我们需要用 `await` 从 `coro` 中获取协程执行结果。若 `coro` 引发了未处理的异常,则将在此处重新引发。
- ⑪ `asyncio.run` 启动事件循环,并仅在事件循环退出时才返回。这是 `asyncio` 的常见使用模式:将 `main` 函数实现为一个协程,并在 `if __name__ == '__main__'` 块内驱动这个 `main` 协程。



Python 3.7 中新增了 `asyncio.get_running_loop` 函数,以便在协程内部使用,如[示例 21.1](#)中③处所示。如果没有运行循环, `asyncio.get_running_loop` 会引发 `RuntimeError`。`asyncio.get_running_loop` 的实现比 `asyncio.get_event_loop` 更简单、更快速。如有需要, `asyncio.get_running_loop` 函数可以启动事件循环。自 Python 3.10 起, `asyncio.get_event_loop` 已弃用,最终将成为 `asyncio.get_running_loop` 的别名。

21.3.1 异步代码阅读技巧

`asyncio` 中有许多新概念需要掌握,但如果采用 Guido van Rossum 自用的技巧去阅读代码, [示例 21.1](#)的整体逻辑还是很容易理解的:眯起眼睛,假装 `async` 和 `await` 关键字不存在。如此一来,您会发现,协程读起

³五元组结构为:(family, type, proto, canonname, sockaddr)

来就像普通的顺序函数一样。

例如,以如下协程主体为例:

```
1  async def probe(domain: str) -> tuple[str, bool]:
2      loop = asyncio.get_running_loop()
3      try:
4          await loop.getaddrinfo(domain, None)
5      except socket.gaierror:
6          return (domain, False)
7      return (domain, True)
```

其工作原理与下面的函数类似,只是上述协程会神奇地从不阻塞。

```
1  def probe(domain: str) -> tuple[str, bool]: # no async
2      loop = asyncio.get_running_loop()
3      try:
4          loop.getaddrinfo(domain, None) # no await
5      except socket.gaierror:
6          return (domain, False)
7      return (domain, True)
```

使用 `await loop.getaddrinfo(...)` 语法可以避免阻塞,因为 `await` 会挂起当前的协程对象。例如,在执行协程 `probe('if.dev')` 时, `getaddrinfo('if.dev', None)` 会创建一个新的协程对象。等待该协程对象启动底层 `addrinfo` 查询,并将控制权交还给事件循环,而不是交还给暂缓执行的 `probe('if.dev')` 协程。然后,事件循环可以驱动其他被挂起的协程对象,如 `probe('or.dev')`。

当事件循环获得 `getaddrinfo('if.dev', None)` 查询的响应时,对应的协程对象将恢复执行,控制权交还给 `probe('if.dev')` (在 `await` 处被挂起),处理可能引发的异常,并返回得到的元组。

到目前为止,只看到 `asyncio.as_completed()` 函数与 `await` 关键字被应用于协程。但实际上,它们可以处理任何可异步调用 (`Awaitable`) 对象,详见下节。

21.4 新概念:异步可调用对象

`for` 关键字可用于处理可调用对象,而 `await` 关键字可用于处理“可异步调用 (`Awaitable`) 对象”。

作为 `asyncio` 的终端用户,日常可见到如下 2 种可异步调用对象:

- 原生协程对象,通过调用原生协程函数得到。
- `asyncio.Task`,通常通过将协程对象传递给 `asyncio.create_task()` 函数来获得。

不过,终端用户代码并不总是需要 `await` 关键字来处理 `Task`,还可以用 `asyncio.create_task(coro())` 将协程 `coro` 调度为并发执行,无需等待 `asyncio.create_task()` 的返回。`spinner_async.py` 脚本中的 `spinner` 协程 (见“[示例 19.4<569页>](#)”④处) 就是这么处理的。若不准备取消任务或等待任务,则无须保存从 `create_task()` 返回的 `Task` 对象。只需要创建任务,就足以调度协程的运行。

相比之下, `await other_coro()` 会立即运行 `other_coro` 并等待其运行完毕,因为继续向下运行之前需要协程 `other_coro` 返回的结果。在 `spinner_async.py` 中,原生协程 `supervisor` 用 `result = await slow()` 执行 `slow` 协程,并获得其结果 (详见“[示例 19.4<569页>](#)”⑥处)。

为了实现异步库或为了向 `asyncio` 库贡献代码,可能还要处理如下底层异步可调用 (`Awaitable`) 对象:

- 一个对象, 拥有返回迭代器的 `__await__` 方法。例如, `asyncio.Future` 实例——`asyncio.Task` 是 `asyncio.Future` 的子类。
- 以其他语言编写的对象, 使用 Python/C API, 提供 `tp_as_async.am_await` 函数,(类似于 `__await__` 方法) 返回一个迭代器。

现存代码库或许还有另一种异步可调用对象: 基于生成器的协程对象——这种对象正被逐步废弃。



PEP 492 指出: “`await` 表达式用 `yield from` 实现, 并多出一个验证参数的步骤。而且 `await` 只接受一个异步可调用对象(Awaitable)”。PEP 492 并未详细说明该实现细节, 而是引用了引入 “`yield from`” 表达式的 “PEP 380”。我在 fluentpython.com 上 “Classic Coroutines” 中 “The Meaning of `yield from`” 一节, 对此做了详细解释。

接下来, 分析下载固定国家国旗图片的 `asyncio` 版本脚本。

21.5 用 `asyncio` 和 `HTTPX` 进行下载

`flags.Asyncio.py` 脚本从 fluentpython.com 下载一组固定的 20 个国旗图片。该脚本首次见 “[20.2 并发网络下载<596页>](#)”, 但现在将用刚刚学到的概念来详细研究它。

从 Python 3.10 开始, `asyncio` 仅直接支持 TCP 与 UDP, 而且 Python 标准库中没有异步 HTTP 客户端与服务器包。所以, 在所有 HTTP 客户端示例中, 我都使用了 `HTTPX` 包。

这里将自下而上逐步分析 `flags.Asyncio.py`。首先, 看一下关于设置的函数(如示例 21.2 所示)。



为使代码更易于阅读, `flags.Asyncio.py` 没有错误处理。在介绍 `async/await` 时, 不妨先关注主要执行逻辑, 了解程序中常规函数和协程是如何调度执行的。从 “[21.7 增强 `asyncio` 下载器<628页>](#)” 开始, 示例将包括错误处理和更多功能。

本章与 “[二十 并发执行器](#)” 中的 `flags_*.py` 示例共用一些代码与数据, 因此我将相关示例都放在 [随书代码库](#) 的 `20-executors/getflags` 目录中。

</> 示例 21.2: `flags.Asyncio.py`: 设置操作的函数

```

1  def download_many(cc_list: list[str]) -> int: ❶
2      return asyncio.run(supervisor(cc_list)) ❷
3
4  async def supervisor(cc_list: list[str]) -> int:
5      async with AsyncClient() as client:
6          to_do = [download_one(client, cc)
7                  for cc in sorted(cc_list)] ❸
8          res = await asyncio.gather(*to_do) ❹
9
10         return len(res)
11
12 if __name__ == '__main__':
13     main(download_many)

```

- ❶ 此函数应是一个常规函数（不是协程），以便传给 `flags.py` 模块（见“[示例 20.2](#)”）中的 `main` 函数，并由 `main` 函数调用。
- ❷ 执行事件循环，以驱动 `supervisor(cc_list)` 协程对象，直至该协程返回。在事件循环运行期间，此行代码将阻塞。该行代码的结果就是 `supervisor` 返回的结果。
- ❸ `HTTPX` 中的异步 HTTP 客户端操作是 `AsyncClient` 的方法，`AsyncClient` 类同时也是 [异步上下文管理器](#)：提供了异步设置与清理方法的上下文管理器（详见“[21.6 异步上下文管理器](#)”）。
- ❹ 为每个要检索的国旗图片调用一次 `download_one` 协程，从而建立一个协程对象列表。
- ❺ 等待 `asyncio.gather` 协程。该协程接受一个或多个异步可调用对象，并等待这些可调用对象全执行完毕。然后，按提交顺序返回由异步可调用对象结果构成的列表。
- ❻ `supervisor` 返回由 `asyncio.gather` 返回的列表长度。

现在，回顾一下 `flags_asyncio.py` 脚本的前半部分（见[示例 21.3](#)）。我重新调整了协程的顺序，使其与事件循环启动协程的顺序保持一致，以便于阅读。

```
</> 示例 21.3: flags_asyncio.py:import 语句与负责下载的函数
1  import asyncio
2
3  from httpx import AsyncClient
4
5  from flags import BASE_URL, save_flag, main ❷
6
7  async def download_one(client: AsyncClient, cc: str): ❸
8      image = await get_flag(client, cc)
9      save_flag(image, f'{cc}.gif')
10     print(cc, end=' ', flush=True)
11     return cc
12
13 async def get_flag(client: AsyncClient, cc: str) -> bytes: ❹
14     url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
15     resp = await client.get(url, timeout=6.1,
16                             follow_redirects=True) ❺
17     return resp.read() ❻
```

- ❶ `HTTPX` 包 未存在于 Python 标准库中，必须自行安装。
- ❷ 复用 `flags.py` 模块 中的代码（见“[示例 20.2](#)”）。
- ❸ `download_one` 必须是原生协程，这样才能用 `await` 处理 `get_flag`（处理 HTTP 请求）。然后，显示下载的国旗图片对应的国家代码，并保存图片。
- ❹ 原生协程 `get_flag` 需要接收 `AsyncClient` 实例，才能发起 `request` 请求。
- ❺ `httpx.AsyncClient` 实例的 `get` 方法返回 `ClientResponse` 对象——一个 [异步上下文管理器](#)。
- ❻ 网络 I/O 操作是以协程方法实现的，因此可由 `ascyncio` 事件循环来异步驱动。



为了获得更好的性能, `get_flag` 内的 `save_flag` 调用应该是异步执行的, 以避免阻塞事件循环。不过, `asyncio` 目前还未提供(类似 Node.js 那种)异步文件系统 API。“21.7.1 使用 `asyncio.as_completed` 和线程<629页>”将说明如何将 `save_flag` 委托给一个线程。

用户编写的代码可用 `await` 将操作显式委托给 `httpx` 协程, 或者也可用异步上下文管理器(如“21.6 异步上下文管理器”中的 `AsyncClient` 与 `ClientResponse`)的特殊方法将操作隐式委托给 `httpx` 协程。

21.5.1 原生协程的秘密:默默无闻的生成器

经典协程示例(见“17.13 经典协程<518页>”)与 `flags asyncio.py` 之间的关键区别在于, 后者没有一目了然的 `.send()` 调用或 `yield` 表达式。用户代码应位于 `asyncio` 库与所用异步库(如 `HTTPX`)之间, 如图 21.1 所示。

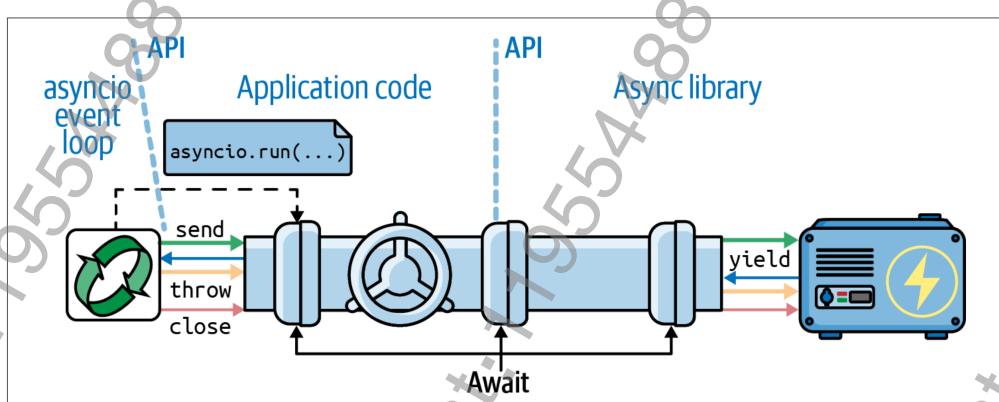


图 21.1: 在异步程序中, 用户函数会启动事件循环, 并用 `asyncio.run` 调度最初的协程。每个用户协程都用 `await` 驱动下一个协程, 形成通道, 以实现 `HTTPX` 等库与事件循环之间的通信。

`asyncio` 事件循环在背后调用 `.send()` 来驱动用户协程, 而用户协程使用 `await` 等待其他协程(包括库提供的协程)。如前所述, `await` 的大部分实现都借鉴了 `yield from`, 后者也会调用 `.send` 来驱动协程。

`await` 链最终会到达底层异步可调用对象(`Awaitable`), 该对象返回一个可由事件循环驱动的生成器, 以作为对事件(如计时器或网络 I/O)的响应。这些 `await` 链末端的底层异步可调用对象(`Awaitable`)和生成器是在库的深处实现的, 不开放为 API, 可能是 Python/C 扩展。

使用 `asyncio.gather()` 和 `asyncio.create_task()` 等函数, 可以启动多个并发 `await` 通道, 从而能够在单个线程中并发执行由单个事件循环驱动的多个 I/O 操作。

21.5.2 非此即彼的问题

请注意, [示例 21.3](#) 无法复用 `flags.py`(见“[示例 20.2<597页>](#)”)中的 `get_flag` 函数。我必须将其重写为一个协程, 才能使用 `HTTPX` 包的异步 API。为了充分发挥 `asyncio` 库的性能, 必须将每个执行 I/O 的函数替换为异步版本, 并使用 `await` 或 `asyncio.create_task()` 激活。这样, 在函数等待 I/O 时, 才能将控制权交还给事件循环。如果无法将导致阻塞的函数重写为 [协程 \(Coroutine\)](#), 则应该在单独的线程或进程中运行那个函数, 详见“[21.8 将任务委托给执行器<635页>](#)”。

这也是本章开篇引文选择那段话的原因——“你应该重写所有代码,以便测底去除阻塞,否则你只是在浪费时间。”

出于同样原因,也不能重用 `flags_threadpool.py`(见“[示例 20.3<599页>](#)”)中的 `download_one` 函数。[示例 21.3](#)用 `await` 关键字驱动 `get_flag` 函数,所以 `download_one` 也必须是协程。对于每个请求,都会在协程 `supervisor` 中创建一个 `download_one` 协程对象,并且它们都由 `asyncio.gather` 协程来驱动。

接下来,研究一下协程 `supervisor`(“[示例 21.2<624页>](#)”)与 `get_flag`(“[示例 21.3<625页>](#)”)中的 `async with` 语句。

21.6 异步上下文管理器

如“[18.2 上下文管理器与 with 块<532页>](#)”所述,如果对象所属的类提供了 `__enter__` 与 `__exit__` 方法,则可在 `with` 块主体的前后用此对象运行代码。

下面看一下[示例 21.4](#),该代码摘自 `asyncpg` 库文档。`asyncpg` 库专为 Python/asyncio 设计的 PostgreSQL 数据库接口库(支持事务)。

</> [示例 21.4](#): 来自 `asyncpg` PostgreSQL 驱动程序文档的示例代码

```
1 tr = connection.transaction()
2 await tr.start()
3 try:
4     await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
5 except:
6     await tr.rollback()
7     raise
8 else:
9     await tr.commit()
```

数据库事务特别适用于使用上下文管理器协议:必须启动事务,数据由 `connection.execute` 更改,然后必须根据更改结果进行回滚或提交。

在异步驱动程序(如`asyncpg`)中,设置工作与清理工作需要由协程来执行,以便其他操作可并发执行。然而,传统 `with` 语句的实现并不支持协程用 `__enter__`/`__exit__` 执行相关工作。

鉴于此,“[PEP 492 -Coroutines with async and await syntax](#)”引入了 `async with` 语句,该语句可与异步上下文管理器搭配使用。[异步上下文管理器](#)是指将特殊方法 `__aenter__`/`__aexit__` 实现为协程方法的对象。

在使用 `async with` 时,[示例 21.4](#)可以写成如下形式(摘自 `asyncpg` 文档):

```
1 async with connection.transaction():
2     await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
```

在 `asyncpg.Transaction` 类中, `__aenter__` 协程方法等待 `self.start()`,而 `__aexit__` 协程等待私有的 `_rollback` 或 `_commit` 协程方法,这取决于异常是否发生。使用协程将 `Transaction` 实现为[异步上下文管理器](#),可以让 `asyncpg` 同时处理多个事务。



Caleb Hattingh 点评 `asyncpg` 库

`asyncpg` 库的另一大优点是, 它还通过为 Postgres 本身的内部连接实施连接池, 补足了 PostgreSQL 缺乏高并发支持的劣势 (PostgreSQL 在服务端为每个连接创建一个服务器进程)。

这意味着, 您无需像 `asyncpg` 文档描述那样使用 `pgbouncer` 等额外工具。^a

^a本提示栏直接原文引用了本书技术审校 Caleb Hattingh 的一条点评。感谢 Caleb!

回到 `flags asyncio.py` 脚本, `httpx` 的 `AsyncClient` 类是一个异步上下文管理器, 因此它可以在其 `__aenter__` 和 `__aexit__` 特殊协程方法中使用异步可调用对象 (`Awaitable`)。



“21.10.1.3 将异步生成器用作上下文管理器<650页>”将介绍如何不编写类, 直接用 `contextlib` 库创建异步上下文管理器。将这些内容放在本章后半部分, 是因为要先介绍其预备知识, 详见“21.10.1 异步生成器函数<645页>”。

接下来, 将增强 `asyncio` 国旗图片下载示例, 为其增加进度条。在此过程中, 将引导我们继续探索更多的 `asyncio` API。

21.7 增强 `asyncio` 下载器

回顾一下“20.5 显示下载进度并处理错误<608页>”, `flags2` 系列示例共享相同的命令行接口, 并在下载时显示进度条。同时, 还包括错误处理。



我建议您用 `flags2` 系列示例直观了解并发 HTTP 客户端是如何运行的。用 `-h` 选项查看“示例 20.9<610页>”所示的帮助屏幕。用 `-a`、`-e` 与 `-l` 选项控制下载数量, 用 `-m` 选项设置并发下载数量。对 `LOCAL`、`REMOTE`、`DELAY` 与 `ERROR` 服务器都测试一下, 找出可最大限度发挥各个服务器吞吐量的最佳并发下载数。若想调整测试服务器选项, 可参考“20.5 显示下载进度并处理错误”中的附注栏“[搭建测试服务器](#)”。

例如, [示例 21.5](#) 展示了用 100 个并发请求 (`-m 100`) 从 `ERROR` 服务器获取 100 个国旗图片 (`-al 100`)。结果得到 48 个错误, 要么是 HTTP 418, 要么是超时——`slow_server.py` 的预期 (错误) 行为。

</> [示例 21.5: 运行 `flags2 asyncio.py` 脚本](#)

```

1 $ python3 flags2 asyncio.py -s ERROR -al 100 -m 100
2 ERROR site: http://localhost:8002/flags
3 Searching for 100 flags: from AD to LK
4 100 concurrent connections will be used.
5 100%|██████████| 100/100 [00:03<00:00, 30.48it/s]
6 -----
7 52 flags downloaded.
8 48 errors.
9 Elapsed time: 3.31s

```



测试并发客户端时,要有责任心

尽管线程版 HTTP 客户端与 asyncio 版 HTTP 客户端的总体下载耗时相差无几,但 asyncio 版发送请求的速度更快,因此更有可能触发对服务器的 Dos 攻击。要真正全速运行这些并发客户端,请用本地 HTTP 服务器进行测试,详见“20.5 显示下载进度并处理错误”中的附注栏“[搭建测试服务器](#)”。

现在,分析一下 `flags2_asyncio.py` 是如何实现的。

21.7.1 使用 `asyncio.as_completed` 和线程

如“[示例 21.2<624页>](#)”(❶处)所示,将多个协程提交给 `asyncio.gather()`。`gather()` 按协程提交顺序,返回一个包含协程执行结果的列表。这意味着,只有所有异步可调用对象都运行结束后, `asyncio.gather()` 才返回结果。但为了显示下载进度,我们需要在每个协程运行结束时,立即获取运行结果。

幸好, `asyncio` 也提供了 `asyncio.as_completed()` 生成器函数,其作用与“[示例 20.14<614页>](#)”(❹处)中的 `futures.as_completed()` 函数一样。

[示例 21.6](#) 展示了 `flags2_asyncio.py` 脚本的前半部分,其中用关键字 `async def` 定义了原生协程 `get_flag` 和 `download_one`。“[示例 21.7<631页>](#)”展示了脚本的后半部分,包括 `supervisor` 协程和 `download_many` 函数。由于增加了 `try/except` 错误处理结构,该脚本比 `flags_asyncio.py` 更长。

</> [示例 21.6: flags2_asyncio.py](#)脚本上半部分,余下代码见“[示例 21.7<631页>](#)”

```
1 import asyncio
2 from collections import Counter
3 from http import HTTPStatus
4 from pathlib import Path
5
6 import httpx
7 import tqdm # type: ignore
8
9 from flags2_common import main, DownloadStatus, save_flag
10
11 # low concurrency default to avoid errors from remote site,
12 # such as 503 – Service Temporarily Unavailable
13 DEFAULT_CONCUR_REQ = 5
14 MAX_CONCUR_REQ = 1000
15
16 async def get_flag(client: httpx.AsyncClient, ❶
17                     base_url: str,
18                     cc: str) -> bytes:
19     url = f'{base_url}/{cc}/{cc}.gif'.lower()
20     resp = await client.get(url, timeout=3.1, follow_redirects=True) ❷
21     resp.raise_for_status()
22     return resp.content
23
24 async def download_one(client: httpx.AsyncClient,
25                     cc: str,
```

```

26             base_url: str,
27             semaphore: asyncio.Semaphore,
28         bool) -> DownloadStatus:
29             try:
30                 async with semaphore: ❸
31                     image = await get_flag(client, base_url, cc)
32             except httpx.HTTPStatusError as exc: ❹
33                 res = exc.response
34                 if res.status_code == HTTPStatus.NOT_FOUND:
35                     status = DownloadStatus.NOT_FOUND
36                     msg = f'not found: {res.url}'
37             else:
38                 raise
39             else:
40                 await asyncio.to_thread(save_flag, image, f'{cc}.gif') ❺
41                 status = DownloadStatus.OK
42                 msg = 'OK'
43             if verbose and msg:
44                 print(cc, msg)
45             return status

```

- ❶ 协程 `get_flag` 与“[示例 20.5.1<612页>](#)”中的 `get_flag` 函数作用相差不大。区别 1: 此处需要参数 `client`。
- ❷ 区别 2 与区别 3: `.get()` 是由 `AsyncClient` 提供的一个协程方法, 因此需要用 `await` 关键字。
- ❸ 将信号量 (`semaphore`) 用作异步上下文管理器, 以防止整个脚本出现阻塞。当信号量内的计数器为 0 时, 只有这个协程被挂起。详见后文附注栏“[Python 中的信号量](#)”。
- ❹ 错误处理逻辑与“[示例 20.5.1<612页>](#)”中的 `download_one` 相同。
- ❺ 保存图片是 I/O 操作。为了避免阻塞事件循环, 需要在线程中运行 `save_flag` 函数。

所有网络 I/O 都用 `asyncio` 提供的协程处理, 而文件 I/O 未用协程处理。但是, 文件 I/O 也是阻塞操作——从某种意义上说, 读写文件的耗时要比读写 RAM 长数千倍。如果使用 NAS (网络附加存储), 甚至可能还会涉及到网络 I/O。

从 Python 3.9 开始, `asyncio.to_thread` 协程可轻松地将文件 I/O 委托给 `asyncio` 提供的线程池。若需要向后兼容 Python 3.7 或 3.8, 则需要多增加几行代码, 详见“[21.8 将任务委托给执行器<635页>](#)”。但在此之前, 先完成对 HTTP 客户端代码的分析。

21.7.2 用信号量限制网络请求

通常应该对网络客户端进行限流, 以避免过多的并发请求对服务器造成冲击。

信号量 (Semaphore) 是一种同步原语, 比锁更灵活。信号量可以配置最大数量, 一个信号量可由多个协程持有。因此, 信号量特别适合于限制活动的并发协程数。更多信息, 详见后文附注栏“[Python 中的信号量](#)”。

`flags2_threadpool.py` (见“[示例 20.14<614页>](#)”) 脚本中, `download_many` 函数在实例化 `futures.ThreadPoolExecutor` 时, 将必填参数 `max_workers` 设为 `concur_req`, 以实现限流。`flags2_asyncio.py` (如示例 21.7 所示) 脚本中, 协程函数 `supervisor` 会创建一个 `asyncio.Semaphore` 实例, 并将其作为参数 `semaphore` 传递给 [示例 21.6](#) 中的协程函数 `download_one`。

Python 中的信号量

信号量 (Semaphore) 由计算机科学家 Edsger W. Dijkstra 在 20 世纪 60 年代初发明。信号量概念简单, 但使用非常灵活, 以至于大多数同步对象 (如锁和屏障 (barrier)) 都可构建在信号量之上。Python 标准库中有 3 个 Semaphore 类: `threading.Semaphore`、`multiprocessing.Semaphore`、`asyncio.Semaphore`^a。此处介绍的是最后一个。

`asyncio.Semaphore` 在内部管理一个计数器, 每当用 `await` 关键字执行协程 `.acquire()` 时, 该计数器都会递减; 而每次调用 `.release()` 方法时, 该计数器就会递增。`.release()` 不是协程, 因为它从不阻塞。计数器的初始值在实例化 `asyncio.Semaphore` 时设置:

```
1 semaphore = asyncio.Semaphore(concur_req)
```

若计数器大于 0, 则用 `await` 执行协程 `.acquire()` 没有延迟; 若计数器为 0, 则 `.acquire()` 会挂起待处理的协程, 直至其他协程在同一信号量上调用 `.release()`, 从而递增计数器。一般不会直接调用 `.acquire()` 与 `.release()`, 而是推荐将信号量当作 [异步上下文管理器](#) 来使用, 这样更安全。如下所示 (摘自“[示例 21.7<631页>](#)”中的 `download_one`):

```
1 async with semaphore:
2     image = await get_flag(client, base_url, cc)
```

协程方法 `Semaphore.__aenter__` 异步等待 `.acquire()`, 而协程方法 `Semaphore.__aexit__` 则调用 `.release()`。该代码片段可保证处于活动状态的 `get_flag` 协程数量, 在任何时刻都不会超过 `concur_req`。

标准库中的每个 `Semaphore` 类都有一个子类 `BoundedSemaphore`, 该子类强制执行额外的约束: 当 `.release()` 操作次数多于 `.acquire()` 操作时, 内部计数器永远不会大于初始值。^b

^a 一个 `semaphore` 管理着一个内部计数器, 每次调用 `acquire()` 时计数器都会递减, 每次调用 `release()` 时计数器都会递增。计数器永远不会小于 0; 当 `acquire()` 发现计数器为 0 时, 操作就会阻塞, 直到某个任务调用 `release()`。

^b 感谢 Guto Maia, 他在阅读本章初稿时指出, 信号量的概念没有得到解释。感谢 Maia!

现在, 看一下脚本 `flags2 asyncio.py` 的余下部分, 如[示例 21.7](#)所示。

</> [示例 21.7: flags2 asyncio.py](#)脚本余下代码, 接续“[示例 21.6<629页>](#)”

```
1 async def supervisor(cc_list: list[str],
2                     base_url: str,
3                     verbose: bool,
4                     concur_req: int) -> Counter[DownloadStatus]: ❶
5     counter: Counter[DownloadStatus] = Counter()
6     semaphore = asyncio.Semaphore(concur_req) ❷
7     async with httpx.AsyncClient() as client:
8         to_do = [download_one(client, cc, base_url, semaphore, verbose)
9                 for cc in sorted(cc_list)] ❸
10        to_do_iter = asyncio.as_completed(to_do) ❹
11        if not verbose:
12            to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list)) ❺
13        error: httpx.HTTPError | None = None ❻
14        for coro in to_do_iter:
15            try:
16                status = await coro ❻
```

```

17     except httpx.HTTPStatusError as exc:
18         error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
19         error_msg = error_msg.format(resp=exc.response)
20         error = exc
21
22     except httpx.RequestError as exc:
23         error_msg = f'{exc} {type(exc)}'.strip()
24         error = exc
25
26     except KeyboardInterrupt:
27         break
28
29     if error:
30         status = DownloadStatus.ERROR
31         if verbose:
32             url = str(error.request.url)
33             cc = Path(url).stem.upper()
34             print(f'{cc} error: {error_msg}')
35         counter[status] += 1
36
37     return counter
38
39 def download_many(cc_list: list[str],
40                   base_url: str,
41                   verbose: bool,
42                   concur_req: int) -> Counter[DownloadStatus]:
43     coro = supervisor(cc_list, base_url, verbose, concur_req)
44     counts = asyncio.run(coro)
45
46     return counts
47
48 if __name__ == '__main__':
49     main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)

```

- ❶ 协程 supervisor 与函数 download_many 接受相同的参数。但是，不能在 main() 函数中直接调用 supervisor，因为 supervisor 是协程，而 download_many 是常规函数。
- ❷ 创建一个 `asyncio.Semaphore` 实例。使用此信号量的活动协程数，不能超过 `concur_req`。`concur_req` 的值由 `flags2_common.py` 中的 `main` 函数根据命令行选项和各示例设置的常量来计算。
- ❸ 创建一个协程对象列表，每个元素对应一次 `download_one` 协程调用。
- ❹ 获取一个迭代器，用于返回处理完毕的协程对象。我未将 `as_completed` 调用直接放在下面的 `for` 循环中，因为我需要根据用户选择的详细模式，确定是否为 `as_completed` 迭代器外套一层显示进度条的 `tqdm` 迭代器。
- ❺ 用生成器函数 `tqdm.tqdm()` 封装 `as_completed` 迭代器，以显示下载进度。
- ❻ 声明变量 `error`，初始化为 `None`。该变量将用于保存 `try/except` 语句之外的异常（如果引发）。
- ❼ 遍历已执行完毕的协程对象；此循环与“[示例 20.14<614页>](#)”中 `download_many` 内的循环类似。
- ❽ 异步等待协程，并获取其结果。这一步不会阻塞，因为生成器函数 `asyncio.as_completed()` 只会生成（产生）已执行完毕的协程。
- ❾ 此赋值是必要的，因为变量 `exc` 的作用域仅限于此 `except` 子句内，而我需要保存它的值，以便在后面使

用。

⑩ 同标号 9。

⑪ 如果出现错误,则设置变量 status。

⑫ 在详细模式下,从引发的异常中提取 URL...

⑬ ...再提取文件名,以便在下一行显示国家代码。

⑭ `download_many` 实例化协程对象 `supervisor`,并通过 `asyncio.run` 将其传递给事件循环;在事件循环结束时,收集 `supervisor` 返回的计数器。

在 [示例 21.7](#) 中,不能像“[示例 20.14<614页>](#)”那样,使用将 `future` 对象映射到国家代码的字典。因为, `asyncio.as_completed()` 返回的异步可调用对象与接收的异步可调用对象相同。`asyncio` 的内部机制可能会用其他异步可调用对象替换我们提供的异步可调用对象,但最终产生的结果是一样的。⁴



由于下载失败时,不能以异步可调用对象为“键(key)”,从字典中获取国家代码。所以,不得不从异常中提取国家代码。为此,我将异常保存在变量 `error` 中,以便在 `try/except` 语句之外检索国家代码。Python 语言不使用块级作用域,比如循环与 `try/except` 之类的语句不会在它们管理的块中创建局部作用域。但是,如果 `except` 子句将异常绑定到变量上(如[示例 21.7](#)中的变量 `exc`),则绑定的变量仅存在于该特定 `except` 子句所在的块内。

至此,用 `asyncio` 实现了与 `flags2_threadpool.py`(见“[示例 20.14<614页>](#)”)同等功能的 `flags2 asyncio.py`。对此示例的讨论到此结束。

下一示例将演示一种简单的编程模式:用协程逐个执行异步任务。我认为这一模式值得关注,因为任何有 JavaScript 经验的程序员都知道,为了逐个运行异步函数,需要不断地嵌套回调,这种编程模式被称为“末日金字塔”。`await` 关键字可以根除这种编程模式。所以,Python 与 JavaScript 现在都提供了关键字 `await`。

21.7.3 为每次下载发起多个请求

假设想为每个国旗图片保存国家名称和国家代码,而不仅仅是国家代码。现在,需要为每个国旗图片发出 2 个 HTTP 请求:一个是获取国旗图片本身,另一个是获取与图片同目录下的 `metadata.json` 文件——这是记录国家名称的地方。

在线程版脚本中协调同一任务中的多个请求很容易:只需发起一个请求,然后发起另一个请求,阻塞线程 2 次,并将 2 条数据(国家代码与国家名称)保存在局部变量中,以便在保存文件时使用。如果要在异步脚本中用回调实现同样的操作,则需要嵌套函数,将国家代码与名称保存在闭包中,以便在保存文件时可用,因为各个回调都在不同的局部作用域内运行。`await` 关键字缓解了这一问题,允许逐个驱动异步请求,共享驱动协程的局部作用域。

⁴我在 `python-tulip` 群组中发起了一个题为“[Which other futures my come out of asyncio.as_completed?](#)”的主题,其中对此进行了详细讨论。Guido 对此做出了回应,并就 `as_completed` 的实现以及 `asyncio` 中 `future` 和协程之间的密切关系发表了自己的见解。



如果您在用现代 Python 进行异步应用程序编程，并且用了许多回调函数，那么说明您使用的可能是旧的编程模式，已不适应现代 Python 环境。如果您正在编写一个库，用于与不支持协程的遗留代码或底层代码进行交互，那么使用些回调函数则是合理的。StackOverflow 上的问答“What is the use case for `future.add_done_callback()`?”解释了为什么在底层代码中需要回调函数，而在当今的 Python 应用程序级别代码中并不是非常有用。

用 `asyncio` 实现的第 3 版国旗图片下载脚本有几处更改：

- `get_country`

新增的协程，用于获取国家代码对应的 `metadata.json` 文件，并从中获取国家名称。

- `download_one`

该协程现在用关键字 `await` 委托 `get_flag` 协程与新增的 `get_country` 协程，并 `get_country` 返回的结果创建要保存的文件名。

先从 `get_country` 的代码开始，如示例 21.8 所示。请注意，它与“示例 21.6<629页>”中的 `get_flag` 非常相似。

```
</> 示例 21.8: flags3 asyncio.py: get_country 协程
1  async def get_country(client: httpx.AsyncClient,
2                      base_url: str,
3                      cc: str) -> str: ❶
4      url = f'{base_url}/{cc}/metadata.json'.lower()
5      resp = await client.get(url, timeout=3.1, follow_redirects=True)
6      resp.raise_for_status()
7      metadata = resp.json()
8      return metadata['country'] ❷ ❸
```

❶ 如果一切顺利，该协程将返回一个包含国家名称的字符串。

❷ `metadata` 是一个 Python 字典，根据响应的 JSON 内容构建。

❸ 返回国家名称。

示例 21.9 是修改后的 `download_one` 协程，与“示例 21.6<629页>”相比，只改动了几行。

```
</> 示例 21.9: flags3 asyncio.py: download_one 协程
1  async def download_one(client: httpx.AsyncClient,
2                      cc: str,
3                      base_url: str,
4                      semaphore: asyncio.Semaphore,
5                      verbose: bool) -> DownloadStatus:
6
7      try:
8          async with semaphore: ❶
9              image = await get_flag(client, base_url, cc)
10         async with semaphore: ❷
11             country = await get_country(client, base_url, cc)
12     except httpx.HTTPStatusError as exc:
```

```

12     res = exc.response
13     if res.status_code == HttpStatus.NOT_FOUND:           status =
14         DownloadStatus.NOT_FOUND
15     else:
16         raise
17     else:
18         filename = country.replace(' ', '_')    ❸
19         await asyncio.to_thread(save_flag, image, f'{filename}.gif')
20         status = DownloadStatus.OK
21         msg = 'OK'
22     if verbose and msg:
23         print(cc, msg)
24     return status

```

- ❶ 持有信号量 (semaphore), 异步等待 `get_flag`...
- ❷ ... 同样, 持有信号量, 异步等待 `get_country`。
- ❸ 用国家名称创建文件名。作为命令行用户, 我不喜欢在文件名中看到空格。

这比嵌套回调好多了!

我将 `get_flag` 和 `get_country` 调用分开放在 2 个用 `semaphore` 控制的 `with` 块中, 因为持有信号量 (semaphore) 与锁的时间越短越好。

我可以使用 `asyncio.gather` 并行调度 `get_flag` 和 `get_country`, 但如果 `get_flag` 引发异常, 则没有可保存的图片, 此时运行 `get_country` 也毫无意义。不过, 有些时候应使用 `asyncio.gather` 同时请求多个 API, 而不是等待一个请求得到响应之后, 再发起下一个请求。

在 `flags3 asyncio.py` 文件中, `await` 语法出现了 6 次, 而 `async with` 出现了 3 次。希望您已掌握 Python 异步编程的要领。异步编程的一个挑战是确定何时必须用 `await` 关键字以及何时不能用 `await`。其实, 答案很简单: 协程与异步可调用对象 (如 `asyncio.Task`) 应使用 `await` 关键字。但是, 有些 API 可能会比较混乱, 将协程与常规函数混在一起, 无章法可循。例如, “[示例 21.14<643页>](#)” 中使用的 `asyncio.StreamWriter` 类。

在分析过 [示例 21.9](#) 之后, 对 `flags` 系列示例的讨论至此结束。接下来, 讨论如何在异步编程中使用线程执行器或进程执行器。

21.8 将任务委托给执行器

与 Python 相比, Node.js 在异步编程方面的一个重要优势是 Node.js 标准库, 它为所有 I/O (包括网络 I/O) 提供了异步 API。在 Python 中, 如果不小心, 文件 I/O 会明显降低异步应用程序的性能, 因为主线程中读写存储会阻塞事件循环。

在 “[示例 21.6<629页>](#)” (❶处) 中, `download_one` 协程用如下代码将下载的图片保存到磁盘:

```

1     await asyncio.to_thread(save_flag, image, f'{cc}.gif')

```

如前所述, `asyncio.to_thread` 是在 Python 3.9 中新增的协程。如果要向后兼容 Python 3.7 或 3.8, 那么需将上述代码替换为 [示例 21.10](#) 中的 3 行代码。

</> 示例 21.10: 替换 `await asyncio.to_thread` 的 3 行

```
1 loop = asyncio.get_running_loop() ①
2 loop.run_in_executor(None, save_flag, ②
3                     image, f'{cc}.gif') ③
```

① 获取指向事件循环的引用。

② 第 1 个参数是要使用的执行器; 这里传入 `None`, 会选择 `asyncio` 事件循环中始终可用的默认执行器 `futures.ThreadPoolExecutor`。

③ 可通过位置参数传入要运行的函数, 但如果传入关键字参数, 则需要利用 `functools.partial()` 函数, 如 `asyncio.loop.run_in_executor` 文档所述。

新增的 `asyncio.to_thread` 函数更易用, 也更灵活, 因为它也接受关键字参数。

`asyncio` 自身的实现也多次使用了 `run_in_executor`。如, “示例 21.1<621页>” 中的 `loop.getaddrinfo()` 协程是通过调用 `socket.getaddrinfo()` 实现的, 而此函数要阻塞数秒才能返回, 具体取决于 DNS 解析速度。

异步 API 的常见编程模式是: 在内部用 `loop.run_in_executor` 将实现细节中的阻塞调用封装成协程。如此一来, 提供了一致的协程接口, 都可以用 `await` 驱动, 同时隐藏了需要使用的线程。MongoDB 的异步驱动 Motor 提供了与 `async/await` 兼容的 API, 将与数据库服务器通信的线程核心封装起来。Motor 首席开发人员 A. Jesse Jiryu Davis 在 “Response to “Asynchronous Python and Databases”” 中解释了他的理由。剧透: Davis 发现在数据库驱动的特殊使用场景下, 线程池的性能更高——尽管有传言认为对网络 I/O 来说, 异步方案总比线程速度更快。

将 `Executor` 显式传给 `loop.run_in_executor`, 主要是为了使用 `futures.ProcessPoolExecutor` 在不同的 Python 进程中运行 CPU 密集型函数, 以避免 GIL 争用。由于启动进程的开销较高, 因此最好在 `supervisor` 中启动 `futures.ProcessPoolExecutor`, 并将其传递给需要使用它的协程。

本书技术审校 Caleb Hattingh⁵ 建议我添加如下警告, 指出有关执行器和 `asyncio` 的注意事项。



Caleb 对 `run_in_executor` 的警告

使用 `run_in_executor` 可能会导致难以调试的问题, 因为撤销行为可能与多数人的预期不符。使用执行器的协程只是给人一种可以撤销底层线程(如果是 `ThreadPoolExecutor`)的假象, 而实际没有撤销机制。例如, 在 `run_in_executor` 调用中创建的长期运行的线程可能会阻止 `asyncio` 程序正常关闭: `asyncio.run` 将等待执行器完全关闭才会返回, 如果执行器的任务无法自行停止, 则 `asyncio.run` 将永远等待下去。在我看来, 将这个函数命名为 `run_in_executor_uncancellable` 更为合适。

接下来, 将讨论如何用 `asyncio` 编写 HTTP 服务器。

21.9 用 `asyncio` 编写服务器

演示 TCP 服务器时, 通常用回显服务器。我们将构建两个有趣的示例服务器, 用于搜索 Unicode 字符。第 1 个服务器用 `FastAPI` 处理 HTTP, 第 2 个服务器仅用 `asyncio` 处理 TCP。

⁵Caleb Hattingh 是《Using Asyncio in Python》的作者之一。

这 2 个服务器可允许用户根据 Unicode 标准名称所含的单词检索 Unicode 字符。图 21.2 显示了在 `web_mojifinder.py` (这是要构建的第 1 个服务器) 中查询的某次会话。

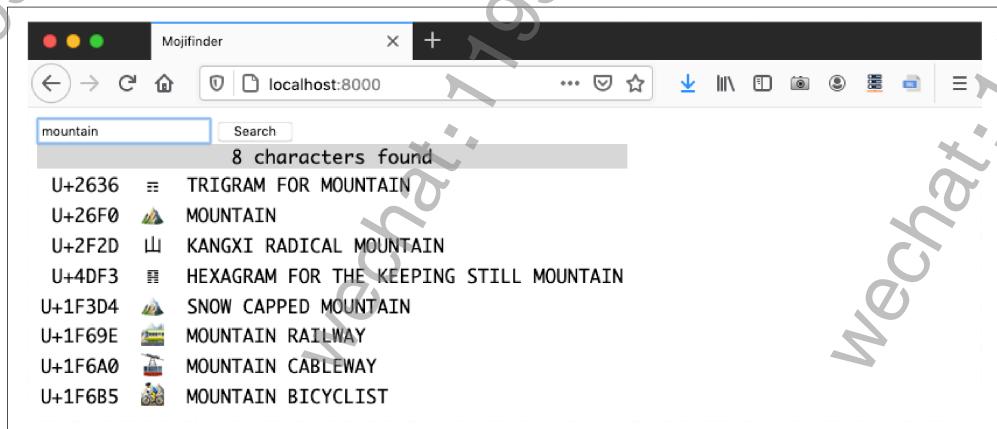


图 21.2: 浏览器窗口中显示了在 `web_mojifinder.py` 服务器中搜索 “mountain” 的结果

这些示例中的 Unicode 检索逻辑位于本书随书代码库 中 `charindex.py` 模块 的 `InvertedIndex` 类中。此模块内容不多,未涉及并发,相关介绍见下方标记栏 “倒排索引简介”,可根据实际情况选读。若觉得没有必要,可以直接跳到“[21.9.1 FastAPI Web 服务](#)”,了解 HTTP 服务器的实现。

倒排索引简介

倒排索引通常将单词映射到包含该单词的文档。mojifinder 示例中的“文档”是指 Unicode 字符。InvertedIndex 类为 Unicode 字符标准名称中的每个单词建立(倒排)索引，并将此索引存储在 defaultdict 中。例如，索引字符“U+0037—DIGIT SEVEN”时，InvertedIndex 初始化方法会将字符‘7’追加到 defaultdict 中的键“DIGIT”与“SEVEN”之下。将 Python 3.9.1 自带的 Unicode 13.0.0 数据索引完毕之后，“DIGIT”对应 868 个字符，“SEVEN”对应 143 个字符，包括“U+1F556(CLOCK FACE SEVEN OCLOCK)”与“U+2790(DINGBAT NEGATIVE CIRCLED SANS-SERIF DIGIT SEVEN)”。

查看“CAT”与“FACE”条目,得到的结果如下图所示^a:

InvertedIndex.search 方法会将查询拆分成单词，并返回各个单词对应条目的交集。所以，搜索“FACE”能找到 171 个结果，搜索“CAT”能找到 14 个结果，而搜索“cat face”只能找到 10 个结果。

这就是倒排索引背后的美妙思想:倒排索引是信息检索的基石,是搜索引擎背后的理论基础。如需了解更多信息,请参阅维基百科文章“[Inverted Index](#)”。

^a屏幕截图中的方框问号并不是书籍的缺陷。它是“U+101EC (PHAISTOS DISC SIGN CAT)”字符，而我使用的终端字体中缺少这个字符，因此无法正常显示。斐斯托斯圆盘 (Phaistos Disc) 是一种刻有象形文字的古代文物，发现于克里特岛。

21.9.1 FastAPI Web 服务

现在,用 FastAPI 编写下一示例 (`web_mojifinder.py`),即“19.7.4 WSGI 应用服务器<586页>”中提示框“**ASGI——异步服务器网关接口**”提到的 Python ASGI Web 框架之一。图 21.2 是前端页面截图。这是个很简单的 SPA(单页面应用程序):初始 HTML 下载完毕后,用户 UI 由客户端 JavaScript 与服务器通信进行更新。

FastAPI 旨在为 SPA 和移动应用程序实现后端,这些应用程序主要由返回 JSON 响应(而不是服务器渲染的 HTML)的 Web API 端点组成。FastAPI 借助装饰器、**类型提示**(Type Hints)和**代码内省**(Introspection)消除了 Web API 的大量模板代码。此外, FastAPI 还能为我们创建的 API 自动发布交互式 OpenAPI(又称 Swagger)文档。图 21.3 显示了 FastAPI 为 `web_mojifinder.py` 自动生成的 /docs 页面。

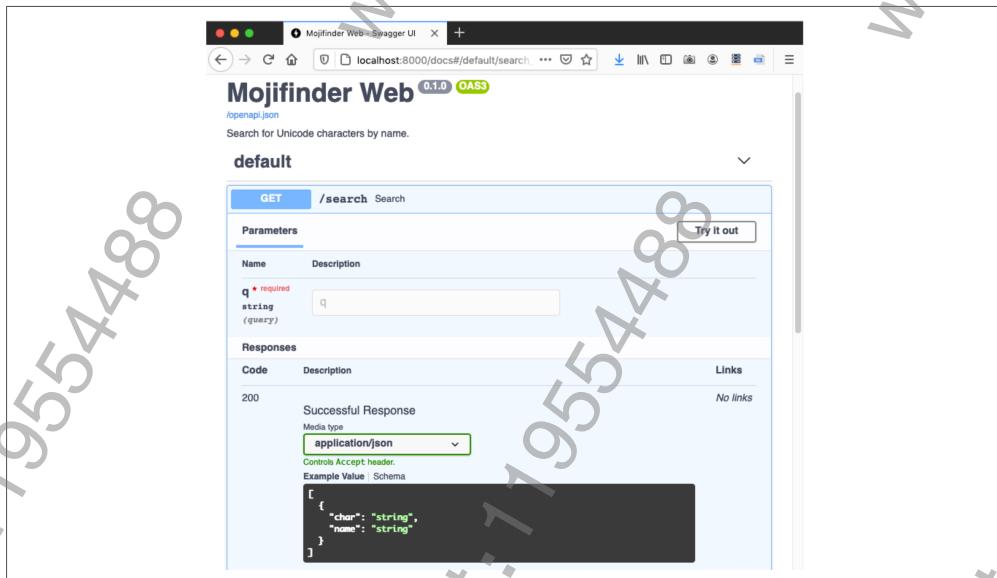


图 21.3: 为 /search 端点自动生成的 OpenAPI 模式

`web_mojifinder.py` 示例的源码见 [示例 21.11](#),但那只是后端代码。当访问根(/)URL 时,服务器会发送 `form.html` 文件。`form.html` 文件包含 81 行代码,其中 54 行是 JavaScript 代码,负责与服务器通信,并将结果填入表格。如果您对阅读未使用框架的纯 JavaScript 代码感兴趣,请阅读[随书代码库](#)的“`21-async/mojifinder/static/form.html`”文件。

运行 `web_mojifinder.py` 之前,需要先安装 2 个包及其依赖项:FastAPI 和 `uvicorn`⁶。开发模式下用 `uvicorn` 运行 `web_mojifinder.py` 的命令如下所示:

```
1 $ uvicorn web_mojifinder:app --reload
```

参数列举如下:

- `web_mojifinder:app`
包名、冒号(:)以及包内定义的 ASGI 应用名称(通常命名为 `app`)。
- `--reload`

让 `uvicorn` 监视应用程序源文件的更改并自动重新加载这些更改。仅在开发过程中有用。

下面分析 `web_mojifinder.py` 的源码。

⁶你可以使用其他 ASGI 服务器来代替 `uvicorn`,如 `hypercorn` 或 `Daphne`。更多信息,请参阅有关实现的 [ASGI 官方文档](#)页面。

</> 示例 21.11: `web_mojifinder.py` 完整源码

```
1  from pathlib import Path
2  from unicodedata import name
3
4  from fastapi import FastAPI
5  from fastapi.responses import HTMLResponse
6  from pydantic import BaseModel
7
8  from charindex import InvertedIndex
9
10 STATIC_PATH = Path(__file__).parent.absolute() / 'static' ❶
11
12 app = FastAPI(          ❷
13     title='Mojifinder Web',
14     description='Search for Unicode characters by name.',
15 )
16
17 class CharName(BaseModel): ❸
18     char: str
19     name: str
20
21 def init(app):          ❹
22     app.state.index = InvertedIndex()
23     app.state.form = (STATIC_PATH / 'form.html').read_text()
24
25 init(app)              ❺
26
27 @app.get('/search', response_model=list[CharName]) ❻
28 async def search(q: str): ❻
29     chars = sorted(app.state.index.search(q))
30     return ({'char': c, 'name': name(c)} for c in chars) ❻
31
32 @app.get('/', response_class=HTMLResponse, include_in_schema=False)
33 def form():             ❾
34     return app.state.form
35
36 # no main function      ❿
```

❶ 与本章主题无关,不过值得注意的是:pathlib 重载了运算符 “/”,使写出的代码很优雅。⁷

❷ 此行定义 ASGI 应用程序。可简写为 `app = FastAPI()`。提供的参数是自动生成文档的元数据。

❸ 一个用于 JSON 响应的 pydantic 模式⁸,包含字段 `char` 和 `name`。

❹ 构建索引 `index`,并加载静态 HTML 表单 `form`,并将二者附加到 `app.state` 以供后续使用。

❺ 当 ASGI 服务器加载该模块时,运行 `init`。

❻ /search 端点的路由;`response_model` 使用❸处定义的 pydantic 模型 `CharName` 描述响应格式。

⁷感谢技术审校 Miroslav Šedivý 提出本示例适合使用 `pathlib`。

⁸如“[八 函数中的类型提示](#)”所述,pydantic 会在运行时强制执行类型提示,以进行数据验证。

- ⑦ FastAPI 假定函数或协程签名中的参数,若不在路由路径中,都将在 HTTP 查询字符串中传递,如: /search?q=cat。由于 q 无默认值,若查询字符串中缺少 q, FastAPI 将返回 422(不可访问实体)状态。
- ⑧ 返回由字典构成的可迭代对象,与 response_model 模式兼容,可让 FastAPI 根据 @app.get 装饰器中的 response_model 构建 JSON 响应。
- ⑨ 常规函数(即非异步运行)也可用于生成响应。
- ⑩ 该模块没有 main 函数,由 ASGI 服务器(本例是 uvicorn)加载并驱动。

示例 21.11 未直接使用 `asyncio`。FastAPI 基于 Starlette ASGI 工具包构建,而 Starlette ASGI 工具包又使用了 `asyncio`。

还要注意的是,search 的主体内并未使用 `await`、`async with` 或 `async for`,因此它可以是一个常规函数。我将 search 定义为一个协程,只是为了表明 FastAPI 知道如何处理它。在实际应用程序中,大多数端点都会查询数据库或访问其他远程服务器,因此 FastAPI 和 ASGI 框架的关键优势就是支持协程,以便能够利用异步库处理网络 I/O。



我编写的用于加载和提供静态 HTML 表单的 init 和 form 函数是为了让示例更简短、更易于运行。而推荐的最佳做法是在 ASGI 服务器前设置一个代理/负载均衡器来处理所有静态资产,并尽可能使用 CDN(内容分发网络)。Traefik 就是这样一个代理/负载均衡器,它自称是“边缘路由器”,“代表你的系统接收请求,并找出负责处理这些请求的组件”。FastAPI 提供的 `项目生成脚本` 能为您准备好相关代码。

您或许已注意到,search 与 form 的返回值未含类型提示。其实,FastAPI 依靠路由装饰器中的关键字参数 “`response_model`=” 判断返回值类型。FastAPI 文档的“[Response Model](#)”对此做了解释:

在 FastAPI 的路径函数中,使用 `response_model` 参数声明响应模型,而不是使用函数返回值类型注释声明响应模型。这是因为路径函数可能不会直接返回响应模型,而可能会返回一个字典、数据库对象或其他模型,然后使用 `response_model` 对字段进行限制和序列化。

例如,在搜索中,我返回一个产出 dict 的生成器,而不是 CharName 对象构成的列表,但这足够 FastAPI 和 pydantic 验证数据并构建适合的 JSON 响应(与 `response_model=list[Char Name]` 兼容)之用。

现在,我们将重点关注响应查询请求的 `tcp_mojifinder.py` 脚本,效果如 图 21.4 所示。

21.9.2 异步 TCP 服务器

`tcp_mojifinder.py` 程序使用纯 TCP 与 Telnet 或 Netcat 等客户端通信,因此无需借助外部依赖,只用 `asyncio` 即可编写该程序,而且也无需重新发明 HTTP。图 21.4 显示了基于文本的 UI。

该程序的长度是 `web_mojifinder.py` 的 2 倍,因此我将内容分为 3 部分演示:示例 21.12、示例 21.14 和示例 21.15。`tcp_mojifinder.py` 的前半部分(包含 import 语句)见示例 21.14,但首先要介绍的是驱动程序运行的 supervisor 协程和 main 函数。

</> 示例 21.12: `tcp_mojifinder.py`:一个简单的 TCP 服务器(后续见“[示例 21.14<643页>](#)”)

```
1  async def supervisor(index: InvertedIndex, host: str, port: int) -> None:
2      server = await asyncio.start_server(
```

```

TW-LR-MBP:~ luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> fire
U+2632 ☰ TRIGRAM FOR FIRE
U+2EA3 ☲ CJK RADICAL FIRE
U+2F55 火 KANGXI RADICAL FIRE
U+322B 灑 PARENTHESIZED IDEOGRAPH FIRE
U+328B ☲ CIRCLED IDEOGRAPH FIRE
U+4DDD 灑 HEXAGRAM FOR THE CLINGING FIRE
U+1F525 🔥 FIRE
U+1F692 🚒 FIRE ENGINE
U+1F6F1 🚒 ONCOMING FIRE ENGINE
U+1F702 🔮 ALCHEMICAL SYMBOL FOR FIRE
U+1F9EF 🔬 FIRE EXTINGUISHER
11 found

```

图 21.4: 与 `tcp_mojifinder.py` 服务器的 Telnet 会话:查询“fire”。

```

3     functools.partial(finder, index),           ②
4     host, port)                            ③
5
6     socket_list = cast(tuple[TransportSocket, ...], server.sockets) ④
7     addr = socket_list[0].getsockname()
8     print(f'Serving on {addr}. Hit CTRL-C to stop.')      ⑤
9     await server.serve_forever()            ⑥
10
11 def main(host: str = '127.0.0.1', port_arg: str = '2323'):
12     port = int(port_arg)
13     print('Building index.')
14     index = InvertedIndex()                ⑦
15     try:
16         asyncio.run(supervisor(index, host, port))      ⑧
17     except KeyboardInterrupt:
18         print('\nServer shut down.')
19
20 if __name__ == '__main__':
21     main(*sys.argv[1:])

```

- ❶ 此 `await` 表达式会快速获取一个 `asyncio.Server` 实例, 即一个 TCP 套接字服务器。默认情况下, `start_server` 会创建并启动服务器, 以便随时接受连接。
- ❷ `start_server` 的第 1 个参数是 `client_connected_cb`, 这是一个在新客户端连接启动时运行的回调。回调可以是函数, 也可以是协程, 但必须接受 2 个参数: `asyncio.StreamReader` 与 `asyncio.StreamWriter`。不过, 我的 `finder` 协程还需要获取一个索引, 因此我用 `functools.partial` 绑定了参数 `index`, 得到了一个接收 `asyncio.StreamReader` 与 `asyncio.StreamWriter` 的回调函数。为适应回调 API 而改造用户函数是 `functools.partial` 最常见的用途之一。
- ❸ `start_server` 的第 2 和第 3 个参数, 分别是 `host` 和 `port`。完整签名详见 `asyncio` 文档。
- ❹ 由于 `typeshed` 在 2021 年 5 月为服务器类的 `sockets` 属性提供了过时的类型提示, 因此需要进行此转换。详见 [Issue#5535 on typeshed](#)。⁹

⁹ 问题 [Issue#5535](#) 已于 2021 年 10 月关闭, 但此后 Mypy 没有发布新版本, 因此错误依然存在。

- ⑤ 显示服务器第一个套接字的地址和端口。
- ⑥ 虽然 `start_server` 已经以并发任务启动了服务器, 但此处仍需用 `await` 处理 `server_forever`, 以便 `supervisor` 协程可在此处挂起。若无此行, `supervisor` 会立即返回, 结束用 `asyncio.run(supervisor())` 启动的循环, 并退出程序。`Server.server_forever` 文档指出: “若服务器已在接受连接, 则可调用此方法”。
- ⑦ 构建倒排索引。¹⁰
- ⑧ 启动事件循环运行 `supervisor` 协程。
- ⑨ 捕获中断 (`KeyboardInterrupt`), 防止在终端中按 `Ctrl-C` 键停止服务器时输出太多调用跟踪 (`traceback`), 扰乱用户视线。

观察 `tcp_mojifinder.py` 在服务器控制台上生成的输出 (如示例 21.13 所示), 或许更容易理解 `tcp_mojifinder.py` 中控制权的流转情况。

</> 示例 21.13: `tcp_mojifinder.py`: “图 21.4<641页>” 中描述的会话服务器端。

```

1 $ python3 tcp_mojifinder.py
2 Building index. ❶
3 Serving on ('127.0.0.1', 2323). Hit Ctrl-C to stop. ❷
4 From ('127.0.0.1', 58192): 'cat face' ❸
5   To ('127.0.0.1', 58192): 10 results.
6 From ('127.0.0.1', 58192): 'fire' ❹
7   To ('127.0.0.1', 58192): 11 results.
8 From ('127.0.0.1', 58192): '\x00' ❺
9 Close ('127.0.0.1', 58192). ❻
10 ^C ❻
11 Server shut down. ❻
12 $
```

- ❶ 由 `main` 输出。在我的设备中, 在显示下一行之前有 0.6 秒的延迟, 这是构建索引的时间。
- ❷ 由 `supervisor` 协程输出。
- ❸ `finder` 协程中 `while` 循环的第 1 次遍历。TCP/IP 协议栈为我的 Telnet 客户端分配了 58192 端口。如果有多个客户端连接到服务器, 就会在输出中看到它们的端口各不相同。
- ❹ `finder` 协程中 `while` 循环的第 2 次遍历。
- ❺ 我在客户端按下 `Ctrl-C`, `finder` 中的 `while` 循环退出。
- ❻ `finder` 协程会显示这条信息, 然后退出。与此同时, 服务器仍在运行, 准备为其他客户端提供服务。
- ❼ 我在服务器终端按下 `Ctrl-C`; `server.serve_forever` 被撤消, 结束了 `supervisor` 协程和事件循环。
- ❽ 由 `main` 输出。

在函数 `main` 建立索引并启动事件循环后, `supervisor` 协程会迅速显示 “`Serving on...`” 消息, 并在 `await server.serve_forever()` 一行 (示例 21.12 ❻ 处) 挂起。此时, 控制权进入事件循环并停留在那里, 偶尔会回到 `finder` 协程, 每当需要等待网络发送或接收数据时, `finder` 协程就会将控制权交还给事件循环。

当事件循环处于活动状态时, 对于连接到服务器的每个客户端, 都将启动一个新的 `finder` 协程实例。这样, 这个简单的服务器就可以并发处理多个客户端。这种处理方式会持续进行, 直到服务器引发 `Keyboard-`

¹⁰ 技术审校 Leonardo Rochael 指出, 可以在 `supervisor` 协程中用 `loop.run_with_executor()` 将建立索引的工作委托给另一个线程, 这样服务器就可以在建立索引的同时, 立即接受请求。的确如此, 但查询索引是此服务器唯一要做的事情, 因此 Leonardo Rochael 的提议对本示例意义不大。

Interrupt 或服务器进程被操作系统终止。

现在,看一下 `tcp_mojifinder.py` 的前半部分,包括 `finder` 协程。

</> 示例 21.14: `tcp_mojifinder.py`脚本的前半部分(接续“示例 21.12<640页>”)

```
 1 import asyncio
 2 import functools
 3 import sys
 4 from asyncio.trsock import TransportSocket
 5 from typing import cast
 6
 7 from charindex import InvertedIndex, format_results
 8
 9 CRLF = b'\r\n'
10 PROMPT = b'?> '
11
12 async def finder(index: InvertedIndex,
13                   reader: asyncio.StreamReader,
14                   writer: asyncio.StreamWriter) -> None:
15     client = writer.get_extra_info('peername')
16     while True:
17         writer.write(PROMPT) # 不能用 await
18         await writer.drain() # 必须用 await
19         data = await reader.readline()
20         if not data:
21             break
22         try:
23             query = data.decode().strip()
24         except UnicodeDecodeError:
25             query = '\x00'
26         print(f' From {client}: {query!r}')
27         if query:
28             if ord(query[:1]) < 32:
29                 break
30             results = await search(query, index, writer)
31             print(f' To {client}: {results} results.')
32
33         writer.close()
34         await writer.wait_closed()
35         print(f'Close {client}.')
```

- ❶ `format_results` 用于在基于文本的用户 UI (如命令行或 Telnet 会话) 中, 格式化显示 `InvertedIndex.search` 的结果。
- ❷ 为了将 `finder` 传递给 `asyncio.start_server`, 我用 `functools.partial` 对其进行了封装, 因为服务器预期传入的协程或函数只接收 2 个参数:`asyncio.StreamReader` 与 `asyncio.StreamWriter`。
- ❸ 获取套接字所连接的远程客户端地址。
- ❹ 循环处理一个对话框, 直到从客户端收到控制字符为止。
- ❺ `StreamWriter.write` 不是协程, 只是一个常规函数; 该行发送?<> 提示符。

- ⑥ `StreamWriter.drain` 会清空写入缓冲区；它是一个协程，因此必须用 `await` 驱动。
- ⑦ `StreamWriter.readline` 是协程，返回字节序列（`bytes`）。
- ⑧ 如果没有收到字节序列，则客户端关闭连接，从而退出循环。
- ⑨ 使用默认 UTF-8 编码将字节序列（`bytes`）解码为 `str`。
- ⑩ 当用户按下 `Ctrl-C`，或用 `Telnet` 客户端发送控制字节序列时，可能会引发 `UnicodeDecodeError`；如果发生这种情况，为简单起见，请用空字符替换查询。
- ⑪ 将查询记录到服务器控制台。
- ⑫ 如果收到控制字符或空字符，则退出循环。
- ⑬ 执行实际的搜索；`search` 协程的源码如示例 21.15 所示。
- ⑭ 将响应记录到服务器控制台。
- ⑮ 关闭 `asyncio.StreamWriter`。
- ⑯ 等待 `asyncio.StreamWriter` 关闭。如 `StreamWriter.close()` 文档所述，这是推荐做法。
- ⑰ 将此客户端会话的结束消息，记录到服务器控制台。

本示例的最后一部分是 `search` 协程，如示例 21.15 所示。

```
</> 示例 21.15: tcp_mojifinder.py:search 协程
1  async def search(query: str,
2                     index: InvertedIndex,
3                     writer: asyncio.StreamWriter) -> int:
4      chars = index.search(query)
5      lines = (line.encode() + CRLF for line
6                in format_results(chars))
7      writer.writelines(lines)
8      await writer.drain()
9      status_line = f'{"-" * 66} {len(chars)} found' ⑥
10     writer.write(status_line.encode() + CRLF)
11     await writer.drain()
12     return len(chars)
```

- ① `search` 必须是协程，因为它会写入 `asyncio.StreamWriter`，且必须用 `StreamWriter.drain()` 协程方法。
- ② 查询倒排索引。
- ③ 该生成器表达式将生成以 UTF-8 编码的字节（`byte`）字符串，其中包含 Unicode codepoint、实际字符、字符名称和 CRLF 序列（例如，`b'U+0039\t9\tDIGIT NINE\r\n'`）。
- ④ 发送 `lines`。奇怪的是，`writer.writelines()` 不是协程。
- ⑤ 但 `writer.drain()` 是一个协程。不要忘了 `await`。
- ⑥ 构建状态行（`status_line`），然后发送。

请注意，在 `tcp_mojifinder.py` 中，所有网络 I/O 都是字节（`byte`）类型；我们需要解码从网络接收到的字节（`byte`），并在发送之前将其编码为字符串。在 Python 3 中，默认编码是 UTF-8，这也是本示例中所有 `encode` 与 `decode` 调用中隐式使用的编码。



请注意，有些 I/O 方法是协程，必须用关键字 `await` 进行驱动，而有些则是简单函数。例如，`StreamWriter.write()` 是常规函数，因为它将内容写入缓冲区；而 `StreamWriter.drain`（刷新缓冲区，并执行网络 I/O）是协程，`StreamWriter.readline` 也是协程，但 `StreamWriter.writelines` 不是协程！在我撰写本书第 1 版时，`asyncio` API 文档进行了改进，明确标注了哪些是协程。

`tcp_mojifinder.py` 代码利用 `asyncio` 提供的高级 Streams API，只需实现一个处理函数（可以是普通回调函数，也可以是协程），即可创建一个现成的服务器。此外，受 Twisted 框架对传输与协议抽象的启发，`asyncio` 也提供了底层传输与协议 API，见 `asyncio` 文档，文档中还包括用该底层 API 实现的 TCP 和 UDP 回显服务器和客户端。

下一话题是 `async for` 及其背后的对象。

21.10 异步迭代器和异步可迭代对象

如“21.6 异步上下文管理器<627页>”所述，`async with` 可以处理实现了 `__aenter__`/`__aexit__` 方法的对象。`__aenter__`/`__aexit__` 方法通常以协程对象的形式，返回异步可调用对象（`Awaitable`）。

同样，`async for` 也适用于异步可迭代对象——实现了 `__aiter__` 的对象。但是，`__aiter__` 必须是常规方法，而不是协程方法，而且它必须返回一个异步迭代器。

异步迭代器提供了 `__anext__` 协程方法，该方法返回一个异步可调用对象，通常是一个协程对象。异步迭代器还应实现 `__aiter__` 方法，该方法通常返回 `self`。这与“17.5.2 勿让可迭代对象变成其自己的迭代器<489页>”所述的“可迭代对象与迭代器之间的区别”是一样的。

PostgreSQL 异步驱动 `aiopg` 文档中有一个示例，说明如何用 `async for` 遍历数据库游标中的各行：

```
1  async def go():
2      pool = await aiopg.create_pool(dsn)
3      async with pool.acquire() as conn:
4          async with conn.cursor() as cur:
5              await cur.execute("SELECT 1")
6              ret = []
7              async for row in cur:      ❶
8                  ret.append(row)
9              assert ret == [(1,)]
```

本示例中的查询仅返回 1 行记录，但现实中，一个 `SELECT` 查询可能会返回数千行记录。对于大型查询，游标不会一次性加载所有行。因此，在游标等待其他行时，`async for row in cur`（❶处）绝不能阻塞事件循环，这一点非常重要。通过将游标实现为异步迭代器，`aiopg` 在每次调用 `__anext__` 时，可将控制权归还给事件循环；稍后当 PostgreSQL 收到更多行记录时，再重新获取控制权。

21.10.1 异步生成器函数

若想实现异步生成器，可以编写一个类，并实现特殊方法 `__anext__` 与 `__aiter__`。不过，还有一种更简单的方法：用 `async def` 声明一个函数，并在函数主体中使用 `yield` 来实现异步迭代器。这与用生成器函数简化经典迭代器模式的方式类似。

下面分析一个用 `async for` 实现异步生成器的简单示例。“[示例 21.1<621页>](#)”中的 `blogdom.py` 是一个用于探测域名的脚本。现在,假设发现该脚本中定义的 `probe` 协程还有其他用途,并决定将其放在新模块 `domainlib.py` 中。同时,在此模块中还将新定义一个异步生成器 `multi_probe`——接受一个域名列表,并生成相应域名的探测结果。

`domainlib.py` 脚本的实现稍后再讲,先看一下如何在 Python 新增的异步控制台中使用该脚本。

21.10.1.1 在 Python 异步控制台中实验

自 Python 3.8 开始,用命令行选项 `-m asyncio` 运行 Python 解释器,可得到一个“异步 REPL”:一个导入了 `asyncio` 模块的 Python 控制台,提供了一个正在运行中的事件循环。在此控制台的顶层提示符中可接受 `await`、`async for` 和 `async with` 等关键字,而这些关键字在原生协程之外使用时会引发语法错误。¹¹

要实验 `domainlib.py`,请进入[随书代码库](#)本地副本的 `21-async/domains/asyncio/` 目录。运行如下命令:

```
1 $ python -m asyncio
```

控制台启动后,可看到类似如下的输出:

```
1 asyncio REPL 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
2 [Clang 6.0 (clang-600.0.57)] on darwin
3 Use "await" directly instead of "asyncio.run()".
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> import asyncio      ❶
6 >>>
```

控制台头部信息指出,可用 `await` 代替 `asyncio.run()`,以驱动协程和其他异步可调用对象 (Awaitable)。另外, `asyncio` 模块是自动导入的,无需显式导入。❶处的代码是为了向用户明确表明 `asyncio` 模块已导入。

现在导入 `domainlib.py` 并使用其中的 2 个协程: `probe` 和 `multi_probe` (如[示例 21.16](#)所示)。

</> [示例 21.16](#): 运行 `python3 -m asyncio` 后,试验 `domainlib.py`

```
1 >>> await asyncio.sleep(3, 'Rise and shine!') ❶
2 'Rise and shine!'
3 >>> from domainlib import *
4 >>> await probe('python.org')                  ❷
5 Result(domain='python.org', found=True)          ❸
6 >>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split() ❹
7 >>> async for result in multi_probe(names):    ❺
8 ...     print(*result, sep='\t')
9 ...
10 golang.org      True
11 no-lang.invalid False
12 python.org      True
13 rust-lang.org   True
14 >>>
```

❶ 简单试用 `await`,看看异步控制台的运行情况。提示: `asyncio.sleep()` 接受一个可选的参数,设置函数返

¹¹ 这就像 Node.js 控制台一样,非常适合实验。感谢 Yury Selivanov 为 Python 异步编程所做的杰出贡献。

回之前等待的秒数。

- ❷ 驱动 `probe` 协程。
- ❸ `domainlib` 模块中的 `probe` 协程返回一个具名元组 `Result`。
- ❹ 构建一个域名列表。`.invalid` 是保留的顶级域名, 用于测试。针对此类域名的 DNS 查询, 始终会从 DNS 服务器获得 `NXDOMAIN` 响应, 表示“该域名不存在”。¹²
- ❺ 用 `async for` 遍历异步生成器 `multi_probe`, 以显示结果。
- ❻ 注意, 显示域名探测结果的顺序与向 `multi_probe` 提供域名的顺序不同。结果的显示顺序取决于 DNS 解析的速度。

示例 21.16 表明 `multi_probe` 是一个异步生成器, 因为它与 `async for` 语法兼容。接下来, 再多做一些实验, 如示例 21.17 所示。

</> 示例 21.17: 更多实验, 接续示例 21.16

```
1 >>> probe('python.org')          ❶
2 <coroutine object probe at 0x10e313740>
3 >>> multi_probe(names)         ❷
4 <async_generator object multi_probe at 0x10e246b80>
5 >>> for r in multi_probe(names): ❸
6 ...     print(r)
7 ...
8 Traceback (most recent call last):
9 ...
10 TypeError: 'async_generator' object is not iterable
```

- ❶ 调用原生协程, 得到一个协程对象 (`coroutine`)。
- ❷ 调用异步生成器, 得到一个异步生成器对象 (`async_generator`)。
- ❸ 不能用常规的 `for` 循环遍历异步生成器。因为异步生成器实现的是 `__aiter__`, 而不是 `__iter__`。

异步生成器由 `async for` 驱动, 而 `async for` 可以是一个块语句 (如示例 21.16 所示)。另外, `async for` 也可以用在异步推导式中, 详见下文。

21.10.1.2 实现一个异步生成器

下面分析 `domainlib.py` 的代码, 其中包含 `multi_probe` 异步生成器 (如示例 21.18 所示)。

</> 示例 21.18: `domainlib.py`: 探测域名的函数

```
1 import asyncio
2 import socket
3 from collections.abc import Iterable, AsyncIterator
4 from typing import NamedTuple, Optional
5
6 class Result(NamedTuple):          ❶
7     domain: str
8     found: bool
```

¹²详见 RFC 6761—Special-Use Domain Names。

```

9
10 OptionalLoop = Optional[asyncio.AbstractEventLoop] ❷
11
12 async def probe(domain: str, loop: OptionalLoop = None) -> Result: ❸
13     if loop is None:
14         loop = asyncio.get_running_loop()
15     try:
16         await loop.getaddrinfo(domain, None)
17     except socket.gaierror:
18         return Result(domain, False)
19     return Result(domain, True)
20
21 async def multi_probe(domains: Iterable[str]) -> AsyncIterator[Result]: ❹
22     loop = asyncio.get_running_loop()
23     coros = [probe(domain, loop) for domain in domains] ❺
24     for coro in asyncio.as_completed(coros): ❻
25         result = await coro ❼
26         yield result ❼

```

- ❶ 使用 NamedTuple, probe 的结果更易于阅读与调试。
- ❷ 此类型别名是为了防止下一行（❸处）代码过长。
- ❸ 协程 probe 现在接受一个可选参数 loop, 以避免当 multi_probe 驱动 probe 协程时, 重复调用 get_running_loop。
- ❹ 异步生成器函数生成一个异步生成器对象, 该对象可注释为 AsyncIterator[SomeType]。
- ❺ 构建一个 probe 协程对象列表, 列表中的每个 probe 协程对象对应一个域名。
- ❻ 此处不能用 async for, 因为 asyncio.as_completed 是一个经典生成器。
- ❼ 用关键字 await 异步等待协程对象, 获取协程结果。
- ❼ 生成 result。此行使 multi_probe 成为异步生成器。



示例 21.18（❻~❼）中的 for 循环可以更简洁。

```

1 for coro in asyncio.as_completed(coros):
2     yield await coro

```

Python 将 for 循环的主体解析为 yield (await coro), 因此可按预期工作。

我觉得在第一个异步生成器示例中就用这种简写方式, 可能让人不解其意。所以,

示例 21.18 把它分成了 2 行。

有了 domainlib.py, 就可以在 domaincheck.py 中演示 multi_probe 异步生成器的使用。domainlib.py 脚本将搜索由 Python 短关键字与指定后缀构成的域名。

如下是 domainlib.py 脚本的示例输出：

```

1 $ ./domaincheck.py net
2 FOUND          NOT FOUND
3 ===          =====
4 in.net
5 del.net
6 true.net

```

```
7  for.net
8  is.net
9  try.net
10  none.net
11  from.net
12  and.net
13  or.net
14  else.net
15  with.net
16  if.net
17  as.net
18  elif.net
19  pass.net
20  not.net
21  def.net
```

有了 `domainlib.py` 模块, `domaincheck.py` 的代码简单直观, 如 [示例 21.19](#) 所示。

</> [示例 21.19: domaincheck.py](#): 用 `domainlib` 模块探测域名

```
1  #!/usr/bin/env python3
2  import asyncio
3  import sys
4  from keyword import kwlist
5
6  from domainlib import multi_probe
7
8  async def main(tld: str) -> None:
9      tld = tld.strip('.')
10     names = (kw for kw in kwlist if len(kw) <= 4) ❶
11     domains = (f'{name}.{tld}'.lower() for name in names) ❷
12     print('FOUND\tNOT FOUND') ❸
13     print('=====\\t=====')
14     async for domain, found in multi_probe(domains):
15         indent = '' if found else '\\t\\t' ❹
16         print(f'{indent}{domain}') ❺
17
18 if __name__ == '__main__':
19     if len(sys.argv) == 2:
20         asyncio.run(main(sys.argv[1])) ❻
21     else:
22         print('Please provide a TLD.', f'Example: {sys.argv[0]} COM.BR')
```

❶ 生成长度不超过 4 的关键字。

❷ 生成以给定后缀作为顶级域名 (TLD, Top Level Domain) 的域名。

❸ 格式化表格输出的表头。

❹ 异步遍历 `multi_probe(domains)`。

❺ 将缩进 (indent) 设置为 0 或 2 个制表符 (\t), 以便将结果放在合适的列中。

❻ 用给定的命令行参数, 运行 `main` 协程。

生成器还有一个与遍历无关的用途：可以用作上下文管理器。这也适用于异步生成器，可用作[异步上下文管理器](#)。

21.10.1.3 将异步生成器用作上下文管理器

我们自己的代码中很少需要编写[异步上下文管理器](#)，万一需要，可以考虑使用 Python 3.7 在 `contextlib` 模块中新增的 `@asynccontextmanager` 装饰器。这与“[18.2.2 使用 @contextmanager](#)”中用到的 `@contextmanager` 类似。

《Using Asyncio in Python》(Caleb Hattingh 著)一书中给出了一个将 `@asynccontextmanager` 与 `loop.run_in_executor` 结合使用的有趣示例。该示例代码如[示例 21.20](#) 所示(略有改动)。

</> [示例 21.20: @asynccontextmanager 与 loop.run_in_executor 用法示例](#)

```

1  from contextlib import asynccontextmanager
2
3  @asynccontextmanager
4  async def web_page(url):          ❶
5      loop = asyncio.get_running_loop() ❷
6      data = await loop.run_in_executor(❸
7          None, download_webpage, url)
8      yield data                      ❹
9      await loop.run_in_executor(None, update_stats, url) ❺
10
11 async with web_page('google.com') as data:          ❻
12     process(data)

```

- ❶ 被装饰的函数必须是一个[异步生成器](#)。
- ❷ 对 Caleb 代码的轻微改动：用轻量级的 `get_running_loop` 取代 `asyncio.get_event_loop`。
- ❸ 假设 `download_webpage` 是一个用 `requests` 库实现的阻塞函数；在单独的线程中运行该函数，以避免阻塞事件循环。
- ❹ 此 `yield` 表达式之前的所有行将成为装饰器构建的[异步上下文管理器](#)的 `__aenter__` 协程方法。`data` 的值将绑定到❻处 `async with` 语句中 `as` 子句的 `data` 变量。
- ❺ `yield` 表达式之后的所有行将成为[异步上下文管理器](#)的 `__aexit__` 协程方法。此处，将另一个阻塞调用委托给线程执行器。
- ❻ 用 `async with` 结构，调用 `web_page`。

这与顺序执行代码的 `@contextmanager` 装饰器非常相似。更多细节详见“[18.2.2 使用 @contextmanager](#)”，包括如何在 `yield` 一行处理错误。有关 `@asynccontextmanager` 的另一个示例，请参阅 `contextlib` 文档。

最后，对比一下[异步生成器与原生协程](#)。

21.10.1.4 异步生成器与原生协程

异步生成器函数与原生协程之间的主要异同点，如表 21.1 所示：

表 21.1: 异步生成器与原生协程对比

对比项	异步生成器	原生协程
声明方式	用 <code>async def</code> 声明, 主体总含 <code>yield</code> 表达式	用 <code>async def</code> 声明, 主体不含 <code>yield</code> 表达式
<code>return</code> 返回值	不含 <code>return</code> 语句	含 <code>return</code> 语句, 可能返回 <code>None</code> 之外的值
对象类型	异步可迭代对象	异步可调用对象
驱动方式	由 <code>async for</code> 或异步推导式驱动	由 <code>await</code> 驱动, 也可传给 <code>asyncio</code> 库中接受异步可调用对象的函数 (如 <code>create_task</code>)

接下来, 该介绍异步推导式了。

21.10.2 异步推导式和异步生成器表达式

“[PEP 530 -Asynchronous Comprehensions](#)”从Python 3.6开始, 为推导式与生成器表达式语法引入了`async for`与`await`关键字。

“[PEP 530](#)”中定义的结构中, **只有异步生成器表达式可以出现在`async def`主体之外**。

21.10.2.1 定义并使用一个异步生成器表达式

有了“[示例 21.18<647页>](#)”中的`multi_probe`异步生成器之后, 可以再编写一个异步生成器, 只返回找到的域名。下面的演示将在使用“`-m asyncio`”选项启动的异步控制台中进行:

```

1 $ python -m asyncio
2 >>> from domainlib import multi_probe
3 >>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split()
4 >>> gen_found = (name async for name, found in multi_probe(names) if found) ❶
5 >>> gen_found
6 <async_generator object <genexpr> at 0x10a8f9700> ❷
7 >>> async for name in gen_found: ❸
8 ...     print(name)
9 ...
10 golang.org
11 python.org
12 rust-lang.org

```

- ❶ 用`async for`表明这是一个异步生成器表达式。可定义在Python模块的任意位置。
- ❷ 异步生成器表达式会生成一个`async_generator`对象, 与`multi_probe`(见“[21.18 domainlib.py: 探测域名的函数<647页>](#)”等异步生成器函数返回的对象类型完全相同)。
- ❸ 异步生成器对象由`async for`语句驱动, 而`async for`语句又只能出现在`async def`主体内或在本例使用的这种异步控制台中。

综上所述, **异步生成器表达式可以在程序的任何位置定义, 但只能在原生协程或异步生成器函数内使用**。

“[PEP 530](#)”引入的其他结构只能在原生协程或异步生成器函数内定义和使用。

21.10.2.2 异步推导式

Yury Selivanov(“[PEP 530](#)”作者)通过3段简短代码证明了异步推导式的必要性, 摘录如下:

我们一致认为,如果能将如下代码:

```
1 result = []
2 async for i in aiter():
3     if i % 2:
4         result.append(i)
```

重写为下面这样,该多好:

```
1 result = [i async for i in aiter() if i % 2]
```

此外,对于给定 [原生协程](#) `fun`,应该可以这样编写代码:

```
1 result = [await fun() for fun in funcs]
```



在列表推导式中使用 `await` 类似于使用 `asyncio.gather`。不过,得益于可选参数 `return_exceptions` 的存在, `asyncio.gather` 可以让你对异常处理进行更多控制。Caleb Hattingh 建议始终设置 `return_exceptions=True`(默认为 `False`)。更多信息请参阅 [asyncio.gather](#) 文档。

回到神奇的 Python 异步控制台(见“[21.10.1.1 在 Python 异步控制台中实验](#)”[<646页>](#)”)。

```
1 >>> names = 'python.org rust-lang.org golang.org no-lang.invalid'.split()
2 >>> names = sorted(names)
3 >>> coros = [probe(name) for name in names]
4 >>> await asyncio.gather(*coros)
5 [Result(domain='golang.org', found=True),
6 Result(domain='no-lang.invalid', found=False),
7 Result(domain='python.org', found=True),
8 Result(domain='rust-lang.org', found=True)]
9 >>> [await probe(name) for name in names]
10 [Result(domain='golang.org', found=True),
11 Result(domain='no-lang.invalid', found=False),
12 Result(domain='python.org', found=True),
13 Result(domain='rust-lang.org', found=True)]
```

注意,我对域名列表 `names` 进行了排序,以表明两种情况下的结果都是按照提交的顺序排列的。

“[PEP 530](#)”允许在列表推导式、集合推导式以及字典推导式中使用 `async for` 与 `await`。例如,下面是一个字典推导式,用于在异步控制台中存储 `multi_probe` 的结果:

```
1 >>> {name: found async for name, found in multi_probe(names)}
2 {'golang.org': True, 'python.org': True, 'no-lang.invalid': False, 'rust-lang.org':
True}
```

`await` 关键字可以在 `for` 或 `async for` 子句之前的表达式中使用,也可以在 `if` 语句之后的表达式中使用。如下是在异步控制台中的集合推导式,仅收集找到的域名。

```
1 >>> {name for name in names if (await probe(name)).found}
2 {'rust-lang.org', 'python.org', 'golang.org'}
```

由于 `__getattr__` 运算符(即符号`.`)的优先级较高,我不得不为 `await` 表达式多加一层括号。

同样,所有这些推导式只能出现在 `async def` 主体内部或神奇的异步控制台中。

现在,让我们了解一下 `async` 语句、`async` 表达式,以及用它们创建的对象的重要特性。这些语法结构通常与 `asyncio` 一起使用,但实际上二者是相互独立的。

21.11 asyncio 之外的异步世界:Curio

Python 的 `async/await` 语言构造并不与任何特定的事件循环或库绑定。¹³由于特殊方法提供了可扩展的 API,任何有足够动机的人都可以编写自己的异步运行时环境和框架,以驱动原生协程、异步生成器等。

David Beazley 在他的 [Curio 项目](#)¹⁴中就是这么做的。他有兴趣重新思考如何在一个从零开始构建的框架中使用这些新的语言特性。回想一下, `asyncio` 是在 Python 3.4 中发布的,它使用 “`yield from`” 而不是 `await`,因此它的 API 无法利用异步上下文管理器、异步迭代器以及其他依托 `async/await` 关键字的结构。因此,与 `asyncio` 相比, `Curio` 的 API 更简洁,实现也更简单。

示例 21.11 展示了为使用 `Curio`,而重写的 `blogdom.py` 脚本(见“示例 21.1<621页>”。

```

1  #!/usr/bin/env python3
2  from curio import run, TaskGroup
3  import curio.socket as socket
4  from keyword import kwlist
5
6  MAX_KEYWORD_LEN = 4
7
8  async def probe(domain: str) -> tuple[str, bool]: ❶
9      try:
10         await socket.getaddrinfo(domain, None) ❷
11     except socket.gaierror:
12         return (domain, False)
13     return (domain, True)
14
15 async def main() -> None:
16     names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN)
17     domains = (f'{name}.dev'.lower() for name in names)
18     async with TaskGroup() as group:
19         for domain in domains:
20             await group.spawn(probe, domain) ❸
21         async for task in group:
22             domain, found = task.result
23             mark = '+' if found else '-'
24             print(f'{mark} {domain}') ❹
25
26     if __name__ == '__main__':
27         run(main()) ❺

```

¹³相比之下,JavaScript 中的 `async/await` 结构仅限于内置事件循环和运行时环境,即浏览器、Node.js 或 Deno。

¹⁴`Curio` 是一个基于协程的库,用于并发 Python 系统编程。它提供了标准的编程抽象,如任务、套接字、文件、锁和队列。您会发现它很熟悉、小巧、快速且有趣。

- ❶ 协程 probe 无须获取事件循环,因为 ...
- ❷ ...getaddrinfo 是 curio.socket 的顶层函数,而不像 asyncio 那样,是 loop 对象的方法。
- ❸ TaskGroup 是 Curio 中的一个核心概念,用于监视和控制多个协程,并确保它们全都得到执行和清理。
- ❹ TaskGroup.spawn() 启动协程,将协程封装到 Task 中,使协程由特定的 TaskGroup 实例管理和控制。
- ❺ 用 async for 遍历 TaskGroup,在任务完成时生成 Task 实例。这对应于“[示例 21.1<621页>](#)”中❻处“for ... in asyncio.as_completed(...):”那几行。
- ❻ Curio 开创了这种在 Python 中启动异步程序的明智方式。

展开最后一点:如果看过本书第 1 版中的 `asyncio` 示例,您会看到如下几行反复出现:

```
1 loop = asyncio.get_event_loop()
2 loop.run_until_complete(main())
3 loop.close()
```

Curio TaskGroup 是一种[异步上下文管理器](#),它取代了 `asyncio` 中的多个临时 API 和编程模式。[示例 21.11](#)的❻处指出,我们可以遍历 TaskGroup,而无须调用 `asyncio.as_completed()` 函数。此外,如下代码摘自“[Task Groups](#)文档”,展示了从 TaskGroup 中搜集所有任务的结果,而无需使用`asyncio.gather()`函数:

```
1 async with TaskGroup(wait=all) as g:
2     await g.spawn(coro1)
3     await g.spawn(coro2)
4     await g.spawn(coro3)
5     print('Results:', g.results)
```

任务组支持[结构化并发](#)¹⁵:这是一种并发编程范式,它将一组异步任务的所有活动限制在单个人口点和出口点中。这类似于结构化编程,它摒弃了 GOTO 命令,并引入了块语句来限制循环和子程序的入口点和出口点。当 TaskGroup 作为[异步上下文管理器](#)使用时,它确保在封闭块退出时,所有在封闭块中启动的任务都会被完成或撤销,并处理任何可能引发的异常。



在即将发布的 Python 版本中, `asyncio` 可能会采用[结构化并发](#)。Python 3.11 中已批准的“[PEP 654 –Exception Groups and except*](#)”就是明显迹象。“[Motivation](#)”提到了 Trio 的“托儿所(nurseries)”(任务组的称呼):“受 Trio Nurseries 的启发,为 `asyncio` 实现更好的任务派生 API,是此 PEP 的主要动机。”

Curio 的另一个重要特性是:对在同一代码库中使用协程和线程编程提供了更好的支持——这对大部分重要的异步程序来说是必不可少的。用 `await spawn_thread(func, ...)` 启动线程,返回一个与 Task 接口类似的 `AsyncThread` 对象。该线程可以调用协程,这要归功于特殊的 `AWAIT(coro)` 函数——该函数全部大写命名,因为 `wait` 是关键字。

Curio 还提供了 `UniversalQueue`,可以用于协调线程、Curio 协程和 `asyncio` 协程之间的工作。没错,Curio 与 `asyncio` 可以在同一进程中一起运行,二者通过 `UniversalQueue` 与 `UniversalEvent` 进行通信。这些“通用”类的 API 在协程内部和外部是相同的,但在协程内,需要在调用前(`prefix`)加上 `await`。

截至我在 2021 年 10 月写下这些文字时,HTTPX 是第一个兼容 Curio 的 HTTP 客户端库,但我还尚未听说是否有异步数据库代码库支持 Curio。在 Curio 存储库中有一系列令人印象深刻的[网络编程示例](#),包括一个

¹⁵结构化并发:详见维基百科 https://en.wikipedia.org/wiki/Structured_concurrency。

使用 WebSocket 的示例。还有一个实现了“RFC 8305—Happy Eyeballs”并发算法,用于连接到 IPv6 端点,并在需要时快速回退到 IPv4。

Curio 的设计具有很大的影响力。由 Nathaniel J. Smith 发起的 Trio 框架深受 Curio 的启发。Curio 还促使 Python 贡献者对的可用性做出了改进。例如,在 `asyncio` 早起版本中,用户需要频繁获取和传递 `loop` 对象。因为一些重要函数要么是 `loop` 对象的方法,要么就需要 `loop` 作为参数。在最新版本 Python 中,不再需要直接访问 `loop` 对象。而且,某些接受可选参数 `loop` 的函数,也开始废弃该参数了。

下一个主题是异步类型的 [类型提示 \(Type Hints\)](#)。

21.12 异步对象的类型提示

原生协程的返回类型描述了原生协程主体内 `return` 语句中的对象类型,指明了用 `await` 关键字处理协程时得到的对象类型。¹⁶

本章提供了多个注解原生协程的示例,包括 [示例 21.11 \(①处\)](#) 中的 `probe` 协程。

```
1  async def probe(domain: str) -> tuple[str, bool]: ①
2      try:
3          await socket.getaddrinfo(domain, None) ②
4      except socket.gaierror:
5          return (domain, False)
6      return (domain, True)
```

如果需要注释一个参数,该参数接受协程对象,则使用的 [泛型 \(Generic Type\)](#) 为:

```
1  class typing.Coroutine(Awaitable[V_co], Generic[T_co, T_contra, V_co]):
2      ...
```

上述类型,以及 Python 3.5 和 3.6 引入的如下类型,用于注解异步对象:

```
1  class typing.AsyncContextManager(Generic[T_co]):
2      ...
3
4  class typing.AsyncIterable(Generic[T_co]):
5      ...
6
7  class typing.AsyncIterator(AsyncIterable[T_co]):
8      ...
9
10 class typing.AsyncGenerator(AsyncIterator[T_co], Generic[T_co, T_contra]):
11     ...
12
13 class typing.Awaitable(Generic[T_co]):
14     ...
```

对于 Python ≥ 3.9 ,应使用 `collections.abc` 中的对应类型。

关于这些 [泛型 \(Generic Type\)](#),我想强调如下 3 点:

- 第一,第 1 个类型参数都是 [协变 \(covariance\)](#) 的,此类型参数用于描述异步对象生成(产出)的项的类型。正如“[15.7.4.4 型变经验法则](#)”第 1 条所述:

¹⁶这与经典协程的注解不同,详见“[17.13.3 经典协程的泛型注解](#)”。

如果形式 (Formal) 类型参数定义了从对象中产生的数据类型,那么该形式类型参数可能是协变 (covariance) 的。

- 第二, `AsyncGenerator` 和 `Coroutine` 在倒数第 2 个参数上是逆变 (contravariance) 的。该参数是底层 `send()` 方法的参数类型,事件循环调用该方法来驱动异步生成器和协程 (`Coroutine`) 的执行。因此,属于一种“输入”类型。因此,根据“15.7.4.4 型变经验法则<444页>”第 2 条所述,该参数可能是逆变 (contravariance) 的:

如果一个形式 (Formal) 类型参数定义的是在对象初始化后,向对象中输入的数据类型,则该形式类型参数可能是逆变 (contravariance) 的。

- 第三, `AsyncGenerator` 没有返回值类型,这与“17.13.3 经典协程的泛型注解<524页>”所述的 `typing.Generator` 不同。如“17.13 经典协程<518页>”所述,通过引发 `StopIteration(value)` 来返回一个值,是将生成器用作协程并支持“yield from”的一种技巧。异步对象之间不存在这种重叠:`AsyncGenerator` 对象不会返回值,并且完全独立于原生协程对象。原生协程对象用 `typingCoroutine` 进行注解。

最后一节,简要讨论一下异步编程的优势与挑战。

21.13 异步工作原理与陷阱

本章最后几节,抛开所用语言及库,讨论有关异步编程的高级思想。

首先,说明异步编程吸引人的首要原因;然后,指出一个常见误区,以及如何应对它。

21.13.1 围绕阻塞型调用的循环运行

Ryan Dahl (Node.js 发明者) 在介绍 Node.js 的理念时说:“我们对待 I/O 的方法完全错了。”¹⁷他将文件 I/O 或网络 I/O 相关的函数定义为“阻塞型函数”,并认为我们不能像对待非阻塞函数那样对待阻塞型函数。为了解释原因,他给出了表 21.2 第 2 列中的数字。

表 21.2: 现代计算机从不同设备读取数据的延迟¹⁹

设备类型	CPU 循环 (计算机时间)	换算为人类时间
L1 缓存	3	3 秒
L2 缓存	14	14 秒
RAM	250	250 秒
磁盘	41,000,000	1.3 年
网络	240,000,000	7.6 年

要理解表 21.2,需要牢记:具有 GHz 时钟的现代 CPU 每秒可运行数十亿个周期。假设 CPU 每秒正好运行 10 亿个周期。那么,1 秒内该 CPU 可读取 L1 缓存超过 3.33 亿次,或者在同样时间内读取 4 次网络。表 21.2 的第 3 列通过对第 2 列乘以一个常数因子,进行换算来让这些数字更具体化。因此,在另一个宇宙中,

¹⁷ 视频:“Introduction to Node.js”,4 分 55 秒处。

¹⁹ 第 3 列是按一定比例换算的时间,方便人类理解。

如果读取 L1 缓存需要耗时 3 秒,那么读取网络将要耗时 7.6 年。这个比较突出显示了读取 L1 缓存的速度相比读取网络而言有多快。

表 21.2 表明,规范的异步编程方法可以提高服务器的性能。而实现这种规范是一种挑战。第一步是认清一个事实,即“I/O 密集型系统”只是一种幻觉。

21.13.2 I/O 密集型系统的误区

人们常说“异步编程适合用于 I/O 密集型系统”。而我从艰辛的经验中认识到,其实并不存在所谓的“I/O 密集型系统”。您遇到的可能是 I/O 密集型函数。也许您系统中绝大多数函数都是 I/O 密集型函数——与处理数据相比,耗费在等待 I/O 操作上的时间更多。在等待 I/O 之时,它们将控制权交给事件循环,事件循环可以驱动其他挂起的任务。然而,不可避免的是,任何非平凡系统都会有一些组件是 CPU 密集型的。即便是微不足道的系统,也会在压力下揭示这一点。在本章“21.15 延伸阅读”的“杂谈”中,我讲了一个故事——2 个异步程序因受 CPU 密集型函数牵连,拖慢事件循环速度,从而严重影响性能。

鉴于任何重要的系统都会有 CPU 密集型函数,因此处理这些函数就是异步编程成功的关键。

21.13.3 避免 CPU 密集型陷阱

如果正在大规模使用 Python,应该备有专门的自动化测试,以便在出现性能衰退时,尽早发现性能问题。这对异步代码至关重要,而且因为 GIL 的存在,这对 Python 多线程代码也不可或缺。如果等到速度慢到影响开发团队的工作,那就太晚了。那时再修复可能需要做出伤筋动骨的改动。

当发现 CPU 占用出现瓶颈时,可采取如下措施:

- 将任务委托给 Python 进程池。
- 将任务委托给外部任务队列。
- 使用 CPython、C、Rust,或可编译为机器码,并能与 Python/C API 交互的其他语言(最好能释放 GIL)重写相关代码。
- 如若可以承受性能损失,则可以什么也不做。但要将此决定记录下来,以便以后更容易恢复它。

在项目开始时就应该尽快选择和集成外部任务队列,以便团队成员在需要时拿来即用,无需犹豫。

最后一项措施是什么多不做,但这容易留下技术债务。

并发编程是一个引人入胜的话题,我想多写一些关于它的内容。但该主题并非本书重点,而且本章内容已经很长了。所以,就此打住吧!

21.14 本章小结

常规异步编程方法的问题在于它们都是非此即彼的命题。你应该重写所有代码,以便测底去除阻塞,否则你只是在浪费时间。

——Alvaro Videla 与 Jason J. W. Williams,《RabbitMQ in Action》

本章开篇引用这段引文有 2 个原因。总体而言,它提醒我们通过将较慢的任务委托给其他处理单元(如简单的线程或分布式任务队列均可),可避免阻塞事件循环。具体而言,这也是一个警告:一旦编写了第 1 个

async def, 你的程序不可避免地会有越来越多的 async def、await、async with 和 async for。那时, 再想使用非异步库, 面临的挑战可想而知。

“[十九 Python 中的并发模型](#)”简单实现了几个 spinner 示例, 本章关注的重点是用[原生协程](#)进行异步编程。本章首先分析了 DNS 探测示例 `blogdom.py`, 随后介绍了异步可调用对象。在阅读 `flags_asyncio.py` 源码过程中, 见到了首个[异步上下文管理器](#)示例。

高级版的国旗图片下载程序引入了 2 个强大的函数:`asyncio.as_completed`生成器和 `loop.run_in_executor` 协程。随后, 介绍了信号量的概念和应用——通过信号量限制并发下载数, 规范的 HTTP 客户端本该如此。

通过 `mojifinder` 示例, 介绍了服务器端异步编程: `FastAPI` Web 服务、`tcp_mojifinder.py`。后者仅使用了 `asyncio` 与 TCP 协议。

异步迭代器与异步可迭代对象是下一个主要主题, 其中介绍了 `async for`、Python 异步控制台、异步生成器、异步生成器表达式与异步推导式。

本章最后一个示例是用 `Curio` 框架重写的 `blogdom.py`, 以展示 Python 的异步特性并不依赖于 `asyncio` 包。`Curio` 还展示了[结构化并发](#)的概念, 此概念对业界有一定影响, 使写出的并发代码更清晰易懂。

最后, “[21.13 异步工作原理与陷阱<656页>](#)”介绍了异步编程的主要吸引力、对“[I/O 密集型系统](#)”的误解, 以及如何处理程序中不可避免的 CPU 密集型部分。

21.15 延伸阅读

David Beazley 在 PyOhio 2016 的主题演讲 “[Fear and Awaiting in Async](#)” 非常精彩, 通过现场编程介绍了 Yury Selivanov 为 Python 3.5 贡献的 `async/await` 关键字的潜力。在演讲中, Beazley 抱怨 `await` 不能在列表推导式中使用, 但 Selivanov 在 “[PEP 530 -Asynchronous Comprehensions](#)” 中对此问题进行了修复, 并在同年发布的 Python 3.6 中实现。除此之外, Beazley 演讲的内容现在看来也不过时, 他未借助任何框架, 只用一个简单的 `run` 函数通过 `.send(None)` 驱动协程运行, 就揭示了异步对象的工作原理。Beazley 在演讲最后才提到 `Curio`。这是他在那一年开始做的一个实验——在不用回调与 `future` 对象的情况下, 仅用协程做异步编程能完成多少工作。结果表明, 可以做很多事情——`Curio`一直有新版发布, 还启发 Nathaniel J. Smith 创造了 `Trio`。`Curio` 文档中有[更多链接](#), 指向 Beazley 在该主题上的演讲。

除了创建 `Trio`, Nathaniel J. Smith 还撰写了两篇深度博文, 我强烈推荐: 一篇名为 “[Some thoughts on asynchronous API design in a post-async/await world](#)”, 对比了 `Curio` 与 `asyncio` 的设计; 另一篇是关于[结构化并发](#)的 “[Notes on structured concurrency, or: Go statement considered harmful](#)”。此外, Smith 在 StackOverflow 上对问题 “[What is the core difference between asyncio and trio?](#)” 做出了详尽回答。

要了解更多关于 `asyncio` 包的内容, 可以看一下我在本章开头提到的最好的文字资料: 经 Yury Selivanov 于 2018 年开始修订的[官方文档](#), 以及《[Using Asyncio in Python](#)》(Caleb Hattingh 著)一书。在官方文档中, 务必阅读 “[Developing with asyncio](#)”, 其中介绍了 `asyncio` 调试模式, 并讨论了常见的错误和陷阱以及如何避开它们。

Miguel Grinberg 在 PyCon 2017 上的演讲 “[Asynchronous Python for the Complete Beginner](#)”介绍了异步编程的通用知识, 也涉及了 `asyncio`。该演讲只有 30 分钟, 非常通俗易懂。Michael Kennedy 的演讲 “[Demystifying Python's Async and Await Keywords](#)”, 也是一个很棒的入门资料。在该演讲中, 我学到了一些关于 `unsync` 库的知识, 该库提供了一个装饰器, 可根据需要将协程、I/O 密集型函数、CPU 密集型函数的执行, 委托给 `asyncio`、`threading` 或 `multiprocessing`。

在 EuroPython 2019 上,PyLadies 全球领导者 Lynn Root 做了题为“Advanced asyncio: Solving Real-world Production Problems”的精彩演讲,其内容是她作为 Spotify 工程师使用 Python 的经验之谈。

2020 年,Łukasz Langa 录制了一系列关于 `asyncio` 的精彩视频,首先是“Learn Python's AsyncIO #1 - The Async Ecosystem”。Langa 还为 PyCon 2020 制作了超酷的视频“`AsyncIO + Music`”,该视频不仅展示了 `asyncio` 在具体的面向事件领域中的应用,还对 `asyncio` 进行了深入浅出的讲解。

另一个以事件为导向的编程领域是嵌入式系统。所以,Damien George 为 MicroPython²⁰ 添加对 `async/await` 的支持。在 PyCon Australia 2018 上,Matt Trentini 演示了 `uasyncio` 库,这是 `asyncio` 的一个子集,是 MicroPython 标准库的一部分。

有关 Python 异步编程的更高层次思考,请阅读 Tom Christie 的博文“`Python async frameworks - Beyond developer tribalism`”。

最后,我推荐阅读 Bob Nystrom 的文章“What Color is Your Function?”,其中讨论了常规函数与异步函数(即协程)在 JavaScript、Python、C# 等语言中不兼容的执行模型。剧透警告:Nystrom 的结论是,做对了这一点的语言是 Go,即 Go 语言的所有函数都是同一颜色。我喜欢 Go 的这种做法。但我也认为 Nathaniel J. Smith 在“`Go statement considered harmful`”文中中说的也有道理。世间不完美,并发编程始终是一个难题。

杂谈

一个速度慢的函数,差点破坏 uvloop 基准测试

2016 年,Yury Selivanov 发布了 `uvloop`,它是“一个快速、可直接替换内置 `asyncio` 事件循环的库。”Selivanov 在 2016 年发布该库的博客文章中提供的基准测试结果令人印象非常深刻。他写道:“`uvloop` 至少比 nodejs、gevent 以及其他任何 Python 异步框架快 2 倍。基于 `uvloop` 的 `asyncio` 的性能接近于 Go 程序的性能。”

但是,该文章也指出,若想与 Go 性能相当,需满足如下 2 个条件:

1. Go 配置为使用单线程。这使得 Go 运行时的行为类似于 `asyncio`:并发性是通过事件循环驱动的多个协程实现的,所有这些都在同一个线程中。^a
2. 除了 `uvloop` 本身之外,Python 3.5 代码还要使用 `httptools`。

Selivanov 解释说,他在使用 `aiohttp`(第一个基于 `asyncio` 构建的全功能 HTTP 库之一)对 `uvloop` 进行基准测试后编写了 `httptools`:

然而, `aiohttp` 的性能瓶颈竟然是其 HTTP 解析器,该解析器速度如此之慢,以至于底层 I/O 库多快都几乎无济于事。为了让事情更有趣,我们为 `http-parser`(最初为 NGINX 开发的 Node.js HTTP 解析器库,用 C 实现)创建了一个 Python 绑定。该库名为 `httptools`,可在 Github 和 PyPI 上找到。

现在想想看:Selivanov 的 HTTP 性能测试包括在不同的语言/库中编写的一个简单的回显服务器;然后,使用 `wrk` 基准测试工具来对其进行测试。大多数开发人员会认为一个简单的回显服务器应该是一个“I/O 密集型系统”,对吧?但事实证明,解析 HTTP 头是一个 CPU 密集型任务,而在 Selivanov 在 2016 年进行基准测试时, `aiohttp` 中用 Python 实现的 HTTP 头部解析速度较慢。每当用 Python 编写的函数在解析 HTTP 头时,事件循环都会被阻塞。这种影响很明显,以至于 Selivanov 不得不额外编写了 `httptools`。

²⁰MicroPython:是一款针对微控制器的 Python 解释器。

如果不优化 CPU 密集型代码, 那么事件循环带来的性能提升将会被抵消。

积羽沉舟

想象一个复杂且不断发展的 Python 系统, 它具有数万行异步代码, 与许多外部库交互, 而不是简单的回显服务器。几年前, 我被要求帮助诊断此类系统中的性能问题。它是用 Python 2.7 和 Twisted 框架编写的。Twisted 是一个可靠的库, 在很多方面可称得上是 asyncio 的先驱。

Python 用于构建 Web UI 的外观, 集成了现有库和用其他语言编写的命令行工具提供的功能, 但不是为并发执行而设计的。

该项目雄心勃勃; 它已经开发了一年多, 但尚未投产。^b 随着时间的推移, 开发人员注意到整个系统的性能正在下降, 并且很难找到瓶颈。

问题现状是: 随着新功能的增加, 就会有更多的 CPU 密集型代码拖慢 Twisted 的事件循环。Python 作为一种黏合语言, 意味着需要进行大量的数据解析和格式转换。瓶颈并非出现在单个点上: 问题分散在数月开发过程中添加的无数个小型函数中。要解决此问题, 需要重新思考系统架构, 并重写大量代码。可能需要利用任务队列, 也可能需要使用适合于 CPU 密集型并发处理的微服务或自定义库。然而, 项目方并未准备好进行这种额外投资, 项目随后不久被取消了。

当我把这个故事告诉了 Twisted 项目创始人 Glyph Lefkowitz 时, 他表示, 在准备开发一个异步编程项目时, 他的首要任务之一是决定使用哪些工具来处理 CPU 密集型任务。受到此次对话的启发, 我撰写了“[21.13.3 避免 CPU 密集型陷阱](#)”一节。

^a 在 Go 1.5 发布之前, 使用单线程是默认设置。几年前, Go 已经因支持高并发的网络系统而赢得了当之无愧的声誉。又一个证据表明并发不需要多个线程或 CPU 核心。

^b 抛开技术选择, 该项目中最大的问题是: 项目方未采用 MVP (Minimum Viable Product) 方法——尽快交付最简可行产品; 然后, 稳步增加功能。

PART V

第五部分

元编程

- 第 22 章 动态属性和特性
- 第 23 章 属性描述符
- 第 24 章 [类元编程 \(Class Metaprogramming \)](#)

wechat: 119554488

动态属性和特性

特性 (property) 之所以极其重要, 是因为它的存在使得开发者可将公共数据属性作为类的公共接口的一部分, 进行暴露是完全安全且确实可取的。

——Martelli, Ravenscroft 与 Holden, “特性的重要性”^a

^a摘自《Python in a Nutshell, 3rd Ed》(Alex Martelli, Anna Ravenscroft, Steve Holden 合著) 第 123 页。

在 Python 中, 数据属性和方法 (Method) 被统称为“属性 (attribute)”。方法 (Method) 是一种可调用的属性。动态属性 (Dynamic Attribute) 提供与数据属性 (即 obj.attr) 相同的接口, 但要按需计算。这与 Bertrand Meyer 所说的“统一访问原则 (Uniform Access Principle)”相符:

模块应使用统一的表示法为用户提供所有服务, 这样做不会泄露模块内的服务实现方式 (通过存储实现, 还是通过计算实现)。^a

^a摘自《Object-Oriented Software Construction, 2nd Ed》(Bertrand Meyer 著) 第 57 页。

Python 有多种可实现动态属性的方式。本章介绍最简单的 2 种: `@property` 装饰器和 `__getattr__()` 方法。用户定义的类通过实现 `__getattr__` 方法, 可以实现一种我称之为“虚拟属性 (Virtual Attribute)”的动态属性: 这种虚拟属性不会在类的源码中显式声明, 也不在实例的 `__dict__` 中, 但可以在用户尝试读取不存在的属性 (如 `obj.no_such_attr`) 时, 可被另行检索或即时动态计算。

创建动态属性与虚拟属性是编写框架时, 常用的一种“元编程 (Metaprogramming)”技术。然而, 在 Python 中, 相关的基础技术很简单, 因此也可在日常数据处理任务中使用这 2 种属性。下面以这种任务, 开始本章主题的介绍。

22.1 本章新增内容

本章的改动大多是为了讨论 `@functools.cached_property` (Python 3.8 引入), 及 `@property` 与 `@functools.cache` (Python 3.9 新增) 的结合使用。受此影响, “[22.3 计算的特性](#)”中的 Record 类与 Event 类的代码都有所改动。同时, 还进行了重构, 以便利用“[PEP 412 -Key-Sharing Dictionary](#)”提出的优化措施。

为了突出更相关的特性，并保持示例的可读性，我删除了一些非必要的代码——将旧的 `DbRecord` 类并入到 `Record` 中，用 `dict` 替换了 `shelve.Shelve`，并删除了下载 OSCON 数据集的逻辑——现在的示例从“[随书源码库](#)”中的本地文件中读取数据。

22.2 利用动态属性进行数据转换

接下来的几个示例将利用动态属性处理 O'Reilly 为 OSCON 2014 大会发布的日程数据集（JSON 格式）。示例 22.1 显示了该数据集¹中的 4 条记录。

</> 示例 22.1: `osconfeed.json` 文件的示例记录（省略了部分字段内容）

```

1  {
2      "Schedule": {
3          "conferences": [
4              {
5                  "serial": 115
6              },
7              {
8                  "events": [
9                      {
10                         "serial": 34505,
11                         "name": "Why Schools Don't Use Open Source to Teach Programming",
12                         "event_type": "40-minute conference session",
13                         "time_start": "2014-07-23 11:30:00",
14                         "time_stop": "2014-07-23 12:10:00",
15                         "venue_serial": 1462,
16                         "description": "Aside from the fact that high school programming...",
17                         "website_url": "http://oscon.com/oscon2014/public/schedule/detail/34505",
18                         "speakers": [
19                             157509,
20                         ],
21                         "categories": [
22                             "Education"
23                         ]
24                     },
25                     {
26                         "speakers": [
27                             {
28                                 "serial": 157509,
29                                 "name": "Robert Lefkowitz",
30                                 "photo": null,
31                                 "url": "http://sharewave.com/",
32                                 "position": "CTO",
33                                 "affiliation": "Sharewave",
34                                 "twitter": "sharewaveteam",
35                                 "bio": "Robert 'r0ml' Lefkowitz is the CTO at Sharewave, a startup..."
36                         }
37                     ],
38                     "venues": [
39                         {
40                             "serial": 1462,
41                             "name": "F151",
42                             "category": "Conference Venues"
43                         }
44                     ]
45                 }
46             ]
47         }
48     }
49 
```

¹由于 COVID-19 的原因，OSCON（O'Reilly 开源会议）近几年未举办。自 2021 年 1 月 10 日起，这些示例所用的那个 744 KB 原始 JSON 文件已不可在线访问。但可在示例代码存储库中找到 `osconfeed.json` 副本。

```
36 ]  
37 }
```

该 JSON 文件中共有 895 条记录, [示例 22.1](#) 仅显示了其中的 4 条。整个日程数据集是一个包含键为 “Schedule” 的 JSON 对象, 其值是另一个映射, 包含 4 个键: “conferences”、“events”、“speakers” 和 “venues”。这 4 个键中的每个键对应的值都是一个记录列表。在完整的数据集中, 列表 “events”、“speakers” 和 “venues” 中有数十或数百条记录, 而 “conferences” 只包含 [示例 22.1](#) 中显示的那一条记录。每条记录都有一个 “serial” 字段, 这是列表中各条记录的唯一标识符。

为了了解此日程数据集的结构, 我在 Python 控制台中进行了探索, 如 [示例 22.2](#) 所示。

</> [示例 22.2](#): 在交互式控制台中探索 `osconfeed.json`

```
1  >>> import json  
2  >>> with open('data/osconfeed.json') as fp:  
3  ...     feed = json.load(fp)         ❶  
4  >>> sorted(feed['Schedule'].keys())   ❷  
5  ['conferences', 'events', 'speakers', 'venues']  
6  >>> for key, value in sorted(feed['Schedule'].items()):  
7  ...     print(f'{len(value):3} {key}')   ❸  
8  ...  
9  1 conferences  
10 484 events  
11 357 speakers  
12 53 venues  
13 >>> feed['Schedule']['speakers'][-1]['name']   ❹  
14 'Carina C. Zona'  
15 >>> feed['Schedule']['speakers'][-1]['serial'] ❺  
16 141590  
17 >>> feed['Schedule']['events'][40]['name']  
18 'There *Will* Be Bugs'  
19 >>> feed['Schedule']['events'][40]['speakers'] ❻  
20 [3471, 5199]
```

- ❶ `feed` 的值是一个字典, 该字典中包含嵌套字典与列表, 存储着字符串与整数值。
- ❷ 列出 `Schedule` 键中的 4 个记录容器。
- ❸ 显示各个容器中的记录数。
- ❹ 浏览嵌套的字典与列表, 获取最后一个演讲者的名字。
- ❺ 获取❹处演讲者的序号 (`serial`)。
- ❻ 每个活动都有一个演讲者 (`speakers`) 列表, 其中包含 0 个或多个演讲者序号。

22.2.1 用动态属性访问 JSON 类数据

[示例 22.2](#) 相当简单, 但 `feed["Schedule"]["events"][40]["name"]` 这种语法有些繁琐。在 JavaScript 中, 可以用 `feed.Schedule.events[40].name` 获取相同的值。在 Python 中, 很容易实现一个类似于字典的类², 可

²例如, `AttrDict` 与 `addict`

以实现同样的功能——网上有许多示例。我自己编写了 FrozenJSON 类, 比大多数示例更简单。因为它只支持读取: 即只能用于访问数据。不过, FrozenJSON 类也支持递归, 可自动处理嵌套的映射和列表。

示例 22.3 展示了 FrozenJSON 类的用法, 源码如示例 22.4 所示:

</> 示例 22.3: FrozenJSON 不仅可读取属性, 还可调用方法

```

1  >>> import json
2  >>> raw_feed = json.load(open('data/osconfeed.json'))
3  >>> feed = FrozenJSON(raw_feed)
4  >>> len(feed.Schedule.speakers) ❶
5  357
6  >>> feed.keys()
7  dict_keys(['Schedule'])
8  >>> sorted(feed.Schedule.keys()) ❷
9  ['conferences', 'events', 'speakers', 'venues']
10 >>> for key, value in sorted(feed.Schedule.items()): ❸
11 ...     print(f'{len(value):3} {key}')
12 ...
13 1 conferences
14 484 events
15 357 speakers
16 53 venues
17 >>> feed.Schedule.speakers[-1].name ❹
18 'Carina C. Zona'
19 >>> talk = feed.Schedule.events[40]
20 >>> type(talk) ❺
21 <class 'explore0.FrozenJSON'>
22 >>> talk.name
23 'There *Will* Be Bugs'
24 >>> talk.speakers
25 [3471, 5199]
26 >>> talk.flavor ❻
27 Traceback (most recent call last):
28 ...
29 KeyError: 'flavor'
```

- ❶ 传入由嵌套字典与列表组成的 raw_feed, 创建一个 FrozenJSON 实例。
- ❷ FrozenJSON 允许用属性表示法遍历嵌套的字典; 此处获取了演讲者列表 (speakers) 的长度。
- ❸ 也可以用底层字典的方法 (如.keys()), 获取存放记录的容器名。
- ❹ 用 items() 可以获取存放记录的容器名及其内容; 然后, 用 len() 显示各个容器的长度。
- ❺ 列表 (如 feed.Schedule.speakers) 仍是列表, 但若其中的项是映射, 则该项会被转换为 FrozenJSON。
- ❻ event 列表中索引位为 40 的项是一个 JSON 对象, 而现在它是一个 FrozenJSON 实例。
- ❼ 事件 (event) 记录中有一个 speakers 列表, 其中列出了演讲者的序号 (serial)。
- ❽ 尝试读取缺失的属性会引发 KeyError, 而不是通常的 AttributeError。

FrozenJSON 类的关键是 `__getattr__` 方法, “[12.6 Vector 类第 3 版: 动态访问属性](#)”的 Vector 例子中用到了该方法, 那时是为了按字母检索 Vector 分量, 如 `v.x`、`v.y`、`v.z` 等。重点是要记住: 仅当无法通过常规

方式获取属性(即,在实例、类或超类中找不到指定的属性)时,Python 解释器才会调用特殊方法 `__getattr__`。

示例 22.3 最后一行(❸处)揭示了代码中的一个小问题:尝试读取缺失属性时,本应引发 `AttributeError`,而不是示例中所示的 `KeyError`。其实,起初我对此异常做了处理,但 `__getattr__` 方法的代码量增加了 1 倍,而且偏离了我想要展示的重要逻辑。鉴于用户知道 `FrozenJSON` 是由映射和列表构建的,所以我想 `KeyError` 也不至于太令人困惑。

</> 示例 22.4: `explore0.py`: 将 JSON 数据集转换为包含嵌套 `FrozenJSON` 对象、列表和简单类型的

```

1 from collections import abc
2
3 class FrozenJSON:
4     """A read-only façade for navigating a JSON-like object
5     using attribute notation
6     """
7
8     def __init__(self, mapping):
9         self.__data = dict(mapping) ❶
10
11     def __getattr__(self, name): ❷
12         try:
13             return getattr(self.__data, name) ❸
14         except AttributeError:
15             return FrozenJSON.build(self.__data[name]) ❹
16
17     def __dir__(self): ❺
18         return self.__data.keys()
19
20     @classmethod
21     def build(cls, obj): ❻
22         if isinstance(obj, abc.Mapping): ❼
23             return cls(obj)
24         elif isinstance(obj, abc.MutableSequence): ❼
25             return [cls.build(item) for item in obj]
26         else: ❼
27             return obj

```

- ❶ 用参数 `mapping` 构建一个字典。这样可以确保传入的内容可以被转换为映射。`__data` 的双下划线前缀,使其成为一个“私有属性”。
- ❷ 仅当不存在指定名为 `name` 的属性时,Python 解释器才会调用 `__getattr__`。
- ❸ 如果 `name` 匹配了字典 `__data` 中的某个属性,则返回该属性。`feed.keys()` 调用,就是这样处理的:`keys` 方法是字典 `__data` 的一个属性。
- ❹ 否则,从 `self.__data` 中获取 `name` 键对应的项,并返回对该项调用 `FrozenJSON.build()` 的结果。³
- ❺ 实现为内置函数 `dir()` 提供支持的 `__dir__` 方法。`dir()` 可以支持在标准 Python 控制台、IPython、Jupyter Notebook 等环境中的自动补全。`__dir__` 方法的代码很简单,将基于 `self.__data` 中的键实现递归自动

³ 表达式 `self.__data[name]` 可能会引发 `KeyError` 异常。理想情况下,应该处理此异常并引发 `AttributeError`。因为这才是 `__getattr__` 期望引发的异常种类。建议勤奋的读者实现此错误处理的代码,以作为练习。

补全。因为 `__getattr__` 方法能动态构建 `FrozenJSON` 实例, 以便于用交互方式探索数据。

- ❶ 这是一个备选的构造方法, 这是`@classmethod`装饰器的常见用途。
- ❷ 如果 `obj` 是映射, 则用它构建 `FrozenJSON`。此处利用了 [大鹅类型 \(Goose Typing\)](#), 详见“[13.5 大鹅类型 \(Goose Typing\) <358页>](#)”。
- ❸ 如果 `obj` 是 `MutableSequence` 对象, 则它一定是一个列表。⁴因此, 将 `obj` 中的每个项都递归地传递给`.build()`, 以构建一个列表。
- ❹ 如果 `obj` 既不是字典, 也不是列表, 则原样返回 `obj`。

`FrozenJSON` 实例将 `__data` 私有实例属性存储在名称为 `_FrozenJSON__data` 的属性中 (详见“[11.10 Python 私有属性与“受保护”的属性<310页>](#)”)。尝试通过其他名称检索属性将触发 `__getattr__` 方法。该方法首先查看 `self.__data` 字典是否具有该名称的属性 (而不是键); 这使得 `FrozenJSON` 实例能够处理 `dict` 的方法, 例如可以将`.items()` 委托给 `self.__data.items()`。如果 `self.__data` 中没有给定名称的属性, `__getattr__` 将以该名称为“键 (Key)”, 从字典 `self.__data` 中检索项, 并将该项传递给 `FrozenJSON.build` 方法。这样便能深入 JSON 数据的嵌套结构, 因为类方法 `build` 将每个嵌套映射都转换为另一个 `FrozenJSON` 实例。

注意, `FrozenJSON` 不会转换或缓存原始数据集。在遍历数据过程中, `__getattr__` 反复地创建 `FrozenJSON` 实例。对于这种体量的数据集, 以及仅用于探索或转换数据的脚本来说, 这么做是可以的。

从随机数据源中生成 (或仿效) 动态属性名的 Python 脚本都会面临一个问题: 原始数据中的键可能不适合用作属性名称。下一节, 将解决这个问题。

22.2.2 属性名无效的问题

`FrozenJSON` 类无法处理与 Python 关键字同名的属性名。例如, 对于以下对象:

```
1 >>> student = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

将无法读取 `student.class`, 因为 `class` 是 Python 的保留关键字:

```
1 >>> student.class
2   File "<stdin>", line 1
3     student.class
4       ^
5 SyntaxError: invalid syntax
```

但可以这样做:

```
1 >>> getattr(student, 'class')
2 1982
```

但 `FrozenJSON` 的理念是提供方便的数据访问, 因此更好的解决方案是检查传给 `FrozenJSON.__init__` 方法的映射中, 是否存在键名为关键字的键。如果存在, 则在关键字后面加上后缀 `_`, 这样就可以像这样读取属性了:

```
1 >>> student.class_
2 1982
```

⁴数据源是 JSON 格式, 而 JSON 中的容器类型只有 `dict` 和 `list`。

将示例 22.4 中的 `__init__` 替换为示例 22.5 中的版本, 即可实现这一目标。

</> 示例 22.5: `explore1.py`: 为名称是 Python 关键字的属性增加后缀 `_`

```
1 import keyword
2
3 # 省略多行代码
4 def __init__(self, mapping):
5     self.__data = {}
6     for key, value in mapping.items():
7         if keyword.iskeyword(key): ❶
8             key += '_'
9         self.__data[key] = value
```

❶ `keyword.iskeyword(...)` 函数正是我们所需要的; 要使用该函数, 必须导入 `keyword` 模块。

如果 JSON 记录中的“键 (Key)”不是有效的 Python 标识符, 也会出现类似的问题:

```
1 >>> x = FrozenJSON({'2be': 'or not'})
2 >>> x.2be
3   File "<stdin>", line 1
4     x.2be
5     ^
6 SyntaxError: invalid syntax
```

在 Python 3 中, 这种有问题的“键 (Key)”很容易检测到, 因为 `str` 类提供了 `s.isidentifier()` 方法, 它可以根据语法判断 `s` 是否是有效的 Python 标识符。但要把一个不是有效标识符的“键 (Key)”变成一个有效的属性名并非易事。一种解决方案是实现特殊方法 `__getitem__`, 以允许使用 `x['2be']` 表示法来访问属性。本示例为了简单起见, 将忽略这个问题。

在对动态属性的名称稍作处理后, 接下来分析 `FrozenJSON` 的另一个基本特性: 类方法 `build` 的处理逻辑。`FrozenJSON.build` 被 `__getattr__` 使用, 以根据所访问属性的值返回不同类型的对象: 嵌套结构被转换为 `FrozenJSON` 实例或由 `FrozenJSON` 实例组成的列表。

同样的逻辑除了可以用类方法实现, 还可以用特殊方法 `__new__` 实现。详见下一节。

22.2.3 用 `__new__` 灵活创建对象

我们经常将 `__init__` 称作“构造方法”, 这是因为沿用了其他语言中的术语。在 Python 中, `__init__` 将 `self` 作为第 1 个参数, 可见当 Python 解释器调用 `__init__` 时, 对象已经存在。另外, `__init__` 无法返回任何内容。因此, 这确实是一个初始化方法, 而不是构造方法。

当调用一个类来创建实例时, 为了构建实例, Python 调用的特殊方法是 `__new__`。它是一个经过特殊处理的类方法 (Class Method), 所以不必为其应用 `@classmethod` 装饰器。Python 将类方法 `__new__` 返回的实例传递给 `__init__` 的第 1 个参数 `self`。我们几乎无需自己编写 `__new__` 方法, 因为从 `object` 继承的实现足以满足绝大多数的使用场景。

如有必要, 类方法 `__new__` 也可以返回其他类的实例。当发生这种情况时, Python 解释器不会调用 `__init__`。换句话说, Python 构建对象的逻辑类似于下面的伪代码:

```
1 # 构建对象的伪代码
```

```

2 def make(the_class, some_arg):
3     new_object = the_class.__new__(some_arg) ①
4     if isinstance(new_object, the_class):
5         the_class.__init__(new_object, some_arg)
6     return new_object
7
8 # 以下2个语句的作用基本等同
9 x = Foo('bar')
10 x = make(Foo, 'bar')

```

示例 22.6 展示了 FrozenJSON 的一个变体, 将之前类方法 build 中的处理逻辑移到了类方法 __new__ 中。

</> 示例 22.6: explore2.py:

```

1 from collections import abc
2 import keyword
3
4 class FrozenJSON:
5     """一个只读接口, 该接口用属性表示法访问JSON类对象
6     """
7
8     def __new__(cls, arg):
9         if isinstance(arg, abc.Mapping): ①
10            return super().__new__(cls)
11        elif isinstance(arg, abc.MutableSequence): ②
12            return [cls(item) for item in arg]
13        else: ③
14            return arg
15
16     def __init__(self, mapping):
17         self.__data = {}
18         for key, value in mapping.items():
19             if keyword.iskeyword(key):
20                 key += '_'
21             self.__data[key] = value
22
23     def __getattr__(self, name):
24         try:
25             return getattr(self.__data, name)
26         except AttributeError:
27             return FrozenJSON(self.__data[name]) ④
28
29     def __dir__(self):
30         return self.__data.keys()

```

① __new__ 是类方法 (Class Method), 其第 1 个参数是类本身, 其余参数与 __init__ 相同 (但没有 self)。

② 默认行为是委托给超类的类方法 __new__。在本例中, 调用的是基类 object 的 __new__, 传入的唯一参数是 FrozenJSON 类。

- ③ 类方法 `__new__` 的余下代码与先前的 `build` 方法完全一致。
- ④ 之前此处调用的是 `FrozenJSON.build`；现在只需调用 `FrozenJSON` 类，Python 将调用 `FrozenJSON.__new__` 来处理该类。

类方法 `__new__` 将获取类作为第 1 个参数，因为通常创建的对象就是该类的实例。因此，在 `FrozenJSON.__new__` 中，当表达式 `super().__new__(cls)` 有效地调用 `object.__new__(FrozenJSON)` 时，`object` 类构建的实例实际上是 `FrozenJSON` 的实例。新实例的 `__class__` 属性将保存对 `FrozenJSON` 类的引用。不过，真正的构建操作是由 Python 解释器在内部调用 `object.__new__`（用 C 语言实现）完成的。

OSCON 日程数据集的 JSON 结构不太适合交互式探索。例如，索引为 40 的活动（活动名为 “There *Will* Be Bugs”）有 2 位演讲者，分别是 3471 和 5199。但查找他们的姓名非常困难，因为提供的是序号，而 `Schedule.speakers` 列表并没有按序号编制索引。所以，要找到每个演讲者，必须遍历列表 `Schedule.speakers`，直至找到与序号匹配的记录。我们的下一项任务是调整数据结构，为自动获取所链接的记录做好准备。

22.3 计算的特性

在 “11.7 可哈希的 `Vector2d`” 一节中，首次见到 `@property` 装饰器。“示例 11.7” 中的 `Vector2d` 用到了 2 个特性（`property`），只是为了将 `x` 和 `y` 分量都设置为只读属性。本节将介绍计算值的特性（`property`），顺带讨论如何缓存计算得到的值。

在 OSCON 日程数据（JSON 格式）中，“活动（events）”列表中的记录包含一些整数序号（`serial`），这些序号指向“演讲者（speakers）”与“会场（venues）”列表中的记录。例如，如下是关于某次会议演讲的一条记录（省略了描述）：

```

1 { "serial": 33950,
2   "name": "There *Will* Be Bugs",
3   "event_type": "40-minute conference session",
4   "time_start": "2014-07-23 14:30:00",
5   "time_stop": "2014-07-23 15:10:00",
6   "venue_serial": 1449,
7   "description": "If you're pushing the envelope of programming...",
8   "website_url": "http://oscon.com/oscon2014/public/schedule/detail/33950",
9   "speakers": [3471, 5199],
10  "categories": ["Python"]
11 }
```

我们将实现一个具有 `venue` 与 `speakers` 属性的 `Event` 类，以便自动返回所链接的数据。换句话说，“取消引用”序号。给定一个 `Event` 实例，示例 22.7 展示了 `Event` 类要实现的行为。

</> 示例 22.7：读取 `venue` 与 `speakers`，返回 `Record` 对象

```

1 >>> event
2 <Event 'There *Will* Be Bugs'>
3 >>> event.venue
4 <Record serial=1449>
5 >>> event.venue.name
6 'Portland 251'
7 >>> for spkr in event.speakers: ④
```

```

8 ...     print(f'{spkr.serial}: {spkr.name}')
9 ... 3471: Anna Martelli Ravenscroft
10 5199: Alex Martelli

```

- ❶ 给定一个 Event 实例 ...
- ❷ ... 读取 event.venue 会返回一个 Record 对象, 而不是序号。
- ❸ 现在, 很容易就可获取场地 (venue) 的名称。
- ❹ event.speakers 特性 (property) 会返回包含 Record 实例的列表。

像往常一样, 我们将逐步编写代码, 以实现 [示例 22.7](#) 中的预期行为。先从 Record 类和一个读取 JSON 数据并返回字典 (字典中包含 Record 实例) 的函数开始。

22.3.1 第 1 步: 数据驱动的属性创建

[示例 22.8](#) 展示了指导第一步的 doctest。

```

</> 示例 22.8: 测试 schedule_v1.py (见 示例 22.9)
1 >>> records = load(JSON_PATH)      ❶
2 >>> speaker = records['speaker.3471'] ❷
3 >>> speaker                      ❸
4 <Record serial=3471>
5 >>> speaker.name, speaker.twitter ❹
6 ('Anna Martelli Ravenscroft', 'annaraven')

```

- ❶ 用 load 函数加载字典形式的 JSON 数据。
- ❷ records 中的“键 (Key)”是由记录类型与序号 (serial) 组成的字符串。
- ❸ speaker 是 Record 类 ([见示例 22.9](#)) 的实例。
- ❹ 原始 JSON 数据中的字段可通过 Record 实例属性进行检索。

`schedule_v1.py` 的代码如 [示例 22.9](#) 所示。

```

</> 示例 22.9: schedule_v1.py: 调整 OSCON 日程数据结构
1 import json
2
3 JSON_PATH = 'data/osconfeed.json'
4
5 class Record:
6     def __init__(self, **kwargs):
7         self.__dict__.update(kwargs) ❶
8
9     def __repr__(self):
10        return f'{self.__class__.__name__} serial={self.serial!r}' ❷
11
12 def load(path=JSON_PATH):
13     records = {}
14     with open(path) as fp:

```

```
15 raw_data = json.load(fp)          ❶
16 for collection, raw_records in raw_data['Schedule'].items():      ❷
17     record_type = collection[:-1] ❸
18     for raw_record in raw_records:
19         key = f'{record_type}.{raw_record["serial"]}'           ❹
20         records[key] = Record(**raw_record)                      ❺
21
return records
```

- ❶ 这是一种常用的编程捷径,根据关键字参数构建带属性的实例(详细解释见下文)。
- ❷ 使用 serial 字段构建自定义的 Record 表示形式,如 [示例 22.8](#) 的 ❸ 所示。
- ❸ load 函数最终将返回包含 Record 实例的字典。
- ❹ 解析 JSON,返回原生 Python 对象:如列表、字典、字符串、数字等。
- ❺ 遍历名为“conferences”、“events”、“speakers”、“venues”的 4 个顶层列表。
- ❻ record_type 列表名称去掉最后一个字符得到的结果,如 speakers 变成了 speaker。在 Python 3.9 中,可以用 collection.remove suffix('s') 更明确地执行此操作,详见“[PEP 616 –String methods to remove prefixes and suffixes](#)”。
- ❼ 创建“speaker.3471”格式的“键(Key)”。
- ❽ 创建一个 Record 实例,并将其与❹处的“键(Key)”一起保存在 records 字典中。

[示例 22.9](#) 中的 Record.`__init__` 方法展示了一种古老的 Python 编程技巧。如“[11.11 用 __slots__ 节省内存<311页>](#)”所述,Python 将每个实例的属性都存储在实例中名为 `__dict__` 的字典中,除非在类中声明了类属性 `__slots__`。因此,用映射更新实例的 `__dict__` 属性,可以快速为该实例创建一大批(Bunch)属性。⁵



如“[22.2.2 属性名无效的问题<668页>](#)”所述,根据应用程序的不同,Record 类可能需要处理不可作为属性名称的“键(Key)”。不过,若分心处理此问题,将偏离本示例的主题。而且对本示例目前读取的数据集来说,也不存在这样的问题。

[示例 22.9](#) 中定义的 Record 类非常简单。您或许会想,为何一开始不这么做,而是使用更为复杂的 FrozenJSON 类。原因有二:其一,FrozenJSON 类需要递归转换嵌套映射与列表,而 Record 类不需要这么做,因为转换后的数据集没有嵌套(nested)在映射(或列表)中的映射。记录中只包含字符串、数字、字符串列表、整数列表。其二,FrozenJSON 允许访问 `__data` 的字典属性(用以调用`keys()`等方法),而现在不再需要这个功能了。



Python 标准库提供了与 Record 类似的类,可通过传给特殊方法 `__init__` 的关键字参数,来构建实例的属性。这些类包括: `type.SimpleNamespace`、`argparse.Namespace` 和 `multiprocessing.managers.Namespace`。为此,我编写了一个更简单的 Record 类,以强调这些类的基本思想:用特殊方法 `__init__` 更新实例的 `__dict__` 属性。

重新调整日程数据集的结构之后,可以增强 Record 类,以自动检索“活动(event)”记录中引用的“会场(venue)”和“演讲者(speaker)”记录。在接下来的示例中,将使用特性(property)来实现这一想法。

⁵顺便说一下,Bunch(大批)是 Alex Martelli 在 2001 年分享技巧时使用的类名,该技巧名为“The simple but handy “collector of a bunch of named stuff” class”。

22.3.2 第 2 步: 用特性检索链接的记录

下一版代码的目标是: 给定一条“活动 (event)”记录, 读取其“会场 (venue)”特性 (property) 将得到一个 Record 对象。这与 Django ORM 访问 ForeignKey 字段的行为类似: 得到的不是“键 (Key)”, 而是链接的模型对象。

本小节将从“会场 (venue)”特性 (property) 开始。示例 22.10 中的交互, 演示了本小节中 `schedule_v2.py` 将要实现的预期行为。

</> 示例 22.10: 摘自 `schedule_v2.py` 中的 doctests

```

1  >>> event = Record.fetch('event.33950') ❶
2  >>> event
3  <Event 'There *Will* Be Bugs'>
4  >>> event.venue
5  <Record serial=1449>
6  >>> event.venue.name
7  'Portland 251'
8  >>> event.venue_serial
9  1449

```

❶ Record.fetch 静态方法从数据集中获取一个 Record 对象或一个 Event 对象。

❷ 注意, `event` 是 Event 类的实例。

❸ 访问 `event.venue`, 将得到一个 Record 实例。

❹ 现在, 可以轻松地获取 `event.venue` 的名称。

❺ Event 实例还有一个来自 JSON 数据的 `venue_serial` 属性。

Event 是 Record 的子类, 增加了 `venue` 特性 (property), 用于检索链接的记录; 还定制了特殊方法 `__repr__`。

本节的代码详见“随书代码库”中的 `22-dyn-attr-prop/oscon/schedule_v2.py` 模块。该模块包含近 60 行代码, 因此我将其拆分为多个部分, 分别展示。首先, 展示增强的 Record 类的源码, 如示例 22.11 所示。

</> 示例 22.11: `schedule_v2.py`: 增加 `fetch` 方法的 Record 类

```

1  import inspect ❶
2  import json
3
4  JSON_PATH = 'data/osconfeed.json'
5
6  class Record:
7
8      __index = None ❷
9
10     def __init__(self, **kwargs):
11         self.__dict__.update(kwargs)
12
13     def __repr__(self):
14         return f'{self.__class__.__name__} serial={self.serial!r}'
15

```

```

16     @staticmethod ❸
17     def fetch(key):      if Record.__index is None: ❹
18         Record.__index = load()
19     return Record.__index[key] ❺

```

- ❶ `inspect` 模块 将在 `load` 函数中使用,如 [示例 22.13](#) 所示。
- ❷ 私有类属性 `__index` 最终将保存对 `load()` 函数(见[示例 22.13](#))返回的字典的引用。
- ❸ `fetch` 是一个静态方法 (Static Method),明确表示其效果不受调用该方法的实例或类的影响。
- ❹ 如有必要,填充 `Record.__index`。
- ❺ 通过 `Record.__index` 获取指定 `key` 对应的记录。



本示例展示了 `@staticmethod` 装饰器的合理用途。静态方法 `fetch()` 始终只处理类属性 `Record.__index`, 即便在子类中调用亦是如此(如 `Event.fetch()`, 稍后说明)。若将 `fetch` 声明为类方法 (Class Method) 容易让人误解, 因为用不到第 1 个参数 `cls`。

接下来,在 `Event` 类中使用一个特性 (property),如 [示例 22.12](#) 所示。

```

</>示例 22.12: schedule_v2.py:Event 类

1  class Event(Record): ❶
2
3      def __repr__(self):
4          try:
5              return f'{self.__class__.__name__} {self.name!r}' ❷
6          except AttributeError:
7              return super().__repr__()
8
9      @property
10     def venue(self):
11         key = f'venue.{self.venue_serial}'
12         return self.__class__.fetch(key) ❸

```

- ❶ `Event` 类扩展(继承)了 `Record` 类。
- ❷ 如果实例有 `name` 属性, 则用该属性定制字符串表示形式。否则, 将此定制操作委托 `Record` 类的 `__repr__`。
- ❸ `venue` 特性 (property) 根据 `venue_serial` 属性构建一个“键 (Key)”, 将此“键 (Key) ”传给从 `Record` 类继承的类方法 `fetch()`(使用 `self.__class__` 的原因稍后说明)。

`venue` 方法的 `return` 语句中 ([示例 22.12 ❸处](#)) 调用了 “`self.__class__.fetch(key)`”。为何不直接调用 `self.fetch(key)` 呢? 因为, 后者只适用于特定的 OSCON 数据集——即任何“活动 (event)”记录中都不含名为“`fetch`”的“键 (Key)”。但如果一个“活动 (event)”记录中包含名为“`fetch`”的“键 (Key)”, 则当在那个 `Event` 实例中调用 `self.fetch` 时将获取该字段(即键为 `fetch`)的值, 而不是 `Event` 类从 `Record` 类中继承的 `fetch()` 类方法。这是个不易察觉的 BUG, 很容易逃过测试。因为它的行为取决于所用的数据集。



当从数据中创建实例属性名称时,很容易遮蔽类属性(如方法),从而导致BUG;或者意外覆盖现有的实例属性,导致数据丢失。或许正是因为这些问题的存在,Python字典与JavaScript对象有着本质的区别。

如果Record类的行为更像映射,实现了动态`__getitem__`方法,而不是动态`__getattr__`方法,则不会有因覆盖或遮蔽而导致BUG的风险。其实,将Record类实现为映射,或许更符合Python风格。然而,倘若真那么做,就无法研究动态属性编程的技巧与陷阱了。

本示例(`schedule_v2.py`)的最后一部分是重写的`load()`函数,如示例22.13所示。

```
</>示例22.13: schedule_v2.py:load函数
1 def load(path=JSON_PATH):
2     records = {}
3     with open(path) as fp:
4         raw_data = json.load(fp)
5         for collection, raw_records in raw_data['Schedule'].items():
6             record_type = collection[:-1] ①
7             cls_name = record_type.capitalize() ②
8             cls = globals().get(cls_name, Record) ③
9             if inspect.isclass(cls) and issubclass(cls, Record): ④
10                 factory = cls ⑤
11             else:
12                 factory = Record ⑥
13             for raw_record in raw_records: ⑦
14                 key = f'{record_type}.{raw_record["serial"]}' ⑧
15                 records[key] = factory(**raw_record)
16
17 return records
```

- ① 目前为止,与`schedule_v1.py`(见“示例22.9<672页>”)相比,`load`函数暂未做改动。
- ② 将`record_type`首字母大写,以尝试获取类名;例如,“event”变为“Event”。
- ③ 从模块全局作用域中获取相应名称的Python对象;若不存在相应的Python对象,则获取`Record`类。
- ④ 如果③处检索的Python对象是一个类,并且是`Record`的子类...
- ⑤ ...则将`factory`与之绑定。这意味着`factory`可以是`Record`的任何子类,具体取决于`record_type`。
- ⑥ 否则,将`factory`绑定到`Record`类。
- ⑦ 用于创建“键(Key)”与保存记录的`for`循环与之前的相同,只是...
- ⑧ ...`records`中存储的对象是由`factory`构造的。根据`record_type`的不同,存储的对象可以是`Record`类,也可以是其子类(如`Event`)。

注意,唯一有自定义类的`record_type`是`Event`。如果也定义了`Speaker`类或`Venue`类,那么`load()`函数将自动使用这些类构建和保存记录,不再使用默认的`Record`类。

接下来,用同样的方式为`Event`类增加`speakers`特性(property)。

22.3.3 第3步:用特性覆盖现有属性

示例 22.12 中的 venue 特性 (property) 名称与“活动 (events)”容器中的字段名不匹配,该名称来自于 venue_serial 字段名。相反,“活动 (events)”容器中的每条记录都有一个包含序号 (serial) 列表的 speakers 字段。我们希望将该信息作为 Event 实例的 speaker 特性 (property) 进行公开,而 Event 实例会返回 Record 实例的列表。正如 [示例 22.14](#) 所示,这种名称冲突需要特别注意。

</> [示例 22.14: schedule_v3.py](#):speakers 特性

```

1 @property
2 def speakers(self):
3     spkr_serials = self.__dict__['speakers'] ❶
4     fetch = self.__class__.fetch
5     return [fetch(f'speaker.{key}') ❷
6             for key in spkr_serials]
```

- ❶ 我们想要的数据位于 speakers 属性中,但必须直接从实例的 __dict__ 中获取,以避免递归调用 speakers 特性 (property)。
- ❷ 获取“键 (Key)”与 spkr_serials 中的数字匹配的所有记录;返回由这些记录构成的列表。

在 speakers 方法中,尝试读取 self.speakers 将调用 speakers 特性 (property) 本身,很快就会引发递归错误。但是,如果通过 self.__dict__['speakers'] 读取相同的数据,就会绕过 Python 检索属性的常规算法,不会调用特性 (property),也就避免了递归。因此,直接通过对象的 __dict__ 属性读写数据,是 Python 元编程常用的技巧。



Python 解释器在求解 obj.my_attr 时,首先会查看 obj 类。如果该类有一个名称为 my_attr 的特性 (property),则该特性 (property) 会遮盖同名的实例属性。[“22.5.1 特性 \(property\) 覆盖实例属性<683页>”](#) 中将举例说明这一点。[“二十三 属性描述符”](#) 将揭示属性是作为描述符实现的——一种更强大、更通用的抽象。

当编写 [示例 22.14](#) (❷处) 中的列表推导式时,直接告诉我:“此操作消耗的资源估计不少。”而事实并非如此,因为 OSCON 数据集的“活动 (events)”没有几个“演讲者 (speakers)”。所以,将问题复杂化属于过早优化。不过,缓存特性 (property) 是一种常见需求,而且还有一些注意事项。接下来的几个示例,将演示缓存特性 (property) 的具体做法。

22.3.4 第4步:定制的特性缓存

缓存特性 (property) 是一种常见需求,因为人们期望类似 event.venue 这样的表达式应消耗较少的资源⁶。如果 Event 特性 (property) 背后用到的 Record.fetch 方法需要查询数据库或 Web API,则特性 (property) 缓存就更是必不可少的。

在本书第 1 版中,我为 speakers 方法手工实现了缓存逻辑,如 [示例 22.15](#) 所示。

</> [示例 22.15: 用 hasattr 禁用“键共享 \(Key-Sharing\)”优化措施,手工实现缓存逻辑](#)

⁶实际上,这是本章开头提到的 Meyer 提出的“统一访问原则”的一个缺点。若对相关讨论感兴趣,请阅读本章后文的“杂谈”。

```

1 @property
2 def speakers(self):
3     if not hasattr(self, '__speaker_objs'): ❶
4         spkr_serials = self.__dict__['speakers']
5         fetch = self.__class__.fetch
6         self.__speaker_objs = [fetch(f'speaker.{key}') for key in spkr_serials]
7
8     return self.__speaker_objs ❷

```

- ❶ 若实例中没有名为 `__speaker_objs` 的属性，则获取 `speaker` 对象，并存入 `__speaker_objs` 属性中。
 ❷ 返回 `self.__speaker_objs`。

示例 22.15 中手工实现的缓存逻辑很简单，但在实例初始化后创建属性违背了“PEP 412 -Key-Sharing Dictionary”提出的优化措施，详见“3.9 dict 实现方式对实践的影响<84页>”。如果数据集规模较大，内存使用量的差异还是会很大的。

手工实现缓存方案时，若要兼顾“键共享 (Key-Sharing)”优化措施，则需在 `Event` 类中定义 `__init__` 方法，将必要的属性 `__speaker_objs` 初始化为 `None`；然后，在 `speakers` 方法中检查 `__speaker_objs` 属性值。

</> 示例 22.16：`__init__` 中定义的存储，以利用“键共享 (Key-Sharing)”优化措施

```

1 class Event(Record):
2     def __init__(self, **kwargs):
3         self.__speaker_objs = None
4         super().__init__(**kwargs)
5
6     # 省略 15 行
7
8     @property
9     def speakers(self):
10         if self.__speaker_objs is None:
11             spkr_serials = self.__dict__['speakers']
12             fetch = self.__class__.fetch
13             self.__speaker_objs = [fetch(f'speaker.{key}') for key in spkr_serials]
14
15     return self.__speaker_objs

```

示例 22.15 与示例 22.16 展示了在传统 Python 代码库中相当常见的缓存技术。然而，在多线程程序中，像这样手工实现的缓存容易引入争用，可能导致数据损坏。如果 2 个线程正在读取先前未缓存的属性，线程 1 需将计算得到的数据存入缓存属性（本例中的 `__speaker_objs`）中；而当线程 2 从缓存中读取数据时，线程 1 的计算过程可能还未结束。

幸运的是，Python 3.8 引入了线程安全的 `@functools.cached_property` 装饰器。不过，使用此装饰器有一些注意事项，下一节将详细说明。

22.3.5 第 5 步：用 `functools` 缓存特性

`functools` 模块提供了 3 个用于缓存的装饰器。“9.9.1 用 `functools.cache` 进行记忆化<260页>”中已介绍过 `@cache` 与 `@lru_cache`，还有一个是 Python 3.8 引入的 `@cached_property`。

`functools.cached_property` 装饰器将方法的结果缓存到与方法同名的实例属性中。例如,在 [示例 22.17](#) 中, `venue()` 方法将计算得到的值存储在 `self` 中名为 `venue` 的属性中。之后,当客户端代码尝试读取 `venue` 时,数据将来自新创建的实例属性 `venue`,而不再调用 `venue()` 方法。

</> [示例 22.17: `@cached_property` 的简单用法](#)

```

1 @cached_property
2 def venue(self):
3     key = f'venue.{self.venue_serial}'
4     return self.__class__.fetch(key)

```

如 “[22.3.3 第 3 步:用特性覆盖现有属性<677页>](#)” 所述,特性 (`property`) 会遮蔽同名的实例属性。既然如此, `functools.cached_property` 装饰器是如何工作的呢? 如果特性 (`property`) 覆盖了实例属性,那么实例属性 `venue` 将被忽略,并且总是调用 `venue()` 方法,每次都计算 `key` 并运行 `fetch!`

答案可能会令人意外: `cached_property` 名称不是很恰当。`@cached_property` 装饰器 并不会创建完整的特性 (`property`),而是创建了一个非覆盖型描述符 (Nonoverriding Descriptor)。描述符是一个对象,负责管理访问另一个类中的属性,详见 “[二十三 属性描述符](#)”。`property` 装饰器是一个高级 API,用于创建覆盖型描述符 (Overriding Descriptor)。“[二十三 属性描述符](#)”将详细介绍覆盖型与非覆盖型描述符。

现在,暂且搁置底层实现,从用户的角度出发,重点关注 `cached_property` 与 `property` 之间的区别。Raymond Hettinger 在 [Python 文档](#)中已经解释地很清楚了:

`cached_property()` 的机制与 `property()` 有些不同。常规特性 (`property`) 会阻止属性写入,除非定义了 [设值 \(setter\)](#) 方法。与之相反, `cached_property` 允许属性写入。

`cached_property` 装饰器只在查找时运行,而且只在同名属性不存在时运行。当运行时, `cached_property` 会写入同名属性。后续的属性读取和写入都优先于 `cached_property` 方法,其工作方式与普通属性相同。

可以通过删除属性来清除缓存的值。如此一来,便可再次运行 `cached_property` 方法。^a

^a源自 [@functools.cached_property 文档](#)。我之所以知道这些解释出自 Raymond Hettinger 之手,是因为这几段内容出现在他对 “[issue#42781 functools.cached_property docs should explain that it is non-overriding](#)” 的回复中。Hettinger 是 Python 官方文档与标准库的主要贡献者。他还编写了出色的 “[Descriptor Guide](#)”,这是 “[?? ??](#)” 中的主要参考资料。

回到 `Event` 类: `@cached_property` 的特定行为使得它不适合用来装饰 `speakers` 方法,因为该方法依赖于一个同样命名为 `speakers` 的现有属性。而该属性已存在,存储了活动演讲者的序号 (`serial`)。



`@cached_property` 有一些重要的限制:

- 如果被装饰的方法已经依赖于一个同名实例属性,则 `@cached_property` 不能直接替代 `@property`。
- 不能在定义了类属性 `__slots__` 的类中使用 `@cached_property`。
- `@cached_property` 违背对实例属性 `__dict__` 的“键共享 (Key-Sharing)”优化措施,因为`@cached_property`会在 `__init__` 初始化实例之后创建一个实例属性。

尽管存在局限,但 `@cached_property` 可以满足大多数常见需求,并且是线程安全的。在 `@cached_property` 的 Python 源码 (`Lib/functools.py`) 可以看到,该装饰器用到了“[重入锁 \(Reentrant Lock \)](#)”。

speakers 特性 (property) 可以使用 @cached_property 文档推荐的另一种方案:为 speakers() 方法堆叠装饰器 @property 和 @cache,如示例 22.18所示。

</> 示例 22.18: 为 speakers 方法堆叠装饰器 @property 与 @cache

```

1 @property ❶
2 @cache ❷
3 def speakers(self):
4     spkr_serials = self.__dict__['speakers']
5     fetch = self.__class__.fetch
6
7     return [fetch(f'speaker.{key}') for key in spkr_serials]

```

❶ 装饰器的堆叠顺序很重要:@property 在上 ...

❷ ...@cache 在下。

根据“9.9.1 用 functools.cache 进行记忆化<260页>”的提示栏“堆叠 (Stacked) 装饰器”对堆叠语法的说明,示例 22.18前 3 行代码等同于如下内容:

```
1 speakers = property(cache(speakers))
```

@cache 被应用于 speakers() 方法,返回一个新函数。然后,这个新函数再被 @property 装饰,被替换为一个新构建的特性 (property)。

借助 OSCON 数据集 (JSON 格式) 对只读特性 (property) 与缓存装饰器的讨论,到此结束。下一节,将通过一系列新示例,创建读写特性 (property)。

22.4 用特性验证属性

特性 (property) 除了可用于计算属性值之外,还可用于执行业务规则——即在不影响客户端代码的情况下,将公开属性更改为受 读值 (getter) 和 设值 (setter) 保护的属性。本节将通过一个详尽的示例进行说明。

22.4.1 LineItem 类第 1 版:表示订单中商品的类

假设有一个批量销售有机食品的电商应用程序,顾客可以按重量订购坚果、干果或谷物。在该系统中,每个订单将包含一系列商品,每个商品都可用??中 LineItem 类的实例表示。

</> 示例 22.19: bulkfood_v1.py:最简单的 LineItem 类

```

1 class LineItem:
2
3     def __init__(self, description, weight, price):
4         self.description = description
5         self.weight = weight
6         self.price = price
7
8     def subtotal(self):
9         return self.weight * self.price

```

此类很精简,或许太简单了。示例 22.20 揭示了 LineItem 类的一个问题。

```
</> 示例 22.20: 重量为负值时,金额小计也为负值
1 >>> raisins = LineItem('Golden raisins', 10, 6.95)
2 >>> raisins.subtotal()
3 69.5
4 >>> raisins.weight = -20 # garbage in...
5 >>> raisins.subtotal() # garbage out...
6 -139.0
```

此示例像玩具一样,但并没想象中那么好玩儿。如下是亚马逊早期的一个真实故事:

我们发现,顾客在订购图书时会将数量设为负数!然后,我们会在顾客的信用卡上记账负数的金额,并苦苦等待他们将书寄送过来(妄想)。

——Jeff Bezos, 亚马逊创始人与 CEO^a

^a摘自《华尔街日报》的文章“Birth of a Salesman”(October 15, 2011),这是 Jeff Bezos 的原话。注意,截至 2021 年为止,需要订阅才能阅读此文。

这个问题该如何解决呢?可以更改 LineItem 类的接口,用读值 (getter) 方法与设值 (setter) 方法管理 weight 属性。这是 Java 采用的方式,完全可行。

但是,如果能直接设定商品的 weight 属性,则会显得更自然。另外,此系统可能运行在生产环境中,而其他功能模块已直接访问 item.weight 了。在这种情况下,符合 Python 风格的做法是将数据属性替换为特性 (property)。

22.4.2 LineItem 类第 2 版:能验证的特性 (property)

实现特性 (property) 之后,将允许我们使用读值 (getter) 方法与设值 (setter) 方法,但 LineItem 的接口保持不变(即设置 LineItem 的 weight 仍可写为 raisins.weight = 12)。

读写 weight 特性 (property) 的代码,如示例 22.21 所示。

</> 示例 22.21: bulkfood_v2.py: 定义了 weight 特性 (property) 的 LineItem 类

```
1 class LineItem:
2
3     def __init__(self, description, weight, price):
4         self.description = description
5         self.weight = weight ❶
6         self.price = price
7
8     def subtotal(self):
9         return self.weight * self.price
10
11     @property ❷
12     def weight(self): ❸
13         return self.__weight ❹
```

```

15 @weight.setter          ❸
16 def weight(self, value): ❹ if value > 0:
17     self._weight = value ❺
18 else:
19     raise ValueError('value must be > 0') ❻

```

- ❶ 此处已经使用了特性 (property) 的设值 (setter) 方法, 确保不会创建 weight 为负值的实例。
- ❷ 用装饰器 @property 装饰 读值 (getter) 方法。
- ❸ 实现特性 (property) 的所有方法, 其方法名都应与公开属性的名称一样: 本例为 weight。
- ❹ 实际值存储在私有属性中, 本例为 _weight。
- ❺ 被装饰的 读值 (getter) 方法有一个 setter 属性, 此属性也是一个装饰器, 用于装饰设值 (setter) 方法。此装饰器可将 读值 (getter) 方法与 设值 (setter) 方法绑在一起。
- ❻ 如果 value > 0, 则设置私有属性 _weight 的值。
- ❼ 否则, 引发 ValueError 异常。

注意, 现在无法创建 weight 为无效值的 LineItem 对象了。

```

1 >>> walnuts = LineItem('walnuts', 0, 10.00)
2 Traceback (most recent call last):
3 ...
4 ValueError: value must be > 0

```

现在, 禁止用户为 weight 提供负值。尽管顾客通常无法设置商品的价格 (price), 但工作人员的失误或应用程序 BUG, 都可能导致 LineItem 对象的 price 属性为负值。为防止出现此种情况, 也可以将属性 price 转换为特性 (property)。但是, 这样会使代码出现一些重复。

如“[十七 迭代器、生成器和经典协程](#)”开篇引自 Paul Graham 的引文所述“当我在我的程序中发现了模式 (patterns) 时, 我认为可能是哪里出现了问题。”消除重复代码的良方是抽象。抽象特性 (property) 的定义 (definition) 有 2 种方式: 一是使用特性 (property) 工厂函数, 二是用描述符类。后者更灵活, 详见“[二十三 属性描述符](#)”。实际上, 特性 (property) 本身就是用描述符类实现的。但在本章, 为了继续探讨特性 (property), 我们将采用第一种方式——实现一个特性 (property) 工厂。

但在实现特性 (property) 工厂之前, 需要先对特性 (property) 有更深入的了解。

22.5 全面了解特性 (property)

虽然内置的 property 经常被用作装饰器, 但实际上它是一个类。在 Python 中, 函数与类通常可以互换使用。因为二者都是可调用的, 并且 Python 没有实例化对象的 new 运算符。所以, 调用构造函数与调用工厂函数没有区别。此外, 只要能返回新的可调用对象, 取代被装饰的函数, 函数与类都可以用作装饰器。

property 构造函数的完整签名, 如下所示:

```
1 property(fget=None, fset=None, fdel=None, doc=None)
```

所有参数都是可选参数, 如果没有为某个参数提供函数, 则生成的特性 (property) 对象不允许执行相应的操作。

特性 (property) 类型是在 Python 2.2 引入的, 但直至 Python 2.4 才出现 @ 装饰器语法。因此, 在过去几

年中,若要定义特性 (property),只能将存取函数传给 property() 构造函数的前 2 个参数。

不使用装饰器 (即不用 @ 符号) 定义特性的传统语法,如示例 22.22 所示。

</> 示例 22.22: bulkfood_v2b.py:效果同示例 22.21,只是未使用装饰器

```
1 class LineItem:
2
3     def __init__(self, description, weight, price):
4         self.description = description
5         self.weight = weight
6         self.price = price
7
8     def subtotal(self):
9         return self.weight * self.price
10
11    def get_weight(self):          ❶
12        return self.__weight
13
14    def set_weight(self, value):  ❷
15        if value > 0:
16            self.__weight = value
17        else:
18            raise ValueError('value must be > 0')
19
20    weight = property(get_weight, set_weight) ❸
```

❶ 普通的读值 (getter) 方法。

❷ 普通的设值 (setter) 方法。

❸ 为 property 构造函数传入存取函数,以便构建 property 对象,并将对象赋值给一个公开的类属性:本例为 wweight。

传统语法形式在某些场景下比装饰器语法更合适,稍后将讨论的特性 (property) 工厂的就是一个例子。但是,如果在具有许多方法的类主体中,装饰器语法可以更明确地看出哪些是读值 (getter) 方法与设值 (setter) 方法,而无需依赖方法名前缀——如 get 和 set 前缀的约定。

类中的特性 (property) 会影响实例属性的查找方式,这种查找方式起初可能会令人惊讶。下一节,将详细说明。

22.5.1 特性 (property) 覆盖实例属性

特性 (property) 始终是类属性,但实际上特性 (property) 管理的是实例中的属性存取。

如“11.12 覆盖类属性 (Class Attribute) <315页>”所述,当实例与它的类拥有同名的数据属性时,实例属性会覆盖 (或遮蔽) 类属性——至少通过实例读取数据属性时,是这样的。示例 22.23 展示了这一点。

</> 示例 22.23: 实例属性遮蔽了类中的数据属性

```
1 >>> class Class:          ❶
2 ...     data = 'the class data attr'
```

```

3 ...     @property
4 ...     def prop(self):...             return 'the prop value'
5 ...
6 >>> obj = Class()
7 >>> vars(obj)                  ❷
8 {}
9 >>> obj.data                  ❸
10 'the class data attr'
11 >>> obj.data = 'bar'          ❹
12 >>> vars(obj)                  ❺
13 {'data': 'bar'}
14 >>> obj.data                  ❻
15 'bar'
16 >>> Class.data                ❻
17 'the class data attr'

```

- ❶ 定义 Class 类, 该类包含 2 个类属性: data 属性与 prop 特性 (property)。
- ❷ vars 函数会返回 obj 的 __dict__ 属性。从中可知 obj 中没有实例属性。
- ❸ 读取 obj.data, 获取的是类属性 Class.data 的值。
- ❹ 而为 obj.data 赋值, 则会创建一个实例属性 data。
- ❺ vars 函数会返回 obj 的 __dict__ 属性。从中可知 obj 中多了一个实例属性 data。
- ❻ 现在, 读取 obj.data, 获取的是实例属性 data 的值。从 obj 实例读取属性时, 实例属性会覆盖 (遮蔽) 同名的类属性。
- ❼ 类属性 Class.data 的值完好无损。

下面尝试覆盖 obj 实例的 prop 特性 (property)。继续示例 22.23 的控制台会话, 如示例 22.24 所示。

```

</> 示例 22.24: 实例属性不会遮蔽类中的特性(接续示例 22.23)
1 >>> Class.prop          ❶
2 <property object at 0x1072b7408>
3 >>> obj.prop          ❷
4 'the prop value'
5 >>> obj.prop = 'foo'    ❸
6 Traceback (most recent call last):
7 ...
8 AttributeError: can't set attribute
9 >>> obj.__dict__['prop'] = 'foo'  ❹
10 >>> vars(obj)          ❺
11 {'data': 'bar', 'prop': 'foo'}
12 >>> obj.prop          ❻
13 'the prop value'
14 >>> Class.prop = 'baz'  ❼
15 >>> obj.prop          ❻
16 'foo'

```

- ❶ 直接读取 Class.prop 特性 (property), 获取的是特性 (property) 对象本身, 无需运行特性 (property) 的读值 (getter) 方法。

- ② 读取 obj.prop, 将执行特性 (property) 的读值 (getter) 方法: 本例为 prop()。
- ③ 尝试设置实例属性 prop 时, 失败了。
- ④ 但可以直接将 'prop' 存入 obj.__dict__ 中。
- ⑤ vars 函数会返回 obj 的 __dict__ 属性。从中可知 obj 现在有 2 个实例属性: data 与 prop。
- ⑥ 但是, 读取 obj.prop 仍然会运行特性 (property) 的读值 (getter) 方法。该特性 (property) 未被同名的实例属性遮盖。
- ⑦ 覆盖 Class.prop 特性 (property), 将销毁特性 (property) 对象。
- ⑧ 读取 obj.prop, 获取的是实例属性。因为 Class.prop 不再是特性 (property), 所以不会覆盖实例属性 obj.prop。

最后在举一个例子, 为 Class 类新增一个特性 (property), 覆盖实例属性。示例 22.25 接续示例 22.24

</> 示例 22.25: 新的类特性会遮盖现有的实例属性 (接续示例 22.24)

```

1  >>> obj.data          ❶
2  'bar'
3  >>> Class.data       ❷
4  'the class data attr'
5  >>> Class.data = property(lambda self: 'the "data" prop value') ❸
6  >>> obj.data          ❹
7  'the "data" prop value'
8  >>> del Class.data    ❺
9  >>> obj.data          ❻
10 'bar'

```

- ❶ obj.data 获取的是实例属性 data。
- ❷ Class.data 获取的是类属性 data。
- ❸ 用一个新特性 (property) 覆盖 Class.data。
- ❹ 现在, 实例属性 obj.data 被 Class.data 特性 (property) 遮盖了。
- ❺ 删除特性 (property) Class.data。
- ❻ 现在恢复原样, obj.data 又可获取实例属性 data 了。

本节的要点是: 像 obj.data 这样的表达式, 查找 data 的顺序是先从类 (即 obj.__class__) 中查找 data, 只有类中不存在名为 data 的特性 (Property) 时, Python 才会从实例 (即 obj) 中查找 data。这条规则适用于所有覆盖型描述符, 包括特性 (property)。关于描述符的进一步介绍, 详见“[二十三 属性描述符](#)”。

现在回到特性 (property)。每个 Python 代码单元 (如模块、函数、类、方法) 都可以有一个 docstring。下一个话题是如何为特性 (property) 添加文档 (docstring)。

22.5.2 特性的文档 (docstring)

当诸如控制台的 help() 函数或集成开发环境 (IDE) 需要显示特性 (property) 的文档 (docstring) 时, 它们会从特性 (property) 的 __doc__ 属性中提取信息。

如果使用传统的特性 (property) 语法, 则可通过 property 函数的 doc 参数为特性 (property) 对象指定文档 (docstring), 如下所示:

```
1 weight = property(get_weight, set_weight, doc='weight in kilograms')
```

读值 (getter) 方法 (即应用了@property装饰器的方法) 的 docstring 作为一个整体, 变成了特性 (property) 的文档。图 22.1 展示了示例 22.26 中的代码所生成的帮助界面。

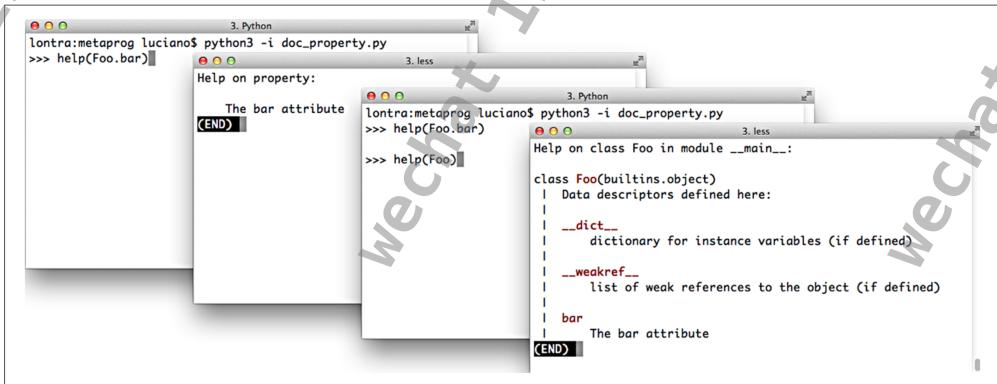


图 22.1: Python 控制台执行 help(Foo.bar) 与 help(Foo) 的截屏, 源码见示例 22.26

</> 示例 22.26: 一个特性 (property) 的文档 (docstring)

```
1 class Foo:
2
3     @property
4     def bar(self):
5         """The bar attribute"""
6         return self.__dict__['bar']
7
8     @bar.setter
9     def bar(self, value):
10        self.__dict__['bar'] = value
```

至此, 特性 (property) 相关的重要知识已介绍完毕。下面回过头解决前面遇到的问题: 保护 LineItem 对象的 weight 属性与 price 属性, 使其只接受大于 0 的值, 但无需手动实现两对几乎一致的设值 (setter) / 读值 (getter) 方法。

22.6 定义一个特性工厂函数

我们将定义一个名为 quantity 的特性 (property) 工厂函数 (Factory Function)。之所以用这个名称, 是因为在此应用中表示数量的托管 (Managed) 属性不能为负数或零。示例 22.27 是 LineItem 类的简洁版, 用到了 2 个 quantity 特性 (property) 实例: 一个用于管理 weight 属性, 另一个用于管理 price 属性。

</> 示例 22.27: bulkfood_v2prop.py: 使用特性 (property) 工厂函数

```
1 class LineItem:
2     weight = quantity('weight') ①
3     price = quantity('price')    ②
4
5     def __init__(self, description, weight, price):
```

```

6     self.description = description
7     self.weight = weight      ③
8     self.price = price
9
10    def subtotal(self):
11        return self.weight * self.price ④

```

- ① 用工厂函数 (Factory Function) 将第一个自定义特性 (property) weight 定义为类属性。
- ② 用工厂函数 (Factory Function) 定义另一个自定义特性 (property), 即 price。
- ③ 此处, 特性 (property) 已激活, 可确保 weight 不被设为负数与 0.
- ④ 此处也用到了特性 (property), 用于检索存储在实例中的值。

如前所述, 特性 (property) 是类属性。在构建各个 quantity 特性 (property) 对象时, 需要传入由特性 (property) 管理的 LineItem 实例属性名称。遗憾的是, 这一行必须输入 2 次 weight。

```
1 weight = quantity('weight')  #传入实例属性名, 将返回的特性对象绑定到类属性 weight
```

此处很难避免这种重复输入, 因为 特性 (property) 不知道它会被绑定到哪个类属性上。记住, 赋值语句的右侧表达式被先求解, 因此调用 quantity('weight') 时, 左侧的类属性 weight 还不存在。



改进 quantity 特性 (property) 以避免重复输入属性名, 是一个复杂 (nontrivial) 的元编程问题。将在“[二十三 属性描述符](#)”中解决此问题。

示例 22.28 给出了 quantity 特性 (property) 工厂函数的实现。⁷

</> 示例 22.28: `bulkfood_v2prop.py`: quantity 特性工厂函数

```

1 def quantity(storage_name):          ①
2
3     def qty_getter(instance):          ②
4         return instance.__dict__[storage_name] ③
5
6     def qty_setter(instance, value):      ④
7         if value > 0:
8             instance.__dict__[storage_name] = value ⑤
9         else:
10            raise ValueError('value must be > 0')
11
12    return property(qty_getter, qty_setter) ⑥

```

- ① storage_name 参数确定各个特性 (property) 的数据存储位置; 对于 weight 特性来说, 存储的名称为“weight”。
- ② qty_getter 函数的第一个参数可命名为 self, 但这样会很奇怪, 因为该函数没在类主体中。instance 指的是存储属性的 LineItem 实例。

⁷ 这段代码改编自《[Python Cookbook, 3rd Ed](#)》(David Beazley、Brian K. Jones 合著) 一书的“[Recipe 9.21. Avoiding Repetitive Property Methods](#)”。

- ③ qty_getter 引用了 storage_name, 将 storage_name 保存在 qty_getter 函数的闭包中; 直接从 instance.__dict__ 获取值, 以绕过特性 (property), 并避免无限递归。
- ④ 定义 qty_setter 函数, 第 1 个参数也是 instance。
- ⑤ 将 value 直接存入 instance.__dict__, 这么做也是为了绕过 特性 (property)。
- ⑥ 构建并返回一个自定义的特性 (property) 对象。

示例 22.28 中值得仔细分析的代码是与 storage_name 变量相关的部分。当用传统方式 (即应用 @property 装饰器) 定义 特性 (property) 时, 用于存储属性值的属性名称会硬编码在 读值 (getter) 方法与 设值 (setter) 方法中 (如 “示例 22.21<681页>” 所示)。但这里的 qty_getter 和 qty_setter 函数是通用的, 二者依靠 storage_name 变量来判断从 __dict__ 中获取哪个属性, 或设置哪个属性。每次调用 quantity 工厂函数构建 特性 (property) 时, 为参数 storage_name 传入的值必须是唯一的。

在工厂函数最后一行 (示例 22.28 的⑥处) 创建的 特性 (property) 对象, 会将函数 qty_getter 与 qty_setter 包装起来。稍后, 当调用 qty_getter 与 qty_setter 执行任务时, 二者将从各自的 闭包 (closures) 中读取 storage_name, 以确定从哪里检索/存储 托管 (Managed) 属性 的值。

示例 22.29 中, 创建并审查了一个 LineItem 实例, 并公开了属性值的保存位置。

</> 示例 22.29: bulkfood_v2prop.py: 探索特性 (property) 与真用用于保存值的实例属性

```

1  >>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
2  >>> nutmeg.weight, nutmeg.price ❶
3  (8, 13.95)
4  >>> nutmeg.__dict__ ❷
5  {'description': 'Moluccan nutmeg', 'weight': 8, 'price': 13.95}

```

❶ 通过 特性 (property) 读取 weight 与 price, 这会遮盖同名的实例属性。

❷ 通过 nutmeg.__dict__ 审查 nutmeg 实例, 查看真正用于保存值的实例属性。

注意, 工厂函数构建的 特性 (property) 利用了 “22.5.1 特性 (property) 覆盖实例属性<683页>” 所述的行为: weight 特性 (property) 覆盖了实例中同名的 weight 属性, 因此对 self.weight 或 nutmeg.weight 的每次引用都由 特性 (property) 函数处理, 绕过 特性 (property) 逻辑的唯一方法是直接访问实例的 __dict__ 属性。

示例 22.28 的代码可能有些晦涩难懂, 但它很简洁: 其行数与 “示例 22.21<681页>” 中用 @property 装饰器声明 读值 (getter) / 设值 (setter) 方法的行数相同, 但那里仅定义了 weight 特性 (property)。示例 22.27 中定义的 LineItem 类, 因为没有 读值 (getter) / 设值 (setter) 方法的干扰, 看起来清爽多了。

在真实的系统中, 类似的验证可能会出现在跨越多个类的许多字段中, 因此最好将 quantity 特性 (property) 工厂函数放置在单独的实用模块中, 以便重复使用。最终, 这个简单的 特性 (property) 工厂函数将被重构为一个更具可扩展性的 描述符 (Descriptor) 类, 使用专门的子类执行不同类型的验证逻辑。详见 “二十二 属性描述符”。

接下来, 分析属性删除问题, 以此结束对 特性 (property) 的探讨。

22.7 处理属性删除操作

使用 del 语句不仅可以删除变量, 还可以删除属性:

```
1  >>> class Demo:
2      ...
3      ...
4      ...
5      >>> d = Demo()
6      >>> d.color = 'green'
7      >>> d.color
8      'green'
9      >>> del d.color
10     >>> d.color
11     Traceback (most recent call last):
12     File "<stdin>", line 1, in <module>
13     AttributeError: 'Demo' object has no attribute 'color'
```

实际上,在 Python 中很少需要删除属性,而用特性 (property) 来处理属性的删除操作就更不常见了。但是,Python 是支持这种操作的,为此我可以虚构一个示例,来演示用特性 (property) 删除属性的方式。

在特性 (property) 定义中,可以用 @my_property.deleter 装饰器包装一个方法,由该方法负责删除特性 (property) 管理的属性。接下来,兑现承诺,虚构一个示例。此示例灵感来自电影《Monty Python and the Holy Grail》⁸中的黑衣骑士角色,如示例 22.30 所示。

```
</> 示例 22.30: blackknight.py
1  class BlackKnight:
2
3      def __init__(self):
4          self.phrases = [
5              ('an arm', "'Tis but a scratch."),
6              ('another arm', "It's just a flesh wound."),
7              ('a leg', "I'm invincible!"),
8              ('another leg', "All right, we'll call it a draw.")
9          ]
10
11     @property
12     def member(self):
13         print('next member is:')
14         return self.phrases[0][0]
15
16     @member.deleter
17     def member(self):
18         member, text = self.phrases.pop(0)
19         print(f'BLACK KNIGHT (loses {member}) -- {text}'')
```

blackknight.py 脚本的 doctest 如示例 22.31 所示。

```
</> 示例 22.31: blackknight.py:示例 22.30 的 doctest (黑衣骑士绝不屈服)
```

```
1  >>> knight = BlackKnight()
2  >>> knight.member
```

⁸中文名:《巨蟒与圣杯》

```

3 next member is:
4 'an arm'
5 >>> del knight.member
6 BLACK KNIGHT (loses an arm) -- 'Tis but a scratch.
7 >>> del knight.member
8 BLACK KNIGHT (loses another arm) -- It's just a flesh wound.
9 >>> del knight.member
10 BLACK KNIGHT (loses a leg) -- I'm invincible!
11 >>> del knight.member
12 BLACK KNIGHT (loses another leg) -- All right, we'll call it a draw.

```

在不使用装饰器的传统调用语法中, `fdel` 参数用于设置 `删值 (deleter)` 函数。例如, `BlackKnight` 类主体中可以像下面这样创建 `member` 特性 (property)。

```
1 member = property(member_getter, fdel=member_deleter)
```

如果不使用特性 (property), 也可以通过实现较底层的 `__delattr__` 特殊方法来处理属性删除, 详见“22.8.3 用于属性处理的特殊方法<691页>”。以下练习留给拖拉的读者去完成: 虚构一个类, 并实现特殊方法 `__delattr__`。

特性 (property) 是一个强大的功能, 但有时更简单或更底层的替代方案可能更受欢迎。在本章的最后一节, 将介绍 Python 为动态属性编程提供的一些核心 API。

22.8 处理属性的重要属性与函数

在本章以及本章之前的章节中, 多次用到 Python 为处理动态属性提供的内置函数与特殊方法。由于这些函数与方法的文档散落在 Python 官方各处, 所以这一节将它们汇总到一起进行集中介绍。

22.8.1 影响属性处理的特殊属性

“22.8.2 用于属性处理的内置函数”与“22.8.3 用于属性处理的特殊方法”中所列的许多函数与特殊方法的行为都受如下 3 个特殊属性影响:

- 实例属性 `__class__`

对象所属类的引用 (如 `obj.__class__` 与 `type(obj)` 的作用相同)。Python 只在对象的类中查找特殊方法 (如 `__getattr__`), 而不在实例中查找。

- 实例属性 `__dict__`

存储对象或类的可写属性的映射。具有 `__dict__` 的对象, 可以随时随意设置新的实例属性。对于具有类属性 `__slots__` 的类, 其实例可能没有 `__dict__` 属性。详见后文 `__slots__` 的说明。

- 类属性 `__slots__`

为了节省内存, 可以在类中定义类属性 `__slots__`。`__slots__` 属性值是一个字符串元组, 列出了类的实例中允许存在的属性。⁹如果 `__slots__` 属性值中不含 “`__dict__`”, 则该类的实例就不会有自己的 `__dict__` 属性, 只有 `__slots__` 中列出的属性才允许在这些实例中使用。详见“11.11 用 `__slots__` 节省

⁹Alex Martelli 指出, 虽然 `__slots__` 的值可以是一个列表, 但最好使用元组。因为在类主体处理完毕后, 再更改 `__slots__` 中的列表, 不会产生任何影响, 所以此处使用一个可变序列会产生误导。

内存<311页>”。

22.8.2 用于属性处理的内置函数

如下 5 个内置函数可执行对象的读取、写入与 [内省 \(Introspection\)](#) 操作：

- **dir([object])**

可列出对象的大多数属性。[官方文档](#)指出, `dir()` 主要为了方便在交互式提示符中使用, 因此不会列出完整的属性名称, 而是只列出一些重要的属性名称。`dir()` 可以审查含有或不含 `__dict__` 属性的对象。`dir` 函数不会列出 `__dict__` 属性本身, 但会列出 `__dict__` 中的“键”。几个特殊的类属性(如 `mro`、`bases` 和 `name`)也不会被 `dir()` 列出。可通过实现特殊方法 `__dir__` 来定制 `dir()` 的输出, 如[“示例 22.4<667页>”](#)所示。如果不提供可选的 `object` 参数, 则 `dir()` 列出当前作用域中的属性名称。

- **getattr(object, name[, default])**

从对象 `object` 中获取由 `name` 指定的属性。主要用于检索事先不知道名称的属性(或方法)。获取的属性可能来自对象所属的类或超类。如果 `name` 指定的属性不存在, `getattr()` 将引发 `AttributeError` 异常或返回 `default` 参数指定的值(如果提供了)。在 Python 标准库中, `cmd` 包的 `Cmd.onecmd` 方法就很好地利用了 `getattr()`, 用于获取并执行用户定义的命令。

- **hasattr(object, name)**

如果对象 `object` 中存在由 `name` 指定的属性, 或者能以某种方式(如继承)通过对象 `object` 获取由 `name` 指定的属性, 则返回 `True`。[文档](#)解释说:“这是通过调用 `getattr(object, name)` 并查看是否引发 `AttributeError` 来实现的。”

- **setattr(object, name, value)**

如果对象 `object` 允许, 为对象 `object` 的属性 `name` 赋值。这可能会创建一个新属性或覆盖现有属性。

- **vars([object])**

返回对象的 `__dict__` 属性。如果实例所述的类定义了类属性 `__slots__`, 且实例中没有 `__dict__` 属性, 这样的实例无法被 `vars()` 处理。如果不指定参数, 函数 `vars()` 的作用与 `locals()` 相同: 返回一个表示局部作用域的字典。

22.8.3 用于属性处理的特殊方法

在用户定义的类中, 如下特殊方法可用于获取、设置、删除、列出属性。

使用点号表示法(如 `obj.attr`)或内置函数 `getattr`、`hasattr` 和 `setattr` 进行属性读写, 都会触发本节列出的相应[特殊方法](#)。但是, 直接通过实例的 `__dict__` 属性来读写属性, 不会触发这些特殊方法。必要时, 这是绕过特殊方法的常用方式。

《[The Python Language Reference](#)》的“3.3.11. Special method lookup”警告说:

对于用户自定义的类, 如果特殊方法被正确定义在类中, 则特殊方法的隐式调用可被正确执行。但如果特殊方法被定义在实例的字典(如 `__dict__` 属性)中, 则特殊方法的隐式调用无法保证正确性。

换句话说, 即便操作目标是一个实例, 也会在实例所属的类上检索特殊方法。这意味着, 类中定义的特殊方法不会被同名的实例属性所遮蔽。

在如下示例中,假定有一个名为 Class 的类,obj 是 Class 类的实例,attr 是实例 obj 的属性。

无论用点号表示法(如 obj.attr),还是用“22.8.2 用于属性处理的内置函数”所列的内置函数,都会触发以下特殊方法中的某一个。例如,obj.attr 与 getattr(obj,'attr',42) 都会触发特殊方法 Class.__getattribute__(obj,'attr')。

- `__delattr__(self, name)`

当试图用 del 语句删除属性时,就会调用此方法。例如,del obj.attr 语句会触发 Class.__delattr__(obj,'attr')。若 attr 是一个 **特性 (property)**,而且 Class 类实现了 `__delattr__` 方法,则永远不会调用 **特性 (property)** 的 **删值 (deleter)** 方法。

- `__dir__(self)`

当在对象上调用 dir 函数时会调用此方法,以列出对象中的属性;例如,dir(obj) 会触发 Class.__dir__(obj)。所有现代 Python 控制台在 tab 自动补全时,也会调用此方法。

- `__getattr__(self, name)`

仅当尝试获取指定的属性失败时,才调用此方法。即在搜索了 obj、Class 及其超类之后,才调用此方法。表达式 obj.no_such_attr、getattr(obj, 'no_such_attr') 与 hasattr(obj, 'no_such_attr') 可能会触发 Class.getattr(obj, 'no_such_attr'),但前提是在 obj 或其所属的 Class 及其超类中都找不到名为 no_such_attr 的属性。

- `__getattribute__(self, name)`

当 Python 代码直接尝试获取 name 指定的属性时,始终调用此方法。在某些情况下(如获取 `__repr__` 方法),Python 解释器可能会绕过此方法。点号表示法、getattr 和 hasattr 内置函数都会触发此方法。`__getattribute__` 仅在 `__getattribute__` 之后被调用,并且仅当 `__getattribute__` 引发 AttributeError 时,`__getattr__` 才会被调用。为了在获取实例 obj 的属性时不触发无限递归,`__getattribute__` 方法的实现应使用 super().__getattribute__(obj, name)。

- `__setattr__(self, name, value)`

当尝试设置指定名称的属性时,始终会调用此方法。点号表示法和内置函数 setattr 都会触发此方法;例如,obj.attr = 42 和 setattr(obj, 'attr', 42) 都会触发 Class.__setattr__(obj, 'attr', 42)。



在实践中,特殊方法 `__getattribute__` 和 `__setattr__` 总是被无条件地调用,几乎影响每一次属性的存取。因此,比 `__getattr__` 更难使用。`__getattr__` 仅用于处理不存在的属性名称。与本节的这些特殊方法相比,使用 **特性 (property)** 或描述符相对不易出错。

至此,对 **特性 (property)**、特殊方法与其他动态属性编程技术的讨论就告一段落了。

22.9 本章小结

在开始讨论动态属性时,展示了一些定义简单类的示例,以便能更容易处理 JSON 数据集。第 1 个示例是 FrozenJSON 类,它将嵌套的字典与列表转换为嵌套的 FrozenJSON 实例以及由 FrozenJSON 实例组成的列表。FrozenJSON 示例代码展示了在读取属性时,如何使用特殊方法 `__getattribute__` 即时转换数据结构。最后一版 FrozenJSON 示例代码展示了使用 `__new__` 构造方法将一个类转换为灵活的对象 **工厂函数 (Factory Function)**,不受实例自身的限制。

随后,将 JSON 数据集转换为一个字典,其中存储了 Record 类的实例。第 1 版的 Record 类仅有几行代码,引入了“Bunch(大批)”惯用法:使用传递给 `__init__` 的关键字参数,调用 `self.dict.update(**kwargs)` 构建任意属性。第 2 版代码中增加了 Event 类,通过特性 (property) 实现了自动获取链接的记录。有时,计算得到的特性 (property) 值需要缓存,所以随后介绍了几种缓存实现方式。

在意识到 `@functools.cached_property` 并非适用所有场景之后,介绍了另一种替代方案:按顺序叠放 `@property` 与 `@functools.cache`,前者在上,后者在下。

接下来,继续介绍特性 (property)。在 `LineItem` 类中定义了一个特性 (property),其目的是为了保护 `weight` 属性,防止属性值出现无业务含义的负值或零。在深入研究特性 (property) 语法和语义之后,创建了一个特性 (property) 工厂函数——`quantity()`,以便在不编写多个读值 (getter)/设值 (setter) 方法的前提下,为属性 `weight` 和 `price` 强制执行同样的验证逻辑。特性 (property) 工厂函数利用几个精妙的概念(如闭包、特性覆盖实例属性),以更少的代码为定义特性 (property) 提供了更优雅的通用解决方案。

最后,简单介绍了如何用特性 (property) 处理属性删除,然后概述了 Python 核心语言为支持属性元编程提供的特殊属性、内置函数和特殊方法。

22.10 延伸阅读

官方文档中关于属性处理和内省 (Introspection) 的内置函数位于《The Python Standard Library》的“Built-in Functions”中。相关的特殊方法和类属性 `__slots__` 在《The Python Language Reference》的“3.3.2. Customizing attribute access”中。绕过实例调用特殊方法的语义解释位于“3.3.12. Special method lookup”。在《The Python Standard Library》的“Special Attributes”中包括了 `__class__` 与 `__dict__` 属性。

《Python Cookbook, 3rd Ed》(David Beazley 与 Brian K. Jones 著)中有多个涉及本章主题的示例,但我要重点介绍其中最出色的 3 个:“8.8 扩展子类中的特性”解决了在从超类继承的特性 (property) 中覆盖方法这个棘手问题;“8.15 委托属性访问”实现了一个代理类,展示了本书“22.8.3 用于属性处理的特殊方法<691页>”中所列的大多数特殊方法;还有出色的“9.21 避免重复的特性方法”是本章“示例 22.28<687页>”中特性 (property) 工厂函数的基础。

《Python in a Nutshell, 3rd Ed》(Alex Martelli、Anna Ravenscroft、Steve Holden 著)一书严谨而客观。该书只用了 3 页来介绍特性 (property),但这是因为该书采用了逻辑化的行文方式:之前的 15 页已经对 Python 中的类做了详尽说明,包括描述符,而特性 (property) 背后就是用描述符实现的。因此,当作者讲到特性 (property) 时,用较少的篇幅发表了很多见解,例如本章开篇的那段引文。

本章开头引用的“统一访问原则”出自 Bertrand Meyer,他开创了契约设计方法论,设计了 Eiffel 语言,并撰写了优秀著作《Object-Oriented Software Construction, 2nd Ed》。该书的前 6 章是我所见过的面向对象分析与设计最好的概念性介绍材料之一。第 11 章介绍了契约式设计,第 35 章提供了 Meyer 对一些有影响力的面向对象语言的评估,包括 Simula、Smalltalk、CLOS (Common Lisp Object System)、Objective-C、C++ 和 Java,并对其他一些语言进行了简短评论。直至该书的最后一页他才透露,他使用的机具可读性的“表示法”看似伪代码,实际出自 Eiffel 语言。

杂谈

从美学角度看, Meyer 提出的“统一访问原则”很有吸引力。作为使用 API 的程序员,我不应关心 `product.price` 只是获取数据属性还是执行计算。但作为消费者,我更应该关心:在当今的电子商务中,

product.price 的值往往取决于查询者是谁, 因此它肯定不仅仅是一个单纯的数据属性。事实上, 通常的做法是, 如果查询来自网店外部(如比价引擎), 那么 price 的值会低一些。显然, 这对只喜欢浏览特定网店的忠实消费者来说, 利益受到了损害。但我离题了。

上段内容虽然离题了, 但却提出了一个与编程相关的问题: 尽管统一访问原则在理想世界中非常有意义, 但在现实中, API 用户可能需要知道读取 product.price 是否过于耗费资源或时间。一般来说, 这是编程抽象的问题: 抽象之后很难推算求解表达式的运行时成本。但另一方面, 抽象又能让用户用更少的代码完成工作的工作。这是一种权衡。Ward Cunningham 在其[原创维基网站](#)上, 对软件工程方面的“统一访问原则”优缺点做了富有洞察力的论述。

在面向对象编程语言中, 是否违反“统一访问原则”通常体现在语法上: 是直接读写公开的数据属性, 还是调用[读值\(getter\)](#)/[设值\(setter\)](#)方法。

Smalltalk 和 Ruby 以一种简单而优雅的方式解决了这个问题: 它们根本不支持公共数据属性。在这些语言中, 每个实例属性都是私有的, 因此对它们的每次访问都必须通过方法进行。但它们的语法让这一点变得简单易行: 在 Ruby 中, product.price 调用了 price 的[读值\(getter\)](#)方法; 而在 Smalltalk 中, 只需使用 product.price。

Java 采用的是另一种方式, 即让程序员在 4 种访问级别修饰符中选择, 包括没有名称的默认级别——Java 教程称之为“package-private”。

不过, 大部分用户并不认同 Java 设计者制定的这种语法。Java 世界的程序员都认为, 属性应该是私有的, 而且每次都必须明确指定 private, 因为默认访问级别不是 private。如果所有属性都是私有的, 那么从类外部访问属性就必须使用访问器(accessor)。一些 Java IDE 提供了自动生成访问器方法的快捷方式。不幸的是, 当 6 个月后不得不阅读代码时, IDE 的帮助就显得不太够用了。您需要费力地在一堆无实际作用的访问器方法中, 找出实现了某些业务逻辑的那一个。

Alex Martelli 将访问器方法称为“愚蠢的惯用法”, 这道出了 Python 社区中多数人的心声。他列举了如下 2 个示例, 二者虽然外观差异较大, 但作用相同。^a

```
1 someInstance.widgetCounter += 1
2 # rather than...
3 someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

在设计 API 时, 我有时会想: 是否应该将不带参数(self 除外)、有返回值(None 除外), 并且是纯函数(即没有副作用)的方法都替换为只读特性(property)? 在本章中, LineItemsubtotal 方法(见“[示例 22.27<686页>](#)”)就可以替换为一个只读特性(property)。当然, 用于修改对象的方法(如 my_list.clear())不在此列。将这样的方法也转换为特性(property), 是一个糟糕的想法, 因为只要访问 my_list.clear 就会删除列表中的内容。

在我参与创作的 Pingo GPIO 库(见“[3.5.2 __missing__ 方法<76页>](#)”)中, 大部分用户级 API 都是基于特性(property)实现的。例如, 要读取模拟引脚的当前值, 用户需要写入 pin.value, 而设置数字引脚模式则需要写入 pin.mode = OUT。在幕后, 读取模拟引脚值或设置数字引脚模式可能涉及大量代码, 具体取决于特定的开发板驱动。之所以决定在 Pingo 中使用特性(property), 是因为我们希望这些 API 即使在交互式环境(如 Jupyter Notebook 等)中也能被舒服地使用, 而且我们认为 pin.mode = OUT 比 pin.set_mode(OUT) 更易于观察和操作。

虽然我觉得 Smalltalk 和 Ruby 的处理方式更简洁, 但我认为 Python 比 Java 的处理方式更有意

义。一开始,从简单的方式入手,将数据成员编码为公开属性,因为我们知道这些属性始终可以用[特性 \(property\)](#)(或[“二十三 属性描述符”](#)中的描述符)包装。

`__new__` 方法比 `new` 运算符更好

在 Python 中,还有一处体现了[“统一访问原则”](#)(或其变体):函数调用与对象实例化使用相同的语法。如 `my_obj=foo()`,其中 `foo` 可以是类或其他可调用对象。

其他受 C++ 语法影响的语言提供了 `new` 运算符,致使实例化看起来与调用不同。多数时候,API 用戶不关心 `foot` 是函数,还是类。数年来,我一直以为 `property` 是个函数,实际上它是个类。不过正常使用时,并没什么区别。

用[工厂函数 \(Factory Function\)](#)取代构造方法有许多充足的理由。^b一个重要的原因是:通过返回先前构建的实例来限制实例的数量(体现了单例模式)。一个相关的用途是:缓存构建过程开销较大的对象。此外,有时便于根据提供的参数,返回不同类型的对象。

编写构造方法更简单;提供工厂函数虽然增加了灵活性,但需要以编写更多的代码为代价。在具有 `new` 运算符的编程语言中,API 的设计者必须提前决定,究竟是坚持使用简单的构造函数,还是使用[工厂函数 \(Factory Function\)](#)。若最初的选择是错误的,那么纠正错误的代价可能会很高——这一切都因为 `new` 是运算符。

有时,采用其他方式可能也很方便,用类取代简单的函数。

在 Python 中,类与函数在许多情况下可以互换使用。这不仅是因为 Python 中没有 `new` 运算符,还因为 Python 中有特殊方法 `__new__`。`__new__` 可以将一个类变成工厂函数,生成不同类型的对象(参见[“22.2.3 用 `__new__` 灵活创建对象”](#));也可以返回事先构建好的实例,而不是每次都创建一个新实例。

如果[“PEP 8 –Style Guide for Python Code”](#)^a没推荐类的命名应采用“驼峰式”,则函数与类的[二元性 \(Duality\)](#)会更易于使用。不过,标准库中也有许多类名都是小写的(如 `property`、`str`、`defaultdict` 等)。因此,使用小写的类名可能是一种特色,而不是 BUG。但无论如何,Python 标准库在类名大小写上的不一致,都会导致可用性问题。

虽然调用函数与调用类没什么不同,但如果能明确区分,还是很有好处的。因为类还有其他功能:子类化。因此,我在编写类时都会采用驼峰式命名,而且希望 Python 标准库中的所有类都使用同样的约定。`collections.OrderedDict`与`collections.defaultdict`,我在盯着你你们呢!

^a参见《Python in a Nutshell, 2nd Ed》(Alex Martelli 著)第 101 页。

^b

wechat: 119554488

属性描述符

学习描述符不仅可以访问更大的工具集,还能加深对 Python 工作原理的理解,并欣赏其设计的优雅之处。

——Raymond Hettinger, Python 核心大师级开发者^a

^aRaymond Hettinger: [Descriptor HowTo Guide](#)。

描述符 (Descriptor) 是在多个属性中重复使用相同访问逻辑的一种方式。例如,ORM (如 Django ORM 和 SQLAlchemy) 中的字段类型就是描述符,它管理着从数据库记录中的字段到 Python 对象属性的数据流,反之亦然。

描述符是一个实现动态协议的类,其中包括 `__get__`、`__set__` 和 `__delete__` 方法。特性 (Property) 类实现了完整的描述符协议。与动态协议一样,只实现部分特殊方法也是可以的。事实上,真实代码中的大多数描述符都只实现了 `__get__` 和 `__set__` 方法,而且许多描述符只实现了其中的一个方法。

描述符是 Python 的一个显著特性,它不仅可部署在应用程序级别,也可部署在语言基础结构中。用户定义的函数就是描述符。我们将看到描述符协议是如何允许将方法作为绑定或非绑定方法运行的,这取决于它们是如何被调用的。

理解描述符是掌握 Python 的关键。这就是本章的主题。

在本章中,将重构 “[22.4 用特性验证属性](#)” 的 `bulkfood` 示例,使用描述符替换特性 (property)。这样就可以更容易地在不同的类中重复使用属性验证逻辑。本章将介绍覆盖 (overriding) 描述符与非覆盖 (nonoverriding) 描述符的概念,并意识到 Python 函数就是描述符。最后,将介绍一些实现描述符的技巧。

23.1 本章新增内容

由于 Python 3.6 在描述符协议中新增了 `__set_name__` 特殊方法,“[23.2.2 Lineltem 类第 4 版: 存储 \(Storage\) 属性的自动命名](#)” 中的 `Quantity` 描述符示例得到了极大的简化。

我删除了以前在 “[23.2.2 Lineltem 类第 4 版: 存储 \(Storage\) 属性的自动命名](#)” 中的特性 (property) 工厂示例,因为它存在的目的是展示 `Quantity` 问题的另一种解决方法,但有了 `__set_name__` 之后,描

述符解决方案变得简单多了。

“23.2.3 Lineltem 类第 5 版:一种新型描述符<704页>”曾出现过的 AutoStorage 类也被移除,因为新增了 `_set_name_` 之后,不再需要 AutoStorage 类。

23.2 描述符示例:属性验证

如“22.6 定义一个特性工厂函数<686页>”所述,特性 (property) 工厂函数 (Factory Function) 可借助函数式编程模式,避免重复编写读值 (getter) 方法与设值 (setter) 方法。特性 (property) 工厂函数 (Factory Function) 是高阶函数,在闭包 (closures) 中保存 `storage_name` 等设置,由参数决定创建哪些访问器函数,再使用访问器函数构建一个定制的特性 (property) 实例。解决这种问题的面向对象方式是使用描述符类。

本章将接续“22.6 定义一个特性工厂函数<686页>”中的 Lineltem 系列示例,将 `quantity` 特性 (property) 工厂函数 (Factory Function) 重构为 `Quantity` 描述符类。这将使其更易于使用。

23.2.1 Lineltem 类第 3 版:一个简单的描述符

正如本章开篇所述,实现了特殊方法 `_get_`、`_set_` 或 `_delete_` 的类就是描述符。描述符的用法是将描述符实例声明为另一个类的类属性。

本示例将创建 1 个 `Quantity` 描述符类, `Lineltem` 类将使用 2 个 `Quantity` 实例:一个用于管理 `weight` 属性,另一个用于管理 `price` 属性。如图 23.1 所示。

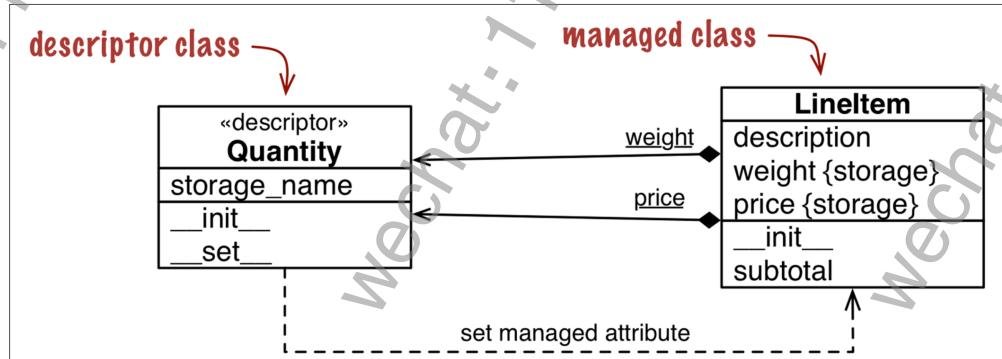


图 23.1: Lineltem 使用 `Quantity` 描述符类的 UML 类图¹

请注意,在图 23.1 中 `weight` 这个词出现了 2 次,因为实际上有 2 个不同的名为 `weight` 的属性:一个是 `Lineltem` 的类属性,另一个是各个 `Lineltem` 对象的实例属性。属性 `price` 也是如此。

23.2.1.1 理解描述符的术语

实现和使用描述符涉及多个组件,因此在命名这些组件时要准确。在描述本章的示例时,将使用以下术语和定义。虽然在看到代码之后,会更容易理解这些术语和定义,但我还是想把它们提到代码之前,以便你在需要时可以参考。

- 描述符 (Descriptor) 类

¹UML 图中加下划线的属性是类属性。请注意, `weight` 和 `price` 是附加到 `Lineltem` 类中的 `Quantity` 实例,但 `Lineltem` 实例也有自己的 `weight` 和 `price` 属性,并保存了属性值。

实现了描述符协议的类。如图 23.1 中的 `Quantity` 类。

- **托管 (Managed) 类**

将描述符 (Descriptor) 实例声明为类属性的类。如图 23.1 中的 `LineItem` 类。

- **描述符 (Descriptor) 实例**

指描述符 (Descriptor) 类的实例。每个描述符 (Descriptor) 实例都被声明为托管 (Managed) 类的类属性。在图 23.1 中, 每个描述符 (Descriptor) 实例都用带下划线名称与箭头组合表示 (下划线在 UML 中表示类属性)。黑色菱形 (◆) 指向包含描述符 (Descriptor) 实例的 `LineItem` 类。

- **托管 (Managed) 实例**

指托管 (Managed) 类的实例。在本例中, `LineItem` 实例就是托管实例 (图 23.1 中未显示)。

- **存储 (Storage) 属性**

指托管 (Managed) 实例的属性, 用于保存特定实例的托管 (Managed) 属性值。在图 23.1 中, `LineItem` 实例属性 `weight` 和 `price` 都是存储 (Storage) 属性。它们与描述符 (Descriptor) 实例不同, 后者始终是类属性。

- **托管 (Managed) 属性**

指托管 (Managed) 类中的公开属性, 由描述符 (Descriptor) 实例处理, 其值存储在存储 (Storage) 属性中。换句话说, 描述符 (Descriptor) 实例和存储 (Storage) 属性为托管 (Managed) 属性提供了基础架构。

必须认识到, `Quantity` 实例是 `LineItem` 的类属性。图 23.2 中的工坊 (Mills) 和小玩意儿 (Gizmos) 强调了这一要点。

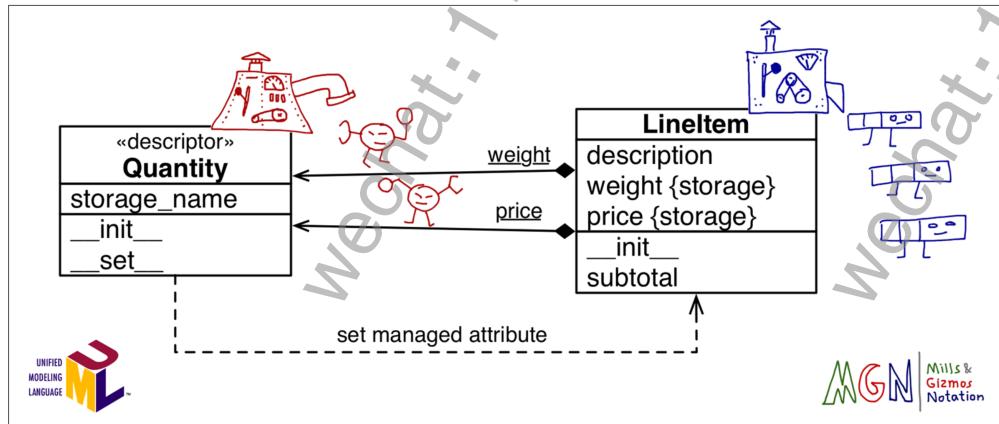


图 23.2: 用 MGN (Mills & Gizmos Notation) 注释的 UML 类图²

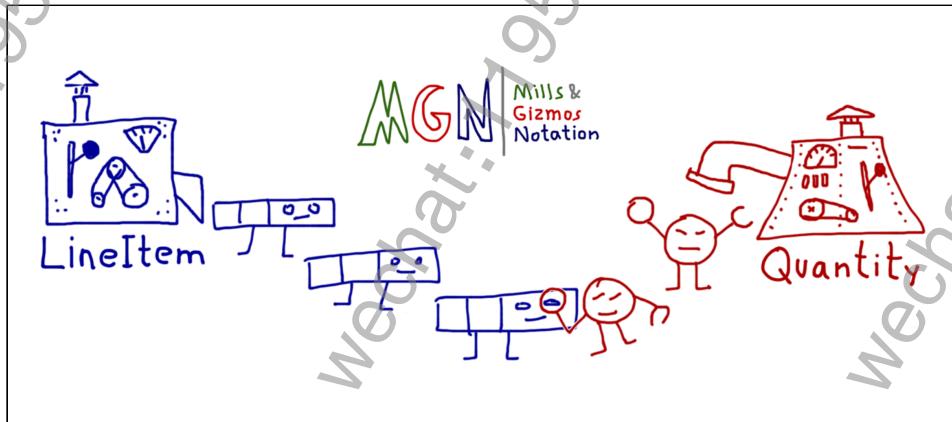
MGN (Mills & Gizmos Notation) 简介

在多次介绍描述符之后, 我意识到 UML 并不擅长展示涉及类和实例的关系, 比如托管 (Managed) 类和描述符 (Descriptor) 实例之间的关系。^a 因此, 我发明了自己的“语言”——Mills & Gizmos Notation (MGN), 我用它来注释 UML 图。

MGN 的设计旨在明确区分类与实例, 如下图所示。在 MGN 中, 类被画成一个“工坊 (Mill)”, 即一个生产小玩意儿 (gizmos) 的复杂机器。类 (即图中的工坊) 是带有杠杆和刻度盘的机器。而小玩意 (gizmos) 则

²类是生产小工具 (Gizmos, 可理解为实例) 的工坊 (Mills)。Quantity 工坊生产 2 个圆头小工具 (Gizmos), 二者被连接到 LineItem 工坊: weight 与 price。LineItem 工坊生产的矩形小工具有自己的 weight 和 price 属性, 并保存了属性值。

是实例,看起来要简单得多。当以彩色呈现本书时,小玩意儿的颜色与制造它的工坊颜色相同。



在上图中,我将 `LineItem` 实例绘制为“表格发票”中的行,其中的 3 个单元格代表 3 个属性(`description`、`weight` 和 `price`)。因为 `Quantity` 实例是描述符,所以它们用“放大镜”来 `__get__` 值,用“爪子”来 `__set__` 值。等我们讨论元类时,你会感谢我的这些涂鸦。

^a在 UML 类图中,类和实例都被画成矩形。二者虽然在视觉上有区别,但类图中很少出现实例,因此开发人员认不出。

涂鸦到此为止,接下来看一下代码:示例 23.1 展示了 `Quantity` 描述符类,示例 23.2 展示了新版的 `LineItem` 类,其中用到了 2 个 `Quantity` 实例。

</> 示例 23.1: `bulkfood_v3.py`:Quantity 描述符不接受负值

```

1  class Quantity:
2
3      def __init__(self, storage_name):
4          self.storage_name = storage_name
5
6      def __set__(self, instance, value):
7          if value > 0:
8              instance.__dict__[self.storage_name] = value
9          else:
10             msg = f'{self.storage_name} must be > 0'
11             raise ValueError(msg)
12
13     def __get__(self, instance, owner):
14         return instance.__dict__[self.storage_name]

```

❶ 描述符是基于协议实现的,无需子类化。

❷ 每个 `Quantity` 实例都有一个 `storage_name` 属性:这是存储(Storage)属性的名称,用于在托管(Managed)实例中保存值。

❸ 当试图对托管(Managed)属性赋值时,会调用 `__set__`。在此处, `self` 是描述符(Descriptor)实例(即 `LineItem.weight` 或 `LineItem.price`), `instance` 是托管(Managed)实例(一个 `LineItem` 实例), `value` 是要分配的值。

❹ 必须将属性值直接存储到 `__dict__` 中;调用 `set_attr(instance, self.storage_name)` 会再次触发 `__set__`

方法，导致无限递归。

- ❸ 需要实现 `__get__`，因为托管（Managed）属性的名称可能与 `storage_name` 不同。稍后将解释 `owner` 参数。

实现 `__get__` 是有必要的，因为用户可能会编写出如下的代码：

```
1 class House:
2     rooms = Quantity('number_of_rooms')
```

在 `House` 类中，托管（Managed）属性是 `rooms`，但存储（Storage）属性是 `number_of_rooms`。给定一个名为 `chaos_manor` 的 `House` 实例，读写 `chaos_manor.rooms` 会经过依附在 `rooms` 上的 `Quantity` 描述符（Descriptor）实例，但读写 `chaos_manor.number_of_rooms` 则会绕过描述符实例。

请注意，`__get__` 接收 3 个参数：`self`、`instance` 和 `owner`。`owner` 参数是对托管（Managed）类（如 `LineItem`）的引用，在用描述符检索类属性时会用到——也许是为模仿 Python 的缺省行为——即在实例中找不到属性名时，继续检索类属性。

如果通过类（如 `LineItem.weight`）检索托管属性（如 `weight`），描述符 `__get__` 方法会接收 `None` 作为实例参数值。

为了支持用户的内省（Introspection）和其他元编程技巧，当通过类访问托管（Managed）属性时，最好让 `__get__` 返回描述符（Descriptor）实例。为此，应下下面这样编写 `__get__` 方法：

```
1 def __get__(self, instance, owner):
2     if instance is None:
3         return self
4     else:
5         return instance.__dict__[self.storage_name]
```

示例 23.2 演示了如何在 `LineItem` 类中使用 `Quantity` 类。

</> 示例 23.2: `bulkfood_v3.py`: 用 `Quantity` 描述符管理 `LineItem` 中的属性

```
1 class LineItem:
2     weight = Quantity('weight') ❶
3     price = Quantity('price') ❷
4
5     def __init__(self, description, weight, price): ❸
6         self.description = description
7         self.weight = weight
8         self.price = price
9
10    def subtotal(self):
11        return self.weight * self.price
```

❶ 第 1 个描述符（Descriptor）实例将管理 `weight` 属性。

❷ 第 2 个描述符（Descriptor）实例将管理 `price` 属性。

❸ 类主体的余下代码与 `bulkfood_v1.py` 中的原始代码（“示例 22.19<680页>”）一样简洁。

示例 23.2 中的代码可按预期运行，防止以 0 美元的价格出售松露（White truffle）³：

³白松露（White truffle）每磅售价高达数千美元。不允许以 0.01 美元的价格出售松露，是留给有进取心的读者的一个练习。我认识一

```

1 >>> truffle = LineItem('White truffle', 100, 0)
2 Traceback (most recent call last):
3 ...
4 ValueError: value must be > 0

```



在编写描述符方法 `__get__` 和 `__set__` 时, 请牢记 `self` 和 `instance` 参数的含义: `self` 是描述符 (Descriptor) 实例, 而 `instance` 是托管 (Managed) 实例。管理实例属性的描述符应该将属性值存储在托管 (Managed) 实例中。这就是 Python 为描述符方法提供 `instance` 参数的原因。

直接在描述符 (Descriptor) 实例中存储各个托管 (Managed) 属性的值可能很诱人, 但却是错误的。换句话说, 在 `__set__` 方法中, 应该像下面这样编码:

```
1 instance.__dict__[self.storage_name] = value # instance 指托管实例
```

而错误的写法是:

```
1 self.__dict__[self.storage_name] = value # self 指托管属性
```

要理解为什么这么做是错误的, 请想想 `__set__` 方法前 2 个参数 (`self` 和 `instance`) 的含义。`self` 是描述符实例, 实际上是托管 (Managed) 类的类属性。内存中可能同时有成千上万个 `LineItem` 实例, 但只有 2 个描述符 (Descriptor) 实例: 即类属性 `LineItem.weight` 和 `LineItem.price`。因此, 存储在描述符 (Descriptor) 实例中的内容实际上会成为 `LineItem` 类属性的一部分, 从而由所有 `LineItem` 实例共享。

示例 23.2 有个缺点, 即在托管 (Managed) 类主体中实例化描述符时需要重复输入属性名称。如果 `LineItem` 类能像下面这样声明就更好了:

```

1 class LineItem:
2     weight = Quantity()
3     price = Quantity()
4
5     # remaining methods as before

```

目前的示例 23.2 需要明确命名各个 `Quantity`, 不仅麻烦而且危险: 如果程序员复制粘贴代码时, 忘记编辑名称, 比如写成 `price = Quantity('weight')`, 程序将会出现错误的行为, 每当设置价格时都会破坏 `weight` 的值。

可问题是, 如“[六 对象引用、可变性与垃圾回收](#)”所述, 赋值语句右侧的表达式会先执行, 而语句左侧的变量此刻还不存在。表达式 `Quantity()` 在求值时会创建一个描述符 (Descriptor) 实例, 而此刻 `Quantity` 类中的代码无法猜出要将描述符实例绑定到哪个变量 (`weight` 还是 `price`)。

值得庆幸的是, 描述符协议现在支持了一个新的特殊方法——`__set_name__`。接下来, 看看如何使用它。

个人, 因为网上商店 (这里不是亚马逊网站) 的一个错误, 他以 18 美元的价格买到了一本价值 1800 美元的统计百科全书。



为描述符的存储 (Storage) 属性自动命名，曾经是一个棘手的问题。在本书第 1 版中，我在本章和下一章用了好几页篇幅和多行代码介绍了不同的解决方案，涉及类装饰器与元类（见“[二十四 类元编程 \(Class Metaprogramming\)](#)”）。但是，这一切在 Python 3.6 中得到了极大的简化。

23.2.2 LineItem 类第 4 版：存储 (Storage) 属性的自动命名

为避免在描述符 (Descriptor) 实例中重复输入属性名，本例将实现特殊方法 `__set_name__` 为各个 `Quantity` 实例设置 `storage_name`。特殊方法 `__set_name__` 是在 Python 3.6 中新增到描述符协议中的。如果描述符实现了 `__set_name__` 方法，Python 解释器就会在类主体中找到的各个描述符上调用 `__set_name__` 方法⁴。

在 [示例 23.3](#) 中，`LineItem` 描述符类不需要 `__init__` 方法。取而代之的是用特殊方法 `__set_item__` 保存存储 (Storage) 属性的名称。

</> [示例 23.3：bulkfood_v4.py](#): 用 `__set_name__` 为每个 `Quantity` 描述符实例设置名称

```

1  class Quantity:
2
3      def __set_name__(self, owner, name): ❶
4          self.storage_name = name ❷
5
6      def __set__(self, instance, value): ❸
7          if value > 0:
8              instance.__dict__[self.storage_name] = value
9          else:
10              msg = f'{self.storage_name} must be > 0'
11              raise ValueError(msg)
12
13      # no __get__ needed ❹
14
15  class LineItem:
16      weight = Quantity() ❺
17      price = Quantity()
18
19      def __init__(self, description, weight, price):
20          self.description = description
21          self.weight = weight
22          self.price = price
23
24      def subtotal(self):
25          return self.weight * self.price

```

❶ `self` 是描述符 (Descriptor) 实例，而不是托管 (Managed) 实例；`owner` 是托管 (Managed) 类，即描述

⁴ 更确切地说，是 `type.__new__`（即代表类的对象的构造函数）调用了 `__set_name__`。内置 `type` 实际上是一个元类，是用户自定义类所属的默认类。这一点起初很难理解，但请放心：“[二十四 类元编程 \(Class Metaprogramming\)](#)” 将专门讨论类的动态配置，包括元类的概念。

符所绑定的类;name 表示将描述符分配给 owner 类中的哪个属性。

- ❷ 此处与“[示例 23.1<700页>](#)”中 ❷ 处 `__init__` 所做的工作一样。
- ❸ 这里的 `__set__` 方法与“[示例 23.1<700页>](#)”中的完全相同。
- ❹ 由于 **存储 (Storage)** 属性的名称与 **托管 (Managed)** 属性的名称一致,因此无需实现 `__get__`。表达式 `product.price` 可直接从 `LineItem` 实例中获取 `price` 属性。
- ❺ 现在,无需将 **托管 (Managed)** 属性名称传递给 `Quantity` 构造函数。这也是本版 `LineItem` 代码的目标。

看过 [示例 23.3](#) 之后,您或许会想,仅为了管理几个属性,有必要编写这么多代码么?但重点是要知道,现在描述符的逻辑已经被分离到一个单独的代码单元中:`Quantity` 类。通常,不会在使用描述符的模块中定义描述符,而是将描述符定义在单独的实用工具模块中,以便可在整个应用程序中复用——如果你正在开发一个库或框架,甚至可以在多个应用程序中使用。

有鉴于此,示例 23.4 更好地体现了描述符的典型用法。

</> [示例 23.4: `bulkfood_v4c.py`](#): 简化 `LineItem` 的定义; `Quantity` 描述符类位于 `model_v4c` 模块中

```

1  import model_v4c as model ❶
2
3
4  class LineItem:
5      weight = model.Quantity() ❷
6      price = model.Quantity()
7
8      def __init__(self, description, weight, price):
9          self.description = description
10         self.weight = weight
11         self.price = price
12
13     def subtotal(self):
14         return self.weight * self.price

```

❶ 导入实现了 `Quantity` 类的 `model_v4c` 模块。

❷ 使用 `model.Quantity` 类。

Django 用户会注意到[示例 23.4](#)看起来很像模型定义。这并非巧合:Django 的模型字段就是描述符。

由于描述符是以类的形式实现的,因此可以利用继承来复用一些代码来实现新的描述符。详见下一节。

23.2.3 `LineItem` 类第 5 版:一种新型描述符

我们虚构的有机食品商店遇到了一个问题:不知何故,某个 `LineItem` 实例的 `description` 属性中包含空白字符,导致订单无法完成。为了避免这种问题,我们将创建一个新的描述符 `NonBlank`。在设计 `NonBlank` 时,我们发现除了验证逻辑外,它与 `Quantity` 描述符非常相似。

据此,我们判断需要重构,定义一个抽象类 `Validated`,覆盖 `__set__` 方法。`__set__` 方法中调用必须由子类实现的 `validate` 方法。

然后,重写 `Quantity`,通过继承 `Validated` 类并只编写 `validate` 方法来实现 `NonBlank` 类。

`Validated`、`Quantity`、`NonBlank` 之间的关系,体现了《设计模式》一书中提出的“**模板方法 (Template**

Method)”:

模板方法定义了一个算法框架,其中包含一系列抽象操作,而子类可以重写这些操作以实现特定于自身需求的具体行为。

在 [示例 23.5](#) 中,Validated.__set__ 是模板方法, self.validate 是抽象操作。

</> [示例 23.5: model_v5.py: 抽象基类 Validated](#)

```
1 import abc
2
3 class Validated(abc.ABC):
4
5     def __set_name__(self, owner, name):
6         self.storage_name = name
7
8     def __set__(self, instance, value):
9         value = self.validate(self.storage_name, value) ❶
10        instance.__dict__[self.storage_name] = value ❷
11
12     @abc.abstractmethod
13     def validate(self, name, value): ❸
14         """return validated value or raise ValueError"""

```

❶ __set__ 将验证操作委托给 validate 方法 ...

❷ ... 然后,用返回 value 更新存储 (self.storage_name) 的值。

❸ validate 是一个抽象方法,即模板方法。

Alex Martelli 更喜欢将这种设计模式称为“自我委托”,我也认为这个名字更确切:__set__ 方法的第 1 行将职责自我委托给 validate 方法。⁵

本例中,Quantity 与 NonBlank 是 Validated 的具体子类 (Concrete Subclass),如 [示例 23.6](#) 所示。

</> [示例 23.6: model_v5.py: Validated 的具体子类 \(Concrete Subclass \) : Quantity 与 NonBlank](#)

```
1 class Quantity(Validated):
2     """a number greater than zero"""
3
4     def validate(self, name, value): ❶
5         if value <= 0:
6             raise ValueError(f'{name} must be > 0')
7         return value
8
9
10    class NonBlank(Validated):
11        """a string with at least one non-space character"""
12
13        def validate(self, name, value):

```

⁵ 摘自 Alex Martelli 的“Python Design Patterns”演讲的第 50 张幻灯片。

```

14     value = value.strip()
15     if not value:          ②
16         raise ValueError(f'{name} cannot be blank')
17     return value           ③

```

- ① 实现抽象方法 `Validated.validate` 要求的模板方法。
- ② 如果去除首尾空白后,没有任何剩余,则剔除该值。
- ③ 具体方法 `validate` 必须返回通过验证的值。此方法可对接收到的数据进行清理、转换或规范化,以确保最终返回的值符合预期格式。在本例中,返回的值首尾不含空白字符。

`model_v5.py` 的用户无需了解这些细节。只需要知道,他们可以使用 `Quantity` 和 `NonBlank` 自动验证实例属性。详见 [示例 23.7](#) 中最新版的 `LineItem` 类。

</> [示例 23.7: `bulkfood_v5.py`: 使用描述符 `Quantity` 和 `NonBlank` 的 `LineItem` 类](#)

```

1 import model_v5 as model           ①
2
3 class LineItem:
4     description = model.NonBlank()  ②
5     weight = model.Quantity()
6     price = model.Quantity()
7
8     def __init__(self, description, weight, price):
9         self.description = description
10        self.weight = weight
11        self.price = price
12
13    def subtotal(self):
14        return self.weight * self.price

```

- ① 导入 `model_v5` 模块,为其指定一个更友好的名字。
- ② 使用 `model.NonBlank` 描述符。其余代码未变。

本章列举的几个 `LineItem` 示例展示了描述符的典型用途,即管理数据属性。`Quantity` 这种描述符被称为 [覆盖型描述符 \(Overriding Descriptor\)](#),因为它的 `_set_` 方法会覆盖(插手接管) [托管 \(Managed\)](#) 实例中同名的实例属性的设置。除此之外,还有 [非覆盖型描述符 \(Nonoverriding Descriptor\)](#)。我们将在下一节详细探讨二者之间的区别。

23.3 覆盖型描述符与非覆盖型描述符的对比

如前所述,Python 处理属性的方式特别不对等。通过实例读取属性时,通常返回的是实例属性,但如果实例中不存在指定的属性,则会获取类属性;而为实例属性赋值时,如果实例中不存在指定的属性,往往会在实例中创建该属性,而根本不会影响类。

这种不对等的处理方式也会影响到描述符。实际上,根据是否实现了特殊方法 `_set_`,描述符可分为两大类:实现了 `_set_` 方法的描述符类是 [覆盖型描述符 \(Overriding Descriptor\)](#);而未实现 `_set_` 方法的描述符类是 [非覆盖型描述符 \(Nonoverriding Descriptor\)](#)。分析如下几个示例后,您将会对这两个术语有深

人的理解。

为了观察这两类描述符的差异, 需要用到几个类。为此, 将用示例 23.8 中的代码作为后续几节的测试平台。



在示例 23.8 中, 每个 `__get__` 和 `__set__` 方法都调用了 `print_args` 函数, 目的是以一种更易读的方式显示调用过程。没必要理解 `print_args` 以及辅助函数 `cls_name` 和 `display`, 不要被它们分散了注意力。

</> 示例 23.8: `descriptorkinds.py`: 用于研究描述符覆盖行为的几个类

```
1  ### 辅助函数, 仅用于显示调用过程 ####
2  def cls_name(obj_or_cls):
3      cls = type(obj_or_cls)
4      if cls is type:
5          cls = obj_or_cls
6      return cls.__name__.split('.')[-1]
7
8  def display(obj):
9      cls = type(obj)
10     if cls is type:
11         return f'<class {obj.__name__}>'
12     elif cls in [type(None), int]:
13         return repr(obj)
14     else:
15         return f'<{cls_name(obj)} object>'
16
17 def print_args(name, *args):
18     pseudo_args = ', '.join(display(x) for x in args)
19     print(f'-> {cls_name(args[0])}.__{name}__({pseudo_args})')
20
21
22 ### 对本示例比较重要的几个类 ####
23
24 class Overriding: ❶
25     """又叫数据描述符或强制描述符"""
26
27     def __get__(self, instance, owner):
28         print_args('get', self, instance, owner) ❷
29
30     def __set__(self, instance, value):
31         print_args('set', self, instance, value)
32
33
34 class OverridingNoGet: ❸
35     """没有 ``__get__`` 方法的覆盖型描述符"""
36
37     def __set__(self, instance, value):
38         print_args('set', self, instance, value)
```

```

39
40 class NonOverriding: ④
41     """又叫非数据描述符或遮盖型描述符"""
42
43     def __get__(self, instance, owner):
44         print_args('get', self, instance, owner)
45
46
47 class Managed: ⑤
48     over = Overriding()
49     over_no_get = OverridingNoGet()
50     non_over = NonOverriding()
51
52     def spam(self): ⑥
53         print(f'--> Managed.spam({display(self)})')

```

- ① 含有 `__get__` 方法与 `__set__` 方法的 覆盖型描述符 (Overriding Descriptor)。
- ② 本例中的每个描述符方法都会调用 `print_args` 函数。
- ③ 不含 `__get__` 方法的 覆盖型描述符 (Overriding Descriptor)。
- ④ 不含 `__set__` 方法, 所以这是一个 非覆盖型描述符 (Nonoverriding Descriptor)。
- ⑤ 托管 (Managed) 类, 用各个描述符 (Descriptor) 类的实例定义托管类中的属性。
- ⑥ 将 `spam` 方法放到此处是为了对比, 因为方法也是描述符。

接下来的几小节, 将分析对 `Managed` 类及其实例做属性读写时的行为, 还会讨论 [示例 23.8](#) 中定义的各个描述符。

23.3.1 覆盖型描述符

实现了特殊方法 `__set__` 的描述符被称为 “[覆盖型描述符 \(Overriding Descriptor\)](#)”, 因为虽然描述符是类属性, 但是实现了 `__set__` 方法的描述符将覆盖对实例属性的赋值操作。[“示例 23.3<703页>”](#) 中的 `Quantity` 描述符就是这样实现的。特性 (property) 也是覆盖型描述符 (Overriding Descriptor): 如果未提供设值 (setter) 函数, 则特性 (property) 类的 `__set__` 方法将引发 `AttributeError` 异常, 以表明相应的属性是只读的。可以用 [示例 23.8](#) 中的代码测试覆盖型描述符 (Overriding Descriptor) 的行为, 测试结果如[示例 23.9](#)所示。



在讨论这些概念时, Python 的贡献者和作者使用了不同的术语。而我采用了《Python in a Nutshell》一书中的覆盖型描述符 (Overriding Descriptor)。Python 官方文档使用 “[数据描述符 \(Data Descriptor\)](#)”, 但 覆盖型描述符 (Overriding Descriptor) 更能凸显它的特殊行为。覆盖型描述符 (Overriding Descriptor) 也被称为 “[强制型描述符 \(Enforced Descriptor\)](#)”。非覆盖型描述符 (Nonoverriding Descriptor) 的又称 “[非数据描述符 \(Nondata Descriptor\)](#)” 或 “[遮盖型描述符 \(Shadowable Descriptor\)](#)”。

</> 示例 23.9: 覆盖型描述符的行为

```

1  >>> obj = Managed()          ❶
2  >>> obj.over               ❷
3  -> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
4  >>> Managed.over           ❸
5  -> Overriding.__get__(<Overriding object>, None, <class Managed>)
6  >>> obj.over = 7            ❹
7  -> Overriding.__set__(<Overriding object>, <Managed object>, 7)
8  >>> obj.over               ❺
9  -> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
10 >>> obj.__dict__['over'] = 8 ❻
11 >>> vars(obj)              ❼
12 {'over': 8}
13 >>> obj.over               ❽
14 -> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)

```

- ❶ 创建 Managed 对象, 供测试使用。
- ❷ obj.over 将触发描述符 __get__ 方法, 将托管 (Managed) 实例 obj 传给 __get__ 方法的第 2 个参数 instance。
- ❸ Managed.over 将触发描述符 __get__ 方法, 将 None 传给 __get__ 方法的第 2 个参数 instance。
- ❹ 为 obj.over 赋值将触发描述符 __set__ 方法, 并将数字 7 传给 __set__ 方法的最后一个参数 value。
- ❺ 读取 obj.over 仍会调用描述符的 __get__ 方法。
- ❻ 绕过描述符, 直接通过 obj.__dict__ 为实例属性 over 赋值。
- ❼ 验证❻指定的新值是否位于字典 obj.__dict__ 名为 over 的键下。
- ❽ 但是, 即使使用名为 over 的实例属性, Managed.over 描述符仍然会覆盖读取 obj.over 的操作, 即与❷一样, 触发 __get__ 方法。

23.3.2 不含 __get__ 方法的覆盖型描述符

特性 (property) 和其他覆盖型描述符 (Overriding Descriptor) (如 Django 模型字段) 都实现了特殊方法 __set__ 和 __get__, 但也可以只实现 __set__, “示例 23.8<707页>” 的❸处所示。此时, 描述符只处理写操作。通过实例访问描述符时, 由于没有处理读取操作的 __get__ 方法, 因此会返回描述符对象本身。如果直接通过实例的 __dict__ 属性创建了与描述符同名的实例属性, 并赋予了一个新值, 则后续再为此属性赋值时, 仍由 __set__ 方法接管, 但读取该属性时将直接从实例中返回新赋予的值, 而不再返回描述符对象。也就是说, 在属性读取时, 实例属性会遮盖描述符, 但在属性写入时仍由描述符处理赋值操作。如示例 23.10 所示。

</> 示例 23.10: 不含 __get__ 方法的覆盖型描述符

```

1  >>> obj.over_no_get        ❶
2  <__main__.OverridingNoGet object at 0x665bcc>
3  >>> Managed.over_no_get   ❷
4  <__main__.OverridingNoGet object at 0x665bcc>
5  >>> obj.over_no_get = 7    ❸
6  -> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
7  >>> obj.over_no_get        ❹

```

```

8 <__main__.OverridingNoGet object at 0x665bcc>
9 >>> obj.__dict__['over_no_get'] = 9      ❸
10 >>> obj.over_no_get      ❹
11 9
12 >>> obj.over_no_get = 7      ❺
13 -> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
14 >>> obj.over_no_get      ❻
15 9

```

- ❶ 此覆盖型描述符 (Overriding Descriptor) 中不含 `__get__` 方法。因此, `obj.over_no_get` 会从托管 (Managed) 类中获取描述符 (Descriptor) 实例。
- ❷ 直接读取 `Managed.over_no_get`, 亦是如此——从托管 (Managed) 类中获取描述符 (Descriptor) 实例。
- ❸ 尝试为 `obj.over_no_get` 赋值, 会调用描述符的 `__set__` 方法。
- ❹ 因为 `__set__` 方法中未修改属性, 所以再次读取 `obj.over_no_get` 仍会从托管 (Managed) 类中获取描述符 (Descriptor) 实例。
- ❺ 直接通过实例的 `__dict__` 属性, 设置名为 `over_no_get` 的实例属性。
- ❻ 现在, 实例属性 `over_no_get` 遮蔽了描述符, 但只有读取操作是这样的。
- ❼ 尝试为 `obj.over_no_get` 赋值, 赋值操作仍由描述符的 `__set__` 方法处理。
- ❽ 但在属性读取时, 只要存在与描述符同名的实例属性, 描述符就会被该实例属性遮盖。

23.3.3 非覆盖型描述符

未实现特殊方法 `__set__` 的描述符属于非覆盖型描述符 (Nonoverriding Descriptor)。如果设置了同名的实例属性, 那么非覆盖型描述符 (Nonoverriding Descriptor) 就会被该实例属性遮盖, 使描述符无法处理该特定实例中的特定属性。所有方法与 `@functools.cached_property` 都是以非覆盖型描述符 (Nonoverriding Descriptor) 形式实现的。示例 23.11 展示了非覆盖型描述符 (Nonoverriding Descriptor) 的操作。

</> 示例 23.11: 非覆盖型描述符的行为

```

1 >>> obj = Managed()
2 >>> obj.non_over      ❶
3 -> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
4 >>> obj.non_over = 7      ❷
5 >>> obj.non_over      ❸
6 7
7 >>> Managed.non_over      ❹
8 -> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
9 >>> del obj.non_over      ❺
10 >>> obj.non_over      ❻
11 -> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)

```

- ❶ `obj.non_over` 将触发描述符的 `__get__` 方法, 并将 `obj` 传给 `__get__` 的第 2 个参数 `instance`。
- ❷ `Managed.non_over` 是个非覆盖型描述符 (Nonoverriding Descriptor), 因此不含干涉赋值操作的 `__set__` 方法。

- ③ 现在, 对象 obj 有个名为 non_over 的实例属性, 会遮盖托管 (Managed) 类 (即 Managed) 中的同名描述符属性。
- ④ Managed.non_over 描述符仍然存在, 会通过类来截获这次访问。
- ⑤ 若将实例属性 non_over 删除了 ...
- ⑥ ... 则读取 obj.non_over 时, 与①处行为相同, 仍将触发描述符的 __get__ 方法, 并将托管 (Managed) 实例 (即 obj) 传给 __get__ 的第 2 个参数 instance。

在上述几个示例中, 多次为几个与描述符同名的实例属性赋值, 得到的结果依据描述符是否含有 __set__ 方法而有所不同。

依附在类上的描述符无法控制类属性的赋值操作。其实, 这意味着为类属性赋值可以破坏类中的描述符属性。详见下一节。

23.3.4 覆盖类中的描述符

无论描述符是什么类型的, 为类属性 (Class Attribute) 赋值都可以覆盖 (破坏) 类中的描述符。这是一种猴子补丁 (Monkey-patch) 技术, 但在示例 23.12 中, 描述符被替换为整数, 这实际上会导致依赖描述符的类无法正常执行操作。

</> 示例 23.12: 通过类属性赋值, 可以覆盖类中的任何描述符

```
1  >>> obj = Managed()      ❶
2  >>> Managed.over = 1     ❷
3  >>> Managed.over_no_get = 2
4  >>> Managed.non_over = 3
5  >>> obj.over, obj.over_no_get, obj.non_over ❸
6  (1, 2, 3)
```

- ❶ 为后面的测试创建一个实例。
- ❷ 为类属性 (Class Attribute) 赋值, 可覆盖 (破坏) 类中的描述符属性。
- ❸ 描述符真的不见了。

示例 23.12 揭示了读写属性的另一种不对等: 类属性的读取可以由依附在托管类上带有 __get__ 方法的描述符处理, 但类属性的写入却不由依附在托管类上带有 __set__ 方法的描述符处理。



要想控制类属性 (Class Attribute) 的操作, 需要将描述符依附在类所属的类上, 即依附在元类上。默认情况下, 用户定义类的元类是 type, 并且不能为 type 添加属性。但在“二十四类元编程 (Class Metaprogramming)”中, 我们将创建自己的元类。

下面调转话题, 介绍一下如何用描述符实现 Python 中的方法。

23.4 方法也是描述符

由于用户定义的所有函数中都会有一个特殊方法 `__get__`，当将函数依附到类中之后，这些函数就相当于描述符。所以，当在实例上调用用户函数时，类中的函数就会变成绑定方法。示例 23.13 展示了从“示例 23.8<707页>”定义的 `Managed` 类中读取 `spam` 方法。

</> 示例 23.13: 方法也是非覆盖型描述符

```

1  >>> obj = Managed()
2  >>> obj.spam      ❶
3  <bound method Managed.spam of <descriptorkinds.Managed object at 0x74c80c>>
4  >>> Managed.spam ❷
5  <function Managed.spam at 0x734734>
6  >>> obj.spam = 7 ❸
7  >>> obj.spam
8  7

```

❶ 读取 `obj.spam` 将获取一个绑定方法对象。

❷ 但是，读取 `Managed.spam` 获取的是一个函数。

❸ 为 `obj.spam` 赋值，会遮蔽类属性（Class Attribute），导致无法通过 `obj` 实例访问 `spam` 方法。

函数中不会实现特殊方法 `__set__`，因为函数是非覆盖型描述符（Nonoverriding Descriptor），如示例 23.13（❸处）的最后一行所示。

从示例 23.13 可得到的另一个重要信息是：`obj.spam` 与 `Managed.spam` 获取的是不同的对象。与描述符一样，当通过托管（Managed）类访问时，函数的 `__get__` 会返回函数自身的引用。但当通过实例访问时，函数的 `__get__` 返回一个绑定（Bound）方法对象：这是一种可调用对象，将函数包装在内部，并将托管（Managed）实例（如 `obj`）绑定到函数的第一个参数（即 `self`）。这个行为与 `functools.partial` 函数一样，详见“7.8.2 用 `functools.partial` 冻结参数<200页>”。为了深入理解这一机制，请参阅示例 23.14。

</> 示例 23.14: `method_is_descriptor.py`: `Text` 类，衍生自 `UserString` 类

```

1  import collections
2
3  class Text(collections.UserString):
4
5      def __repr__(self):
6          return 'Text={!r}'.format(self.data)
7
8      def reverse(self):
9          return self[::-1]

```

下面来分析一下 `Text.reverse` 方法，如示例 23.15 所示。

</> 示例 23.15: 试验一个方法

```

1  >>> word = Text('forward')
2  >>> word      ❶
3  Text('forward')

```

```

4  >>> word.reverse()          ❷
5  Text('drawrof')>>> Text.reverse(Text('backward'))          ❸
6  Text('drawkcab')
7  >>> type(Text.reverse), type(word.reverse)          ❹
8  (<class 'function'>, <class 'method'>)
9  >>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')])) ❺
10 ['diaper', (30, 20, 10), Text('desserts')]
11 >>> Text.reverse.__get__(word)          ❻
12 <bound method Text.reverse of Text('forward')>
13 >>> Text.reverse.__get__(None, Text)          ❼
14 <function Text.reverse at 0x101244e18>
15 >>> word.reverse          ❽
16 <bound method Text.reverse of Text('forward')>
17 >>> word.reverse.__self__          ❾
18 Text('forward')
19 >>> word.reverse.__func__ is Text.reverse ❿
20 True

```

- ❶ Text 实例的 `repr` (字符串表示形式) 看起来像一个 Text 构造函数调用, 它会生成一个相等的实例。
- ❷ 通过实例调用 `reverse` 方法, 将返回反向拼写的单词。
- ❸ 通过类调用 `reverse` 方法, 相当于调用 `reverse()` 函数。
- ❹ 注意❷与❸处是不同的: 前者是调用绑定到实例的方法, 后者是调用类中的函数。
- ❺ `Text.reverse` 相当于函数, 甚至可以处理 Text 实例之外的其他对象。
- ❻ 所有函数都是非覆盖型描述符 (Nonoverriding Descriptor)。若调用函数的 `__get__` 方法时传入实例, 则得到的是绑定到那个实例上的方法: 即 `word.reverse`。
- ❼ 若调用函数的 `__get__` 方法时, 为参数 `instance` 传入 `None`, 则得到的是函数本身。
- ❽ 表达式 `word.reverse` 实际上是调用 `Text.reverse.__get__(word)`, 返回对应的绑定 (Bound) 方法。
- ❾ 绑定方法对象有一个 `__self__` 属性, 其值是调用该方法的实例的引用。
- ❿ 绑定方法对象还有一个 `__func__` 属性, 其值是依附在托管 (Managed) 类上那个原始函数的引用。

绑定方法对象中还有一个 `__call__` 方法, 用于处理实际的调用。此方法将调用 `__func__` 中引用的原始函数, 并将方法的 `__self__` 属性传给原始函数的第一个参数 (即 `self`)。这就是形参 `self` 的隐式绑定方式。

将函数转化为绑定 (Bound) 方法的方式是描述符在语言中被用作基础设施的最典型例子。

在深入探讨描述符和方法的工作原理后, 下面看一下用法方面的实际建议。

23.5 描述符用法建议

如下根据前文探讨的描述符特征, 给出一些实用性结论:

- 使用特性 (property), 以保持简单。

内置特性 (property) 是覆盖型描述符 (Overriding Descriptor), 即便未定义设值 (setter) 方法⁶, 它也会实现特殊方法 `__set__` 与 `__get__`。特性 (property) 的 `__set__` 会默认引发 “`AttributeError: can't set attribute`” 异常。所以, 创建只读属性的最简单方法就是使用特性 (property), 这能避免下一条中所

⁶即使未定义删值 (deleter) 方法, property 装饰器也会提供 `__delete__` 方法。

述的问题。

- 只读描述符必须含有 `__set__` 方法一样

用描述符 (Descriptor) 类实现只读属性时要记住, `__set__` 与 `__get__` 这 2 个方法必须都定义。否则, 描述符会被同名的实例属性所遮蔽。只读属性的 `__set__` 方法只需引发 `AttributeError` 异常, 并提供合适的错误消息。⁷

- 用于验证的描述符可以只含有 `__set__` 方法一样

在仅用于属性验证的描述符中, `__set__` 方法应检查参数 `value` 所得到的值。如果有效, 就用描述符实例的名称作为“键 (key)”, 直接在实例的 `__dict__` 属性中设置。这样, 从实例中读取同名属性会很快, 因为不用经过 `__get__` 处理。详见“[示例 23.3<703页>](#)”中的代码。

- 仅含 `__get__` 方法的描述符可用于实现高效缓存

若代码中只编写了 `__get__` 方法, 则得到的是非覆盖型描述符 (Nonoverriding Descriptor)。这种描述符可用于执行某些耗费资源的计算, 然后通过为实例设置同名属性来缓存结果。⁸同名的实例属性会遮盖描述符, 因此后续访问应直接从实例的 `__dict__` 属性中获取值, 而不再触发描述符的 `__get__` 方法。`@functools.cached_property` 装饰器实际上生成的就是非覆盖型描述符 (Nonoverriding Descriptor)。

- 非特殊方法可以被实例属性遮盖

函数和方法都只实现了 `__get__`, 所以它们都是非覆盖型描述符 (Nonoverriding Descriptor)。像 `my_obj.the_method=7` 这样的简单赋值之后, 后续通过该实例访问 `the_method`, 得到的是数字 7——但是不影响类和其他实例。然而, 特殊方法并不受此问题的影响。Python 解释器只在类中查找特殊方法, 也就是说, `repr(x)` 执行的其实是 `x.__class__.__repr__(x)`。所以, 实例 `x` 中定义的 `__repr__` 属性不会影响 `repr(x)` 调用。出于同样的原因, 实例的 `__getattr__` 属性也不会破坏常规的属性访问规则。

实例的非特殊方法可以轻易地被覆盖 (破坏), 这听起来似乎很脆弱且容易出错。但是我个人在 20 多年的 python 编程生涯中, 从未遇到过这样的问题。但是, 如果要创建大量的动态属性, 且属性名来自不受控制的数据集, 则了解这个问题是有必要的。或许可以实现某种机制, 如筛选或转义动态属性的名称, 以维持数据的健全性。



“[示例 22.4<667页>](#)”中的 `FrozenJSON` 类不存在实例属性遮盖实例方法的问题, 因为该类只有几个特殊方法与一个 `build` 类方法。只要通过类来访问类方法, 则类方法就是安全的, 在“[示例 22.4<667页>](#)”中我就是这么调用 `FrozenJSON.build` 方法的。在“[示例 22.6<670页>](#)”中, `build` 方法被换成了 `__new__` 方法。“[22.3 计算的特性<671页>](#)”中的 `Record` 类与 `Event` 类也是安全的, 因为二者只实现了特殊方法、静态方法与特性 (property)。特性 (property) 是覆盖型描述符 (Overriding Descriptor), 因此不会被实例属性遮盖。

在本章最后, 将介绍描述符中还未涉及的, 但在讨论特性 (property) 时已介绍过的 2 个特性: 文档和对删除托管属性的处理。

⁷Python 为此类异常提供的错误消息不一致。若试图修改 `complex` 数值的 `c.real` 属性, 则得到的错误消息是“`AttributeError: read-only attribute`”。但是, 尝试修改 `c.conjugate` (`complex` 对象的方法), 则得到的错误消息是“`AttributeError: 'complex' object attribute 'conjugate' is read-only`”。甚至连“`read-only`”的拼写方式都不一致。

⁸但是, 如“[3.9 dict 实现方式对实践的影响<84页>](#)”所述, 在 `__init__` 方法运行之后创建实例属性, 会违背“键共享 (Key-Sharing)”内存优化。

23.6 描述符的文档字符串与覆盖删除操作

描述符类的 docstring 将被用作 [托管 \(Managed\)](#) 类 中各个 [描述符 \(Descriptor\)](#) 实例的文档。图 23.3 展示了“[示例 23.6<705页>](#)”与“[示例 23.7<706页>](#)”含有描述符 `Quantity` 与 `NonBlank` 的 `LineItem` 类的帮助界面。

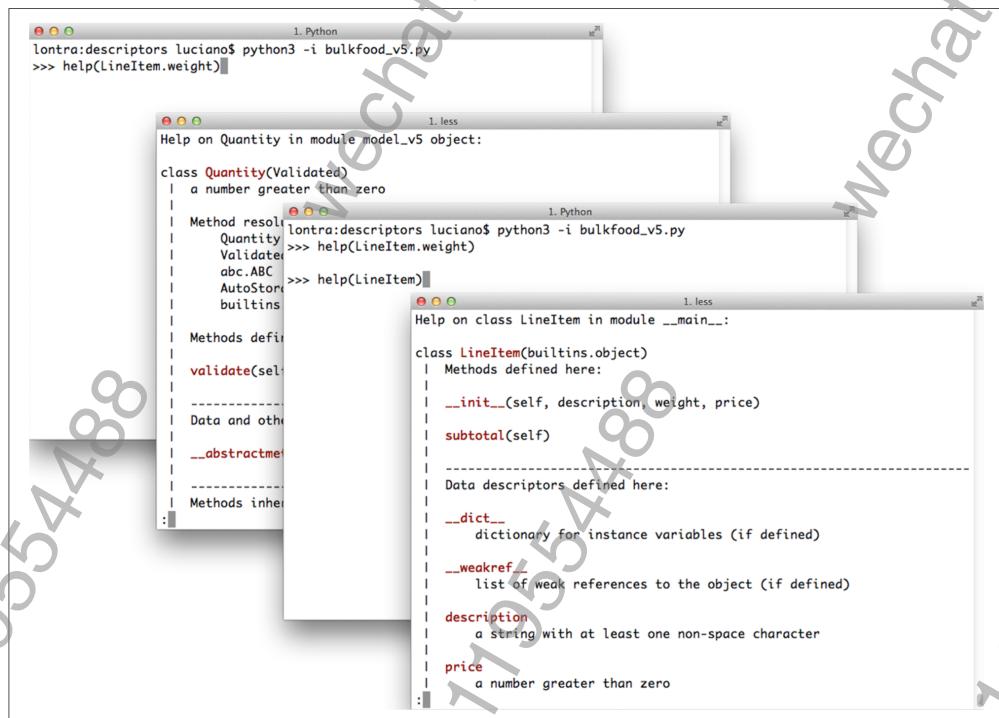


图 23.3: 发出 `help(LineItem.weight)` 和 `help(LineItem)` 命令时的 Python 控制台截图。

如图所见,帮助界面提供的信息有些不足。例如,在 `LineItem` 类中,最好增加 `weight` 必须以 kg 为单位的说明信息。这对于 [特性 \(property\)](#) 来说是“信手拈来”,因为各个 [特性 \(property\)](#) 都只处理一个特定的 [托管 \(Managed\) 属性](#)。但对于描述符来说,属性 `weight` 与 `price` 使用的是同一个描述符类 `Quantity`。⁹

在介绍 [特性 \(property\)](#) 时,还提到了本章描述符未涉及的另一个细节——对删除 [托管 \(Managed\) 属性](#) 的处理。在描述符类中,除了实现常规的 `__get__` 与 `__set__` 方法,还可以实现 `__delete__` 方法;或者只实现 `__delete__` 方法,以处理删除 [托管 \(Managed\) 属性](#) 操作。我故意省略了 `__delete__` 相关内容,是因为现实中很少用到 `__delete__`。如果确需使用,请阅读“[Python 数据模型文档](#)”的“[Implementing Descriptors](#)”一节。有兴趣的读者,可以编写一个实现了特殊方法 `__delete__` 的描述符类,以作为练习。

23.7 本章小结

本章首个示例是接续“[二十二 动态属性和特性](#)”中的 `LineItem` 系列示例。在“[示例 23.2<701页>](#)”中,用描述符取代了 [特性 \(property\)](#)。了解到,描述符是一个类,它的实例可用于管理 [托管 \(Managed\)](#) 类中的属性。为了介绍这种属性管理机制,本章引入了多个术语,如[托管 \(Managed\) 实例](#)、[存储 \(Storage\) 属性](#)。

在“[23.2.2 LineItem 类第 4 版: 存储 \(Storage\) 属性的自动命名<703页>](#)”中,删除了声明 `Quantity` 描

⁹定制各个描述符实例的帮助文本特别难。有一种方案是为各个描述符实例动态构建包装类。

述符所需的参数 `storage_name`, 因为该参数多余且易出错。新的解决方案是在 `Quantity` 类中实现特殊方法 `__set_name__`, 将托管 (Managed) 属性的名称保存为 `self.storage_name`。

“23.2.3 `LineItem` 类第 5 版: 一种新型描述符<704页>”展示了如何通过子类化抽象描述符 (Descriptor) 类来共享代码, 同时构建具有一些共同功能的专用描述符。

随后, 本章分析了描述符中含有或不含 `__set__` 方法时的不同行为, 从而对覆盖型描述符 (Overriding Descriptor) 与非覆盖型描述符 (Nonoverriding Descriptor) (又称“数据描述符”与“非数据描述符”) 进行了严格区分。通过详细测试, 揭示了描述符何时接管属性, 以及何时被遮盖、被绕过或被覆盖 (破坏)。

接下来, 本章分析了一种特殊的非覆盖型描述符 (Nonoverriding Descriptor) ——方法。通过控制台中的测试揭示了, 当通过实例访问时, 如何利用描述符协议将依附在类上的函数绑定为实例方法。

作为本章的总结, “23.5 描述符用法建议”对描述符的用法给出了一些实用建议; “23.6 描述符的文档字符串与覆盖删除操作”简要介绍了如何为描述符添加文档。



如“23.1 本章新增内容”所述, 由于 Python 3.6 为描述符协议新增了特殊方法 `__set_name__`, 本章中的几个示例变得简单多了。这就是语言的进化!

23.8 延伸阅读

除了《The Python Language Reference》必读的“3. Data model”之外, Raymond Hettinger 编写的“Descriptor Guide”也是一份宝贵的参考资料——这是 Python 官方文档“Python HOWTOs 合集”的一部分。

对于 Python 对象模型相关的话题来说,《Python in a Nutshell, 3rd Ed》(Alex Martelli、Anna Ravenscroft、Steve Holden 著)一书权威且客观。Martelli 还做了一次题为“Python’s Object Model”的演讲, 深入介绍了特性 (property) 与描述符 (详见幻灯片与视频)。



注意, 在 2016 年 Python 通过“PEP 487”之前, 关于介绍描述符的示例代码可能都比现在复杂。因为 Python 3.6 之前, 描述符类还不支持特殊方法 `__set_name__`。

《Python Cookbook, 3rd Ed》(David Beazley、Brian K. Jones 合著)一书提供了更多具有实践意义的示例, 其中包括许多展示描述符用法的经典示例。推荐的章节包括“6.12. Reading Nested and Variable-Sized Binary Structures”、“8.10. Using Lazily Computed Properties”、“8.13. Implementing a Data Model or Type System”、“9.9. Defining Decorators As Classes”。最后一个示例解决了函数装饰器、描述符与方法之间交互的深层问题, 介绍了如何将函数装饰器实现为一个包含 `__call__` 方法的装饰器类。如果希望该装饰器类既可用于装饰函数, 又可用于装饰方法, 除了实现 `__call__` 方法之外, 还要为该装饰器类实现特殊方法 `__get__`。

“PEP 487 -Simpler customisation of class creation”引入了特殊方法 `__set_name__`, 还提供了一个验证描述符示例。

杂谈

self 的设计

在 Python 中, 要求将 `self` 明确声明为方法的第一个参数是一个有争议的设计决定。在使用这种语言 23 年之后, 我已经习惯了。我认为这个决定体现了“越糟越好 (worse is better)”设计理念。计算机科学家 Richard P. Gabriel 在《The Rise of Worse is Better》一书中介绍了这一设计理念。这种理念的首要任务是“简单 (Simplicity)”, Gabriel 将其表述为:

无论是实现, 还是接口, 设计都必须足够简单。简单的实现比简单的接口更重要。简单是设计中最重要的考虑因素。

Python 要求明确声明 `self`, 体现了这种设计理念, 其简单而优雅的实现, 是以牺牲用户接口为代价的: 像 `def zfill(self, width)` 这样的方法签名, 在视觉上与 `label.zfill(8)` 调用并不匹配。

Modula-3 引入了与 Python 中标识符 `self` 相同的约定。但是有一个关键的区别: 在 Modula-3 中, 接口声明与接口实现是分离开来的, 在接口声明中省略了 `self` 参数, 因此从用户的角度来看, 接口声明中的方法用到的参数数量与方法调用时显式传入的参数数量完全相同。

随着时间的推移, Python 为方法参数提供的错误消息越来越清晰。对于除了 `self` 之外仅接受 1 个参数的用户定义方法, 若用户调用 `obj.meth()`, Python 2.7 会引发:

1 `TypeError: meth() takes exactly 2 arguments (1 given)`

而在 Python 3 中, 不再提及令人困惑的参数数量, 而且明确指明了缺少的参数名称, 如下所示:

`TypeError: meth() missing 1 required positional argument: 'x'`

除了要明确声明参数 `self` 之外, 必须用限定符 `self` 访问实例属性的限制也备受批评。例如, 请参阅 A. M. Kuchling 发表的博文 “Python Warts”(已归档); Kuchling 本人并不太讨厌 `self` 限定符, 但他提到了这一点——可能是为了呼应 comp.lang.python 小组的观点。我个人并不介意 `self` 限定符, `self` 有利于区分局部变量和实例属性。我介意的是在 `def` 语句中使用 `self`。

讨厌明确声明 `self` 的程序员, 可以想想 JavaScript 中 `隐式 this` 那种令人费解的语义, 或许就会心理平衡多了。明确声明 `self` 的这种用法有一些合理之处, Guido 在他的博客 “The History of Python” 题目为 “Adding Support for User-defined Classes” 的博文中说明了相关原因。

wechat: 119554488

类元编程 (Class Metaprogramming)

每个人都知道, 程序调试的难度是程序编写的 2 倍。因此, 如果在编写程序时就已经足够精心, 那么在进行调试的时候要如何解决问题呢?

——Brian W. Kernighan, P. J. Plauger, 《The Elements of Programming Style》^a

^a引自《The Elements of Programming Style》的 “Chapter 2, Expression” 第 10 页。

类元编程 (Class Metaprogramming) 是在运行时创建或自定义类的艺术。在 Python 中, 类是一等对象, 因此可以随时使用函数来创建一个新类, 而无需使用 `class` 关键字。类装饰器也是函数, 但其目的是检查、更改甚至用另一个类替换被装饰的类。最后, 元类是最先进的类元编程 (Class Metaprogramming) 工具: 它可以让你创建具有特殊属性的全新类别的类, 比如我们已经见过的抽象基类 (ABCs)。

元类功能强大, 但难以掌握。类装饰器能解决许多同类问题, 并且更容易理解。此外, Python 3.6 实现的 “PEP 487 -Simpler customisation of class creation” 提供了一些特殊方法, 可以完成以前需要元类或类装饰器才能完成的任务。¹

本章将按复杂度从易到难的顺序, 讲解类元编程 (Class Metaprogramming) 技术。



这是一个令人兴奋的主题, 很容易让人忘乎所以。因此必须提出如下警告:

- 为了保证代码可读性和可维护性, 在应用程序开发中应尽量避免使用本章介绍的技术。
- 如果想编写伟大的 Python 框架, 本章介绍的这些工具是必备的。

24.1 本章新增内容

本书第 1 版 “类元编程” 一章中的所有代码依然可正常运行。但是, 鉴于 Python 3.6 以来增加的新特性, 先前的一些示例已不再是最简单的解决方案了。

¹这并不表示 “PEP 487” 破坏了使用这些功能的代码。而是表示在 Python 3.6 之前使用类装饰器或元类的代码, 现在可以用普通类进行重构, 从而使代码更简单、更高效。

我将那些示例替换掉了。这些示例中有些强调了 Python 新增的元编程功能,有些为了使用更高级的技术而增加了新需求。一些新示例利用类型提示 (Type Hints) 实现了类似于 `@dataclasses.dataclass` 装饰器和 `typing.NamedTuple` 的类构建器。

“24.10 元类的实际运用<750页>”是新增的章节,其中包含一些关于元类适用性的高层次思考。



最好的重构方案是,通过使用新的简洁方法解决相同问题,从而移除冗余的代码。这个方案适用于生产代码以及书籍的写作过程。

首先介绍 Python 数据模型为所有类的定义,提供的属性与方法。

24.2 身为对象的类

与 Python 中的大多数程序实体一样,类也是对象。Python 数据模型为每个类都定义了一些属性,详见《The Python Standard Library》中“Built-in Types”一章的“Special Attributes”中。其中,有 3 个属性在本书中已出现过多次:`__class__`、`__name__` 和 `__mro__`。此外,还有如下标准的类属性 (Class Attribute) :

- `cls.__bases__`

由类的基类构成的元组。

- `cls.__qualname__`

类或函数的限定名称,即从模块的全局作用域到类定义处的点(.)分路径。当在类中定义另一个类时,会用到此属性。例如,在 Django 模型类 (如 `Ox`) 中,有一个名为 `Meta` 的内部类。`Meta` 类的 `__qualname__` 是 `Ox.Meta`,但 `Meta` 类的 `__name__` 只是 `Meta`。该属性的规范是“PEP 3155 – Qualified name for classes and functions”。

- `cls.__subclasses__()`

该方法返回由类的直接子类构成的列表。此方法的实现用到了弱引用 (Weak Reference),以防止超类与子类之间的循环引用 (子类在 `__bases__` 属性中存储指向超类的强引用)。此方法返回的列表中是内存里现存的子类,不含尚未导入的模块中的子类。

- `cls.mro()`

Python 解释器在构建类时,调用此方法,以获得存储在类属性 `__mro__` 中的超类元组。元类可以重写此方法,以自定义正在构建的类的MRO (方法解析顺序)。



`dir(...)` 函数不会列出本节提到的任何属性。

既然类也是对象,那么类所属的类又是什么呢?

24.3 `type`: 内置的类工厂函数

我们通常认为 `type` 是一个函数,可返回对象所属的类,即 `type(my_object)` 返回 `my_object.__class__`。

但实际上, `type` 是一个类。当被调用并为其传入 3 个参数时, 可创建一个新类。

以如下这个简单的类为例:

```
1 class MyClass(MySuperClass, MyMixin):  
2     x = 42  
3  
4     def x2(self):  
5         return self.x * 2
```

使用 `type` 构造方法, 可以在运行时用如下代码创建 `MyClass` 类:

```
1 MyClass = type('MyClass',  
2                 (MySuperClass, MyMixin),  
3                 {'x': 42, 'x2': lambda self: self.x * 2},  
4                 )
```

该 `type()` 调用在功能上, 与先前的 “`class MyClass` 语句块” 相同。

Python 解释器在读取 `class` 语句时, 会调用 `type` 构建类对象, 传入的参数如下所示:

- `name`

出现在 `class` 关键字之后的标识符, 如 `MyClass`。

- `bases`

在 `class` 关键字后的括号中列出的超类元组, 如果 `class` 语句中未提到超类, 则为 `(object,)`。

- `dict`

属性名与属性值的映射。如 “[23.4 方法也是描述符](#)” 所述, 可调用对象变成了方法, 其他值成为了类属性 (Class Attribute)。



`type` 构造方法接受可选的关键字参数, 这些参数会被 `type` 本身忽略, 但会原封不动地传给 `__init_subclass__`, 后者必须使用这些参数。将在 “[24.5 介绍 `__init_subclass__`](#)” 详细介绍这个特殊方法, 但我不介绍关键字参数的使用。更多信息, 详见 “[PEP 487 –Simpler customisation of class creation](#)”。

`type` 类是一个 “元类 (MetaClass)”, 即用于构建类的类。换句话说, `type` 类的实例还是类。Python 标准库中还提供了一些其他元类 (MetaClass), 不过 `type` 是默认的元类 (MetaClass)。

```
1 >>> type(7)  
2 <class 'int'>  
3  
4 >>> type(int)  
5 <class 'type'>  
6  
7 >>> type( OSError)  
8 <class 'type'>  
9  
10 >>> class Whatever:  
11     ...  
12     pass  
13  
14 >>> type(Whatever)  
15 <class 'type'>
```

我们将在 “[24.8 元类入门](#)” 中构建自定义元类 (MetaClass)。

接下来,将用内置的 `type` 定义一个用于构建类的函数。

24.4 类工厂函数

Python 标准库中有一个类工厂函数在本书中多次出现——`collections.namedtuple`。“[五 数据类构建器](#)”中,还介绍过 `@dataclasses.dataclass` 装饰器和 `typing.NamedTuple`。这些类构建器都用到了本章所涉及的技术。

先从一个超级简单的类工厂函数开始,该工厂函数生成的类可用于构建可变对象。该类工厂函数可能是 `@dataclass` 的最简单替代方案。

假设我正在编写一个宠物店应用程序,并且想用简单的记录去存储狗的数据。但我不想写如下这样的样板代码 (boilerplate):

```
1 class Dog:
2     def __init__(self, name, weight, owner):
3         self.name = name
4         self.weight = weight
5         self.owner = owner
```

太繁琐了...每个字段都要出现 3 次。而且这个样板代码甚至无法得到友好的 `repr` (字符串表示形式):

```
1 >>> rex = Dog('Rex', 30, 'Bob')
2 >>> rex
3 <__main__.Dog object at 0x2865bac>
```

参考 `collections.namedtuple`,下面创建一个 `record_factory` 函数,可用于即时构建简单的类(如 `Dog`)。此函数用法如 [示例 24.1](#) 所示:

</> [示例 24.1: 测试 record_factory,一个简单的类工厂函数](#)

```
1 >>> Dog = record_factory('Dog', 'name weight owner') ❶
2 >>> rex = Dog('Rex', 30, 'Bob') ❷
3 >>> rex ❸
4 Dog(name='Rex', weight=30, owner='Bob')
5 >>> name, weight, _ = rex ❹
6 >>> name, weight
7 ('Rex', 30)
8 >>> "{2}'s dog weighs {1}kg".format(*rex) ❺
9 "Bob's dog weighs 30kg"
10 >>> rex.weight = 32 ❻
11 >>> rex
12 Dog(name='Rex', weight=32, owner='Bob')
13 >>> Dog.__mro__ ❼
14 (<class 'factories.Dog'>, <class 'object'>)
```

❶ 工厂函数可以像 `collections.namedtuple` 那样调用:先写类名,后跟一个字符串,列出属性名(多个属性以空格分隔)。

❷ 友好的 `repr` (字符串表示形式)

- ③ 实例是可迭代对象,因此在赋值时可以方便地拆包 ...
- ④ ... 传给函数(如format)时,也可以拆包。
- ⑤ 记录(record)实例是可变对象。
- ⑥ 新建的类继承自object,与我们的工厂函数没有关系。

类工厂函数record_factory的代码如示例24.2所示:²

```
</>示例24.2: record_factory:一个简单的类工厂函数
1  from typing import Union, Any
2  from collections.abc import Iterable, Iterator
3
4  FieldNames = Union[str, Iterable[str]]          ❶
5
6  def record_factory(cls_name: str, field_names: FieldNames) -> type[tuple]: ❷
7
8      slots = parse_identifiers(field_names)          ❸
9
10     def __init__(self, *args, **kwargs) -> None: ❹
11         attrs = dict(zip(self.__slots__, args))
12         attrs.update(kwargs)
13         for name, value in attrs.items():
14             setattr(self, name, value)
15
16     def __iter__(self) -> Iterator[Any]: ❺
17         for name in self.__slots__:
18             yield getattr(self, name)
19
20     def __repr__(self): ❻
21         values = ', '.join(f'{name}={value!r}'
22                             for name, value in zip(self.__slots__, self))
23         cls_name = self.__class__.__name__
24         return f'{cls_name}({values})'
25
26     cls_attrs = dict( ❻
27         __slots__=slots,
28         __init__=__init__,
29         __iter__=__iter__,
30         __repr__=__repr__,
31     )
32
33     return type(cls_name, (object,), cls_attrs) ❻
34
35
36     def parse_identifiers(names: FieldNames) -> tuple[str, ...]:
37         if isinstance(names, str):
38             names = names.replace(',', ' ').split()
39         if not all(s.isidentifier() for s in names):
```

²感谢我的朋友J.S.O.Bueno为我提供了这个示例。

```

40     raise ValueError('names must all be valid identifiers')
41     return tuple(names)

```

- ❶ 用户可以用单个字符串或可产出字符串的可迭代对象提供字段名称。
- ❷ 接受的参数与 `collections.namedtuple` 的前 2 个参数类似, 返回一个 `type`, 即一个行为与元组类似的类。
- ❸ 构建一个包含属性名的元组, 这将成为新建类的 `__slots__` 属性的值。
- ❹ 此函数将成为新建类的 `__init__` 方法, 可接受位置参数和关键字参数。³
- ❺ 按 `__slots__` 设定的顺序产出字段值。
- ❻ 遍历 `__slots__` 与 `self`, 生成友好的 `repr` (字符串表示形式)。
- ❼ 组建类属性 (Class Attribute) 字典。
- ❽ 调用 `type` 构造函数, 构建并返回新类。
- ❾ 将以空格或逗号分隔的 `names` 转换为字符串列表。

示例 24.2 是我们首次在 [类型提示 \(Type Hints\)](#) 中见到 `type`。若❷处的注解只用 “`-> type`”, 则表示 `record_factory` 会返回一个类——这也没错。但使用注解 “`-> type[tuple]`” 更准确, 表示返回的类将是 `tuple` 的子类。

示例 24.2 中 `record_factory` 的最后一行 (❸处) 构建了一个类, 类名为 `cls_name` 的值。此类的直接基类是 `object`, 并且在命名空间中加载了 `__slots__`、`__init__`、`__iter__` 与 `__repr__`, 其中最后 3 个是实例方法。

我们本可以将类属性 (Class Attribute) `__slots__` 命名为其他名称, 但那样的话就必须实现 `__setattr__` 方法来验证被赋值属性的名称。因为对于这种记录类, 我们希望属性始终是固定的那些, 且顺序固定。然而, 如[“11.11 用 `__slots__` 节省内存<311页>”](#) 所述, 类属性 `__slots__` 的主要特点是在处理数百万实例时可节省内存, 而使用 `__slots__` 也有一些缺点。



由 `record_factory` 函数创建的类的实例是不可序列化的, 也就是说, 它们不能用 `pickle` 模块中的 `dump` 函数导出。解决这个问题超出了本示例的范围, 本示例旨在展示如何用 `type` 类满足简单的需求。要获得完整的解决方案, 请研究 `collections.namedtuple` 的源码, 并搜索关键词 “`pickling`”。

接下来, 介绍如何模拟更现代的类构建器 (如 `typing.NamedTuple`)。它接受一个用户定义的类, 这个类是用 `class` 语句编写的, 并自动增强类的功能。

24.5 介绍 `__init_subclass__`

`__init_subclass__` 与 `__set_name__` 都是在[“PEP 487 - Simpler customisation of class creation”](#) 中提出的。[“23.2.2 LineItem 类第 4 版: 存储 \(Storage\) 属性的自动命名<703页>”](#) 中首次介绍过描述符的特殊方法 `__set_name__`。本节将介绍特殊方法 `__init_subclass__`。

如[“五 数据类构建器”](#) 所述, 借助 `typing.NamedTuple` 与 `@dataclasses.dataclass`, 程序员可以用 `class` 语句为新类指定属性。然后, 类构建器会自动为新类添加 `__init__`、`__repr__`、`__eq__` 等方法, 以增强新类的功能。

³我没有为参数添加[类型提示 \(Type Hints\)](#), 因为实际类型是 `Any`。我为返回值类型添加了[类型提示 \(Type Hints\)](#), 否则 `Mypy` 不会在方法内部进行检查。

能。

这两个类构建器都能读取用户在 class 语句中添加的 [类型提示 \(Type Hints\)](#)，以增强类的功能。静态类型检查工具还可以通过这些 [类型提示 \(Type Hints\)](#)，验证用于设置属性或读取属性的代码。但是，在程序运行时，`typing.NamedTuple` 与 `@dataclasses.dataclass` 无法利用 [类型提示 \(Type Hints\)](#) 进行属性验证（或检查）。但“[示例 24.5<727页>](#)”中的 Checked 类则可以执行这种运行时验证。



运行时类型检查无法涵盖所有静态类型提示。或许正因如此，`typing.NamedTuple` 与 `@dataclasses.dataclass` 都没有做运行时检查。不过，有些类型也是具体类，可以与 Checked 一起使用。这包括经常用于字段内容的简单类型，如 `str`、`int`、`float` 和 `bool`，以及这些类型构成的列表。

[示例 24.3](#)展示了如何用 Checked 类构建一个 Movie 类。

</> [示例 24.3: initsub/checkedlib.py](#):

```
1  >>> class Movie(Checked): ❶
2  ...     title: str ❷
3  ...     year: int
4  ...     box_office: float
5  ...
6  >>> movie = Movie(title='The Godfather', year=1972, box_office=137) ❸
7  >>> movie.title
8  'The Godfather'
9  >>> movie ❹
10 Movie(title='The Godfather', year=1972, box_office=137.0)
```

- ❶ Movie 继承自 Checked 类，将在“[示例 24.5<727页>](#)”中定义 Checked 类。
- ❷ 用构造函数注解各个属性。这里用的是内置类型。
- ❸ 必须用关键字参数创建 Movie 实例。
- ❹ 字符串表示形式十分友好。

属性的 [类型提示 \(Type Hints\)](#) 所使用的构造函数可以是任何可调用对象，这些可调用对象接受 0 个或 1 个参数，并返回一个符合预定字段类型的值或引发 `TypeError` 或 `ValueError` 拒绝传入的参数。

在 [示例 24.3](#) 中，使用内置类型进行 [类型提示 \(Type Hints\)](#)，意味着这些属性值必须可被这些类型的构造函数接受。例如，对于 `int` 类型，无论传入的 `x` 是什么类型，`int(x)` 都必须返回一个 `int`。而对于 `str` 类型，运行时可接受任何类型的值。因为在 Python 中，`str(x)` 可以处理任何类型的 `x` 值。⁴

若调用构造函数时未传入参数，则构造函数应返回相应类型的默认值。⁵

Python 内置类型构造函数的标准行为如下所示：

```
1  >>> int(), float(), bool(), str(), list(), dict(), set()
2  (0, 0.0, False, '', [], {}, set())
```

⁴ 的确如此，除非 `x` 所属的类重写了从 `object` 类继承的 `__str__` 或 `__repr__` 方法，并且实现方式有误。

⁵ 该解决方案避免了将 `None` 用作默认值。避免使用空值是个好主意。在一般情况下，很难做到这一点，但在某些情况下却很容易。在 Python 和 SQL 中，我更喜欢在文本字段中用空字符串来表示缺失的数据，而不是 `None` 或 `NULL`。Go 强制施行了这个做法：在 Go 中，主类型的变量和结构字段默认用“零值 (zero value)”初始化。如果感到好奇，请参阅《[A Tour of Go](#) 在线教程》中的“[Zero values](#)”。

在 Movie 这样的 Checked 子类中, 在创建实例时, 如果缺少参数, 则对应的参数值将使用字段构造函数返回的默认值。例如:

```
1 >>> Movie(title='Life of Brian')
2 Movie(title='Life of Brian', year=0, box_office=0.0)
```

在实例化过程中, 可以使用构造函数进行属性验证; 在为实例直接设定属性时, 也可以使用构造函数进行属性验证:

```
1 >>> blockbuster = Movie(title='Avatar', year=2009, box_office='billions')
2 Traceback (most recent call last):
3 ...
4 TypeError: 'billions' is not compatible with box_office:float
5 >>> movie.year = 'MCMLXXII'
6 Traceback (most recent call last):
7 ...
8 TypeError: 'MCMLXXII' is not compatible with year:int
```



Checked 的子类与静态类型检查

对于 Movie 类 (如 [示例 24.3](#) 所定义) 的实例 movie, Mypy 在做静态类型检查时, 将会报告如下赋值语句有类型错误:

```
1 movie.year = 'MCMLXXII'
```

但是, Mypy 无法检测该构造函数调用中的类型错误:

```
1 blockbuster = Movie(title='Avatar', year='MMIX')
```

这是因为 Movie 继承了 Checked.__init__, 而为了可构建任意的用户定义类, 该 __init__ 方法的签名必须可接受任意类型的关键字参数。

此外, 若在 Checked 子类中用类型提示 list[float] 声明一个字段, 那么当为此字段赋予不兼容的值时, Mypy 会报错。但是 Checked 会忽略 list[float] 中的参数 float, 从而将 list[float] 视作 list。

下面看一下 `initsub/checkedlib.py` 的实现。第 1 个类是 Field 描述符, 如 [示例 24.4](#) 所示。

```
</> 示例 24.4: initsub/checkedlib.py:
1 from collections.abc import Callable
2 from typing import Any, NoReturn, get_type_hints
3
4 class Field:
5     def __init__(self, name: str, constructor: Callable) -> None: ❶
6         if not callable(constructor) or constructor is type(None): ❷
7             raise TypeError(f'{name!r} type hint must be callable')
8         self.name = name
9         self.constructor = constructor
10
11     def __set__(self, instance: Any, value: Any) -> None:
```

```

12     if value is ...:
13         value = self.constructor()           ④
14     else:
15         try:
16             value = self.constructor(value)  ⑤
17         except (TypeError, ValueError) as e:
18             type_name = self.constructor.__name__
19             msg = f'{value!r} is not compatible with {self.name}:{type_name}'
20             raise TypeError(msg) from e
21         instance.__dict__[self.name] = value  ⑦

```

- ❶ 注意,从Python 3.9开始,类型提示(Type Hints)中使用的 Callable 类型是collections.abc 模块中的抽象基类(ABCs),而不是已弃用的 typing.Callable。
- ❷ 这是一个极简的 Callable 类型提示(Type Hints),constructor 的形参(Parameter)类型与返回值类型都隐式为 Any。
- ❸ 使用内置函数 callable⁶执行运行时检查。针对 typeNone 的测试不可省略,因为 type(None) 的结果为 NoneType(即 None 所属的类),也是可调用的,但 NoneType 的构造函数仅会返回没用的 None。
- ❹ 如果 Checked.__init__ 将 value 设置为 ... (即 Ellipsis 内置对象),则调用无参数的 constructor()。
- ❺ 否则,调用 constructor() 时,传入参数 value,即 constructor(value)。
- ❻ 如果 constructor(value) 引发 TypeError 或 ValueError,则在 except 块中抛出 TypeError,并给出包含字段名与构造函数名的帮助信息。例如,“MMIX is not compatible with year:int”。
- ❼ 如果未引发异常,则将字段名 name 与字段值 value 保存在 instance.__dict__。

在 __set__ 方法中,需要捕获 TypeError 与 ValueError。因为根据传入的参数不同,内置构造函数有可能引发二者中的一个。例如, float(None) 会引发 TypeError,而 float('A') 会引发 ValueError。另外, float('8') 不会引发异常,而是返回 8.0——在此声明,对这个简单示例来说,这个行为是可以接受的,不是 BUG。



如“23.2.2 LineItem 类第4版:存储(Storage)属性的自动命名<703页>”所述,描述符(Descriptor)类中有一个方便的特殊方法 __set_name__。Field 类中用不到该方法,因为描述符不是在客户端源码中实例化的。用户声明的类型是构造函数,如 Movie 类(“示例 24.3<725页>”中所示。相反,Field 描述符实例是在运行时由 Checked.__init_subclass__ 方法创建的,如示例 24.5 所示。

现在,分析一下 Checked 类,该类的源码被划分为 2 部分展示。示例 24.5 是 Checked 类的前半部分,包含最重要的几个方法。余下部分见“示例 24.6<729页>”。

</> 示例 24.5: initsub/checklib.py:Checked 类最重要的方法

```

1  class Checked:
2      @classmethod
3      def _fields(cls) -> dict[str, type]:  ①
4          return get_type_hints(cls)
5
6      def __init_subclass__(subclass) -> None:  ②
7          super().__init_subclass__()           ③

```

⁶我认为 callable 应可用作类型提示(Type Hints)。但截止 2021 年 5 月 6 日,相应的 issue#42102 还处于待解决状态。

```

8     for name, constructor in subclass._fields().items(): ④
9         setattr(subclass, name, Field(name, constructor)) ⑤
10
11    def __init__(self, **kwargs: Any) -> None:
12        for name in self._fields():
13            value = kwargs.pop(name, ...) ⑥
14            setattr(self, name, value) ⑦
15        if kwargs:
16            self.__flag_unknown_attrs(*kwargs) ⑧

```

- ❶ 编写这个类方法 (Class Method) 是为了在类的余下部分隐藏对 `typing.get_type_hints` 的调用。若仅需支持 Python ≥ 3.10 , 则可用 `inspect.get_annotations` 取代 `typing.get_type_hints`。关于二者的说明, 详见 “15.5.1 运行时的注解问题<432页>”。
- ❷ 在定义当前类的子类时, 将调用 `__init_subclass__` 方法。该方法的第 1 个参数是新定义的子类, 因此我将参数命名为 `subclass`, 而不是通常的 `cls`。有关参数命名的详细说明, 见后方标记栏 “`__init_subclass__` 不是常规的类方法”。
- ❸ 严格来说, `super().__init_subclass__()` 调用不是必需的。但为了与同一继承图 (inheritance graph) 中其他可能实现了 `__init_subclass__` 方法的类 “和谐共处”, 此处应该调用 `__init_subclass__`。详见 “14.4 多重继承与方法解析顺序<398页>”。
- ❹ 遍历每个字段名和构造函数 ...
- ❺ ... 在子类 `subclass` 中创建一个名为 `name` 的属性, 并将该属性绑定到用 `name` 与 `constructor` 参数化的 `Field` 描述符上。
- ❻ 遍历当前类字段中的各个 `name`...
- ❼ ... 从 `kwargs` 中获取对应的 `value`, 并将该 `value` 从 `kwargs` 中移除。用 ... (Ellipsis 对象) 作为默认值, 可以将赋值为 `None` 的参数与未赋值的参数区分开来。⁷
- ❽ 此 `setattr` 调用, 将触发 `Checked.__setattr__` (见示例 24.6) 方法。
- ❾ 如果 `kwargs` 中还有剩余项, 则说明存在与声明的字段不匹配的名称, `__init__` 将执行失败。
- ❿ 错误由 `__flag_unknown_attrs` (见示例 24.6) 报告。该方法接受的参数 `*names` 是那些未知的属性名称。`*kwargs` 中星号 (*) 的作用是, 将 `kwargs` 的键作为序列传递给方法。

`__init_subclass__` 不是常规的类方法

永远无需为特殊方法 `__init_subclass__` 应用 `@classmethod` 装饰器, 这没什么可意外的。因为, 即使不应用 `@classmethod` 装饰器, 特殊方法 `__new__` 的行为也与类方法 (Class Method) 一样。Python 传给 `__init_subclass__` 方法的第 1 个参数是一个类, 该类实现了 `__init_subclass__` 方法的类新定义的子类。这与 `__new__` 以及我所知道的其他所有类方法 (Class Method) 都不同。因此, 我认为 `__init_subclass__` 不是一般意义上的类方法 (Class Method), 所以将第 1 个参数命名为 `cls` 是一种误导。`__init_subclass__` 文档将第 1 个参数命名为 `cls`, 但解释说 “... 每当包含此方法的类被子类化时, 此方法都会被调用, `cls` 表示新的子类。”

⁷ 如 “19.6.3 多核素数检测器代码<579页>” 中的标记栏 “循环、哨兵值 (Sentinel Value) 和毒丸 (poison pill)” 所述, Ellipsis 对象是一种既省事又安全的哨兵值 (Sentinel Value)。Ellipsis 对象由来已久, 但最近人们又发现了它的更多用途, 例如类型提示与 NumPy 中经常用到。

下面接续示例 24.5，继续分析 Checked 类的余下方法。注意，方法 `_fields` 与 `_asdict` 在名称中增加了单下划线（`_`）前缀，这么做的原因与 `collections.namedtuple` API 一样——即减少与用户定义的字段名发生冲突的概率。

</> 示例 24.6: `initsub/checkedlib.py`: Checked 类余下的方法

```

1  def __setattr__(self, name: str, value: Any) -> None: ❶
2      if name in self._fields(): ❷
3          cls = self.__class__
4          descriptor = getattr(cls, name)
5          descriptor.__set__(self, value) ❸
6      else: ❹
7          self.__flag_unknown_attrs(name)
8
9      def __flag_unknown_attrs(self, *names: str) -> NoReturn: ❺
10         plural = 's' if len(names) > 1 else ''
11         extra = ', '.join(f'{name}!r}' for name in names)
12         cls_name = repr(self.__class__.__name__)
13         raise AttributeError(f'{cls_name} object has no attribute{plural} {extra}')
14
15     def _asdict(self) -> dict[str, Any]: ❻
16         return {
17             name: getattr(self, name)
18             for name, attr in self.__class__.__dict__.items()
19             if isinstance(attr, Field)
20         }
21
22     def __repr__(self) -> str: ❼
23         kwargs = ', '.join(
24             f'{key}={value}!r}' for key, value in self._asdict().items()
25         )
26         return f'{self.__class__.__name__}({kwargs})'

```

- ❶ 拦截一切设置实例属性的操作。这是防止设置未知属性所必需的。
- ❷ 如果属性名称 `name` 是已知的，则获取相应的描述符。
- ❸ 通常无须显式调用描述符的 `__set__` 方法。但在本例中是必须的，因为即便存在 `Field` 这样的覆盖型描述符（*Overriding Descriptor*），`__setattr__` 也会拦截一切设置实例属性的操作（如❶所述）。⁸
- ❹ 否则，如果属性名称 `name` 是未知的，则由 `__flag_unknown_attrs` 方法抛出异常。
- ❺ 构建一条有用的消息，列出所有不接受的（`unexpected`）的实参（`Argument`），并引发 `AttributeError` 异常。此处用到了不常见的特殊类型 `NoReturn`，详见“8.5.12 NoReturn<240页>”。
- ❻ 根据 `Movie` 对象的属性创建一个字典。我原本想将此方法命名为 `_as_dict`，但是按照 `collections.namedtuple` 的命名约定，最终采用了 `_asdict` 方法名。
- ❼ 在此示例中，定义特殊方法 `_asdict` 的主要目的，是为了实现一个友好的 `__repr__` 方法，即友好的字符串表示形式。

⁸ 覆盖型描述符（*Overriding Descriptor*）的具体概念，见“23.3.1 覆盖型描述符<708页>”。

Checked示例演示了,当实现阻碍实例化后随意设置实例属性的`__setattr__`方法时,如何处理覆盖型描述符(Overriding Descriptor)。对于此示例,是否值得实现`__setattr__`还有待商榷。如果没有`__setattr__`方法,则`movie.director = 'Greta Gerwig'`可以执行成功,但是属性`director`不会以任何方式被检查,并且不会包含在`_asdict`返回的字典中,也不会出现在`__repr__`中。方法`_asdict`与`__repr__`的定义,如示例24.6所示。

在`record_factory`(见“示例24.2<723页>”)中,我用类属性(Class Attribute)`__slots__`解决了这个问题。但是,这种简单的解决方案并不适用本示例,其原因详见下一小节。

24.5.1 为何`__init_subclass__`无法配置`__slots__`

类属性(Class Attribute)`__slots__`仅在作为类命名空间中的一个条目,传给`__new__`时才有效。而为现有类添加类属性(Class Attribute)`__slots__`并无任何效果。Python仅在类被构建之后才会调用`__init_subclass__`——那时再配置类属性`__slots__`为时已晚。类装饰器也不能配置类属性`__slots__`,因为它的应用甚至比`__init_subclass__`还晚。此处涉及的时机问题,将在“24.7何时发生:导入时与运行时<733页>”中探讨。

若想在运行时配置类属性`__slots__`,则客户代码必须自己构建类的命名空间,并将此命名空间传给`type.__new__`方法的最后1个参数。为此,可以编写一个类工厂函数(Factory Function)(类似于`record_factory.py`),或采用高端技术,实现元类(MetaClass)。“24.8元类入门<737页>”将介绍如何动态配置类属性`__slots__`。

Python 3.7 实现的“PEP 487”,简化了用`__init_subclass__`创建自定义类的过程。在此之前,类似的功能必须用类装饰器实现,详见下一节。

24.6 用类装饰器增强类的功能

“类装饰器”是一种可调用对象,其行为与函数装饰器类似:以被装饰的类为参数,并且返回一个类来取代被装饰的类。类装饰器通常在通过属性赋值向被装饰类中注入更多方法之后,返回被装饰的类本身。

选择使用类装饰器(而不使用更简单的`__init_subclass__`)的最常见原因,可能是为了避免干扰其他类功能(如继承、元类)。⁹

本节将分析`decorator/checkeddeco.py`模块,它可提供与`initsub/checkedlib.py`相同的服务,不过是用类装饰器实现。先看一下从`checkeddeco.py`的`doctests`中提取的使用示例(如示例24.7所示)。

```
</> 示例 24.7: decorator/checkeddeco.py:用装饰器@checked创建一个Movie类
1  >>> @checked
2  ... class Movie:
3  ...     title: str
4  ...     year: int
5  ...     box_office: float
6  ...
7  >>> movie = Movie(title='The Godfather', year=1972, box_office=137)
8  >>> movie.title
9  'The Godfather'
10 >>> movie
11 Movie(title='The Godfather', year=1972, box_office=137.0)
```

⁹“PEP 557 -Data Classes”在其摘要中,以这个原因解释了为何“数据类”是用类装饰器实现的。

示例 24.7 与 “示例 24.3<725页>” 之间的唯一区别是 Movie 类的声明方式：示例 24.7 使用了 @checked 装饰器，而不是子类化 Checked。除此之外，对外行为是相同的，包括 “示例 24.3<725页>” 之后介绍的类型验证和默认值赋值。

下面分析一下 checkeddeco.py 模块的实现。此模块的 import 语句和 Field 类同 initsub/checkdeco.py 一样，详见 “示例 24.4<726页>”。checkeddeco.py 中没有其他类，只有几个函数。

先前在 `__sub_class__` 方法中实现的逻辑，现在被放在了 `checked` 函数中。类装饰器 `checked` 的实现如示例 24.8 所示。

</> 示例 24.8: checkeddeco.py:类装饰器 checked

```
1 def checked(cls: type) -> type:          ❶
2     for name, constructor in _fields(cls).items(): ❷
3         setattr(cls, name, Field(name, constructor)) ❸
4
5     cls._fields = classmethod(_fields) ❹
6
7     instance_methods = ( ❺
8         __init__,
9         __repr__,
10        __setattr__,
11        __asdict,
12        __flag_unknown_attrs,
13    )
14     for method in instance_methods: ❻
15         setattr(cls, method.__name__, method)
16
17     return cls ❻
```

- ❶ 注意，类是 type 的实例。此处的 **类型提示 (Type Hints)** 充分表明 checked 是一个类装饰器：接受一个类，返回一个类。
- ❷ `_fields` 是在模块后面定义的一个顶层函数（见示例 24.9）。
- ❸ 与 “示例 24.5<727页>” 中 `__sub_class__` 方法的逻辑一样，用 Field 描述符实例取代了 `_field` 方法返回的各个属性。不过，这里还有更多的工作需要处理 …
- ❹ … 将 `_fields` 函数构建为类方法，并将其添加到被装饰的类中。此处使用注解 “`type: ignore`”，是为了避免 Mypy 报错 “The type has no: `_fields`”。
- ❺ 这些模块级函数，将成为被装饰类的实例方法。
- ❻ 将 `instance_methods` 中的各个函数都添加到类 `cls` 中。
- ❼ 返回被装饰后的类 `cls`，履行类装饰器的基本契约。

在 checkeddeco.py 模块中，除了装饰器 `checked` 之外，所有顶层函数都以单下划线（`_`）为名称前缀。采用此种命名约定的原因如下：

- `checked` 是 checkeddeco.py 模块公开接口的一部分，但其他函数并不是。
- 示例 24.9 中的函数将被注入到被装饰的类中，`_` 前缀可减少与用户在被装饰类中定义的属性名或方法名发生冲突的概率。

`checkeddeco.py` 模块的余下代码如示例 24.9 所示。这些模块级函数的代码与 `initsub/checkelib.py` 中 `Checked` 类的相应方法相同, 具体说明详见“示例 24.5<727页>”与“示例 24.6<729页>”。

注意, `_field` 函数在 `checkeddeco.py` 模块中身兼两职: 其一, 在装饰器 `checked` 的第 1 行中被用作常规函数; 其二, 作为类方法, 被注入到被装饰的类中。

```

</> 示例 24.9: checkeddeco.py

1 def _fields(cls: type) -> dict[str, type]:
2     return get_type_hints(cls)
3
4 def __init__(self: Any, **kwargs: Any) -> None:
5     for name in self._fields():
6         value = kwargs.pop(name, ...)
7         setattr(self, name, value)
8
9     if kwargs:
10        self._flag_unknown_attrs(*kwargs)
11
12 def __setattr__(self: Any, name: str, value: Any) -> None:
13     if name in self._fields():
14         cls = self.__class__
15         descriptor = getattr(cls, name)
16         descriptor.__set__(self, value)
17     else:
18         self._flag_unknown_attrs(name)
19
20 def _flag_unknown_attrs(self: Any, *names: str) -> NoReturn:
21     plural = 's' if len(names) > 1 else ''
22     extra = ', '.join(f'{name}!' for name in names)
23     cls_name = repr(self.__class__.__name__)
24     raise AttributeError(f'{cls_name} has no attribute{plural} {extra}')
25
26 def _asdict(self: Any) -> dict[str, Any]:
27     return {
28         name: getattr(self, name)
29         for name, attr in self.__class__.__dict__.items()
30         if isinstance(attr, Field)
31     }
32
33 def __repr__(self: Any) -> str:
34     kwargs = ', '.join(
35         f'{key}={value}!' for key, value in self._asdict().items()
36     )
37     return f'{self.__class__.__name__}({kwargs})'

```

`checkeddeco.py` 模块实现了一个简单且用处不小的类装饰器 `checked`。Python 的 `@dataclass` 装饰器功能更丰富。它支持许多配置选项, 可为被装饰的类添加更多方法; 当其与被装饰的类中用户定义方法出现冲突时, 还可自行处理冲突或发出警告; 甚至还能遍历 `__mro__`, 以收集在被装饰的类的超类中声明的用户定义的属性。在 Python 3.9 中, `dataclasses` 包的源码 (`Lib/dataclasses.py`) 长达 1200 多行。

对类做元编程时, 必须知晓: Python 解释器在构建类时, 何时求解各个代码块。详见下一节。

24.7 何时发生:导入时与运行时

Python程序员通常要区分“导入时(`import time`)”与“运行时(`runtime`)”，但这两个术语并没有严格定义，而且二者之间还存在“灰色”地带。

在导入时(`import`)，Python解释器将执行如下操作：

- 从上至下，一次性解析.py模块的源码。此时可能会引发`SyntaxError`异常。
- 将源码编译为要执行的字节码。
- 执行已编译模块的顶层代码。

如果本地`__pycache__`文件夹中有可用的最新.pyc文件，则Python解释器会跳过解析与编译的步骤，因为字节码已准备好可供运行。

尽管解析与编译肯定是“导入时”的活动，不过在此时期内还可能发生其他事情。因为Python中的几乎每条语句都是可执行的，也就是说它们可能运行用户代码，并可能更改用户程序的状态。

特别是，`import`语句不仅仅是一个声明¹⁰，当模块首次被导入时，导入语句实际上运行了模块的所有顶层代码。后续再导入同一模块时将使用缓存，唯一需要做的是将导入的对象与用户模块中的名称进行绑定。顶层代码可以做任何事情，包括“运行时”的典型操作（如写入日志、连接数据库）。¹¹因此，“导入时(`import time`)”与“运行时(`runtime`)”之间的界限是模糊的：`import`语句可以触发各种“运行时”行为。反过来，“导入时”也可能发生在“运行时”的深处，因为在任何常规函数中都可使用`import`语句与`__import__()`内置函数。

以上说明相当抽象与微妙，下面通过具体实验来分析说明。

24.7.1 实验1:何时求解

假设有一个脚本`evaltime/evaldemo.py`脚本，用到了`evaltime/builderlib.py`模块中定义的装饰器、描述符、基于`__init_subclass__`的类构建器。这两个模块中有多个`print`调用，用于展示幕后执行的操作。除此之外，没有其他用途。这些实验的迷弟是观察这些`print`调用发生的顺序。



将类装饰器与基于`__init_subclass__`实现的类构建器，同时应用于单个类，可能是“过度设计”或者在“极端情况”下采取的行为。不过在本实验中，这种不寻常的组合可能很有用。因为，这种组合可以展示类装饰器与`__init_subclass__`对一个类执行更改的时间顺序。

首先，看一下`builderlib.py`模块。我将此模块一分为二，分别为示例24.10与示例24.11。

</>示例24.10: `builderlib.py`:模块的前半部分

```
1 print('@ builderlib module start')
2
3 class Builder: ❶
4     print('@ Builder body')
5
6     def __init_subclass__(cls): ❷
```

¹⁰相比之下，Java中的`import`语句只是声明，用于告知编译器需要哪些特定的包。

¹¹我并不是说“在导入模块时，就打开数据库连接”是个好主意。我只是指出可以这样做。

```

7     print(f'@ Builder.__init_subclass__({cls!r})')
8     def inner_0(self):      ❸
9         print(f'@ SuperA.__init_subclass__(inner_0({self!r}))')
10
11    cls.method_a = inner_0
12
13    def __init__(self):
14        super().__init__()
15        print(f'@ Builder.__init__({self!r})')
16
17
18    def deco(cls): ❹
19        print(f'@ deco({cls!r})')
20
21        def inner_1(self):      ❺
22            print(f'@ deco:inner_1({self!r})')
23
24        cls.method_b = inner_1
25        return cls ❻

```

- ❶ 这是一个类构建器,用于实现 ...
- ❷ ...__init_subclass__ 方法。
- ❸ 定义一个函数。下面的赋值语句可将此处定义的函数 inner_0 添加到子类中。
- ❹ 一个类装饰器。
- ❺ 要添加到被装饰的类中的函数。
- ❻ 返回通过参数传入的类。

builderlib.py 模块的后半部分,如示例 24.11 所示。

</> 示例 24.11: builderlib.py: 模块的后半部分

```

1  class Descriptor:      ❶
2      print('@ Descriptor body')
3
4      def __init__(self):      ❷
5          print(f'@ Descriptor.__init__({self!r})')
6
7      def __set_name__(self, owner, name):      ❸
8          args = (self, owner, name)
9          print(f'@ Descriptor.__set_name__{args!r}')
10
11     def __set__(self, instance, value):      ❹
12         args = (self, instance, value)
13         print(f'@ Descriptor.__set__{args!r}')
14
15     def __repr__(self):
16         return '<Descriptor instance>'
17
18

```

```
19 print('@ builderlib module end')
```

- ❶ 一个描述符类,用于演示何时 ...
- ❷ ... 创建描述符实例,以及 ...
- ❸ ... 在构造 owner 类的过程中,何时调用 __set_name__。
- ❹ 与其他方法一样,这个 __set__ 方法除了显示其接收的参数之外,什么都不做。

如果在 Python 控制台中导入 `builderlib.py` 模块,将得到如下输出:

```
1 >>> import builderlib
2 @ builderlib module start
3 @ Builder body
4 @ Descriptor body
5 @ builderlib module end
```

请注意, `builderlib.py` 打印的行以 @ 为前缀。

现在,看一下 `evaldemo.py`,它会触发 `builderlib.py` 中的特殊方法,如示例 24.12 所示。

</> **示例 24.12: evaldemo.py: 用 builderlib.py 模块 做实验的脚本**

```
1 from builderlib import Builder, deco, Descriptor
2
3 print('# evaldemo module start')
4
5 @deco      ❶
6 class Klass(Builder): ❷
7     print('# Klass body')
8
9     attr = Descriptor() ❸
10
11     def __init__(self):
12         super().__init__()
13         print(f'# Klass.__init__({self!r})')
14
15     def __repr__():
16         return '<Klass instance>'
17
18
19     def main(): ❹
20         obj = Klass()
21         obj.method_a()
22         obj.method_b()
23         obj.attr = 999
24
25     if __name__ == '__main__':
26         main()
27
28 print('# evaldemo module end')
```

- ❶ 应用一个装饰器。
- ❷ 子类化 Builder, 将触发 Builder 的 `__init_subclass__` 方法 (见 “[示例 24.10<733页>](#)” 的❷处)。
- ❸ 实例化描述符。
- ❹ 仅当以主程序运行此模块时, 才会调用此函数。

`evaldemo.py` 中 `print` 调用显示的行以 `#` 为前缀。如果再次打开控制台并导入 `evaldemo.py`, 得到的输出结果将如[示例 24.13](#)所示。

</> [示例 24.13:](#) 在控制台中, 用`evaldemo.py`做实验

```

1  >>> import evaldemo
2  @ builderlib module start          ❶
3  @ Builder body
4  @ Descriptor body
5  @ builderlib module end
6  # evaldemo module start
7  # Klass body                      ❷
8  @ Descriptor.__init__(<Descriptor instance>) ❸
9  @ Descriptor.__set_name__(<Descriptor instance>,
10    <class 'evaldemo.Klass'>, 'attr') ❹
11  @ Builder.__init_subclass__(<class 'evaldemo.Klass'>) ❺
12  @ deco(<class 'evaldemo.Klass'>) ❻
13  # evaldemo module end

```

- ❶ 前 4 行是 “`from builderlib import ...`” 的结果。若在上一个实验 ([示例 24.12](#)) 之后, 未关闭控制台, 则此处的这 4 行不会出现, 因为 `builderlib.py` 模块已经被加载了。
- ❷ 这表明 Python 开始读取 `Klass` 的主体。此时, 类对象还不存在。
- ❸ 描述符实例被创建, 并被绑定到命名空间中的 `attr`。Python 将此命名空间传递给 `type.__new__`——这是默认的类对象构造方法。
- ❹ 此时, Python 的内置 `type.__new__` 创建了 `Klass` 类对象, 并在提供了 `__set_name__` 方法的描述符类的各个描述符实例上调用 `__set_name__`, 同时将 `Klass` 类对象传给 `__set_name__` 方法的参数 `owner`。
- ❺ 然后, `type.__new__` 在 `Klass` 的超类上调用 `__init_subclass__` 方法, 并将 `Klass` 传给 `__init_subclass__` 方法的第一个参数 (即 “[示例 24.10<733页>](#)” ❷处的 `cls`)。
- ❻ 当 `type.__new__` 返回类对象时, Python 将应用装饰器。在本示例中, 由装饰器 `deco` 返回的类绑定到了模块命名空间中的 `Klass` 类对象上。

`type.__new__` 的实现是用 C 语言编写的。上述行为的相美文档描述, 详见《[The Python Language Reference](#)》中 “3. Data model” 一节的 3.3.3.6. Creating the class object。

注意, 控制台会话 (见[示例 24.13](#)) 未执行 `evaldemo.py` 模块中的 `main()` 函数 (见[示例 24.12](#)❷处)。如果将 `evaldemo.py` 当作脚本运行, 得到的输出与[示例 24.13](#)一样, 只是末尾会多出几行。这些多出的行是运行 `main()` 的结果, 如[示例 24.14](#)所示。

</> [示例 24.14:](#) 将`evaldemo.py`当做程序来运行

```

1  $ ./evaldemo.py
2  [... 9 lines omitted ...]
3  @ deco(<class '__main__.Klass'>) ❶

```

```

4 @ Builder.__init__(<Klass instance>) ②
5 # Klass.__init__(<Klass instance>)
6 @ SuperA.__init_subclass__:inner_0(<Klass instance>) ③
7 @ deco:inner_1(<Klass instance>) ④
8 @ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999) ⑤
9 # evaldemo module end

```

- ① 输出的前 10 行 (含本行) 与 [示例 24.13](#) 相同。
- ② 由 Klass.__init__ 中的 super().__init__() 触发, 详见 [示例 24.12](#)。
- ③ 由 main() 中的 obj.method_a() 触发。而 method_a 方法由 SuperA.__init_subclass__ 注入, 详见 “[示例 24.10](#)” 的③处。
- ④ 由 main() 中的 obj.method_b() 触发。而 method_b 方法由 deco 注入, 详见 “[示例 24.10](#)” 的⑤处。
- ⑤ 由 main() 中的 obj.attr=999 触发。

类装饰器与实现了 __init_subclass__ 的基类都是强大的工具, 但它们仅适用于背后已用 type.__new__ 构建的类。在极少数情况下, 当你需要调整传递给 type.__new__ 的参数时, 则需要使用 [元类 \(MetaClass\)](#)。本章以及本书的最后一个主题就是 [元类 \(MetaClass\)](#)。

24.8 元类入门

[元类 \(MetaClass\)](#) 是更深层次的魔法, 99% 的用户都不需要关注。如果你知道自己是否需要元类, 那就说明你不需要 (因为真正需要元类的程序员, 无须解释原因, 也肯定知道自己需要它们)。

—Tim Peters, Timsort 算法的发明者和多产的 Python 贡献者^a

^a摘自 comp.lang.python 邮件列表中, 于 2002 年 12 月 23 日针对 “[Acrimony in c.l.p.](#)” 话题的回复。“[前言](#)” 中的引文也摘自此处。TimBot 就是在那天获得了灵感

[元类 \(MetaClass\)](#) 是一个用于制造类的工厂, 不过不是函数 (如 “[示例 24.2](#)” 中的 record_factory), 而是类。换句话说, 元类是一个类, 其实例也是一个类。图 24.1 使用 MGN 表示法¹² 描述了元类: 即元类是用于生产工坊 (Mill) 的工坊。

根据 “Python 对象模型” 可知, 类也是对象, 所以类必定是另外某个类的实例。默认情况下, Python 中的类是 type 的实例。换句话说, type 是大多数内置类和用户定义类的元类:

```

1 >>> str.__class__
2 <class 'type'>
3 >>> from bulkfood_v5 import LineItem
4 >>> LineItem.__class__
5 <class 'type'>
6 >>> type.__class__
7 <class 'type'>

```

¹²关于 MGN (Mills & Gizmos Notation) 表示法的介绍, 详见 “[23.2.1.1 理解描述符的术语](#)” 中的标记栏 “[MGN \(Mills & Gizmos Notation \) 简介](#)”。

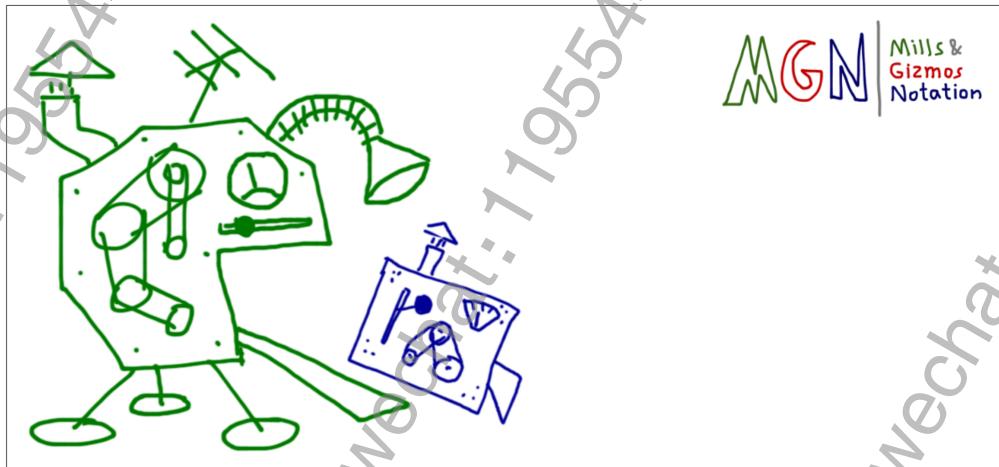


图 24.1: 元类 (MetaClass) 是用于构建类的类

为了避免无限回溯 (regress), type 的类是其自身,如上述代码的最后一行所示。

请注意,我并没有说 str 或 LineItem 是 type 的子类。我的意思是 str 与 LineItem 是 type 的实例,二者都是 object 的子类。图 24.2 可帮助您厘清这个奇怪的事实。

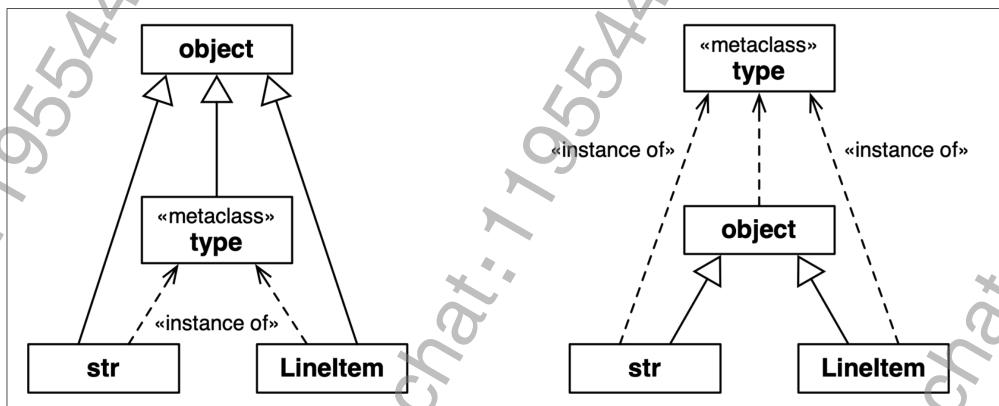


图 24.2: 这 2 个示意图都正确。左图强调 str、type 和 LineItem 是 object 的子类。
右图表明 str、object 和 LineItem 都是 type 的实例,因为它们都是类。



object 类与 type 类之间有一种独特的关系: object 是 type 的实例,而 type 是 object 的子类。这种关系很“神奇”,无法用 Python 代码表述,因为其中一个类必须先存在,然后才能定义另一个类。type 是自身的实例这一事实也很神奇。

如下代码表明 collections.abc.Iterable 所属的类是 abc.ABCMeta。注意,Iterable 是一个抽象类,而 abc.ABCMeta 是一个具体类——然而,Iterable 是 ABCMeta 的实例。

```

1  >>> from collections.abc import Iterable
2  >>> Iterable.__class__
3  <class 'abc.ABCMeta'>
4  >>> import abc
5  >>> from abc import ABCMeta
6  >>> ABCMeta.__class__
7  <class 'type'>

```

向上追溯, `abc.ABCMeta` 最终所属的类也是 `type`。所有类都直接或间接地是 `type` 的实例。但是, 只有元类 (MetaClass) 既是 `type` 的实例, 又是 `type` 的子类。若想理解元类 (MetaClass), 则一定要知道这种关系: 元类 (MetaClass) (如 `ABCMeta`) 从 `type` 类继承了构建类的能力。??说明了这种关系。

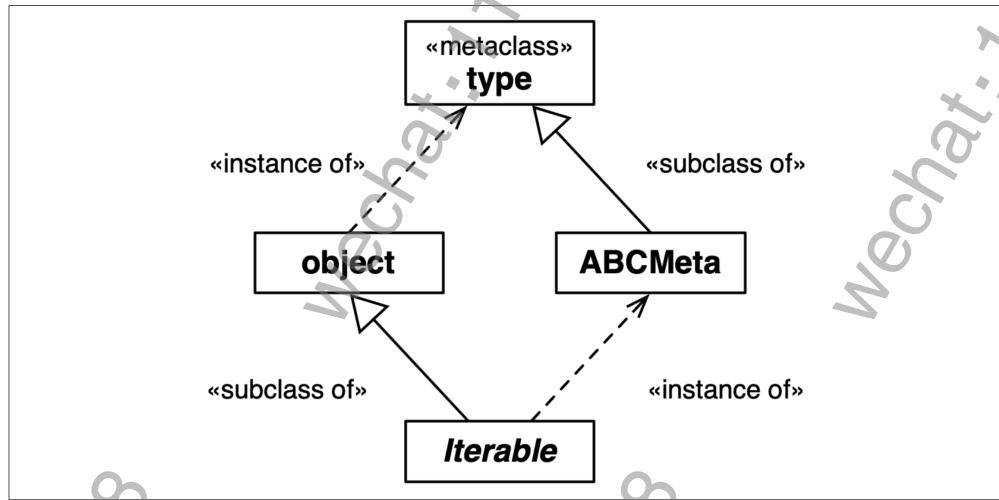


图 24.3: `Iterable` 是 `object` 的子类, 也是 `ABCMeta` 的实例。`object` 和 `ABCMeta` 都是 `type` 的实例, 但这里的关键关系是 `ABCMeta` 也是 `type` 的子类, 因为 `ABCMeta` 是一个元类。在此图中, `Iterable` 是唯一的抽象类。

此处的重点是, 元类 (MetaClass) 是 `type` 的子类, 因此可以用作制造类的工厂。元类 (MetaClass) 可以通过实现特殊方法来定制自己的实例, 详见下一节。

24.8.1 如何用元类定制一个类

若要使用元类 (MetaClass), 务必要理解类上的 `__new__` 方法是如何工作的, 详见“[22.2.3 用 `__new__` 灵活创建对象](#)”。

当元类 (MetaClass) 要创建一个新实例 (即一个类) 时, 同样的机制也会在“元 (meta)”层面上发生。以下的声明为例:

```

1 class Klass(SuperKlass, metaclass=MetaKlass):
2     x = 42
3     def __init__(self, y):
4         self.y = y
  
```

为了处理这个 `class` 语句, Python 将用如下参数调用 `MetaKlass.__new__` 方法:

- `meta_cls`
元类 (MetaClass) 自身 (即 `MetaKlass`), 因为 `__new__` 被当作类方法使用。
- `cls_name`
字符串 `Klass`。
- `bases`
仅含 1 个元素的元组, 即 `(SuperKlass,)`。如果是多重继承, 元组将包含多个元素。
- `cls_dict`
一个映射, 内容类似于 `{x: 42, '__init__': <function __init__ at 0x1009c4040>}`。

在实现 `MetaKlass.__new__` 时, 可以检查并更改这些参数; 然后, 再将这些参数传递给 `super().__new__`; `super().__new__` 最终将调用 `type.__new__` 来创建新的类对象。

`super().__new__` 返回后, 还可以对新创建的类进行进一步处理; 然后, 再返回给 Python。Python 会调用 `SuperKlass.__init_subclass__`, 并传入新创建的类; 然后, 应用类装饰器 (如果有的话)。最后, Python 将类对象绑定到所在命名空间中的名称——如果 `class` 语句是顶层语句, 所在的命名空间通常是模块全局命名空间。

在元类 (MetaClass) 的 `__new__` 方法中, 最常见的处理是添加或替换 `cls_dict` 中的项。`cls_dict` 是一个映射, 表示正在构造的类的命名空间。例如, 在调用 `super().__new__` 之前, 可以通过向 `cls_dict` 添加函数, 来为正在构造的类注入方法。不过, 请注意, 也可以在构建类之后再为类添加方法, 这也是可以用 `__init_subclass__` 或类装饰器为类注入方法的原因。

类属性 `__slots__` 必须在运行 `type.__new__` 之前, 被添加到映射 `cls_dict` 中, 详见 “24.5.1 为何 `__init_subclass__` 无法配置 `__slots__`”¹³。元类的 `__new__` 方法是配置类属性 `__slots__` 的理想位置。下一节将介绍如何配置。

24.8.2 友好的元类示例

本节展示的 `MetaBunch` 元类改编自《Python in a Nutshell, 3rd Ed》(Alex Martelli, Anna Ravenscroft, Steve Holden 合著)一书“第 4 章”的最后一个示例。¹³ 原书示例需要兼顾 Python 2.7 与 Python 3.5, 但我假设您使用 Python ≥ 3.6, 因此对原版代码做了简化。

首先, 看一下 `Bunch` 基类提供的功能:

```

1  >>> class Point(Bunch):
2      ...     x = 0.0
3      ...     y = 0.0
4      ...     color = 'gray'
5      ...
6  >>> Point(x=1.2, y=3, color='green')
7  Point(x=1.2, y=3, color='green')
8  >>> p = Point()
9  >>> p.x, p.y, p.color
10 (0.0, 0.0, 'gray')
11 >>> p
12 Point()

```

还记得吗? `Checked` 类 (见 “示例 24.5”⁷²⁷) 会根据类变量的类型提示 (Type Hints), 为子类中的 Field 描述符指定名称。而这些 Field 描述符实际不会成为类的属性, 因为它们没有值。

而 `Bunch` 子类则不同, 它使用带有值的实际类属性, 这些值将成为实例属性的默认值。生成的 `__repr__` 会省略等于默认值的属性参数。

`MetaBunch` (`Bunch` 的元类) 根据用户类中声明的类属性为新类生成 `__slots__`。如此一来, 实例化与后续赋值都无法使用未声明的属性。

```

1  >>> Point(x=1, y=2, z=3)

```

¹³ 几位作者非常友好地授权我使用他们的示例。`MetaBunch` 类首次出现在 Martelli 于 2002 年 7 月 7 日发布在 comp.lang.python 群组的一则消息中, 标题为 “a nice metaclass example (was Re: structs in python)”。该话题之前讨论的是在 Python 中如何使用记录类型的数据结构。Martelli 在 Python 2.2 中的原始代码在做了一处修改后仍然可以运行: 若要在 Python 3 中使用元类, 必须在类声明中使用 `metaclass` 关键字参数, 如 `Bunch(metaclass=MetaBunch)`。而旧时的约定是添加一个类级属性 `__metaclass__`。

```

2  Traceback (most recent call last):
3    ...AttributeError: No slots left for: 'z'
4  >>> p = Point(x=21)
5  >>> p.y = 42
6  >>> p
7  Point(x=21, y=42)
8  >>> p.flavor = 'banana'
9  Traceback (most recent call last):
10   ...
11 AttributeError: 'Point' object has no attribute 'flavor'

```

现在,让我们深入研究 MetaBunch 的优雅代码,如 [示例 24.15](#) 所示。

</> [示例 24.15: metabunch/from3.6/bunch.py](#):MetaBunch 元类与 Bunch 类

```

1  class MetaBunch(type):      ❶
2  def __new__(meta_cls, cls_name, bases, cls_dict): ❷
3
4      defaults = {}      ❸
5
6      def __init__(self, **kwargs):      ❹
7          for name, default in defaults.items():      ❺
8              setattr(self, name, kwargs.pop(name, default))
9          if kwargs:      ❻
10              extra = ', '.join(kwargs)
11              raise AttributeError(f'No slots left for: {extra!r}')
12
13      def __repr__(self): ❻
14          rep = ', '.join(f'{name}={value!r}'
15                           for name, default in defaults.items()
16                           if (value := getattr(self, name)) != default)
17          return f'{cls_name}({rep})'
18
19      new_dict = dict(__slots__=[], __init__=__init__, __repr__=__repr__) ❻
20
21      for name, value in cls_dict.items():      ❹
22          if name.startswith('__') and name.endswith('__'):
23              if name in new_dict:
24                  raise AttributeError(f"Can't set {name!r} in {cls_name!r}")
25              new_dict[name] = value
26          else:      ❻
27              new_dict['__slots__'].append(name)
28              defaults[name] = value
29      return super().__new__(meta_cls, cls_name, bases, new_dict) ❻
30
31
32  class Bunch(metaclass=MetaBunch):      ❻
33      pass

```

❶ 为了创建元类 (MetaClass),继承 type。

- ② `__new__` 被当作类方法使用,但我们创建的是元类,所以我喜欢将第 1 个参数命名为 `meta_cls` (`mcs` 也是一个常用名称)。其余 3 个参数与直接调用 `type()` 创建类时传入的 3 个参数一样。
- ③ `defaults` 用于保存属性名称到其默认值的映射。
- ④ 此方法将被注入到新类中。
- ⑤ 读取 `defaults`,将相应的实例属性设为从 `kwargs` 中取出 (`pop`) 的值或默认值。
- ⑥ 如果 `kwargs` 中有余项,则意味着没有多余的位置用于存放这些余项。我们应遵从 Fail-Fast (快速失败) 原则,而不是默默忽略这些余项。一种简单而有效的方案是从 `kwargs` 中取出一项,并尝试在实例上设置此余项,故意引发 `AttributeError` 异常。
- ⑦ `__repr__` 返回的字符串 (如 `Point(x=3)`) 看起来像一个构造函数调用,忽略了值为默认值的关键字参数。
- ⑧ 为新类初始化命名空间。
- ⑨ 遍历用户类的命名空间。
- ⑩ 如果 `name` 是带双下划线 (`__`) 的名称,除非它已存在,否则将其赋值到新类的命名空间。这样可以防止用户覆盖 `__init__`、`__repr__` 和其他由 Python 设置的属性,如 `__qualname__` 和 `__module__`。
- ⑪ 如果 `name` 是带双下划线 (`__`) 的名称,则将其追加到 `__slots__` 中,并将其值存入 `defaults`。
- ⑫ 构建并返回新类。
- ⑬ 提供一个基类,这样用户就无需看到 `MetaBunch`。

MetaBunch 之所以能够工作,是因为它能够在调用 `super().__new__` 构建最终类之前配置 `__slots__`。在进行元编程时,理解各个操作的执行顺序至关重要。下面用元类 (`MetaClass`),再做一次求解时间顺序的实验。

24.8.3 实验 2:元类的求解时机

本实验是“24.7.1 实验 1:何时求解<733页>”的变体,为其增加了一个元类。`builderlib.py` 模块与先前的相同,但主脚本更换为 `evaldemo_meta.py`,如示例 24.16 所示。

</> 示例 24.16: `evaldemo_meta.py`:用元类做实验

```

1 #!/usr/bin/env python3
2
3 from builderlib import Builder, deco, Descriptor
4 from metalib import MetaKlass ①
5
6 print('# evaldemo_meta module start')
7
8 @deco
9 class Klass(Builder, metaclass=MetaKlass): ②
10    print('# Klass body')
11
12    attr = Descriptor()
13
14    def __init__(self):
15        super().__init__()
16        print(f'# Klass.__init__({self!r})')
17

```

```

18     def __repr__(self):
19         return '<Klass instance>'
20
21     def main():
22         obj = Klass()
23         obj.method_a()
24         obj.method_b()
25         obj.method_c() ❸
26         obj.attr = 999
27
28
29 if __name__ == '__main__':
30     main()
31
32 print('# evaldemo_meta module end')

```

- ❶ 从 `metalib.py` (见示例 24.18) 导入元类 `MetaKlass`。
- ❷ 将 `Klass` 声明为 `Builder` 的子类和 `MetaKlass` 的实例。
- ❸ 这个方法将由 `MetaKlass.__new__` 方法注入到新建的类中,如示例 24.18所示。



出于实验目的,示例 24.16 打破常规,为 `Klass` 类同时应用了 3 种不同的元编程技术:一个装饰器、一个用 `__init_subclass__` 实现的基类、一个自定义元类。这么做的目的是观察这 3 种技术在类构建过程中的干扰顺序,生产环境中可不能这么做。

与“24.7.1 实验 1:何时求解<733页>”中的实验一样,本示例除了打印揭示执行流程的信息外,什么也不做。示例 24.17 展示了 `metalib.py` 前半部分代码,余下代码如示例 24.18所示。

</> 示例 24.17: `metalib.py`:`NosyDict` 类

```

1  print('% metalib module start')
2
3  import collections
4
5  class NosyDict(collections.UserDict):
6      def __setitem__(self, key, value):
7          args = (self, key, value)
8          print(f'% NosyDict.__setitem__{args!r}')
9          super().__setitem__(key, value)
10
11     def __repr__(self):
12         return '<NosyDict instance>'

```

我编写 `NosyDict` 类是为了覆盖 `__setitem__` 方法,以便在字典中设置项时显示相应的“键 (Key)”与“值 (Value)”。元类 `MetaKlass` 将使用 `NosyDict` 实例来存放待构建类的命名空间,从而进一步揭示 Python 的内部工作原理。

`metalib.py` 的核心内容是示例 24.18 中定义的元类,它实现了特殊方法 `__prepare__`,此方法是 Python 仅在元类上调用的类方法。`__prepare__` 方法为影响创建新类过程提供了最早的时机。



在编写元类 (MetaClass) 时, 我发现对特殊方法的参数采用如下这种命名约定特别有用:

- 在实例方法中使用 `cls` 而不用 `self`, 因为元类的实例是一个类。
- 在类方法中使用 `meta_cls`, 而不用 `cls`, 因为类是一个元类。请注意, 即便未应用 `@classmethod` 装饰器, 特殊方法 `__new__` 的行为也等同类方法 (Class Method)。关于 `__new__` 方法的说明, 详见“22.2.3 用 `__new__` 灵活创建对象<669页>”。

</> 示例 24.18: `metplib.py`:MetaKlass 元类

```

1  class MetaKlass(type):
2      print('% MetaKlass body')
3
4      @classmethod
5      def __prepare__(meta_cls, cls_name, bases): ❶
6          args = (meta_cls, cls_name, bases)
7          print(f'% MetaKlass.__prepare__{args!r}')
8          return NosyDict() ❷
9
10     def __new__(meta_cls, cls_name, bases, cls_dict): ❸
11         args = (meta_cls, cls_name, bases, cls_dict)
12         print(f'% MetaKlass.__new__{args!r}')
13         def inner_2(self):
14             print(f'% MetaKlass.__new__:inner_2({self!r})')
15
16         cls = super().__new__(meta_cls, cls_name, bases, cls_dict.data) ❹
17
18         cls.method_c = inner_2 ❺
19
20     return cls ❻
21
22     def __repr__(cls): ❻
23         cls_name = cls.__name__
24         return f"<class {cls_name!r} built by MetaKlass>"
```

- ❶ `__prepare__` 应声明为类方法 (Class Method)。它不是实例方法, 因为当 Python 调用 `__prepare__` 时, 待构建的类还不存在。
- ❷ Python 在元类上调用 `__prepare__` 方法, 获取一个映射, 用于存放待构建类的命名空间。
- ❸ 返回用作命名空间的 `NosyDict` 实例。关于 `NosyDict` 的定义, 如示例 24.17 所示。
- ❹ `cls_dict` 是由 `__prepare__` 返回的 `NosyDict` 实例。
- ❺ `type.__new__` 方法的最后一个参数必须是真正的字典, 因此为其传入 `NosyDict` 的 `data` 属性。此 `data` 属性是 `NosyDict` 从 `UserDict` 中继承的。
- ❻ 为新创建的类注入一个方法。

- ⑦ 像往常一样, `__new__` 必须返回刚创建的对象——本例中是新创建的类。
 ⑧ 在元类中定义 `__repr__` 方法, 方便定制类对象的字符串表示形式。

在 Python 3.6 之前, `__prepare__` 的主要作用是提供一个 `OrderedDict` 来存放待建类的属性, 这样元类的 `__new__` 方法就可以按照这些属性在用户定义类的源码中出现的顺序来处理它们。现在, 由于 `dict` 可以保留插入顺序, 所以很少需要类方法 `__prepare__`。“[24.11 用 `__prepare__` 实现一些高级功能](#)”一节, 将介绍 `__prepare__` 的一个创意性用途。

在 Python 控制台中导入 `metalib.py`, 得到的结果并无特别之处。注意, 此模块的输出内容将以%为前缀:

```
1  >>> import metalib
2  % metalib module start
3  % MetaKlass body
4  % metalib module end
```

而导入 `evaldemo_meta.py` 会发生很多事情, 如示例 24.19 所示。

</> 示例 24.19: 用 `evaldemo_meta.py` 进行控制台实验

```
1  >>> import evaldemo_meta
2  @ builderlib module start
3  @ Builder body
4  @ Descriptor body
5  @ builderlib module end
6  % metalib module start
7  % MetaKlass body
8  % metalib module end
9  # evaldemo_meta module start
10 % MetaKlass.__prepare__(<class 'metalib.MetaKlass'>, 'Klass', ❶
11     (<class 'builderlib.Builder'>,)) ❷
12 % NosyDict.__setitem__(<NosyDict instance>, '__module__', 'evaldemo_meta') ❸
13 % NosyDict.__setitem__(<NosyDict instance>, '__qualname__', 'Klass')
14 # Klass body
15 @ Descriptor.__init__(<Descriptor instance>) ❹
16 % NosyDict.__setitem__(<NosyDict instance>, 'attr', <Descriptor instance>) ❺
17 % NosyDict.__setitem__(<NosyDict instance>, '__init__',
18     <function Klass.__init__ at >) ❻
19 % NosyDict.__setitem__(<NosyDict instance>, '__repr__',
20     <function Klass.__repr__ at >) ❼
21 % NosyDict.__setitem__(<NosyDict instance>, '__classcell__', <cell at >: empty>)
22 % MetaKlass.__new__(<class 'metalib.MetaKlass'>, 'Klass',
23     (<class 'builderlib.Builder'>,), <NosyDict instance>) ❽
24 @ Descriptor.__set_name__(<Descriptor instance>,
25     <class 'Klass' built by MetaKlass>, 'attr') ❾
26 @ Builder.__init_subclass__(<class 'Klass' built by MetaKlass>)
27 @ deco(<class 'Klass' built by MetaKlass>)
28 # evaldemo_meta module end
```

- ❶ 此行之前的内容, 是导入 `builderlib.py` 和 `metalib.py` 得到的结果。
 ❷ Python 调用 `__prepare__`, 开始处理 `class` 语句。

- ③ 在解析 class 语句主体之前,Python 会将 `__module__` 和 `__qualname__` 条目添加到待构建类的命名空间中。
- ④ 描述符实例已创建 ...
- ⑤ ...将描述符实例绑定到类命名空间中的 `attr`。
- ⑥ 定义了 `__init__` 和 `__repr__` 方法,并将其添加到命名空间中。
- ⑦ 一旦 Python 完成 class 主体的处理,就会调用 `MetaKlass.__new__` 方法。
- ⑧ 在元类的 `__new__` 方法返回新构建的类之后,依次调用 `__set_name__`、`__init_subclass__` 和装饰器。

如果将 `evaldemo_meta.py` 当作脚本运行,则会调用 `main()`,并且会发生更多事情,如示例 24.20 所示。

</> 示例 24.20: 将 `evaldemo_meta.py` 当作脚本运行

```

1 $ ./evaldemo_meta.py
2 [... 20 lines omitted ...]
3 @ deco(<class 'Klass' built by MetaKlass>)      ①
4 @ Builder.__init__(<Klass instance>)
5 # Klass.__init__(<Klass instance>)
6 @ SuperA.__init_subclass__:inner_0(<Klass instance>)
7 @ deco:inner_1(<Klass instance>)
8 % MetaKlass.__new__:inner_2(<Klass instance>)  ②
9 @ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999)
10 # evaldemo_meta module end

```

① 前 21 行(含本行)与示例 24.19 相同。

② 由 `main()` 中的 `obj.method_c()` 触发;`method_c` 方法注入由 `MetaKlass.__new__` 完成。

接下来,回顾一下“24.5 介绍 `__init_subclass__`”中定义的带 `Field` 描述符(见示例 24.5)的 `Checked` 类(见示例 24.4)。其中, `Checked` 类实现了运行时的类型检查。下一节,将介绍如何用元类实现同样的运行时检查功能。

24.9 用元类实现 `Checked` 类

我不鼓励过早优化与过度设计,所以本节将虚构一个场景,此场景适合借助元类,用 `__slots__` 重写 `checkedlib.py`。本节可随意跳过。

一点儿背景

我们用 实现的 `checkedlib.py`,在全公司内取得了成功。并且在生产服务器中,随时都会有数百万个 `Checked` 子类的实例驻留在内存中。

通过论证,我们发现使用 `__slots__` 可以减少云托管费用,原因如下:

- 降低内存使用量,因为各个 `Checked` 实例不需要自己的 `__dict__`。
- 移除 `__setattr__` 后,可提高性能。创建 `__setattr__` 是为了防止预期之外的属性。但是,在实例化时以及在调用 `Field.__set__` 之前的属性设置操作时,都会触发 `__setattr__`。

接下来将要分析的 `metaclass/checkedlib.py` 模块,可以直接取代 `initsub/checkedlib.py`。这两个模块中

的 doctests 以及用于 pytest 的 checkedlib_test.py 文件完全相同。

metaclass/checkedlib.py 已将复杂性从用户层面抽象出来, 用户无需知晓。如下是用到此包的脚本源码:

```

1  from checkedlib import Checked
2
3  class Movie(Checked):
4      title: str
5      year: int
6      box_office: float
7
8  if __name__ == '__main__':
9      movie = Movie(title='The Godfather', year=1972, box_office=137)
10     print(movie)
11     print(movie.title)

```

上述代码中, Movie 类的定义非常简洁, 其背后用到了 Field 验证描述符的 3 个实例、1 个 `__slots__` 配置、从 Checked 继承的 5 个方法, 以及 1 个将这些内容组合在一起的元类。checkedlib 对外唯一可见的部分是 Checked 基类。

在图 24.4 中, MGN 表示法为 UML 类图提供了补充, 使类与实例之间的关系更加清晰明显。

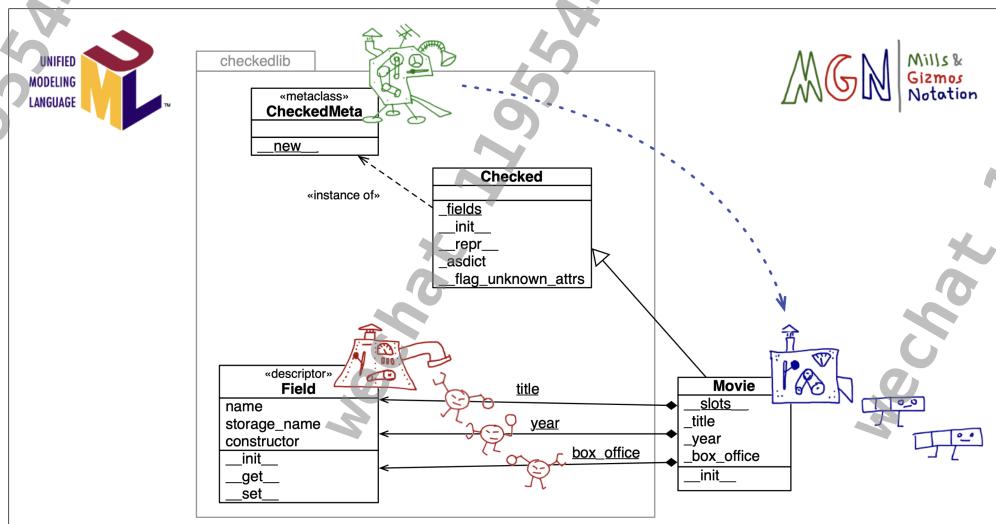


图 24.4: 用 MGN 注解的 UML 类图。CheckedMeta 元工坊构建了 Movie 工坊。Field 工坊构建了 title、year 与 box_office 描述符, 它们是 Movie 的类属性。各个实例的字段数据存储在 Movie 的实例属性 `_title`、`_year` 与 `_box_office` 中。注意 checkedlib 包的边界。Movie 的开发者无需了解 checkedlib.py 中的所有机制。

例如, 使用新版 metaclass/checkedlib.py 的 Movie 类既是 CheckedMeta 类的实例, 也是 Checked 类的子类。此外, Movie 的类属性 title、year 和 box_office 是 3 个独立的 Field 实例。每个 Movie 实例都有自己的 `_title`、`_year` 和 `_box_office` 属性, 用于存放相应字段的值。

接下来, 从新版 metaclass/checkedlib.py 中的 Field 类开始分析, 其源码如示例 24.21 所示。其中的数字标号标注了新版源码中的改动之处。

</> 示例 24.21: metaclass/checkedlib.py: 含有 `storage_name` 属性与 `__get__` 方法的 Field 描述符

```

1  class Field:
2      def __init__(self, name: str, constructor: Callable) -> None:

```

```

3     if not callable(constructor) or constructor is type(None):
4         raise TypeError(f'{name!r} type hint must be callable') self.name
5     = name
6     self.storage_name = '_' + name ❶
7     self.constructor = constructor
8
8     def __get__(self, instance, owner=None):
9         if instance is None:
10             return self
11         return getattr(instance, self.storage_name) ❷
12
13     def __set__(self, instance: Any, value: Any) -> None:
14         if value is ...:
15             value = self.constructor()
16         else:
17             try:
18                 value = self.constructor(value)
19             except (TypeError, ValueError) as e:
20                 type_name = self.constructor.__name__
21                 msg = f'{value!r} is not compatible with {self.name}:{type_name}'
22                 raise TypeError(msg) from e
23             setattr(instance, self.storage_name, value) ❸

```

- ❶ 根据 name 参数计算 storage_name。
- ❷ 如果 __get__ 得到的参数 instance 的值是 None, 说明描述符是从托管 (Managed) 类自身中读取的, 而不是托管 (Managed) 实例。因此我们返回描述符。
- ❸ 否则, 返回存储在名为 storage_name 的属性中的值。
- ❹ __set__ 现在使用 setattr 来设置或更新托管 (Managed) 属性。

Field 描述符类现在有些变化。在先前的“[示例 24.4<726页>](#)”中,各个 Field 描述符实例都将字段值存储在托管 (Managed) 实例的同名属性中。例如,在 Movie 类中, title 描述符将字段值存储在托管 (Managed) 实例的 title 属性中。因此,Field 没必要提供 __get__ 方法。

但是, 使用 __slots__ 之后, 像 Movie 这样的类, 其类属性 (Class Attribute) 与实例属性不能同名。每个描述符实例都是一个类属性 (Class Attribute), 所以现在需要各个实例单独存储属性。代码中选择在描述符名称前添加一个 _ 为前缀。因此, Field 实例有单独的 name 属性和 storage_name 属性, 同时实现了 Field.__get__。

[示例 24.22展示了驱动示例 24.21 的元类的源码。](#)

```

</> 示例 24.22: metaclass/checkedlib.py:CheckedMeta 元类
1 class CheckedMeta(type):
2
3     def __new__(meta_cls, cls_name, bases, cls_dict): ❶
4         if '__slots__' not in cls_dict: ❷
5             slots = []
6             type_hints = cls_dict.get('__annotations__', {})
7             for name, constructor in type_hints.items(): ❸
8                 field = Field(name, constructor) ❹

```

```
9     cls_dict[name] = field
10    slots.append(field.storage_name)      ⑥
11
12    cls_dict['__slots__'] = slots         ⑦
13
14    return super().__new__(               ⑧
15        meta_cls, cls_name, bases, cls_dict) ⑨
```

- ① `__new__` 是 `CheckedMeta` 中唯一实现的方法。
- ② 仅当类的 `cls_dict` 中不含 `__slots__` 时, 才增强类的功能。若 `__slots__` 已存在, 则假定它是 `Checked` 基类, 而不是用户定义类的子类, 并按原样构建该类。
- ③ 在先前的“[示例 24.9<732页>](#)”中, 用 `typing.get_type_hints` 获取类型提示 (Type Hints), 但该函数的第 1 个参数必须是个已存在的类。而此时, 要配置的待构建类还不存在, 所以需要直接从 `cls_dict` 中检索 `__annotations__`。`cls_dict` 是待构建类的命名空间, Python 将其传给元类 `__new__` 方法的最后一个参数。
- ④ 遍历 `type_hints`...
- ⑤ ... 以便为每个带注解的属性, 构建一个 `Field` 实例 ...
- ⑥ ... 用 `Field` 实例覆盖 `cls_dict` 中相应的项 ...
- ⑦ ... 并将字段的 `storage_name` 追加到一个列表中 ...
- ⑧ ... 该列表用于填充 `cls_dict` 中的 `__slots__` 项。`cls_dict` 是待构建类的命名空间。
- ⑨ 最后, 调用 `super().__new__`。

`metaclass/checkedlib.py` 的最后一部分是 `Checked` 基类, 该库的用户将对其进行子类化, 以增强自己的类, 如 `Movie`。

此版本的 `Checked` 代码与 `initsub/checkedlib.py` 中的 `checked` 相同 (见“[示例 24.6<727页>](#)”与“[示例 24.6<729页>](#)”), 但有 3 处改动:

1. 添加了一个空的 `__slots__` 以向 `CheckedMeta.__new__` 表明该类不需要特殊处理。
2. 移除了 `__init_subclass__`。现在, 它的工作由 `CheckedMeta.__new__` 完成。
3. 移除了 `__setattr__`。当为用户定义类添加 `__slots__` 之后, 无法再设置未声明的属性了。所以, 该方法变得多余。

最终版 `Checked` 类的完整代码, 如[示例 24.23](#)所示。

</> [示例 24.23: metaclass/checkedlib.py:新版 Checked 基类的完整源码](#)

```
1  class Checked(metaclass=CheckedMeta):
2      __slots__ = ()  # skip CheckedMeta.__new__ processing
3
4      @classmethod
5      def _fields(cls) -> dict[str, type]:
6          return get_type_hints(cls)
7
8      def __init__(self, **kwargs: Any) -> None:
9          for name in self._fields():
10              value = kwargs.pop(name, ...)
```

```

11     setattr(self, name, value)
12     if kwargs:           self._flag_unknown_attrs(*kwargs)
13
14     def __flag_unknown_attrs(self, *names: str) -> NoReturn:
15         plural = 's' if len(names) > 1 else ''
16         extra = ', '.join(f'{name}!r' for name in names)
17         cls_name = repr(self.__class__.__name__)
18         raise AttributeError(f'{cls_name} object has no attribute{plural} {extra}')
19
20     def __asdict(self) -> dict[str, Any]:
21         return {
22             name: getattr(self, name)
23             for name, attr in self.__class__.__dict__.items()
24             if isinstance(attr, Field)
25         }
26
27     def __repr__(self) -> str:
28         kwargs = ', '.join(
29             f'{key}={value}!r' for key, value in self.__asdict().items()
30         )
31         return f'{self.__class__.__name__}({kwargs})'

```

现在,带有验证描述符的类构建器的第3次实现,至此结束。

下一节,将讨论与元类相关的一些一般性问题。

24.10 元类的实际运用

元类 (MetaClass) 功能强大,但也很复杂。在决定实现元类之前,需要考虑如下几点:

24.10.1 可简化或取代元类的现代特性

随着 Python 语言的发展,元类的一些常见用途被新的语言特性所取代。这些语言特性包括:

- 类装饰器

比元类 (MetaClass) 更容易理解,也不容易与基类和元类发生冲突。

- __set_name__

无需自定义元类逻辑,即可自动设置描述符的名称。¹⁴

- __init_subclass__

提供了一种对最终用户透明的自定义类创建方式,甚至比装饰器更简单。但是,可能会在复杂的类层次结构中引入冲突。

- 保留插入顺序的内置字典

无需使用类方法 __prepare__ 的首要原因——提供一个 OrderedDict 来存放待构建类的命名空间。

Python 只在元类上调用 __prepare__,因此如果需要按照源码中出现的顺序处理类的命名空间,那么在

¹⁴在本书第1版中,高级版的 LineItem 类中使用了元类,而使用元类的原因仅仅是为了设置属性的存储名称。详见第1版随书代码库中的“21-class-metapro/bulkfood”。

Python 3.6 之前必须使用元类。

截至 2021 年,每个积极维护的 CPython 版本都支持上述全部特性。

我之所以一直推荐使用这些特性,是因为我见到业内许多人无意间将问题复杂化,而元类是罪魁祸首。

24.10.2 元类是稳定的语言特性

元类 (MetaClass) 是在 2002 年发布的 Python 2.2 中引入的,同期引入的还有所谓的“新型 (newstyle) 类”、描述符,以及特性 (property)。

值得注意的是,最初由 Alex Martelli 于 2002 年 7 月发布的 MetaBunch 示例,在 Python 3.9 中仍然可以正常运行——唯一需要的改动是指定使用元类的方式,在 Python 3 中使用的语法是 class Bunch(metaclass=MetaBunch):。

“24.10.1 可简化或取代元类的现代特性”中提到的新增特性都没有破坏使用元类的现有代码。但先前使用元类的遗留代码通常可以借助这些新特性进行简化。尤其是如果可以放弃对 Python ≤ 3.6 (不再维护) 的支持,代码可以更加精简。

24.10.3 一个类仅能有一个元类

如果声明的类涉及了 2 个或更多元类,将会得到如下令人费解的错误消息:

```
1 TypeError: metaclass conflict: the metaclass of a derived class  
2 must be a (non-strict) subclass of the metaclasses of all its bases
```

即便未涉及多重继承,也有可能发生此种情况。例如,如下声明可能会引发 TypeError:

```
1 class Record(abc.ABC, metaclass=PersistentMeta):  
2     pass
```

我们知道,abc.ABC 类是元类 abc.ABCMeta 的实例。如果元类 Persistent 自身不是 abc.ABCMeta 的子类,就会产生元类冲突。

处理此种错误的方式有两种:

- 寻找其他方式完成需要做的事情,至少将用到的元类 (MetaClass) 去掉 1 个。
- 编写自己的元类 PersistentABCMeta。利用多重继承将元类 PersistentABCMeta 声明为 abc.ABCMeta 和 PersistentMeta 的子类,并将 PersistentABCMeta 类用作 Record 的唯一元类。¹⁵



可以想象,有些人为了赶工期,可能在截止时间之前将某个元类声明为 2 个元类的子类(即多重继承)。根据我的经验,元类编程所需的时间总是比预期的要长。如果在截止时间之前匆匆使用这种多重继承的方案实现元类编程,会存在一定的风险。如果为了赶工期而这么做,可能会给代码埋下不易察觉的 BUG。即便有意避免已知的 BUG,也应将这种方案视为“技术欠债”,因为这样的代码难以理解和维护。

¹⁵如果您对元类的多重继承所带来的影响感到头晕,实属正常。我也宁愿远离这种解决方案。

24.10.4 元类应作为实现细节

除了 `type` 之外,整个 Python 3.9 标准库中只有 6 个元类 (`MetaClass`)。最为人熟知的应该是 `abc.ABCMeta`、`typing.NamedTupleMeta`¹⁶ 和 `enum.EnumMeta`¹⁷。但是,在用户代码中,不应显式使用其中的任何一个,应将它们视为实现细节。

虽然可以用元类进行一些非常古怪的元编程,但最好还是遵守 **最少惊讶原则** (Principle of Least Astonishment),这样大多数用户才能真正将元类 (`MetaClass`) 视为实现细节。¹⁸

近年来,Python 标准库中的一些元类被其他机制所取代,但并没有破坏相应软件包的公开 API。最简单的方法就是提供一个普通类,用户可以通过子类化这个普通类来访问元类提供的功能,如“[示例 24.15<741页>](#)”中的普通类 `Bunch` 及其元类 `MetaBunch` 那样。

作为类元编程的总结,我将与大家分享我在研究本章时发现的最酷、最小巧的元类示例。

24.11 用 `__prepare__` 实现一些高级功能

当针对本书第 2 版更新本章时,我需要找到简单而又具有启发性的示例,以取代自 Python 3.6 起不再需要元类的 `LineItem` 示例。

最简单、最有趣的元类想法是 João S. O. Bueno 提供给我的,他在巴西 Python 社区被称为 JS。根据他的想法,我创建了一个可以自动生成数字常量的类:

```

1  >>> class Flavor(AutoConst):
2      ...
3      banana
4      ...
5      coconut
6      ...
7      vanilla
8
9      ...
10
11     >>> Flavor.vanilla
12     2
13
14     >>> Flavor.banana, Flavor.coconut
15     (0, 1)

```

是的,该代码就是这样用的! 这实际上是 `autoconst/autoconst_demo.py` 中的 doctest。

下面是用户友好的 `AutoConst` 基类,及其背后的元类在 `autoconst/autoconst.py` 中实现:

```

1  class AutoConstMeta(type):
2      def __prepare__(name, bases, **kwargs):
3          return WilyDict()
4
5  class AutoConst(metaclass=AutoConstMeta):
6      pass

```

就这么简单。

显然,诀窍隐藏在 `WilyDict` 中。

¹⁶`typing.NamedTupleMeta` 是一个无文档 (undocumented) 的元类。

¹⁷Python 3.11 起,将 `enum.EnumMeta` 重命名为 `enum.EnumType`,并将 `enum.EnumMeta` 保留为 `enum.EnumType` 的别名,相关源码见 CPython 实现的“`Lib/enum.py` 1095 行”。

¹⁸在我决定研究 Django 模型字段的实现原理之前,我以从事 Django 开发为生已经好几年了。直至那时,我才了解到描述符和元类。

当 Python 处理用户类的命名空间并读取 `banana` 时,它会在 `__prepare__` 提供的映射(一个 `WilyDict` 实例)中查找该名称。`WilyDict` 实现了 `__missing__` 方法,详见“[小节 3.5.2<76页>](#)”。`WilyDict` 实例中最没有名为 `banana` 的键,因此 `__missing__` 方法被触发。该方法在运行时创建了一个键为“`banana`”、值为 0 的项,并返回该值 0。Python 得到回应后,然后尝试检索“`coconut`”。`WilyDict` 立即添加相应的项,并返回相应的值 1。检索“`vanilla`”也是同样的处理过程,它被映射为 2。

前面章节已见过 `__prepare__` 与 `__missing__`。JS 真正的创意之处是将二者结合起来使用。

如下是 `WilyDict` 类的源码,摘自 `autoconst/autoconst.py`:

```

1  class WilyDict(dict):
2      def __init__(self, *args, **kwargs):
3          super().__init__(*args, **kwargs)
4          self.__next_value = 0
5
6      def __missing__(self, key):
7          if key.startswith('__') and key.endswith('__'):
8              raise KeyError(key)
9          self[key] = value = self.__next_value
10         self.__next_value += 1
11         return value

```

在实验过程中,我发现 Python 在待构建类的命名空间中查找 `__name__`,导致 `WilyDict` 添加了一个 `__name__` 项,并递增了 `__next_value` 的值。因此,我在 `__missing__` 方法中添加了 `if` 语句,以便对看起来像“dunder 属性”¹⁹的键引发 `KeyError`。

`autoconst/autoconst.py` 包要求您精通 Python 动态构建类的机制,也演示了这个机制的强大之处。

我在为元类 `AutoConstMeta` 和普通类 `AutoConst` 添加更多功能的过程中,度过了一段美好时光。但我不会与大家分享我的实验,而是让大家尽情体验 JS 的巧妙创意。

一些实验思路如下:

- 通过值获取常量名称。例如, `Flavor[2]` 返回“`vanilla`”。为此,需要在元类 `AutoConstMeta` 中实现 `__getitem__` 方法。自 Python 3.9 起,可以在 `AutoConst` 类自身中实现 `__class_getitem__` 方法。
- 通过在元类上实现 `__iter__` 方法,以支持对类的迭代。我会让 `__iter__` 以 `(name, value)` 对的形式生成常量。
- 实现一个新的 `Enum` 变体。这将是一项艰巨的任务,因为 `enum` 包中充满了各种技巧,包括有数百行代码的 `EnumMeta` 元类和一个重要的 `__prepare__` 方法。

尽情享受吧!

¹⁹dunder 属性:指 Python 类中以双下划线(`__`)开始和结束的特殊属性或特殊方法。



作为“PEP 585 –Type Hinting Generics In Standard Collections”的一部分，在Python 3.9中添加了特殊方法`__class_getitem__`来支持泛型(Generic Type)。得益于`__class_getitem__`，Python核心开发人员不必再为内置类型编写新的元类，实现编写`list[int]`之类的泛化类型提示所需的`__getitem__`方法。这个功能看似作用不大，但代表了元类更广泛用法的一部分：实现运算符和其他特殊方法，以便可在类级别上执行某些操作，例如将类自身变成可迭代对象，就像`Enum`的子类一样。

24.12 元类总结

元类(MetaClass)，以及类装饰器和的用途如下：

- 子类注册
- 子类结构验证
- 将装饰器一次性应用到多个方法
- 对象序列化
- 对象关系映射
- 基于对象的持久化
- 在类级别实现特殊方法
- 实现其他语言中的类特性，如traits²⁰、面向方面编程²¹。

某些情况下，类元编程还可将运行时重复执行的任务放到导入时执行，以解决性能问题。

最后，回顾一下Alex Martelli在水禽与抽象基类(??页)标记栏中给处的建议：

并且，不要在生产代码中定义抽象基类(ABCs)(或元类)。如果您有这样做的冲动，我敢打赌可能是因为您想“找茬”，就像刚拿到新工具的人都有大干一场的冲动。如果能避开这些深奥的概念，那么您(以及未来的代码维护人员)的生活将更轻松愉快，因为代码会变得简洁明了。

我认为，Martelli的建议不仅适用于抽象基类(ABCs)和元类，也适用于类层次结构、运算符重载、函数装饰器、描述符、类装饰器和用`__init_subclass__`的类构建器。

这些强大的工具主要用于为库开发和框架开发提供支持。应用程序开发自然应使用Python标准库或外部包提供的工具。但是，在应用程序代码中实现这些工具，往往表明抽象不够成熟。

优秀的框架是提炼出来的，而不是发明出来的。^a

——David Heinemeier Hansson, Ruby on Rails的创造者

^a这句话被广泛引用。我在DHH博客2005年的一篇文章中发现了早期的直接引用。

²⁰traits是一种代码重用机制，它允许将一组方法或行为组合到一个单独的单元中，并可以在不同的类型中重复使用。

²¹面向方面编程(AOP, Aspect-Oriented Programming)是一种编程范式，旨在通过分离交叉关注点来提高模块化程度，详见维基百科词条“Aspect-oriented programming”。

24.13 本章小结

本章首先概述了类对象中的属性,如 `__qualname__` 和 `__subclasses__()` 方法。接着,介绍了如何使用内置函数 `type` 在运行时构造类。

随后,介绍了特殊方法 `__init_subclass__`,并定义了第 1 版 Checked 基类,旨在用 `Field` 实例取代用户定义子类中的属性 [类型提示 \(Type Hints\)](#)。`Field` 实例借助构造函数在运行时限定实例属性的类型。

同样的功能也可以用 `@checked` 类装饰器实现。与 `__init_subclass__` 类似,类装饰器可以为用户定义的类添加功能。我们发现, `__init_subclass__` 与类装饰器都不能为类动态配置 `__slots__`,因为它们只在类创建后才起作用。

通过实验展示了在涉及模块、描述符、类装饰器和 `__init_subclass__` 调用时,Python 代码的执行顺序,从而厘清了“导入时”与“运行时”的概念。

接下来介绍了元类。首先简要介绍了元类 `type`,以及用户定义的元类如何用 `__new__` 方法定制待构建的类。然后,自定义了第 1 个元类,即使用 `__slots__` 的经典 `MetaBunch` 示例。接下来,用另一个求解时机实验,证明了元类的 `__prepare__` 方法与 `__new__` 方法均在 `__init_subclass__` 与类装饰器之前被调用,这为进一步定制待构建类提供了机会。

随后,定义了第 3 版 Checked 类构建器。在该示例中,用到了 `Field` 描述符和自定义的 `__slots__` 配置。之后,介绍了在实践中使用元类的一些注意事项。

最后,介绍了 Joāo S. O. Bueno 发明的 `AutoConst` 类,它巧妙地用元类的 `__prepare__` 方法返回一个实现了 `__missing__` 的映射。`autoconst.py` 用不到 20 行的代码,展示了将各项 Python 元编程技术综合在一起的强大功能。

我还没有发现过一种语言能像 Python 这样,既能让初学者感到轻松,又能让专业人士感到实用,还能让黑客们感到兴奋。感谢 Guido van Rossum 和 Python 背后的所有人,是他们让 Python 变得如此简单。

24.14 延伸阅读

本书技术审校 Caleb Hattingh 编写了 [autoslot 包](#),该包提供了一个元类,通过检查 `__init__` 的字节码,找到所有对 `self` 的属性赋值,自动为用户定义的类创建类属性 `__slots__`。它非常有用,也是一个极好的学习示例:`autoslot.py` 中只有 74 行代码,其中 20 行注释说明了最难解决的部分。

《The Python Language Reference》中有关本章的重要资料为“3. Data model”中的“3.3.3. Customizing class creation”一节,该节涵盖了 `__init_subclass__` 与元类。《The Python Standard Library》中,“Built-in Functions”一章的“`type` 类文档”,以及“Built-in Types”一章的“Special Attributes”一节也是必读内容。

在《The Python Standard Library》中,“`types` 模块文档”涵盖了 Python 3.3 中添加的 2 个简化类元编程的函数:`types.new_class` 和 `types.prepare_class`。

类装饰器在 Collin Winter 编写的“[PEP 3129 - Class Decorators](#)”中被正式化,其参考实现由 Jack Diederich 编写。Jack Diederich 在 PyCon 2009 上所做的题为“[Class Decorators: Radically Simpl](#)”演讲,对类装饰器的功能做了简单介绍。除了 `@dataclass` 之外,Python 标准库中还有一个有趣的、更简单的类装饰器示例,即 `@functools.total_ordering`,它可以生成用于对象比较的特殊方法。

关于元类,Python 文档中的主要参考资料是“[PEP 3115 - Metaclasses in Python 3000](#)”,其中引入了特殊方法 `__prepare__`。

《Python in a Nutshell, 3rd Ed》(Alex Martelli、Anna Ravenscroft、Steve Holden 著)一书具有权威性, 但它是在“PEP 487 – Simpler customisation of class creation”发布之前写的。该书中的主要元类示例(即 MetaBunch), 现在仍然不过时, 因为暂时还未出现比它更简单的实现方案。《Effective Python, 2nd Ed》(Brett Slatkin 著)一书中有几个类构建示例用到了最新技术, 包括元类 (MetaClass)。

要了解 Python 中类元编程 (Class Metaprogramming) 的起源, 推荐阅读 Guido van Rossum 在 2003 年发表的论文“Unifying types and classes in Python 2.2”。该文章也适用于现代 Python, 因为它涵盖了当时所谓“新式 (new-style) 类”的语义 (Python 3 中的默认语义), 包括描述符和元类。Guido 在该论文中引用的参考文献之一《Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming》(Ira R. Forman、Scott H. Danforth 合著), 他在亚马逊网站上为这本书打了 5 星好评, 并添加了如下评论:

本书为 Python 2.2 中元类的设计做出了贡献

可惜这本书已经绝版了; Python 通过 super() 函数实现了协作式多重继承, 每当提到这方面难题时, 我都会提起这本书。据我所知, 该书是有关协作式多重继承方面最好的教程。^a

^a我买了一本二手书, 发现这是一本非常具有挑战性的读物。

如果您热衷于元编程, 可能希望 Python 能提供终极的元编程特性: 语法宏, 如 Lisp 系列语言以及最近 Elixir 和 Rust 所提供的那样。语法宏比 C 语言中原始的代码置换宏更强大, 也更容易出错。语法宏是一种特殊函数, 可在编译步骤之前将使用自定义语法的源代码重写为标准代码, 从而使开发人员能够在不更改编译器的情况下引入新的语言结构。与运算符重载一样, 语法宏也可能被滥用。但只要社区理解并管控好这些缺点, 它们就能支持强大且用户友好的抽象, 比如 DSL (领域专用语言)。2020 年 9 月, Python 核心开发者 Mark Shannon 发布了“PEP 638 -Syntactic Macros”, 提出了这样的主张。在最初发布一年后, “PEP 638”仍处于草案阶段, 也没有持续的讨论。显然, 这不是 Python 核心开发人员的首要任务。我希望看到有人能进一步讨论“PEP 638”, 并最终获得批准。借助语法宏, Python 社区可在对核心语言进行永久性修改之前, 先行尝试一些有争议的新特性, 如海象运算符 (PEP 572)、模式匹配 (PEP 634) 和求解类型提示的替代规则 (PEP 563与PEP 649)。目前, 可以用 MacroPy 软件包体验语法宏。

杂谈

在本书最后一个杂谈的开头, 我将引用加州大学 (伯克利和圣巴巴拉) 的 2 位计算机科学教授 Brian Harvey 与 Matthew Wright 的一段长话。他们在合著的《Simply Scheme: Introducing Computer Science》一书中写道:

计算机科学的教学包含 2 种思想流派, 这 2 种流派为:

- **保守派**

计算机程序已经变得过于庞大和复杂, 超过了人类大脑所能承载的限度。因此, 计算机科学教育的任务就是教会人们如何规范自己的工作, 从而让 500 名平庸的程序员能够联合起来, 编制出正确符合规范的程序。

- **激进派**

计算机程序已经变得过于庞大和复杂, 超过了人类大脑所能承载的限度。因此, 计算机科学教育的任务就是教会人们如何扩展思维, 打破常规, 学会用更广博、更强大、更灵活的方式去思考问题。编程思维的各个方面在程序中必会得到充分体现。

——Brian Harvey 与 Matthew Wright,《Simply Scheme》前言^a

^a《Simply Scheme》(Brian Harvey, Matthew Wright 合著), 第 xvii 页。本书全文可在 Berkeley.edu 获取。

Harvey 和 Wright 的夸张描述是关于计算机科学教学的, 但它们同样适用于编程语言设计。现在, 您应该已经猜到我赞同“激进派”的观点, 而且我相信 Python 就是本着这种精神设计的。

Java 实际上一开始就要求使用访问器方法, 而且众多 Java IDE 都提供了生成 **读值 (getter) /设值 (setter)** 方法的快捷键。与此相比, Python 的 **特性 (property)** 算是一大进步。**特性 (property)** 的主要优点是, 一开始编写程序时可先将属性设置为公开的(遵照 [KISS 原则](#))。因为我们可以随时公开属性更改为 **特性 (property)**, 而无需太大的改动。不过, 描述符更进一步, 它提供了一个抽象掉重复访问逻辑的框架。这个框架是如此有效, 以至于 Python 的基本结构在背后也用到了描述符。

另一个强大的想法是将函数视作一等对象, 为高阶函数铺平道路。事实证明, 描述符和高阶函数的结合实现了函数和方法的统一。函数的 `__get__` 方法能将实例绑定到 `self` 参数上, 即时生成方法对象。这种做法相当优雅。^a

最后, 我们将类也视为一等对象。作为一门对初学者友好的编程语言, Python 提供了强大的抽象, 如类构建器、类装饰器, 以及允许用户定义功能完整的元类, 这是一个杰出的设计成就。最重要的是, 这些高级特性的集成方式并没有使 Python 日常编程的适用性变得复杂(实际上, 还潜移默化地改善了 Python 的适用性)。Django 和 SQLAlchemy 等框架的便捷和成功在很大程度上归功于 **元类 (MetaClass)**。多年来, Python 中的 **类元编程 (Class Metaprogramming)** 变得越来越简单, 至少对于常见场景来说是这样。最优秀的语言特性能让所有人都受益, 即便有些 Python 用户感知不到它们的存在。但他们总是可以学习并创建下一个伟大的库。

期待您对 Python 社区与生态的贡献!

^a《Machine Beauty: Elegance and the Heart of Technology》(David Gelernter 著) 一书开篇就对工程作品(从桥梁到软件等)中的优雅与美学进行了深入探讨。后面的章节并不精彩, 但开篇值得一读。

wechat: 119554488

结语

Python 是一种适用于成年人的语言。

——Alan Runyan, Plone 联合创始人

Alan 精辟的定义表达了 Python 的最佳品质之一: 它不会妨碍你, 让你做你必须做的事情。这也意味着它不会为您提供工具, 以限制他人对您的代码及代码所构建的对象的操作。

已经 30 岁的 Python 仍然越来越受欢迎。当然, 它并不完美。最让我恼火的是标准库中对 CamelCase、snake_case 和 joinedwords 的混用。²²但是, 语言的定义和标准库只是语言生态的一部分。由用户和贡献者组成的社区才是 Python 生态中最重要的部分。

这里有一个例子可以说明社区的重要性: 在撰写本书第 1 版关于 [asyncio 包](#) 相关的内容时, 我感到沮丧, 因为 `asyncio` 的 API 包含许多函数, 其中有几十个是 [协程 \(Coroutine\)](#), 你必须用 `yield from` (现在是 `await`) 调用这些协程, 但对于普通函数却不能这样做。虽然这些在 [asyncio 文档](#) 中有说明, 但有时你必须阅读几段文字后才能弄清一个特定函数是否是 [协程 \(Coroutine\)](#)。因此, 我给 `python-tulip` 发了一则消息, 标题为 “[Proposal: make coroutines stand out in the asyncio docs](#)”。Victor Stinner (`asyncio` 核心开发者)、Andrew Svetlov (`aiohttp` 主要作者)、Ben Darnell (`Tornado` 首席开发者) 和 Glyph Lefkowitz (`Twisted` 的发明者) 加入了这个讨论。Darnell 提出了一个解决方案, Alexander Shorin 解释了如何在 `Sphinx` 中实现这一解决方案, Stinner 添加了必要的配置和标记。在我提出此问题后不到 12 小时, 整个 [asyncio 在线文档](#) 就都添加了你今天可以看到的 “`coroutine`” 标签。

在排外的社区中绝不会发生此种事情。任何人都可以加入 `python-tulip` 邮件列表, 而我在编写那个提议之前也只发过几次帖子而已。这个故事表明 Python 社区是一个真正向新想法和新成员开放的社区。Guido van Rossum 以前经常出现在 `python-tulip` 邮件列表中, 并时常回答一些基本问题。

还有一个例子能说明 Python 社区的开放性: Python 软件基金会 (Python Software Foundation, PSF) 一直致力于提高 Python 社区的多样性。目前已经取得了一些令人欣喜的成果。在 2013-2014 年, PSF 董事会首次选出了女性董事——Jessica McKellar 与 Lynn Root。2015 年, 在蒙特利尔举行的 PyCon 北美大会 (Diana Clarke 主持), 约 1/3 的演讲者为女性。PyLadies 成为了一场真正的全球性运动, 我为巴西拥有如此多的 PyLadies 分会而感到自豪。

如果您是一名 Python 爱好者, 但尚未参与社区活动, 我建议您尽快参与。寻找您所在地区的 PyLadies 或 Python 用户组 (Python Users Group, PUG)。如果没有, 那就创办一个。Python 无处不在, 所以您不会孤单。如果可以的话, 可以去参加异地举办的活动, 也可以参加线上活动。在 Covid-19 疫情期间, 我在线上会议的 “[Hallway Track](#)”²³ 中学到了很多。来参加 [PythonBrasil](#)²⁴ 会议吧——多年来这个大会一直定期邀请来自世界各地的演讲者。除了知识分享, 与其他 Python 程序员面对面交流还能带来许多其他好处, 比如获得工作机会和得到真正的友谊。

²²

²³ Hallway Track 是指在 Python 会议或活动中, 除了正式议程和讲座之外的非正式交流和互动环节。

²⁴ PythonBrasil 是巴西的 Python 社区举办的年度大会, 旨在促进 Python 编程语言在巴西的发展和应用。

我知道,如果没有多年来在 Python 社区结识的许多朋友的帮助,我不可能写出这本书。

我的父亲 Jairo Ramalho 常说 “Só erra quem trabalha”,这是葡萄牙语,意思是“只有真正做事的人才会犯错”。这个建议很棒,可以让你不再害怕失败,勇敢向前。在写这本书的过程中,我当然也犯了不少错误。审校、编辑和抢读版的读者帮我找出了其中的许多错误。在第 1 版抢读版发布后的几个小时内,就有读者报告了本书勘误页中的错别字。其他读者报告了更多的错误,还有朋友直接联系我,提出建议和更正。我写完本书后, O'Reilly 的校对人员会在出版过程中找出其他错误。如果书中还有其他错误和词不达意的表述,责任都在我,在此向各位读者表示歉意。

很高兴能为本书第 2 版画上圆满的句号,非常感谢所有在出版过程中给予我帮助的人。

希望很快能在现场活动中见到你。如果你在附近看到我,请过来打个招呼!

延伸阅读

在本书的最后,我将介绍一些关于“Pythonic”的参考资料。“Pythonic”也是本书试图解决的主要问题。

Brandon Rhodes 是一位出色的 Python 教师,他的演讲 “A Python Ästhetic: Beauty and Why I Python” 非常精彩,首先是标题中使用了 Unicode 字符 U+00C6 (LATIN CAPITAL LETTER AE)。另一位出色的老师 Raymond Hettinger,在 2013 年美国 PyCon 大会上也谈到了 Python 之美:“Transforming Code into Beautiful, Idiomatic Python”。

Ian Lee 在 Python-ideas 邮件列表中发起的 “Evolution of Style Guides” 主题值得一读。Lee 是 pep8 软件包的维护者,该软件包用于检查 Python 源代码是否符合 “PEP 8”。为了检查本书中的代码,我使用了 flake8,它封装了 pep8、pyflakes 和 Ned Batchelder 开发的复杂度插件 McCabe。

除了 “PEP 8” 之外,其他有影响力风格指南还有 “Google Python Style Guide” 与 “Pocoo Styleguide”。Pocoo 团队为我们开发了 Flake、Sphinx、Jinja 2 和其他优秀的 Python 库。

《The Hitchhiker's Guide to Python!》由多人维护,是一本关于如何编写 Pythonic 代码的著作。为此著作贡献最多内容的是 Kenneth Reitz。他因开发了特别 Pythonic 的 requests 包,而被 Python 社区视为英雄。David Goodger 在 PyCon US 2008 上发表了题为 “Code Like a Pythonista: Idiomatic Python” 的教程。如果打印出来,教程笔记长达 30 页。Goodger 创建了 reStructuredText 与 docutils,二者是 Sphinx 的基础。Sphinx 是 Python 出色的文档系统(顺便说一下,MongoDB 和许多其他项目的官方文档系统也都是 Sphinx)。

Martijn Faassen 在 “What is Pythonic?” 中,直接回答了这个问题。在 python-list 邮件列表中也有一个相同主题的回帖。Martijn 的回帖是 2005 年的,而那个主题是 2003 年开始讨论的,但 Pythonic 的思想并没有太多变化,Python 语言自身也是如此。“Pythonic way to sum n-th list element?” 主题对 “Pythonic” 做了深入讨论,“十二 序列的特殊方法<321页>”的“杂谈”(??页)中大量引用了该主题。

“PEP 3099: Things that will Not Change in Python 3000”解释了为何经过 Python 3 的大幅调整之后,许多事情仍然保持原样。在很长一段时间里,Python 3 有个昵称——Python 3000。不过,Python 3 比 Python 3000 早来了几个世纪,这可能会让一些人感到沮丧。“PEP 3099”由 Georg Brandl 撰写,汇集了 Guido van Rossum 表达的许多观点。²⁵“Python Essays”上列出了 Guido 本人的多篇文章。

²⁵BDFL (Benevolent Dictator For Life) 意为“终身仁慈独裁者”。这一术语最初用于描述 Guido van Rossum——Python 编程语言的创始人和主要设计者。

作者简介

Luciano Ramalho 在 1995 年 Netscape 上市之前, 他是一名 Web 开发人员。先后用过 Perl 与 Java, 1998 年开始使用 Python。他于 2015 年加入 Thoughtworks, 担任圣保罗办事处首席顾问。他曾在美洲、欧洲和亚洲的 Python 活动中发表过主题演讲、讲座和教程, 还曾在 Go 和 Elixir 会议上发表演讲, 主要关注语言设计主题。Ramalho 是 Python 软件基金会的研究员, 也是巴西第一个骇客 (Hacker) 空间 Garoa Hacker Clube 的联合创始人。

译者简介

ShadowC 一名从事 DBA 岗位的 80 后无产 Python 爱好者, 爱好技术研究、文档撰写、文档翻译等。社交账号 (wx: 119554488)

关于封面

本书封面上的动物是纳马夸沙蜥 (Pedioplanis namaquensis), 分布于纳米比亚的干旱大草原和半沙漠地区。纳马夸沙蜥的身体呈黑色, 背部有 4 条白色条纹, 腿呈褐色, 有白色斑点, 腹部呈白色, 尾巴长而呈粉褐色。它是速度最快的日间活动蜥蜴之一, 以小昆虫为食。它栖息在植被稀疏的沙砾滩上。雌性纳马夸沙蜥在 11 月产下 3~5 枚卵。冬季, 这些蜥蜴都在灌木丛边附近挖掘的洞穴中休眠。

纳马夸沙蜥目前的濒危等级属于“最不受关注”。O'Reilly 出版的图书, 封面上的许多动物都濒临灭绝。它们对世界都很重要。

封面插图由 Karen Montgomery 绘制, 以 Wood 的《Wood's Natural History》中的黑白版画为基础。封面字体为 Gilroy Semibold 和 Guardian Sans。正文字体是 Adobe Minion Pro; 标题字体是 Adobe Myriad Condensed; 代码字体是 Dalton Maag 的 Ubuntu Mono。

卷之三

wechat: 119554488

清单目录

A.1 术语清单

组合模式 (Composition)

在 Python 语言特性中,组合模式 (Composition) 是一种对象组合方式,它涉及到将多个对象组合在一起,以实现比单个对象更复杂的功能。与继承不同,组合模式强调的是“has-a”关系,而不是“is-a”关系。

模式匹配 (Pattern Match)

在 Python 3.10 中引入的新功能 (PEP 634 -Structural Pattern Matching: Specification)。通过 match/case 语法结构来实现的模式匹配,其功能类似于某些编程语言中的 case/when 或 switch/case。但是,模式匹配的功能更强大。因为它支持“析构 (Destructor)”,这是一种高级解包形式。更多关于模式匹配的精彩介绍,详见:PEP 634: Structural Pattern Matching。.

匹配对象 (Subject)

在 match/case 模式匹配 (Pattern Match) 语法结构中,match 关键后的表达式就是匹配对象 (Subject),即各个 case 子句中的模式 (Pattern) 将尝试匹配的目标数据。例如,示例 2.9 中 ① 处的 message。.

模式 (Pattern)

在 match/case 模式匹配 (Pattern Match) 语法结构中,case 关键字后的表达式就是模式 (Pattern)。例如,示例 2.9 中 ② 处的 ['BEEPER', frequency, times] 与 ③ 处的 ['NECK', angle]。.

序列模式 (Sequence Pattern)

序列模式 (Sequence Pattern) 是一种模式匹配 (Pattern Match) 方式,用于匹配和处理序列 (如列表、元组等) 中的元素。序列模式中的方括号 [] 与圆括号 () 语义是一致的,因此可以写成元组或列表的形式。如 ['LED', ident, intensity] 或 [name, _, _, (lat, lon)]。

映射模式 (Mapping Pattern)

映射模式 (Mapping Pattern) 是一种 [模式匹配 \(Pattern Match \)](#) 方式, 用于匹配和处理映射 (如字典) 中的键值 (Key:Value) 对。映射模式示例如 'type': 'book', 'api': 1, 'author': name.

类模式 (Class Pattern)

类模式 (Class Pattern) 是一种 [模式匹配 \(Pattern Match \)](#) 的方式, 用于匹配特定类的实例。

卫 (guard) 语句

卫 (guard) 语句, 即 if 子句, 是指在 Python 模式匹配用于为 case 子句中的 [模式 \(Pattern \)](#) 添加额外判断条件的 if 子句, 以允许在匹配模式时检查更多的条件, 以便更精确地进行模式匹配。

析构 (Destructor)

析构 (Destructor) 是一种用于匹配数据结构, 并将其解构为单个变量的特性。

抽象基类 (ABCs)

[抽象基类 \(ABCs, Abstract Base Classes \)](#) 是一种用于定义接口规范的类, 可以作为其他类的基类, 但不能被直接实例化。简而言之, [抽象基类 \(ABCs \)](#) 可以理解为 Python 中的接口类, 用于约定类的行为和方法的实现。

具体子类 (Concrete Subclass)

具体子类 (Concrete Subclass) 指的是一个实现了 [抽象基类 \(ABCs \)](#) 的具体类。具体子类是对 [抽象基类 \(ABCs \)](#) 进行扩展并实现其 [抽象方法 \(Abstract Method \)](#) 的类, 可以被实例化并用于实际的业务逻辑中。具体子类可以重写从抽象基类继承的方法或属性, 并添加自己的实现。

具体类 (Concrete Class)

具体类 (Concrete Class) 指的是具体的类, 即可以实例化的类。与 [抽象基类 \(ABCs \)](#) 和 [元类 \(Meta-Class \)](#) 相对应, 具体类 (Concrete Class) 可以直接创建对象, 并具有具体的属性和方法。

具体对象 (Concrete Object)

具体对象 (Concrete Object) 是 [具体类 \(Concrete Class \)](#) 的实例化结果, 它拥有了具体类中定义的属性和方法。但具体对象可以有自己的特定属性值, 与其他对象实例相互独立。

具体类型 (Concrete Type)

在 Python 中, 具体类型 (Concrete Type) 是指可以直接实例化和使用的类型, 而不是 [抽象基类 \(ABCs \)](#) 或接口类型。

结构类型 (Structural Typing)

结构类型 (Structural Typing) 指的是一种类型系统, 其中对象的类型是根据其结构 (即其属性和方法) 而不是显式声明的类型来确定的。与 [鸭子类型 \(Duck Typing \)](#) 类似, 结构类型也是强调对象的结构, 而不是它们的特定类型或基类。

抽象方法 (Abstract Method)

抽象方法 (Abstract Method) 是指在父类中声明, 但在子类中必须被重写实现的方法。抽象方法本身不提供具体的实现, 而是为了强制子类提供自己的实现。在 Python 中, 可以通过 [abc 模块](#) 来定义 [抽象基类 \(ABCs \)](#) 和抽象方法。

具体方法 (Concrete Method)

具体方法 (Concrete Method) 是指在父类中已实现，并可直接被子类继承或调用的方法。具体方法 (Concrete Method) 在父类中已提供具体实现逻辑，但子类可以选择性地重写或者扩展它们。.

元类 (MetaClass)

元类 (MetaClass) 是指用于创建类的类，它允许我们定义类的行为和特性，包括类的实例化、属性和方法等。.

虚拟子类 (Virtual Subclass)

虚拟子类 (Virtual Subclass) 是调用抽象基类的 `register()` 方法注册的子类，详见“13.5.6 抽象基类的虚拟子类”。.

实际子类 (Actual Subclass)

实际子类 (Actual Subclass) 是通过继承直接创建的子类。.

可哈希 (hashable)

如果一个对象在其生命周期内有一个永不改变的哈希码 (依托 `__hash__()` 方法)，并且可以与其他对象进行等值比较 (依托 `__eq__()` 方法)，那么这个对象就是“可哈希 (hashable)”。相等的可哈希对象，必须拥有相同的哈希码。.

哈希码

映射对象中的项由“键: 值”对构成。为了避免与映射对象中的“值”相混淆，本书用“哈希码”代替“哈希值”。.

加盐 (Salted)

哈希加盐 (Salted Hash) 是为了提高安全性，而在哈希计算中添加一个随机值，使得相同的输入值在不同的加盐条件下产生不同的哈希码。.

Fail-Fast (快速失败)

Python 的“Fail-Fast (快速失败)”原则是指在代码执行过程中，一旦出现错误就立即抛出异常，停止程序的执行，以便快速发现和解决问题，而不是继续执行可能导致更严重错误的代码。.

字典视图 (Dictionary View)

在 Python3 中，字典视图是字典数据结构的只读投影。它们提供了一种以不同角度查看字典内容的方式，而无需复制原始字典数据或创建额外的列表或迭代器。字典视图是动态的，会同步反映字典的变化。

Python3 提供了 3 种字典视图：

- `dict.keys()`: 返回 `dict_keys` 类的实例，是一个包含字典所有键的视图。
- `dict.values()`: 返回 `dict_values` 类的实例，是一个包含字典所有值的视图。
- `dict.items()`: 返回 `dict_items` 类的实例，是一个包含字典所有“键: 值”对的视图。.

语法糖 (Syntactic Sugar)

语法糖 (Syntactic Sugar) 是一种编程语言中的简化语法或功能，它使代码更易读、更简洁。.

类型提示 (Type Hints)

类型提示 (Type Hints) 又称 “[类型注解 \(Type Annotations \)](#)”，是一种声明函数参数、返回值、变量和属性的预期类型的方法，以便于提高代码的可读性、可靠性和可维护性。类型提示语法详见 “[PEP 526 - Syntax for Variable Annotations](#)”。

类型注解 (Type Annotations)

类型注解 (Type Annotations) 又称 “[类型提示 \(Type Hints \)](#)”，是 Python 中的一种语法，用于在函数参数、返回值和变量上指定其预期的类型，以便为代码的静态类型检查和类型推断提供支持。

前向引用 (Forward Reference)

[类型提示 \(Type Hints \)](#) 的前向引用是指在代码中使用尚未定义的类型，可通过使用类型的字符串全名来解决。例如，当定义一个类 A，在类 A 中引用另一个类 B，而 B 被定义在 A 之后，此时可用字符串来注解类型提示，如: `def method(self, b: 'B'):`。这样可以帮助编译器正确解析类型。

变量注解 (Variable Annotation)

变量注解是用于在变量声明中提供类型信息的语法。它允许您在变量名称后面使用冒号和类型注释来指定变量的类型。这些注释并不会影响程序的运行，但可以提供更明确的[类型提示 \(Type Hints \)](#)和文档。变量注解语法详见 “[PEP 526 - Syntax for Variable Annotations](#)”。

字段注解 (Field Annotation)

字段注解 (Field Annotation) 是一种用于在变量或函数参数添加元数据，以为其指定数据类型的特性。在变量声明或函数签名中，添加 [类型提示 \(Type Hints \)](#) 可提高代码的可读性、可维护性。

代码异味 (Code Smell)

代码异味 (Code Smell) 是指在 Python 中可能存在的问题或不良实践的迹象，可能导致代码难以维护、可读性差、性能问题等。

工厂函数 (Factory Function)

工厂函数 (Factory Function) 又称 “[工厂 \(Factory \)](#)”，是指在 Python 中，用于创建对象实例的函数，而不是直接调用类的构造函数。它们可以根据不同的条件或参数返回不同的对象实例，提供了一种更灵活和可扩展的对象创建方式。

硬编码 (Hardcoded)

硬编码指的是将特定的值或参数直接写入代码中，而不使用变量或配置文件进行存储和管理。其缺点是降低代码的灵活性与可维护性。

框架 (Framework)

框架 (Framework) 是在编程中使用的预先设计好的软件结构，用于提供开发者工具和库，以简化应用程序开发的过程并提供一套标准的参考结构。Python 中典型的框架如：爬虫框架 (Scrapy)、Web 框架 (Flask) 等。

单例 (Singleton)

单例 (Singleton) 是一种设计模式，用于确保一个类只有一个实例，并提供全局访问该实例的方式。这意味着无论在程序中的哪个位置调用该类，都始终获得相同的实例对象。

类属性 (Class Attribute)

类属性是指定义在类级别 (而不是实例级别) 上的属性。类属性可以由所有该类的实例共享和访问，并且它们的值在所有实例之间是共享的。可通过类或实例对象访问类属性。.

类方法 (Class Method)

与类属性类似，类所拥有的方法叫类方法。类方法一般与类属性 (Class Attribute) 配合使用，通常用于访问私有的类属性。通过装饰器 `@classmethod` 定义类方法，且类方法的第一个参数必须是类对象 (习惯为 `cls`)。.

静态方法 (Static Method)

静态方法是指调用时无需传递类对象和实例对象的方法。即静态方法的形参 (Parameter) 里没有 `self/cls`，不会自动绑定实例对象和类对象。通过装饰器 `@staticmethod` 定义静态方法。.

绑定 (Bound) 方法

由于用户定义的所有函数中都会有一个特殊方法 `__get__`，当将函数依附到类中之后，这些函数就相当于描述符。所以，当在实例上调用用户函数时，类中的函数就会变成绑定方法，即绑定到实例上的方法。详见“[23.4 方法也是描述符](#)”。

描述符 (Descriptor)

描述符 (Descriptor) 是一种特殊的 Python 对象，用于控制属性访问和赋值的行为。可以在类定义中被用作属性的 `get`、`set` 和 `delete` 操作的定制化处理器，提供了一种强大而灵活的方式来管理属性的访问权限和行为。.

参数化泛型 (Parameterized Generic Type)

参数化泛型是指在定义类或函数时可以接受不同参数类型的泛型类型 (Generic Type)。例如，`List[int]` 表示一个整型列表，`Dict[str, float]` 表示一个键为字符串，值为浮点型的字典。通过使用参数化泛型，可以提高代码的灵活性和重用性。.

泛型 (Generic Type)

泛型 (Generic Type) 是指具有通用性的类型，可以处理不同类型的对象。它们可以增强代码的可重用性和类型安全性。如，`List` 是 Python 中的泛型，可以用于创建一个可以存储任意类型元素的列表——`List[int]` 代表一个整型元素的列表，`List[str]` 代表一个字符串元素的列表。.

类型参数 (Type Parameters)

类型参数 (Type Parameters)，也称为“类型变量 (type variables)”是用于声明 泛型 (Generic Type) 的参数，用于指定 泛型 (Generic Type) 可以处理的元素的类型。例如，`list[str]` 中的 `str` 限制列表中的元素为 `str` 类型。.

哨兵值 (Sentinel Value)

哨兵值 (Sentinel Value) 指的是被用作特殊标记或标识的特定值，通常用于表示特殊情况或结束条件。.

哨兵 (Sentinel)

哨兵 (Sentinel) 是在 Python 中用于表示特定状态或结束条件的一种特殊类型。它通常用作函数或方法的参数，用于简明地表示特殊情况或结束条件，而无需使用特定值或布尔标志。当哨兵值 (Sentinel Value) 出现时，它可以发出信号，指示程序执行特定操作或终止特定处理过程。.

仅初始化 (Init-only)

仅初始化 (Init-only) 变量是指在类的 `__init__` 方法中定义的, 仅在初始化过程中可赋值的实例变量, 这些变量在初始化后不可更改。.

都柏林核心模式 (Dublin Core Schema)

都柏林核心模式 (Dublin Core Schema) 是一组术语, 可用于描述数字资源 (如视频、图像、网页等)、实物资源 (如书籍、CD 等) 以及艺术品等对象。.

实参 (Argument)

实参 (Argument 或 Actual Argument), 是在函数调用时传递给函数的具体值。它们是函数调用时提供的实际数据, 用于填充函数定义中的形参 (Parameter)。.

形参 (Parameter)

形参 (Parameter 或 Formal Parameter) 是函数定义中的参数, 用于接收传递给函数的 实参 (Argument)。它们用于在函数体内引用传递给函数的值。.

共享调用 (Call by Sharing)

共享调用 (Call by Sharing) 指函数 形参 (Parameter) 获得 实参 (Argument) 引用的副本。形参 (Parameter) 与 实参 (Argument) 都引用同一对象。.

最少惊讶原则 (Principle of Least Astonishment)

“最少惊讶原则 (Principle of Least Astonishment)” 是指设计接口时应尽量避免令人惊讶或困惑的行为, 使用户能够根据直觉和常识来使用接口, 而不需要进行复杂的推理或猜测。.

回调函数 (Callback Function)

回调函数 (Callback Function) 是作为参数传递给另一个函数的函数。当特定事件或条件发生时, 该回调函数会被调用, 以执行特定的操作或处理。.

弱引用 (Weak Reference)

弱引用 (Weak Reference) 是一种特殊的引用。它可以引用一个对象, 但不会增加该对象的引用计数。当所有其他强引用都被解除或销毁时, 弱引用对象可能成为无效的, 并且所引用的对象可能被垃圾回收。弱引用的主要目的是帮助解决循环引用的问题, 防止内存泄漏。.

列表推导式 (Listcomp)

列表推导式是一种简洁的语法形式, 用于通过一种简单的方式生成新列表, 它由一个表达式和一个可迭代的对象组成。.

生成器表达式 (Genexp)

生成器表达式是一种类似于列表推导式 (Listcomp) 的语法形式, 但生成器表达式返回的是一个生成器对象而不是一个列表, 这样可以节省内存并实现惰性计算。.

记忆化 (Memoization)

记忆化 (Memoization) 是计算机科学中一种常用的优化技术, 其目的是通过缓存函数的计算结果, 避免重复执行耗时的计算操作。.

仅限关键字参数 (Keyword-only Argument)

仅限关键字参数 (Keyword-only Argument) 由 PEP 3102 -Keyword-Only Arguments 引入, 是指在函数定义中, 使用了 “*” 后, 参数只能通过关键字参数的形式进行传入, 而不能通过位置参数的形式进行传入。这样可以增强函数的可读性和可维护性。如 `def func1(a,*b)` 与 `def func2(*c,d)` 中的参数 `b`、`c`、`d` 都是 “仅限关键字参数”, 而参数 `a` 是 “位置参数”。

仅限位置参数 (Positional-only Parameter)

仅限位置参数 (Positional-only Parameter) 由 PEP 570 -Python Positional-Only Parameters 引入, 是指在 Python 函数定义中, 通过在参数列表中使用斜杠 “/”, 可以将某些参数限制为只能通过位置参数传入而不能通过关键字参数传入。这样可以确保函数参数的顺序性和一致性。

林奈层次结构 (Linnaean Hierarchies)

林奈层次结构 (Linnaean Hierarchies) 是指将编程语言中的各种元素 (如关键字、数据类型、函数等) 根据特性和层次关系进行分类和命名的层次结构体系。

尾部调用消除 (Tail-call Elimination)

尾部调用消除 (Tail-call Elimination) 是 Python 中的一种内存优化技术, 它在函数的最后一个操作是函数调用时 (如函数递归), 会尝试将函数调用转换为循环, 避免函数调用的堆栈增长, 以减少函数调用的开销。

码点 (Code Point)

Unicode 码点 (Code Point) 是指每个字符在 Unicode 编码表中所对应的唯一数字标识, 用于唯一标识字符集中的每个字符。

机器字 (Machine Word)

机器字 (Machine Word) 是指计算机处理数据的最小单位。它可以是 8 位、16 位、32 位或 64 位, 具体取决于计算机的体系结构。

渐进式类型系统 (Gradual Type System)

渐进式类型系统 (Gradual Type System) 是指允许在 Python 中使用静态类型检查的一种功能, 可以在保持动态类型特性的同时, 提供类型注释和类型检查的选项。

鸭子类型 (Duck Typing)

鸭子类型 (Duck Typing) 是一种动态类型机制, 它关注对象的行为 (方法与属性), 而不是对象的类型。在鸭子类型中, 只要一个对象具有特定的行为 (即方法或属性), 即可在代码中使用该对象, 而无须继承特定的接口或基类。实际上, 鸭子类型 (Duck Typing) 的思想是: “如果一只动物走路外八字、拥有扁嘴巴、叫起来像鸭子, 那么它就是鸭子。”。

静态类型 (Static Typing)

Python 中的静态类型 (Static Typing) 指在代码编写阶段通过 [类型提示 \(Type Hints \)](#) 来声明变量和函数参数的类型, 并借助外部工具 (如 MyPy 或 PyCharm) 进行类型检查, 以提前发现类型相关的错误和提高代码可读性。

静态鸭子类型 (Static Duck Typing)

Python 通过 “[PEP 544 -Protocols: Structural subtyping \(static duck typing\)](#)” 引入的 [Protocol 类](#), 实现了 “静态鸭子类型 (Static Duck Typing)” 这种程序设计理念。此种理念结合了静态类型检查和 [鸭子类](#)

型 (Duck Typing) 的动态特性。在静态鸭子类型中,开发者可以使用静态类型检查器来验证代码是否符合特定的协议或接口,同时可以根据对象的行为进行灵活的编程。.

大鹅类型 (Goose Typing)

大鹅类型 (Goose Typing) 指的是在类型提示 (Type Hints) 中使用泛型 (Generic Type) 来指定参数类型或返回值类型的一种类型实现方式。.

名义类型 (Nominal Typing)

名义类型 (Nominal Typing) 也是一种编程风格。与鸭子类型 (Duck Typing) 不同的是,名义类型 (Nominal Typing) 强调对象的类型,而不注重行为与属性。要求对象显式声明其类型,并限制只能操作相同类型的对象。.

泛化容器 (Generic Collections)

泛化容器 (Generic Collections) 是指一组容器 (Collection) 类型的数据结构,包括序列 (sequences) 和映射 (mappings),如元组 (tuple)、列表 (list)、集合 (set)、字典 (dict) 等。泛化容器 (Generic Collections) 可使用类型提示 (Type Hints) 来指定容器 (Collection) 中元素的类型,这样可以提供更好的类型安全性和代码可读性。.

子类型 (Subtype-of)

子类型 (Subtype-of) 是一种描述类型关系的概念。如果一个类型 A 是类型 B 的子类型,那么可以将一个属于类型 A 的对象赋值给类型 B 的变量。这被称为“Subtype Assignment”。在面向对象的编程语言中,通过 subtype-of 关系,可以实现一些重要的概念,如多态 (polymorphism) 和继承 (inheritance)。子类型可以继承父类型的属性和行为,并且可以在父类型的基础上扩展和重写。.

里氏替换原则 (Liskov Substitution Principle)

里氏替换原则 (Liskov Substitution Principle, LSP) 是面向对象设计中的原则,指子类型必须能够完全替代父类型,保持程序行为的正确性。.

行为子类型 (Behavioral Subtyping)

行为子类型 (Behavioral Subtyping) 要求子类型对象能够完全替代父类型对象,并且在任何使用父类型的地方都能够正确运行,并产生与预期相同的结果。.

相容 (Consistent-with)

“相容 (Consistent-with)” 是渐进式类型系统 (Gradual Type System) 中类型转换的一种保护机制,它要求显式确认类型转换以确保类型之间的兼容性,并确保转换的安全性。.

Geohash

Geohash 是一种地理编码系统,将地球上的经纬度坐标转化为短字符串,以方便存储、传输和检索。.

海象运算符 (:=)

海象运算符 (:=) 是 Python 3.8 引入的一种语法特性,用于在表达式中同时进行赋值和判断。例如,“if name := unicodedata.name(char, ”) 会将 unicodedata.name(char, ”) 的结果赋值给变量 name,同时会将此结果作为 if 语句的判断条件。.

协议 (protocol)

Python 中的协议 (protocol) 是一种非正式的接口概念, 用于定义对象应该具有的方法和属性, 以实现特定的行为或功能。.

高阶函数

高阶函数是指可以接收其他函数作为参数, 或返回其他函数作为结果的函数。.

lambda 函数

lambda 函数, 又称“匿名函数”, 是用 `lambda` 关键字定义的一次性函数, 如同一次性物品一样, 即用即扔。示例 `lambda num1, num2: num1+num2`, 其中冒号 (:) 前的变量为匿名函数的参数, 多个参数用逗号 (,) 分隔。冒号后为匿名函数的返回值表达式 (只能是单个表达式, 不能包含多个表达式或语句)。匿名函数可能无法实现多步骤的复杂逻辑, 即使可以实现, 代码可读性也很差。因此, 匿名函数仅适用于简单的场景。匿名函数特别适用于一次性使用的场景, 除了可用作高阶函数的参数 (如 `calc(7, lambda num: num*5)`); 也可以直接被调用, 例如 `(lambda num1, num2: num1+num2)(2,3)`, 其中 2 与 3 为匿名函数的参数。.

存根 (stub) 文件

存根文件是用来描述模块或库的类型注解和 [类型提示 \(Type Hints \)](#) 的文件。存根文件通常以.pyi 为后缀, 它包含了函数、类、方法等的签名信息, 但没有实际的实现代码。存根文件可以帮助开发者在静态类型检查或 IDE 提示时提供准确的类型信息, 使代码更加可读、可维护。.

泛化类 (Generic Classes)

泛化类 (Generic Classes) 是指能够以一种通用的方式处理不同类型数据的类。泛化类可以通过 [参数化 \(Parameterized \)](#) 类型来实现, 这样一来, 我们就可以在创建实例时指定要处理的 [具体类型 \(Concrete Type \)](#)。.

参数化 (Parameterized) 类型

Python 中的参数化 (Parameterized) 类型允许使用参数来定义类型, 以处理不同类型的数据。例如, `List[int]` 表示整数列表, `Tuple[str, int]` 表示包含字符串和整数的元组。.

型变 (variance)

型变 (variance) 是指 [泛型 \(Generic Type \)](#) 类型参数的 [协变 \(covariance \)](#) 性和 [逆变 \(contravariance \)](#) 性。[协变 \(covariance \)](#) 是指一个类型参数可以被替换为它的子类型, 而 [逆变 \(contravariance \)](#) 是指一个类型参数可以被替换为它的超类型。.

协变 (covariance)

协变 (covariance) 表示子类型可以替换父类型。在协变场景下, 可以将一个子类型的对象赋值给父类型的对象。这意味着可以将一个返回子类型对象的函数赋值给返回父类型对象的函数, 或者将一个子类型的实例传递给一个期望父类型的参数。.

逆变 (contravariance)

逆变 (contravariance) 表示父类型可以替换子类型。在逆变场景下, 可以将一个超类型的对象赋值给子类型的对象。这意味着可以将一个返回父类型对象的函数赋值给返回子类型对象的函数, 或者将一个超类型的实例传递给一个期望子类型的参数。.

闭包 (closures)

在一个函数 (称为外部函数 `f_out`) 内定义另一个函数 (称为内部函数 `f_in`)。若内部函数 `f_in` 引用了外部函数 `f_out` 中的局部变量, 并且外部函数 `f_out` 的返回值为内部函数 `f_in`, 那么内部函数 `f_in` 就被认为 是“闭包 (closures)”。

自由变量 (Free Variable)

自由变量 (Free Variable) 是一个技术术语, 指未在局部作用域中绑定 (只有名称, 没有值) 的变量。自由变量会被保存在闭包 (closures) 中。

泛化函数 (Generic Function)

泛化函数 (Generic Function) 是一组函数, 这组函数会根据第一个实参 (Argument) 的类型, 以不同方式执行相同的操作。

特化函数 (Specialized Function)

特化函数 (Specialized Function) 又称作“专门函数”或“泛化函数 (Generic Function) 的实现”, 是针对特定数据类型或条件 (如参数) 而设计的函数, 用于执行定制化的操作或处理。

内省 (Introspection)

Python 中的内省 (Introspection) 是指通过检查对象的属性和方法来获取有关对象的信息的过程。

猴子补丁 (Monkey-patch)

猴子补丁 (Monkey-patch) 是指在运行时修改或替换 Python 中已有的代码或功能。

读值 (getter)

读值 (getter) 方法是用 `@property` 装饰器 标记的方法。可用于获取属性的值, 并可将读值 (getter) 方法作为普通属性来访问。例如, 实例方法 `A.balance()` 被 `@property` 装饰后, 通过 `A.balance` 即可得到 `A.balance()` 的返回值。

设值 (setter)

设值 (setter) 方法是一种特殊的方法, 在 Python 中用来设置类的属性值, 在属性赋值时执行自定义逻辑或验证操作。

删值 (deleter)

删值 (deleter) 方法是一种特殊方法, 用于在删除实例属性时执行特定的操作。当使用 `del` 关键字删除实例属性时, 如果该属性有一个与之相关联的删值方法, 则该方法会被调用。这允许程序员在属性被删除时执行自定义的清理或其他操作。

特性 (property)

实例中的属性 (attributes) 是对象所拥有的数据, 可以是变量、方法或其他的对象。而 `property` 是一种特殊的属性 (为了以示区分, 简称“特性”), 通过 `@property` 装饰器 来定义, 它可以通过 `getter` 和 `setter` 方法控制对属性 (attributes) 的访问和赋值操作。

身份值 (Identity Value)

在 map-reduce 中, 当使用 `reduce` 操作时, 需要提供一个初始值 (initializer) 来开始 `reduce` 过程。这个初始值就是身份值 (Identity Value), 它在 `reduce` 操作中起到了一个标识初始状态的作用, 保证了

reduce 操作的正确性和准确性。例如,对于 +、|、^ 来说,初始值 (initializer) 应该是 0,但对于 *、& 来说,初始值 (initializer) 应该是 1。.

继承图 (inheritance graph)

继承图是指表示类之间继承关系的一种结构,用于显示类的层次结构和继承关系。.

MRO (方法解析顺序)

MRO (Method Resolution Order, 方法解析顺序) 是用于确定在多重继承情况下,Python 中类的方法调用顺序的规则。.

协作方法 (Cooperative Method)

协作方法 (Cooperative Method) 是指类之间通过调用彼此的方法来共同协作实现功能。而 super() 函数则用于在子类中调用父类的方法,实现协作方法中的继承与重写。.

协作多重继承 (Cooperative Multiple Inheritance)

协作多重继承 (Cooperative Multiple Inheritance) 指多个类之间通过继承关系共同合作实现功能,可以利用 super() 函数解决继承中的 MRO (方法解析顺序) 问题。.

EAFP 原则

EAFP 原则,表示 “it's Easier to Ask for Forgiveness than Permission.” (请求原谅比请求许可更容易)。详见 “[Python 官方术语表](#)”。.

数字塔 (Numeric Tower)

数字塔 (Numeric Tower) 是指在编程语言中支持数字类型继承和操作的体系结构。数字塔将数学运算符和方法按照优先级进行组织,以确保数值运算的正确性和一致性。在 Python 中,数字塔通常涉及不同的数字类型 (如整数、浮点数、复数等) 之间的继承关系和操作规则,从而构建了一个完整的数值计算体系。.

混合 (Mixin) 方法

抽象基类 (ABCs) 中实现的具体方法 (Concrete Method),被称为 “混合 (Mixin) 方法”。.

KISS 原则

KISS 原则,即 “保持简单而愚蠢 (Keep It Simple, Stupid)”,强调简洁而有效的设计和实现,避免过度复杂和不必要的功能。.

形式 (Formal) 类型参数

形式类型参数 (formal type parameters) 指的是泛化类 (Generic Classes) 中声明的形式 (formal) 类型参数。它们允许我们在泛化类中定义一个占位符,例如,Iterable[T] 表示可迭代对象中的元素类型 T 将在实例化时确定。使用 T 表示这个形式 (formal) 类型参数。通过声明形式 (formal) 类型参数,我们可以在泛化类 (Generic Classes) 中编写通用的代码,以便在实例化时使用不同的具体类型。.

实际 (Actual) 类型参数

实际 (Actual) 参数是指在声明参数化 (Parameterized) 类型时,为参数变量提供的类型。例如,Lotto-Blower[int] 中的 int,以及 abc.Mapping[str, float] 中的 str 与 float。.

协程 (Coroutine)

协程 (Coroutine) 是一种能够暂停并恢复执行的函数, 可用于实现非抢占式的多任务处理。.

驱动协程

在 Python 中, 协程 (Coroutine) 自身并不会自动开始执行, 它们需要被显式地启动和管理。“驱动协程 (Driving the coroutine) ” 则是指启动、管理和调度其他协程执行。驱动协程就像是协程的指挥官, 控制着协程的生命周期, 确保它们能够在合适的时机得到执行, 并且能够处理它们之间的协作与通信。.

易并行 (Embarrassingly Parallel)

在编写并行程序过程中, 首先需要将一个计算问题分解成若干子任务, 然后将每个子任务分配给不同的进程或线程分别计算。如果该计算问题能够被分解成一些完全独立的子任务、同时各个子任务之间数据几乎没有依赖, 没有通信。那这个计算问题就叫作 “易并行 (Embarrassingly Parallel) ” 问题。.

非易并发 (Nonembarrassing Concurrency)

Nonembarrassing Concurrency 直译为 “非尴尬并发”, 应译为 “非易并发”, 指的是那些不容易并行化的任务, 这些任务不能简单地分解为独立的子任务, 并行执行。可能需要更复杂的并发处理技术 (如线程间通信或协调), 而不仅仅是简单的线程和锁。.

信号量 (Semaphore)

信号量 (Semaphore) 是一种变量或抽象数据类型, 用于控制多个线程对公共资源的访问, 并避免并发系统 (如多任务操作系统) 中的 **关键区域** 问题。信号量是一种同步原语。普通信号量是根据程序员定义的条件进行更改 (如, 递增或递减或切换) 的普通变量。.

异步生成器

异步生成器是一种特殊生成器, 允许使用 `async for` 语法按需生成异步序列中的值。要定义一个异步生成器, 可以编写一个类, 并实现特殊方法 `__anext__` 与 `__aiter__`。不过还有更简单的实现方法: 用 `async def` 语法定义一个异步函数, 并在其中使用 `yield` 语句来产生值。.

异步上下文管理器

异步上下文管理器是一种特殊类型的上下文管理器, 用于管理异步环境中资源的获取和释放。要定义异步上下文管理器, 可以创建一个实现了 `__aenter__`/`__aexit__` 协程方法的类。`__aenter__` 方法用于进入异步上下文时执行的操作, `__aexit__` 方法用于退出异步上下文时执行的操作。.

协程方法

协程方法是 Python 异步编程的一种方式, 能够使得 I/O 操作更加高效, 提高程序执行效率。定义协程方法时需要使用 `async def` 关键字, 使用 `await` 关键字来等待异步操作或协程的完成。.

原生协程

原生 (Native) 协程, 又称 “异步函数”, 是指用 `async def` 定义的 协程 (Coroutine) 函数。可以用 `await` 关键字——类似于经典协程中的 “`yield from`”, 将一个原生协程委托给另一个原生协程。`async def` 语句 **定义的始终是原生协程, 即使协程主体中没有使用 await 关键字**。在原生协程之外, 不能使用 `await` 关键字。.

经典 (Classic) 协程

一个生成器函数, 通过 `my_coro.send(data)` 调用消耗发送给它的数据, 并使用 `yield` 表达式读取该数据。一个经典协程可以使用 `yield from` 委派给其他经典协程。经典协程无法由 `await` 驱动, 并且不再受 `asyncio` 支持。.

结构化并发

结构化并发是一种编程范式, 旨在通过使用结构化方法进行并发编程来提高计算机程序的清晰度、质量和开发时间。核心概念是通过控制流结构封装并发执行线程 (包括内核态和用户态线程和进程), 控制流结构具有明确的入口点和出口点, 并确保所有生成的线程在退出之前已完成。.

描述符 (Descriptor) 类

指实现了描述符协议的类。如 “图 23.1<698页>” 中的 `Quantity` 类。.

覆盖型描述符 (Overriding Descriptor)

覆盖型描述符是指实现了特殊方法 `__set__` 的描述符 (Descriptor) 类。.

非覆盖型描述符 (Nonoverriding Descriptor)

非覆盖型描述符是指未实现特殊方法 `__set__` 的描述符 (Descriptor) 类。.

托管 (Managed) 类

将描述符 (Descriptor) 实例声明为类属性的类。如 “图 23.1<698页>” 中的 `LineItem` 类。.

描述符 (Descriptor) 实例

指描述符 (Descriptor) 类的实例。每个描述符实例都被声明为托管 (Managed) 类的类属性。在 “图 23.1<698页>” 中, 每个描述符 (Descriptor) 实例都用带下划线名称与箭头组合表示 (下划线在 UML 中表示类属性)。黑色菱形 (◆) 指向包含描述符实例的 `LineItem` 类。.

托管 (Managed) 实例

指托管 (Managed) 类的实例。在 “23.2.1 `LineItem` 类第 3 版: 一个简单的描述符<698页>” 的示例中, `LineItem` 实例就是托管实例 (“图 23.1<698页>” 中未显示)。.

存储 (Storage) 属性

指托管 (Managed) 实例的属性, 用于保存特定实例的托管 (Managed) 属性值。在 “图 23.1<698页>” 中, `LineItem` 实例属性 `weight` 和 `price` 都是存储 (Storage) 属性。它们与描述符 (Descriptor) 实例不同, 后者始终是类属性。.

托管 (Managed) 属性

指托管 (Managed) 类中的公开属性, 由描述符 (Descriptor) 实例处理, 其值存储在存储 (Storage) 属性中。换句话说, 描述符 (Descriptor) 实例和存储 (Storage) 属性为托管 (Managed) 属性提供了基础架构。.

二元性 (Duality)

函数与类的二元性指的是在编程语言中函数 (function) 和类 (class) 之间的对应关系或等效性。具体来说, 它指的是一种概念上的对称性或对应性, 即在某些情况下, 函数可以用类来表达, 反之亦然。.

元编程 (Meta-Programming)

Python 元编程 (Meta-Programming) 是一种在程序运行时动态创建和修改代码的技术, 它允许程序员操作和生成代码以实现更灵活和高效的程序行为。.

类元编程 (Class Metaprogramming)

类元编程 (Class Metaprogramming) 是指在运行时使用编程语言的特性来操作类和对象, 允许程序在运行时动态地创建、修改和操纵类和对象的结构和行为, 或者在运行时进行类的继承和组合等操作。.

Pythonic

Pythonic 是指 Python 语言的一种编程风格, 指的是代码编写风格符合 Python 社区推崇的简洁、优雅、直观和易读的特点。遵循 Pythonic 风格的代码通常更容易理解和维护, 同时也更符合 Python 语言的设计原则。.

ABC 语言

ABC 语言是由荷兰计算机科学家 Edsger W. Dijkstra 等人在 1980 年代初开发的编程语言。其设计目标是简单性和易学性, 旨在提供一种易于理解和使用的高级编程语言, 特别适合教学和初学者。.

A.2 图片清单

1.1 二维向量加法图示	10
1.2 基本容器类型的 UML 类图	13
2.1 元组与数组的内存简图	20
2.2 部分抽象基类的简化 UML 类图	21
2.3 扑克牌点数与花色的笛卡尔积组合	24
2.4 元组引用可变对象	28
2.5 元组增量赋值谜题图解	48
2.6 float 值构成的数组	50
2.7 通过 memoryview 修改内存中的字节	55
3.1 MutableMapping 及其超类的 UML 类图	70
3.2 MutableSet 及其超类的 UML 类图	89
4.1 部分特殊字符的字节表示	100
4.2 Unicode 三明治理论	106
4.3 PowerShell 上运行 stdout_check.py	111
4.4 运行 stdout_check.py 并输出到文件	111
4.5 在 Python 控制台中探索 unicodedata.name()	121
4.6 使用 cf.py 查找微笑的猫	122
4.7 macOS 终端显示数字字符	124
4.8 rammanujan.py 脚本结果截图	125
6.1 将变量视为便利贴,而非盒子	164
6.2 相等性与同一性示意图	166
6.3 执行 l2=list(l1) 赋值	170
6.4 l1 和 l2 的最终状态	171
7.1 factorial 的帮助屏幕	189
9.1 averager 函数的闭包	255
10.1 策略模式实现的订单的 UML 类图	278
10.2 命令设计模式的 UML 类图	288
11.1 开关盒子是一种保护措施	311

12.1 归约函数聚合并计算思路	332
12.2 map-reduce 计算思路	334
13.1 Python 的 4 种类型实现方式	350
13.2 Sequence 相关抽象基类的 UML 类图	353
13.3 MutableSequence 相关抽象基类的 UML 类图	362
13.4 collections.abc 模块中抽象基类的 UML 类图	364
13.5 1 个抽象基类与 3 个子类的 UML 类图	366
13.6 Exception 类的层次结构 ¹	368
13.7 虚拟子类 TomboList 的 UML 类图	372
14.1 协作方法的唤醒序列	398
14.2 Tkinter Text 类的 UML 图及 MRO	402
14.3 django.views.generic.base 模块的 UML 类图	407
14.4 django.views.generic.list 模块的 UML 类图	408
14.5 Tkinter GUI 类层次结构的汇总 UML 图	409
16.1 用 __add__ 与 __radd__ 计算 a+b 的流程图	458
17.1 抽象基类 Iterable 与 Iterator	485
18.1 Scheme lambda 表达式	543
19.1 spinner_thread.py 与 spinner_async.py 的输出	566
19.2 用不同进程数执行 procs.py 的耗时中位数	582
19.3 由多个组件构成的系统可能采用的架构 ²	586
19.4 典型的 WSGI 部署	587
20.1 左上:flags2_threadpool.py 运行中,显示 tqdm 包生成的进度条右下:同一个终端窗口,脚本运行完毕后 .	609
21.1 异步程序中的协程调度	626
21.2 在 web_mojifinder.py 中搜索 mountain 的结果	637
21.3 为/search 端点自动生成的 OpenAPI 模式	638
21.4 TCP 服务器 telnet 会话	641
22.1 Python 控制台的 help 截屏	686
23.1 LineItem 使用 Quantity 描述符类的 UML 类图 ³	698
23.2 用 MGN (Mills & Gizmos Notation) 注释的 UML 类图 ⁴	699
23.3 发出 help(LineItem.weight) 和 help(LineItem) 命令时的 Python 控制台截图	715
24.1 元类 (MetaClass) 是用于构建类的类	738
24.2 元类的 2 种示意图	738
24.3 type、ABCMeta、object、Iterable 之间的关系图	739

24.4 用 MGN 注解的 UML 类图。CheckedMeta 元工坊构建了 Movie 工坊。Field 工坊构建了 title、year 与 box_office 描述符, 它们是 Movie 的类属性。各个实例的字段数据存储在 Movie 的实例属性 _title、_year 与 _box_office 中。注意 checkedlib 包的边界。Movie 的开发者无需了解 checkedlib.py 中的所有机制。 747

A.3 表格清单

1.1 特殊方法名称(不含运算符、核心数学函数)	14
1.2 运算符的符号及背后的特殊方法	15
2.1 列表和元组的方法及属性	29
2.2 一些 Scheme 语法形式和处理语法的 case 序列模式	40
2.3 list 与 array 的方法与属性	52
2.4 list 与 deque 实现的方法	58
3.1 映射类型实现的方法	72
3.2 集合的数学运算符(要么生成一个新集合,要么就地更新可变集合。)	89
3.3 返回布尔值的集合比较运算符和方法	90
3.4 集合的其他实用方法	90
3.5 frozenset、dict_keys、dict_items 实现的方法	91
5.1 三个数据类构建器的功能比较	135
5.2 @dataclass 装饰器可接受的关键字参数	145
5.3 field 函数可接受的关键字参数	147
8.1 部分容器类型及对应的类型提示	221
15.1 涉及类型提示(Type Hints)的 PEP ⁵	447
16.1 中缀运算符与方法名	465
16.2 丰富的比较运算符:若正向方法返回 NotImplemented,就调用反向方法。	465
17.1 用于筛选的生成器函数)	500
17.2 用于映射的生成器函数	501
17.3 用于合并多个可迭代对象的生成器函数)	502
17.4 将输入的各项扩充为多个输出项的生成器函数	505
17.5 用于重新排列项(元素)的生成器函数	507
17.6 读取可迭代对象,并返回单个值的内置函数	508
21.1 异步生成器与原生协程对比	651
21.2 现代计算机从不同设备读取数据的延迟 ⁶	656

A.4 代码清单

1.1 一副有序的纸牌	5
1.2 一个简单的二维向量类	10
2.1 用 for 循环, 基于字符串构建一个 Unicode 码点列表	22
2.2 用列表推导式, 基于字符串构建一个 Unicode 码点列表	22
2.3 用列表推导式和 map/filter 构建列表比较	23
2.4 使用列表推导式生成笛卡尔积	24
2.5 用生成器表达式构建元组和数组	25
2.6 用生成器表达式生成笛卡尔乘积	26
2.7 将元组当作记录使用	26
2.8 02-array-seq/metro_lat_lon.py:解包嵌套元组, 获取经纬度	32
2.9 虚构的 Robot 类中的方法	34
2.10 02-array-seq/match_lat_lon.py 要求 Python 3.10+	34
2.11 lispy/py3.9/lis.py: 不使用 match/case 匹配模式	38
2.12 lispy/py3.10/lis.py: 使用 match/case 匹配模式 (要求 Python 3.10 及以上)	38
2.13 从纯文本形式的发票中, 提取商品信息	42
2.14 列表推导式: 一个列表中嵌套 3 个长度为 3 的子列表	44
2.15 错误做法: 在一个列表中 3 次引用同一个列表对象	45
2.16 一个增量赋值的谜题	47
2.17 增量赋值谜题的执行结果	47
2.18 增量赋值谜题的执行结果	47
2.19 创建、保存、加载一个大型浮点型数组	51
2.20 分别以 1x6, 2x3, 3x2 矩阵视图, 处理 6 字节内存	53
2.21 修改一个 16 位整数数组中某一项的字节, 以改变该项的值。	54
2.22 numpy.ndarray 中行与列的基本操作	55
2.23 处理一个 deque 对象	57
3.1 字典推导式示例	66
3.2 **解包映射	67
3.3 dict 字面量中的 **	67
3.4 用 合并映射	67
3.5 用 = 合并映射	68
3.6 creator.py:get_creators() 函数从出版物记录中提取创作者姓名	68
3.7 测试 creator.py:get_creators() 函数	69

3.8 将多出的 Key:Value 对捕获到一个 dict 中	69
3.9 isinstance 测试	70
3.10 可哈希的 frozenset 与 tuple	71
3.11 用示例 3.12 的脚本, 处理《Python 之禅》的部分输出	73
3.12 index0.py: 用 dict.get 获取并更新单词出现的位置列表	73
3.13 用 dict.setdefault 获取字典的值, 并更新	74
3.14 index_default.py: 用 defaultdict 类代替 setdefault 方法	75
3.15 搜索非字符串键时, StrKeyDict0 将未找到的键转换为字符串	76
3.16 StrKeyDict0 在查找键时, 将非字符串键转换为字符串	77
3.17 在 ChainMap 的多个映射中查找键	79
3.18 更新 ChainMap	79
3.19 ChainMap 模拟 Python 变量查找规则	80
3.20 用 Counter 统计单词中的字母数量	80
3.21 StrKeyDict 在插入、更新和查找时, 始终将非字符串键转换为 str 字符串	81
3.22 MappingProxyType 根据 dict 对象构建只读的 mappingproxy 实例	83
3.23 .values() 方法返回 dict 对象的值视图	83
3.24 字典视图可反映字典的变化	84
3.25 不能手动创建字典视图的实例	84
3.26 set(集合) 的作用	85
3.27 用 dict 去除重复项, 并保留每个项首次出现的位置顺序	86
3.28 统计 needles 中 E-Mail 在 haystack 中出现的次数(二者均为集合类型)	86
3.29 统计 needles 中 E-Mail 在 haystack 中出现的次数(效果与示例 3.28 一致)	86
3.30 统计 needles 中 E-Mail 在 haystack 中出现的次数(支持任何可迭代类型)	86
3.31 构建一个集合, 元素为 Unicode 名称中含有 'SIGN' 的 Latin-1 字符	88
3.32 用 & 获取两个字典中都有的键	91
3.33 字典视图的集合元素符与 set 实例兼容	91
4.1 编码与解码	96
4.2 包含 5 个字节的 bytes 与 bytearray	97
4.3 用数组中的原始数据构建 bytes 对象	99
4.4 用 3 种不同的编解码器, 编码同一段文本	99
4.5 编码到字节, 成功与错误处理	101
4.6 将字节序列解码为 str(有些成功, 有些需要错误处理器)	102
4.7 ola.py: 打印葡萄牙语 "Hello, World!"	103
4.8 一个平台的编码问题	106
4.9 仔细分析在 Windows 中运行的示例 4.8, 找出问题并修正	107
4.10 探索编码默认设置	108
4.11 Windows 10 PowerShell 上的默认编码(cmd.exe 上的输出相同)	109
4.12 stdout_check.py	110
4.13 normeq.py: 规范化 Unicode 字符串比较	115
4.14 simplify.py: 去除所有组合标记的函数	116

4.15	示例 4.14 中函数 <code>shave_marks</code> 的使用示例	116
4.16	从拉丁字符中删除组合标记的函数 ⁷	117
4.17	将一些西方文字符号转换为 ASCII 码 ⁸	118
4.18	示例 4.17 中函数 <code>asciize</code> 的使用示例	118
4.19	<code>locale_sort.py</code> : 使用 <code>locale.strxfrm</code> 函数作为排序键	119
4.20	使用 <code>pyuca.Collator.sort_key</code> 方法	120
4.21	<code>cf.py</code> : 字符查找工具	122
4.22	Unicode 数据库中数字字符的元数据示例	123
4.23	<code>ramanujan.py</code> : 比较简单 <code>str</code> 和 <code>bytes</code> 正则表达式的行为	124
4.24	分别将 <code>str</code> 参数、 <code>bytes</code> 参数传入 <code>listdir</code> , 并查看结果	126
5.1	<code>class/coordinates.py</code>	132
5.2	简单类的功能粗糙	132
5.3	用 <code>namedtuple</code> 构建 <code>Coordinate</code> 类	133
5.4	<code>typing.NamedTuple</code> 为字段添加了类型提示	133
5.5	<code>typing_namedtuple/coordinates.py</code>	134
5.6	<code>dataclass/coordinates.py</code>	134
5.7	定义与使用一个具名元组	136
5.8	具名元组最有用的属性及方法 (续示例 5.7)	137
5.9	构建具名元组, 为字段指定默认值 (续示例 5.8)	138
5.10	<code>typing_namedtuple/coordinates2.py</code>	139
5.11	Python 在运行时, 不强制执行类型提示检查	140
5.12	<code>eaning/demo_plain.py</code> : 带有类型提示的简单类	141
5.13	<code>meaning/demo_nt.py</code> : 用 <code>typing.NamedTuple</code> 构建的数据类	142
5.14	<code>meaning/demo_dc.py</code> : <code>dataclass</code> 装饰器构建的类	143
5.15	<code>dataclass/club_wrong.py</code> : 此数据类将引发 <code>ValueError</code>	146
5.16	<code>ataclass/club.py</code> : 更正示例 5.15 中的 <code>ClubMember</code> 类	146
5.17	<code>ataclass/club_generic.py</code> : 这个 <code>ClubMember</code> 的定义更精确	147
5.18	构建数据类 <code>ClubMember</code>	148
5.19	<code>dataclass/hackerclub.py</code> : <code>HackerClubMember</code> 的 <code>doctest</code>	148
5.20	<code>dataclass/hackerclub.py</code> : 构建 <code>HackerClubMember</code> 类	149
5.21	<code>dataclasses</code> 模块文档中的示例	151
5.22	<code>dataclass/resource.py</code> : <code>Resource</code> 类的代码, 一个基于 Dublin Core 术语的类	152
5.23	<code>dataclass/resource.py</code> : 使用 <code>Resource</code> 类	152
5.24	<code>repr(book)</code> 输出格式	153
5.25	<code>dataclass/resource_repr.py</code> : 在 <code>Resource</code> 类中定制 <code>__repr__</code> 方法	153
5.26	简单类模式	156
5.27	关键字类模式: <code>City</code> 类与 5 个实例	157
5.28	类模式下, 捕获实例属性	157
5.29	模式变量的命名, 无任何限制	158
5.30	位置模式下, 捕获实例属性	158

6.1 变量 a 和 b 保存对同一列表的引用,而不是列表的副本	164
6.2 创建对象之后,才能绑定变量	165
6.3 charles 与 lewis 引用同一对象	165
6.4 经比较,alex 与 charles 相等,但 alex 不是 charles.	166
6.5 起初,t1 与 t2 相等。修改 t1 中一个可变项后,二者不等了。	168
6.6 对包含另一个 list 的 list 做浅拷贝。建议将代码复制到 Online Python Tutor 网站,查看效果。	169
6.7 示例 6.6 的输出	170
6.8 校车乘客在途中有上有下	171
6.9 copy 与 deepcopy 产生的不同效果	171
6.10 循环引用,b 引用 a,又将 b 追加到 a 中,deepcopy 会想办法复制 a。	172
6.11 函数可以更改它接收的可变对象	172
6.12 用一个简单的类,来说明可变默认值的危险	173
6.13 备受“幽灵乘客”折磨的校车	174
6.14 乘客从 TwilightBus 下车后,就消失了	175
6.15 一个简单的类,说明接收可变参数的风险	175
6.16 当对象的引用计数归零时,监控对象生命周期终结的情形	178
6.17 用一个元组构建另一个元组,其实得到的是同一个元组	179
6.18 字符串字面量可能会创建共享的对象	179
7.1 创建一个函数,并检查函数类型。	188
7.2 通过其他名称调用 factorial 函数,并将 factorial 作为参数传递	188
7.3 根据单词长度排序列表	189
7.4 按反向拼写对单词列表进行排序	189
7.5 列表推导式与高阶函数比较	190
7.6 分别用 reduce 与 sum 计算 0-99 之间的整数和	190
7.7 用 lambda 表达式改写示例 7.4(7.4 节)	191
7.8 bingocall.py:从乱序的列表中取出一个元素	193
7.9 函数 tag 用于生成 HTML 标签 ⁹	194
7.10 tag 函数(示例 7.9)的多种调用方式	195
7.11 * 后面的参数 b 是仅限关键字参数(Keyword-only Argument)	195
7.12 模拟 divmod 内置函数	196
7.13 模拟 divmod 内置函数	196
7.14 用 reduce 与 lambda 实现阶乘	197
7.15 用 reduce 与 mul 函数实现阶乘	197
7.16 用 itemgetter 排序一个元组列表 ¹⁰	197
7.17 用 attrgetter 处理示例 7.16 定义的元组 metro_data	198
7.18 methodcaller 示例	199
7.19 用 partial 进行函数改造	200
7.20 用 partial 构建一个便利的 Unicode 规范化函数	200
7.21 将 partial 应用到示例 7.9(7.7)节的 tag 函数上	200
8.1 来自 messages.py 的 show_count,未加类型提示	207

8.2 没有类型提示的 messages_test.py	208
8.3 hints_2/messages.py:有一个可选参数的 show_count 函数	210
8.4 birds.py	212
8.5 在 daffy.py 中试用一下 birds 模块 (示例 8.4)	213
8.6 woody.py	214
8.7 运行时的错误以及 Mypy 可以提供的帮助	214
8.8 带类型提示的 tokenize 函数 (Python ≥ 3.9)	220
8.9 带类型提示的 tokenize 函数 (Python ≥ 3.7)	221
8.10 带类型提示的 tokenize 函数 (Python ≥ 3.5)	221
8.11 coordinates.py:geohash 函数	223
8.12 coordinates_named.py:具名元祖 Coordinates 与 geohash 函数	223
8.13 columnize.py:返回一个列表,列表中的项是字符串元组	224
8.14 charindex.py	225
8.15 replacer.py	228
8.16 sample.py	229
8.17 mode_float.py:可操作 float 及其子类型的 mode 函数 ¹	230
8.18 mode_hashable.py:与示例 8.17类似,但签名更灵活	232
8.19 top 函数,有一个未定义的类型参数 T	233
8.20 comparable.py:定义协议类型 SupportsLessThan	234
8.21 top.py:用上边界为 SupportsLessThan 的 TypeVar 定义函数 top	235
8.22 top_test.py:top 函数测试套件的一部分	235
8.23 mypy top_test.py 的输出 (为便于阅读,将行数进行了拆分)	236
8.24 说明型变 (variance)	238
9.1 装饰器通常将传入的参数,替换为另一个函数	248
9.2 registration.py 模块	249
9.3 读取局部变量和全局变量的函数	251
9.4 b 是局部变量,因为在函数内为其赋值了	251
9.5 反汇编示例 9.3中的 f1 函数	253
9.6 反汇编示例 9.4中的 f2 函数	253
9.7 average_oo.py:一个计算累计平均值的类	254
9.8 average.py:一个计算累计平均值的高阶函数	254
9.9 测试示例 9.8	255
9.10 检查由示例 9.8 中 make_averager 创建的函数	256
9.11 继续示例 9.9	256
9.12 一个有缺陷的计算累计平均值的高阶函数,不保存所有历史值	256
9.13 计算累计平均值,不保存所有历史 (用 nonlocal 修正)	257
9.14 lockdeco0.py:一个显示函数运行时间的简单装饰器	258
9.15 使用装饰器 clock	258
9.16 clockdeco.py:经过改进的 clock 装饰器	260
9.17 生成第 n 个斐波那契数,递归非常耗时	261

9.18 用 @cache 实现示例 9.17,速度更快	262
9.19 htmlize() 为不同 Python 对象类型生成 HTML	264
9.20 用 @singledispatch 创建 @htmlize.register 装饰器,将多个函数绑定为一个泛化函数	265
9.21 “示例 9.2<249 页>” 中 registration.py 模块的删减版,再次给出以方便查看	268
9.22 为了接受参数,新的 register 装饰器必须作为函数调用	268
9.23 使用示例 9.22 中的 registration_param 模块	269
9.24 clockdeco_param.py 模块:参数化的 clock 装饰器	270
9.25 clockdeco_param_demo1.py	271
9.26 clockdeco_param_demo2.py	271
9.27 colckdeco_cls.py 模块:通过类实现的参数化装饰器 clock	272
10.1 实现 Order 类,支持可插入的折扣策略	279
10.2 应用不同促销活动的 Order 类使用示例	281
10.3 用函数实现折扣策略的 Order 类	282
10.4 以函数实现促销折扣的 Order 类使用示例	283
10.5 best_promo 函数计算所有折扣,并返回幅度最大的那一个	284
10.6 best_promo 在函数列表中遍历查找折扣最大的函数	285
10.7 通过检查模块全局命名空间来构建的 promos 列表	285
10.8 对 promotions 模块进行内省,构建 promos 列表	286
10.9 用装饰器 promotion 填充 promos 列表中的值	287
10.10 每个 MacroCommand 实例都在内部维护一个命令列表	289
11.1 Vector2d 实例的多种字符串表示形式	296
11.2 vector2d_v0.py: 目前定义的都是特殊方法	297
11.3 vector2d_v1.py 的一部分: 将被添加到 vector2d_v0.py (示例 11.2) 的 frombytes 类方法	299
11.4 比较 classmethod 与 staticmethod 的行为	299
11.5 Vector2d.__format__ 方法: 版本 1	302
11.6 Vector2d.__format__ 方法, 版本 2: 使用极坐标	302
11.7 vector2d_v3.py: 仅给出使 Vector2d 不可变所需的代码, 完整代码见 “示例 11.11<307 页>”	304
11.8 vector2d_v3.py: 实现 __hash__ 方法	305
11.9 Vector2d 实例的关键字模式匹配 (Python \geq 3.10)	305
11.10 Vector2d 实例的位置模式匹配 (Python \geq 3.10)	306
11.11 vector2d_v3.py: 完整版	307
11.12 私有属性的名称会被改写, 在前面添加 “_类名” 前缀	310
11.13 使用 __slots__ 的 Pixel 类	311
11.14 OpenPixel 是 Pixel 的子类	312
11.15 ColorPixel 是 Pixel 的另一个子类	313
11.16 vector2d_v3_slots.py: 只为 Vector2d 类增加 __slots__ 属性	313
11.17 mem_test.py 使用指定模块中定义的 Vector2d 类创建 10,000,000 个实例	314
11.18 设定原本从类继承的类属性 typecode, 自定义一个实例	315
11.19 ShortVector2d 是 Vector2d 的子类, 仅重写了默认的 typecode	316
11.20 confidential.java: 一个拥有名为 secret 的私有字段的 Java 类	320

11.21 expose.py: 用于读取另一个类中私有字段内容的 Python 代码	320
12.1 测试 Vector.__init__ 方法与 Vector.__repr__ 方法	322
12.2 vector_v1.py: 从 vector2d_v1.py 衍生而来	323
12.3 为方便起见, 此处转转“示例 1.1<5 页>”的代码	324
12.4 检查 __getitem__ 与切片的行为	326
12.5 检查 slice 类的属性	327
12.6 vector_v2.py 部分代码: 为 Vector(“示例 12.2<323 页>”) 添加 __len__ 与 __getitem__	328
12.7 测试示例 12.6 中增强的 __getitem__	328
12.8 vector_v3.py 部分代码: 向 Vector 类添加 __getattr__ 方法	329
12.9 不恰当的行为: 赋值给 v.x 不会引发错误, 但会导致前后不一致	330
12.10 vector_v3.py 部分代码: 在 Vector 类中实现 __setattr__ 方法	330
12.11 计算 0 到 5 整数累加异或(xor)的三种方法	332
12.12 vector_v4.py 部分代码: 在 vector_v3.py 基础上导入 2 个模块, 并增加 __hash__	333
12.13 Vector.__eq__ 的实现: 在 for 循环中使用 zip 函数, 以提高效率	335
12.14 用函数 zip 与 all 实现的 Vector.__eq__, 逻辑与示例 12.13 相同	335
12.15 内置函数 zip 的使用示例	335
12.16 vector_v5.py: Vector 类最终版的 doctest 与完整代码	337
13.1 用 __getitem__ 实现部分序列协议	351
13.2 一副有序的纸牌(同“示例 1.1<5 页>”)	353
13.3 random.shuffle 无法处理 FrenchDeck 实例	355
13.4 为 FrenchDeck 应用猴子补丁, 使其可变并与 random.shuffle 兼容(接续示例 13.3)	355
13.5 用鸭子类型处理字符串或由字符串构成的可迭代对象	357
13.6 frenchdeck2.py: FrenchDeck2 是 collections.MutableSequence 的子类	361
13.7 tombola.py: Tombola 是包含 2 个抽象方法与 2 个具体方法的抽象基类	366
13.8 假冒的 Tombola 不会被识破	368
13.9 bingo.py: BingoCage 是 Tombola 的具体子类	369
13.10 lotto.py: LotteryBlower 是 Tombola 的具体子类, 重写了继承的 inspect 与 loaded 方法	370
13.11 tombolist.py: TomboList 是 Tombola 的虚拟子类	372
13.12 Sized 类的定义(源码见 Lib/_collections_abc.py)	374
13.13 double_protocol.py: 用 Protocol 定义 double 函数	377
13.14 typing.SupportsComplex 协议源代码	378
13.15 vector2d_v4.py: 与 complex 相互转换的方法	381
13.16 vector2d_v5.py: 为当前研究的方法添加注解	382
13.17 randompick.py: 协议 RandomPicker 的定义	382
13.18 randompick_test.py: 使用协议 RandomPicker	383
13.19 randompickload.py: 扩展 RandomPicker 协议	385
14.1 重写的 __setitem__ 方法, 被 dict 的 __init__ 与 __update__ 忽略	396
14.2 AnswerDict 的 __getitem__ 被 dict.update 忽略	397
14.3 DoppelDict2 与 AnswerDict2 可按预期运行, 因为它继承自 UserDict	397
14.4 diamond.py: 图 14.1 中的 Leaf 类、A 类、B 类、Root 类	398

14.5 在 Leaf 对象上调用 ping 与 pong 方法	399
14.6 diamond2.py: 演示 super() 动态行为的类	401
14.7 tkinter.Text 的 MRO (方法解析顺序)	402
14.8 uppermixin.py: UpperCaseMixin 支持不区分大小写的映射	403
14.9 uppermixin.py: 使用 UpperCaseMixin 的两个类	404
14.10 Python 3.10 中 Lib/socketserver.py 文件的部分内容	405
15.1 mysum.py: 具有重载签名的 sum 函数的定义	418
15.2 mymax.py: 用 Python 重写 max 函数	420
15.3 mymax.py: 模块顶部, 包含导入、定义和重载	420
15.4 books.py: 定义 BookDict	424
15.5 使用 BookDict	424
15.6 demo_books.py: 对 BookDict 进行的合法和非法操作	425
15.7 类型检查 demo_books.py	426
15.8 books.py: to_xml 函数	426
15.9 books_any.py: from_json 函数	427
15.10 books.py: 带变量注解的 from_json 函数	427
15.11 demo_not_book.py: 返回一个无效的 BookDict, 但对 to_xml 有效	428
15.12 demo_not_book.py 的 Mypy 报告 (为便于阅读, 重新格式化)	429
15.13 运行 demo_not_book.py 的输出	429
15.14 clipannot.py: 函数 clip 带注释的签名	432
15.15 generic_lotto_demo.py: 使用一个泛化版的 LottoBlower 类	435
15.16 generic_lotto_errors.py: Mypy 报告的错误	435
15.17 generic_lotto.py: 泛化版的 LottoBlower 类	436
15.18 invariant.py: 类型定义和 install 函数	438
15.19 covariant.py: 类型定义和 install 函数	439
15.20 contravariant.py: 类型定义与 install 函数	441
15.21 abs_demo.py: 使用泛化协议 SupportsAbs,label=sec:lst:UseSupportsAbs	444
15.22 generic_randompick.py: 泛化协议 RandomPicker 的定义	445
16.1 vector_v6.py: 为“示例 1.2<10 页>”的 Vector 类增加一元运算符-与 +	454
16.2 算术上下文精度的变化, 可能导致 x 与 +x 不同	454
16.3 一元运算符 +, 将得到一个新的 Counter 实例 (不含计数为零与负值的项)	455
16.4 Vector.__add__ 方法 (第 1 版)	456
16.5 Vector.__add__ (第 1 版), 也支持 Vector 之外的对象	457
16.6 Vector.__add__ (第 1 版), 左操作数不是 Vector 的混合类型加法, 将失败	457
16.7 Vector 的 __add__ 与 __radd__ 方法	458
16.8 Vector.__add__ 方法需要一个可迭代的操作数	459
16.9 Vector.__add__ 方法需要一个包含数字的可迭代对象	459
16.10 vector_v6.py: 添加到 vector_v5.py (见“示例 12.16<337 页>”)的 + 运算符方法	460
16.11 vector_v7.py: 增加 * 运算符方法	461
16.12 vector_v7.py: 矩阵乘法 @ 运算符相关的方法	462

16.13 将 Vector 实例与 Vector 实例、Vector2d 实例、元组进行比较	466
16.14 vector_v8.py:Vector 类改进的 __eq__ 方法	466
16.15 与“示例 16.13<466 页>”一样的测试,最后一个结果有变化	467
16.16 用 += 与 *= 操作 Vector 实例	468
16.17 用运算符 +,新建 AddableBingoCage 实例	469
16.18 可以使用 += 运算符载入现有的 AddableBingoCage 实例 (接续示例 16.17)	470
16.19 bingoaddable.py:AddableBingoCage 扩展了 BingoCage,以支持 + 与 +=	471
16.20 与“示例 16.9<459 页>”一样	474
17.1 src/part04/sentence.py	480
17.2 测试 Sentence 示例能否迭代	481
17.3 abc.Iterator 类的源码 (摘自 Lib/_collections_abc.py 271 行)	485
17.4 sentence_iter.py:用迭代器模式实现 Sentence 类	487
17.5 sentence_gen.py:用生成器实现 Sentence 类	489
17.6 产生 3 个数字的生成器函数	490
17.7 运行时打印消息的生成器函数	492
17.8 sentence_gen2.py:用 re.finditer 函数实现惰性 Sentence 类	493
17.9 生成器函数 gen_AB 先由列表推导式使用,再由生成器表达式使用	494
17.10 sentence_genexp.py:用生成器表达式实现 Sentence 类	494
17.11 演示 ArithmeticProgression 类 (见示例 17.12) 的用法	496
17.12 ArithmeticProgression 类的实现	497
17.13 aritprog_gen 生成器函数	498
17.14 aritprog_v3.py:与“示例 17.13<498 页>”的 aritprog_gen 函数作用相同	499
17.15 演示用于筛选的生成器函数	500
17.16 itertools.accumulate 生成器函数示例	501
17.17 演示用于映射的生成器函数	502
17.18 演示用于合并的生成器函数	503
17.19 演示 itertools.product 生成器函数	503
17.20 演示 count、cycle、pairwise 与 repeat 生成器函数	505
17.21 组合生成器函数为每个输入项生成多个值	506
17.22 演示 itertools.groupby 函数的用法	507
17.23 itertools.tee 生成多个生成器,每个生成器都可生成输入可迭代对象的各项	508
17.24 用 all 与 any 处理几个序列的结果	509
17.25 测试 yield from	510
17.26 yield from 获取子生成器的返回值	511
17.27 tree/step0/tree.py:生成根类的名称,然后停止	512
17.28 tree/step1/tree.py:生成根类与直接子类的名称	513
17.29 tree/step2/tree.py:tree 生成根类的名称,然后委托 sub_tree	513
17.30 tree/step3/tree.py:sub_tree 深度优先遍历第 1 层与第 2 层	514
17.31 tree/step4/tree.py 的 sub_tree 生成器	515
17.32 tree/step5/tree.py:递归地从 sub_tree 中生成项 (只要内存够用)	515

17.33 tree/step6/tree.py:递归调用函数 tree,传入递增的参数 level	516
17.34 replacer.py:返回一个产生字符串元组的迭代器	516
17.35 fibo_gen.py:fibonacci 返回一个产出整数的生成器	517
17.36 tergentype.py:注解迭代器的 2 种方式	517
17.37 coroaverager.py:定义一个计算累计平均值的协程	519
17.38 coroaverager.py:示例 17.37 中定义的累计平均值协程的 docstrings	520
17.39 coroaverager.py:继续示例 17.38	520
17.40 coroaverager2.py:文件上半部分	521
17.41 coroaverager2.py:文件上半部分	522
17.42 coroaverager2.py:关闭协程	522
17.43 捕获 StopIteration 异常,返回一个 Result	523
17.44 coroaverager2.py:捕获 StopIteration 异常,返回一个 Result	523
18.1 将文件对象作为上下文管理器的演示	532
18.2 测试 LookingGlass 上下文管理器类	533
18.3 mirror.py:LookingGlass 上下文管理器类的实现代码	533
18.4 在没有 with 代码块的情况下使用 LookingGlass	534
18.5 mirror_gen.py:用生成器函数实现的上下文管理器	536
18.6 测试 looking_glass 上下文管理器	537
18.7 mirror_gen_exc.py:基于生成器的上下文管理器执行异常处理——外部行为同示例 18.3	538
18.8 looking_glass 上下文管理器也可当作装饰器使用	539
18.9 一个就地重写文件的上下文管理器	539
18.10 用 Scheme 计算最大公约数	540
18.11 作用同示例 18.10,用 Python 编写	540
18.12 lis.py:文件前几行	541
18.13 lis.py:负责解析的函数	541
18.14 lis.py:Environment 类	543
18.15 lis.py:standard_env() 构建并返回全局环境	544
18.16 lis.py:REPL 函数	545
18.17 lispy/py3.10/lis.py:evaluate 函数接受一个表达式,并对其求值	545
18.18 定义一个名为% 的函数,用于计算百分比	550
18.19 mylis_2/lis.py:实现真尾调用 (PTC)	559
18.20 lispy/py3.10/lis.py:摘自“示例 18.17<545 页>”	560
19.1 spinner_thread.py:spin 与 slow 函数	565
19.2 spinner_thread.py:supervisor 与 main 函数	567
19.3 spinner_proc.py:仅展示改动部分,其他代码与 spinner_thread.py 一致	568
19.4 spinner_async.py:main 函数与 supervisor 协程	569
19.5 spinner_async.py:spin 与 slow 协程	570
19.6 spinner_async_experiment.py:用“time.sleep(3)” 替换“await asyncio.sleep(3)”	571
19.7 spinner_async_experiment.py:协程 supervisor 与 slow	571
19.8 spinner_thread.py:线程版 supervisor 函数	572

19.9 spinner_async.py: 异步版 supervisor 协程	572
19.10 primes.py: 易读的素数检查函数 (摘自 Python 文档 “ProcessPoolExecutor 示例”)	573
19.11 spinner_prime_async_nap.py: 协程版的 is_prime	575
19.12 sequential.py: 对一个小型数据集做素数检查 (顺序执行)	577
19.13 procs.py: 多进程素数检测; 导入、类型和函数	579
19.14 procs.py: 多进程版素数检测器中的 main 函数	581
20.1 3 个脚本 flags.py、flags_threadpool.py 和 flags_asyncio.py 的运行结果	596
20.2 flags.py: 顺序下载脚本; 某些函数将被其他脚本复用	597
20.3 flags_threadpool.py: 使用 futures.ThreadPoolExecutor 的线程版下载脚本	599
20.4 flags_threadpool_futures.py: 用 futures.as_completed 重构的 download_many 函数	601
20.5 flags_threadpool_futures.py 的输出	602
20.6 proc_pool.py: 用 ProcessPoolExecutor 重写 procs.py	603
20.7 proc_pool.py 的输出	605
20.8 demo_executor_map.py: 简单演示 ThreadPoolExecutor 类的 map 方法	606
20.9 flags2 系列脚本的帮助界面	610
20.10 用默认值 (LOCAL 服务器、20 个国旗、1 个并发) 运行 flags2_sequential.py 脚本	611
20.11 运行 flags2_sequential.py: 从 DELAY 服务器获取国家代码前缀为 A、B 或 C 的国旗	611
20.12 运行 flags2_sequential.py: 并发 100 (-m 100) 从 ERROR 服务器下载 100 个图片 (-al 100)	611
20.13 flags2_sequential.py: 顺序执行版 download_many() 函数的实现	613
20.14 flags2_threadpool.py: 完整代码清单	614
21.1 blogdom.py: 为一个 Python 博客搜索域名	621
21.2 flags_asyncio.py: 设置操作的函数	624
21.3 flags_asyncio.py: import 语句与负责下载的函数	625
21.4 来自 asyncpg PostgreSQL 驱动程序文档的示例代码	627
21.5 运行 flags2_asyncio.py 脚本	628
21.6 flags2_asyncio.py 脚本上半部分, 余下代码见 “示例 21.7<631 页>”	629
21.7 flags2_asyncio.py 脚本余下代码, 接续 “示例 21.6<629 页>”	631
21.8 flags3_asyncio.py: get_country 协程	634
21.9 flags3_asyncio.py: download_one 协程	634
21.10 替换 await asyncio.to_thread 的 3 行	636
21.11 web_mojifinder.py 完整源码	639
21.12 tcp_mojifinder.py: 一个简单的 TCP 服务器 (后续见 “示例 21.14<643 页>”)	640
21.13 tcp_mojifinder.py: “图 21.4<641 页>” 中描述的会话服务器端。	642
21.14 tcp_mojifinder.py 脚本的前半部分 (接续 “示例 21.12<640 页>”)	643
21.15 tcp_mojifinder.py: search 协程	644
21.16 运行 python3 -m asyncio 后, 试验 domainlib.py	646
21.17 更多实验, 接续 示例 21.16	647
21.18 domainlib.py: 探测域名的函数	647
21.19 domaincheck.py: 用 domainlib 模块探测域名	649
21.20 @asynccontextmanager 与 loop.run_in_executor 用法示例	650

22.1 osconfeed.json 文件的示例记录 (省略了部分字段内容)	664
22.2 在交互式控制台中探索 osconfeed.json	665
22.3 FrozenJSON 不仅可读取属性,还可调用方法	666
22.4 explore0.py: 将 JSON 数据集转换为包含嵌套 FrozenJSON 对象、列表和简单类型的 FrozenJSON	667
22.5 explore1.py: 为名称是 Python 关键字的属性增加后缀 _	669
22.6 explore2.py:	670
22.7 读取 venue 与 speakers, 返回 Record 对象	671
22.8 测试 schedule_v1.py (见示例 22.9)	672
22.9 schedule_v1.py: 调整 OSCON 日程数据结构	672
22.10 摘自 schedule_v2.py 中的 doctests	674
22.11 schedule_v2.py: 增加 fetch 方法的 Record 类	674
22.12 schedule_v2.py: Event 类	675
22.13 schedule_v2.py: load 函数	676
22.14 schedule_v3.py: speakers 特性	677
22.15 用 hasattr 禁用 “键共享 (Key-Sharing)” 优化措施, 手工实现缓存逻辑	677
22.16 __init__ 中定义的存储, 以利用 “键共享 (Key-Sharing)” 优化措施	678
22.17 @cached_property 的简单用法	679
22.18 为 speakers 方法堆叠装饰器 @property 与 @cache	680
22.19 bulkfood_v1.py: 最简单的 LineItem 类	680
22.20 重量为负值时, 金额小计也为负值	681
22.21 bulkfood_v2.py: 定义了 weight 特性 (property) 的 LineItem 类	681
22.22 bulkfood_v2b.py: 效果同示例 22.21, 只是未使用装饰器	683
22.23 实例属性遮蔽了类中的数据属性	683
22.24 实例属性不会遮蔽类中的特性 (接续示例 22.23)	684
22.25 新的类特性会遮盖现有的实例属性 (接续示例 22.24)	685
22.26 一个特性 (property) 的文档 (docstring)	686
22.27 bulkfood_v2prop.py: 使用特性 (property) 工厂函数	686
22.28 bulkfood_v2prop.py: quantity 特性工厂函数	687
22.29 bulkfood_v2prop.py: 探索特性 (property) 与真用用于保存值的实例属性	688
22.30 blackknight.py	689
22.31 blackknight.py: 示例 22.30 的 doctest (黑衣骑士绝不屈服)	689
23.1 bulkfood_v3.py: Quantity 描述符不接受负值	700
23.2 bulkfood_v3.py: 用 Quantity 描述符管理 LineItem 中的属性	701
23.3 bulkfood_v4.py: 用 __set_name__ 为每个 Quantity 描述符实例设置名称	703
23.4 bulkfood_v4c.py: 简化 LineItem 的定义; Quantity 描述符类位于 model_v4c 模块中	704
23.5 model_v5.py: 抽象基类 Validated	705
23.6 model_v5.py: Validated 的具体子类 (Concrete Subclass): Quantity 与 NonBlank	705
23.7 bulkfood_v5.py: 使用描述符 Quantity 和 NonBlank 的 LineItem 类	706
23.8 descriptorkinds.py: 用于研究描述符覆盖行为的几个类	707

23.9 覆盖型描述符的行为	709
23.10 不含 <code>__get__</code> 方法的覆盖型描述符	709
23.11 非覆盖型描述符的行为	710
23.12 通过类属性赋值,可以覆盖类中的任何描述符	711
23.13 方法也是非覆盖型描述符	712
23.14 <code>method_is_descriptor.py</code> :Text 类,衍生自 <code>UserString</code> 类	712
23.15 试验一个方法	712
24.1 测试 <code>record_factory</code> ,一个简单的类工厂函数	722
24.2 <code>record_factory</code> :一个简单的类工厂函数	723
24.3 <code>initsub/checkedlib.py</code> :	725
24.4 <code>initsub/checkedlib.py</code> :	726
24.5 <code>initsub/checkedlib.py</code> :Checked 类最重要的方法	727
24.6 <code>initsub/checkedlib.py</code> :Checked 类余下的方法	729
24.7 <code>decorator/checkeddeco.py</code> :用装饰器 <code>@checked</code> 创建一个 Movie 类	730
24.8 <code>checkeddeco.py</code> :类装饰器 checked	731
24.9 <code>checkeddeco.py</code>	732
24.10 <code>builderlib.py</code> :模块的前半部分	733
24.11 <code>builderlib.py</code> :模块的后半部分	734
24.12 <code>evaldemo.py</code> :用 <code>builderlib.py</code> 模块 做实验的脚本	735
24.13 在控制台中,用 <code>evaldemo.py</code> 做实验	736
24.14 将 <code>evaldemo.py</code> 当做程序来运行	736
24.15 <code>metabunch/from3.6/bunch.py</code> :MetaBunch 元类与 Bunch 类	741
24.16 <code>evaldemo_meta.py</code> :用元类做实验	742
24.17 <code>metalib.py</code> :NosyDict 类	743
24.18 <code>metalib.py</code> :MetaKlass 元类	744
24.19 用 <code>evaldemo_meta.py</code> 进行控制台实验	745
24.20 将 <code>evaldemo_meta.py</code> 当作脚本运行	746
24.21 <code>metaclass/checkedlib.py</code> :含有 <code>storage_name</code> 属性与 <code>__get__</code> 方法的 Field 描述符	747
24.22 <code>metaclass/checkedlib.py</code> :CheckedMeta 元类	748
24.23 <code>metaclass/checkedlib.py</code> :新版 Checked 基类的完整源码	749