

《Python Cookbook 中文版， 》

图书基本信息

书名：《Python Cookbook 中文版，第3版》

13位ISBN编号：9787115379599

出版时间：2015-5-1

作者：David M. Beazley, Brian K. Jones

页数：684

译者：陈舸

版权说明：本站所提供下载的PDF图书仅提供预览和简介以及在线试读，请支持正版图书。

更多资源请访问：www.tushu000.com

《Python Cookbook 中文版，》

内容概要

《Python Cookbook（第3版）中文版》介绍了Python应用在各个领域中的一些使用技巧和方法，其主题涵盖了数据结构和算法，字符串和文本，数字、日期和时间，迭代器和生成器，文件和I/O，数据编码与处理，函数，类与对象，元编程，模块和包，网络和Web编程，并发，实用脚本和系统管理，测试、调试以及异常，C语言扩展等。

本书覆盖了Python应用中的很多常见问题，并提出了通用的解决方案。书中包含了大量实用的编程技巧和示例代码，并在Python 3.3环境下进行了测试，可以很方便地应用到实际项目中去。此外，

《Python Cookbook（第3版）中文版》还详细讲解了解决方案是如何工作的，以及为什么能够工作。

《Python Cookbook（第3版）中文版》非常适合具有一定编程基础的Python程序员阅读参考。

作者简介

David Beazley是一位居住在芝加哥的独立软件开发者以及图书作者。他主要的工作在于编程工具，提供定制化的软件开发服务，以及为软件开发者、科学家和工程师教授编程实践课程。他最为人熟知的工作在于Python编程语言，他已为此创建了好几个开源的软件包（例如Swig和PLY），并且是备受赞誉的图书Python Essential Reference的作者。他也对C、C++以及汇编语言下的系统编程有着丰富的经验。Brain K. Jones是普林斯顿大学计算机系的一位系统管理员。

书籍目录

目录

第1章 数据结构和算法 1

- 1.1 将序列分解为单独的变量 1
- 1.2 从任意长度的可迭代对象中分解元素 3
- 1.3 保存最后N个元素 5
- 1.4 找到最大或最小的N个元素 7
- 1.5 实现优先级队列 9
- 1.6 在字典中将键映射到多个值上 11
- 1.7 让字典保持有序 13
- 1.8 与字典有关的计算问题 14
- 1.9 在两个字典中寻找相同点 15
- 1.10 从序列中移除重复项且保持元素间顺序不变 17
- 1.11 对切片命名 18
- 1.12 找出序列中出现次数最多的元素 20
- 1.13 通过公共键对字典列表排序 22
- 1.14 对不原生支持比较操作的对象排序 23
- 1.15 根据字段将记录分组 25
- 1.16 筛选序列中的元素 26
- 1.17 从字典中提取子集 29
- 1.18 将名称映射到序列的元素中 30
- 1.19 同时对数据做转换和换算 33
- 1.20 将多个映射合并为单个映射 34

第2章 字符串和文本 37

- 2.1 针对任意多的分隔符拆分字符串 37
- 2.2 在字符串的开头或结尾处做文本匹配 38
- 2.3 利用Shell通配符做字符串匹配 40
- 2.4 文本模式的匹配和查找 42
- 2.5 查找和替换文本 45
- 2.6 以不区分大小写的方式对文本做查找和替换 47
- 2.7 定义实现最短匹配的正则表达式 48
- 2.8 编写多行模式的正则表达式 49
- 2.9 将Unicode文本统一表示为规范形式 50
- 2.10 用正则表达式处理Unicode字符 52
- 2.11 从字符串中去掉不需要的字符 53
- 2.12 文本过滤和清理 54
- 2.13 对齐文本字符串 57
- 2.14 字符串连接及合并 59
- 2.15 给字符串中的变量名做插值处理 62
- 2.16 以固定的列数重新格式化文本 64
- 2.17 在文本中处理HTML和XML实体 66
- 2.18 文本分词 67
- 2.19 编写一个简单的递归下降解析器 70
- 2.20 在字节串上执行文本操作 80

第3章 数字、日期和时间 83

- 3.1 对数值进行取整 83
- 3.2 执行精确的小数计算 85
- 3.3 对数值做格式化输出 87

- 3.4 同二进制、八进制和十六进制数打交道 89
- 3.5 从字节串中打包和解包大整数 90
- 3.6 复数运算 92
- 3.7 处理无穷大和NaN 94
- 3.8 分数的计算 96
- 3.9 处理大型数组的计算 97
- 3.10 矩阵和线性代数的计算 101
- 3.11 随机选择 103
- 3.12 时间换算 105
- 3.13 计算上周5的日期 107
- 3.14 找出当月的日期范围 108
- 3.15 将字符串转换为日期 110
- 3.16 处理涉及到时区的日期问题 112
- 第4章 迭代器和生成器 114
 - 4.1 手动访问迭代器中的元素 114
 - 4.2 委托迭代 115
 - 4.3 用生成器创建新的迭代模式 116
 - 4.4 实现迭代协议 118
 - 4.5 反向迭代 121
 - 4.6 定义带有额外状态的生成器函数 122
 - 4.7 对迭代器做切片操作 123
 - 4.8 跳过可迭代对象中的前一部分元素 124
 - 4.9 迭代所有可能的组合或排列 127
 - 4.10 以索引-值对的形式迭代序列 129
 - 4.11 同时迭代多个序列 131
 - 4.12 在不同的容器中进行迭代 133
 - 4.13 创建处理数据的管道 134
 - 4.14 扁平化处理嵌套型的序列 137
 - 4.15 合并多个有序序列, 再对整个有序序列进行迭代 139
 - 4.16 用迭代器取代while循环 140
- 第5章 文件和I/O 142
 - 5.1 读写文本数据 142
 - 5.2 将输出重定向到文件中 145
 - 5.3 以不同的分隔符或行结尾符完成打印 145
 - 5.4 读写二进制数据 146
 - 5.5 对已不存在的文件执行写入操作 149
 - 5.6 在字符串上执行I/O操作 150
 - 5.7 读写压缩的数据文件 151
 - 5.8 对固定大小的记录进行迭代 152
 - 5.9 将二进制数据读取到可变缓冲区中 153
 - 5.10 对二进制文件做内存映射 155
 - 5.11 处理路径名 157
 - 5.12 检测文件是否存在 158
 - 5.13 获取目录内容的列表 159
 - 5.14 绕过文件名编码 161
 - 5.15 打印无法解码的文件名 162
 - 5.16 为已经打开的文件添加或修改编码方式 164
 - 5.17 将字节数据写入文本文件 166
 - 5.18 将已有的文件描述符包装为文件对象 167

- 5.19 创建临时文件和目录 169
- 5.20 同串口进行通信 171
- 5.21 序列化Python对象 172
- 第6章 数据编码与处理 177
 - 6.1 读写CSV数据 177
 - 6.2 读写JSON数据 181
 - 6.3 解析简单的XML文档 186
 - 6.4 以增量方式解析大型XML文件 188
 - 6.5 将字典转换为XML 192
 - 6.6 解析、修改和重写XML 194
 - 6.7 用命名空间来解析XML文档 196
 - 6.8 同关系型数据库进行交互 198
 - 6.9 编码和解码十六进制数字 201
 - 6.10 Base64编码和解码 202
 - 6.11 读写二进制结构的数组 203
 - 6.12 读取嵌套型和大小可变的二进制结构 207
 - 6.13 数据汇总和统计 218
- 第7章 函数 221
 - 7.1 编写可接受任意数量参数的函数 221
 - 7.2 编写只接受关键字参数的函数 223
 - 7.3 将元数据信息附加到函数参数上 224
 - 7.4 从函数中返回多个值 225
 - 7.5 定义带有默认参数的函数 226
 - 7.6 定义匿名或内联函数 229
 - 7.7 在匿名函数中绑定变量的值 230
 - 7.8 让带有N个参数的可调用对象以较少的参数形式调用 232
 - 7.9 用函数替代只有单个方法的类 235
 - 7.10 在回调函数中携带额外的状态 236
 - 7.11 内联回调函数 240
 - 7.12 访问定义在闭包内的变量 242
- 第8章 类与对象 246
 - 8.1 修改实例的字符串表示 246
 - 8.2 自定义字符串的输出格式 248
 - 8.3 让对象支持上下文管理协议 249
 - 8.4 当创建大量实例时如何节省内存 251
 - 8.5 将名称封装到类中 252
 - 8.6 创建可管理的属性 254
 - 8.7 调用父类中的方法 259
 - 8.8 在子类中扩展属性 263
 - 8.9 创建一种新形式的类属性或实例属性 267
 - 8.10 让属性具有惰性求值的能力 271
 - 8.11 简化数据结构的初始化过程 274
 - 8.12 定义一个接口或抽象基类 278
 - 8.13 实现一种数据模型或类型系统 281
 - 8.14 实现自定义的容器 287
 - 8.15 委托属性的访问 291
 - 8.16 在类中定义多个构造函数 296
 - 8.17 不通过调用init来创建实例 298
 - 8.18 用Mixin技术来扩展类定义 299

- 8.19 实现带有状态的对象或状态机 305
- 8.20 调用对象上的方法, 方法名以字符串形式给出 311
- 8.21 实现访问者模式 312
- 8.22 实现非递归的访问者模式 317
- 8.23 在环状数据结构中管理内存 324
- 8.24 让类支持比较操作 327
- 8.25 创建缓存实例 330
- 第9章 元编程 335
 - 9.1 给函数添加一个包装 335
 - 9.2 编写装饰器时如何保存函数的元数据 337
 - 9.3 对装饰器进行解包装 339
 - 9.4 定义一个可接受参数的装饰器 341
 - 9.5 定义一个属性可由用户修改的装饰器 342
 - 9.6 定义一个能接收可选参数的装饰器 346
 - 9.7 利用装饰器对函数参数强制执行类型检查 348
 - 9.8 在类中定义装饰器 352
 - 9.9 把装饰器定义成类 354
 - 9.10 把装饰器作用到类和静态方法上 357
 - 9.11 编写装饰器为被包装的函数添加参数 359
 - 9.12 利用装饰器给类定义打补丁 362
 - 9.13 利用元类来控制实例的创建 364
 - 9.14 获取类属性的定义顺序 367
 - 9.15 定义一个能接受可选参数的元类 370
 - 9.16 在*args和**kwargs上强制规定一种参数签名 372
 - 9.17 在类中强制规定编码约定 375
 - 9.18 通过编程的方式来定义类 378
 - 9.19 在定义的时候初始化类成员 382
 - 9.20 通过函数注解来实现方法重载 384
 - 9.21 避免出现重复的属性方法 391
 - 9.22 以简单的方式定义上下文管理器 393
 - 9.23 执行带有局部副作用的代码 395
 - 9.24 解析并分析Python源代码 398
 - 9.25 将Python源码分解为字节码 402
- 第10章 模块和包 406
 - 10.1 把模块按层次结构组织成包 406
 - 10.2 对所有符号的导入进行精确控制 407
 - 10.3 用相对名称来导入包中的子模块 408
 - 10.4 将模块分解成多个文件 410
 - 10.5 让各个目录下的代码在统一的命名空间下导入 413
 - 10.6 重新加载模块 415
 - 10.7 让目录或zip文件成为可运行的脚本 416
 - 10.8 读取包中的数据文件 417
 - 10.9 添加目录到sys.path中 418
 - 10.10 使用字符串中给定的名称来导入模块 420
 - 10.11 利用import钩子从远端机器上加载模块 421
 - 10.12 在模块加载时为其打补丁 439
 - 10.13 安装只为自己所用的包 441
 - 10.14 创建新的Python环境 442
 - 10.15 发布自定义的包 444

第11章 网络和Web编程 446

- 11.1 以客户端的形式同HTTP服务交互 446
- 11.2 创建一个TCP服务器 450
- 11.3 创建一个UDP服务器 454
- 11.4 从CIDR地址中生成IP地址的范围 456
- 11.5 创建基于REST风格的简单接口 458
- 11.6 利用XML-RPC实现简单的远端过程调用 463
- 11.7 在不同的解释器间进行通信 466
- 11.8 实现远端过程调用 468
- 11.9 以简单的方式验证客户端身份 472
- 11.10 为网络服务增加SSL支持 474
- 11.11 在进程间传递socket文件描述符 481
- 11.12 理解事件驱动型I/O 486
- 11.13 发送和接收大型数组 493

第12章 并发 496

- 12.1 启动和停止线程 496
- 12.2 判断线程是否已经启动 499
- 12.3 线程间通信 503
- 12.4 对临界区加锁 508
- 12.5 避免死锁 511
- 12.6 保存线程专有状态 515
- 12.7 创建线程池 517
- 12.8 实现简单的并行编程 521
- 12.9 如何规避GIL带来的限制 525
- 12.10 定义一个Actor任务 528
- 12.11 实现发布者/订阅者消息模式 532
- 12.12 使用生成器作为线程的替代方案 536
- 12.13 轮询多个线程队列 544
- 12.14 在UNIX上加载守护进程 547

第13章 实用脚本和系统管理 552

- 13.1 通过重定向、管道或输入文件来作为脚本的输入 552
- 13.2 终止程序并显示错误信息 553
- 13.3 解析命令行选项 554
- 13.4 在运行时提供密码输入提示 557
- 13.5 获取终端大小 558
- 13.6 执行外部命令并获取输出 558
- 13.7 拷贝或移动文件和目录 560
- 13.8 创建和解包归档文件 562
- 13.9 通过名称来查找文件 563
- 13.10 读取配置文件 565
- 13.11 给脚本添加日志记录 568
- 13.12 给库添加日志记录 571
- 13.13 创建一个秒表计时器 573
- 13.14 给内存和CPU使用量设定限制 575
- 13.15 加载Web浏览器 576

第14章 测试、调试以及异常 578

- 14.1 测试发送到stdout上的输出 578
- 14.2 在单元测试中为对象打补丁 579
- 14.3 在单元测试中检测异常情况 583

- 14.4 将测试结果作为日志记录到文件中 585
- 14.5 跳过测试, 或者预计测试结果为失败 586
- 14.6 处理多个异常 587
- 14.7 捕获所有的异常 589
- 14.8 创建自定义的异常 591
- 14.9 通过引发异常来响应另一个异常 593
- 14.10 重新抛出上一个异常 595
- 14.11 发出告警信息 596
- 14.12 对基本的程序崩溃问题进行调试 598
- 14.13 对程序做性能分析以及计时统计 600
- 14.14 让你的程序运行得更快 603
- 第15章 C语言扩展 610
- 15.1 利用ctypes来访问C代码 612
- 15.2 编写简单的C语言扩展模块 618
- 15.3 编写一个可操作数组的扩展函数 622
- 15.4 在C扩展模块中管理不透明指针 625
- 15.5 在扩展模块中定义并导出C API 628
- 15.6 从C中调用Python 633
- 15.7 在C扩展模块中释放GIL 639
- 15.8 混合使用C和Python环境中的线程 639
- 15.9 用Swig来包装C代码 640
- 15.10 用Cython来包装C代码 646
- 15.11 用Cython来高效操作数组 652
- 15.12 把函数指针转换为可调用对象 657
- 15.13 把以NULL结尾的字符串传给C库 659
- 15.14 把Unicode字符串传递给C库 663
- 15.15 把C字符串转换到Python中 667
- 15.16 同编码方式不确定的C字符串打交道 669
- 15.17 把文件名传给C扩展模块 672
- 15.18 把打开的文件传给C扩展模块 673
- 15.19 在C中读取文件型对象 674
- 15.20 从C中访问可迭代对象 677
- 15.21 排查段错误 678
- 附录A 补充阅读 680

精彩短评

- 1、翻译得太烂，正文翻译错了就算了，代码的缩进竟然有的地方都和原版不一样。
- 2、超级赞
- 3、学过python 的都需要看看这本书吧，有很多小知识点。
- 4、花了近两个月的时间，看的也差不多了。书中很多实用的技巧，都是基于Python 3.x，这里我用2.x和3.x分别实现了些，除了3.x特有的新功能外，其他的都很ok。个人觉得，最好针对性的去学习，随手翻阅的效果总不如主题性，针对性学习，还有就是学的时候最好进行创新，演变成自己项目中的一部分，增添点趣味才能更好的去把握。
- 5、有 stackoverflow 基本就不需要了
- 6、如果可以给6颗心，我愿意给6颗心。
- 7、翻译的还流畅，就是书中代码缩进不忍直视，举个例子，一个简单的类加上类中方法，也就三行，排版硬是丢了两行缩进，跟买了盗版书一样Orz。。建议看原版，建议看原版，建议看原版。。
- 8、cookbook这种小问题导向的教程，最适合度过小白阶段后的进阶了，我很喜欢。
- 9、算是从头到尾把整本书看完。
- 10、这本书部分地方有些难，不适合初学者。建议作为提升教材
- 11、很棒的书，看完就会做菜了，毕竟不会做菜的程序员不是好PM。元编程和一些类的编程部分太慢了，这次跳过去没看了。一边整理笔记一边回头看。

章节试读

1、《Python Cookbook 中文版，第3版》的笔记-第十章: 模块与包

10.1 构建一个模块的层级包

2、《Python Cookbook 中文版，第3版》的笔记-第一章: 数据结构和算法

1.1 解压序列赋值给多个变量

****问题****: 将N个元素的可迭代对象赋值给N个变量.

****解决方案****: 直接赋值即可.a, b, c = "abc"

```
a, b, c
Out[3]: ('a', 'b', 'c')
```

```
a, b, c = iter("abc")
```

```
a, b, c
Out[5]: ('a', 'b', 'c')
```

1.2 解压可迭代对象赋值给多个变量

****问题****: 将N个元素的可迭代对象赋值给M(M < N)个元素

****解决方案****: 使用*号代表列表.a, *b, c = range(1, 10)

```
a, b, c
Out[7]: (1, [2, 3, 4, 5, 6, 7, 8], 9) **备注**: Python2.x版本不可用
```

1.3 保留最后N个元素

****问题****: 如何保留最后N个元素

****解决方案****: 使用collections.deque, 设定maxlen即可.from collections import deque

```
q = deque(maxlen=3)
```

```
for i in range(10):
```

```
    q.append(i)
```

```
# deque([7, 8, 9], maxlen=3)
```

```
print(q)
```

1.4 查找最大或最小的N个元素

****问题****: 从集合中获得最大或最小的N个元素?

****解决方案****:

- 排序后使用切片

- 使用heapq的nlargest和nsmallestimport heapq

```
dst = [
```

```
    {"name": "a", "num": 3},
```

```
    {"name": "b", "num": 1},
```

```
    {"name": "c", "num": 2},
```

```
    {"name": "d", "num": 4}
```

```
]
```

```
# [{'name': 'd', 'num': 4}, {'name': 'a', 'num': 3}]
```

```
print(heapq.nlargest(2, dst, key=lambda s: s["num"]))
```

```
# [{'name': 'b', 'num': 1}, {'name': 'c', 'num': 2}]
```

```
print(sorted(dst, key=lambda s: s["num"])[0:2])
```

1.5 实现一个优先级队列

****问题**:** 如何实现一个优先级队列?

****解决方案**:** 使用heapq, 通过将优先级权值作为比较对象即可. 为了比较相同优先级权值, 增加一个自增变量即可.import heapq

```
class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0
    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1
    def pop(self):
        return heapq.heappop(self._queue)[-1]
    def __iter__(self):
        return self
    def __next__(self):
        if self._queue:
            return self.pop()
        else:
            raise StopIteration
```

```
q = PriorityQueue()
q.push("a", 4)
q.push("m", 3)
q.push("c", 2)
q.push("d", 6)
q.push("q", 4)
```

```
# d a q m c
for i in q:
    print(i)
```

1.6 字典中的键映射多个值

****问题**:** 怎样实现一个键对应多个值的字典(multidict)

****解决方案**:** 使用defaultdict即可.from collections import defaultdict

```
d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(3)
# [1, 2]
print(d['a'])
```

1.7 字典排序

****问题**:** 对字典进行排序

****解决方案**:** 使用OrderedDictfrom collections import OrderedDict

```
d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
```

```

d['spam'] = 3
d['grok'] = 4
# OrderedDict([('foo', 1), ('bar', 2), ('spam', 3), ('grok', 4)])
print(d)

d1 = {}
d1['foo'] = 1
d1['bar'] = 2
d1['spam'] = 3
d1['grok'] = 4
# {'bar': 2, 'foo': 1, 'grok': 4, 'spam': 3}
print(d1)
## 1.9 查找两字典的相同点
**问题**: 怎样在两个字典中寻找相同点(比如相同的键, 相同的值)
**解决方案**:
- 字典的keys支持集合操作.
- 在查找相同的值时,通过zip将values当做keys, 再执行集合操作.a = {
    "x": 1,
    "y": 2,
    "z": 3
}
b = {
    "w": 10,
    "x": 11,
    "y": 2
}
# {'z'}
print(a.keys() - b.keys())
# {'z', 3}, ('x', 1)}
print(a.items() - b.items())
## 1.10 删除序列相同元素并保持顺序
**问题**: 怎样在一个序列上面保持元素顺序的同时消除重复的值?
**解决方案**: 构建一个集合,并且将元素add到集合中
**备注**: 如果直接使用集合, 则无法保证元素的顺序.def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)

a = [1, 5, 2, 1, 9, 1, 5, 10]
# [1, 5, 2, 9, 10]
print(list(dedupe(a)))
# {1, 2, 10, 5, 9}
print(set(a))
## 1.11 命名切片
**问题**: 拥有大量硬编码切片下标.
**解决方案**: 使用slice函数进行切片命名record = "hello world"
s = slice(2,4)

```

```

# II
print(record[s])
## 1.12 序列中出现次数最多的元素
**问题**: 怎样找出一个序列中出现次数最多的元素呢?
**解决方案**: 使用collections.Counter
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around',
    'the', 'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look',
    'into', 'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
# [('eyes', 8), ('the', 5), ('look', 4)]
print(top_three)
## 1.13 通过某个关键字排序一个字典列表
**问题**: 通过某个或某几个字典字段来排序
**解决方案**:
- 使用operator模块的itemgetter
- 使用lambda表达式
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]

from operator import itemgetter
# [{'lname': 'Beazley', 'uid': 1002, 'fname': 'David'},
# {'lname': 'Cleese', 'uid': 1001, 'fname': 'John'},
# {'lname': 'Jones', 'uid': 1003, 'fname': 'Brian'},
# {'lname': 'Jones', 'uid': 1004, 'fname': 'Big'}]
print(sorted(rows, key=lambda r: r['lname']))
print(sorted(rows, key=itemgetter('lname')))
# [{'lname': 'Beazley', 'uid': 1002, 'fname': 'David'},
# {'lname': 'Cleese', 'uid': 1001, 'fname': 'John'},
# {'lname': 'Jones', 'uid': 1004, 'fname': 'Big'},
# {'lname': 'Jones', 'uid': 1003, 'fname': 'Brian'}]
print(sorted(rows, key=lambda r: (r['lname'], r['fname']))))
print(sorted(rows, key=itemgetter('lname', 'fname')))
## 1.15 通过某个字段将记录分组
**问题**: 对序列进行分组访问
**解决方案**: 通过itertools.groupby()函数来完成
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'}
]

```

```
]
```

```
from itertools import groupby
rows.sort(key=lambda s: s['date'])
for date, items in groupby(rows, key=lambda s: s['date']):
    print(date)
    for i in items:
        print(' ', i)**output**:07/01/2012
        {'date': '07/01/2012', 'address': '5412 N CLARK'}
        {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
        {'date': '07/02/2012', 'address': '5800 E 58TH'}
        {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
        {'date': '07/02/2012', 'address': '1060 W ADDISON'}
```

```
07/03/2012
```

```
        {'date': '07/03/2012', 'address': '2122 N CLARK'}
```

```
07/04/2012
```

```
        {'date': '07/04/2012', 'address': '5148 N CLARK'}
```

```
## 1.18 映射名称到序列元素
```

****问题****: 通过下标访问列表或元组中元素的代码, 但是难以阅读

****解决方案****: 使用collections.namedtuple()from collections import namedtuple

```
Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
```

```
sub = Subscriber('leicj@gmail.com', '2012-12-12')
```

```
sub
```

```
Out[14]: Subscriber(addr='leicj@gmail.com', joined='2012-12-12')
```

```
sub.addr
```

```
Out[15]: 'leicj@gmail.com'
```

```
## 1.20 合并多个字典或映射
```

****问题****: 将多个字典进行合并

****解决方案****:

- 使用collections模块中的ChainMap

- 使用update

****备注****: 如果使用ChainMap, 则新生成的字典是原字典的引用, 而非副本from collections import ChainMap

```
a, b = {'x': 1, 'z': 3}, {'y': 2, 'z': 4}
```

```
c = ChainMap(a, b)
```

```
c
```

```
Out[23]: ChainMap({'x': 1, 'z': 3}, {'z': 4, 'y': 2})
```

```
c['x'], c['y'], c['z']
```

```
Out[24]: (1, 2, 3)
```

```
c.keys(), c.values()
Out[25]:
(KeysView(ChainMap({'x': 1, 'z': 3}, {'z': 4, 'y': 2})),
 ValuesView(ChainMap({'x': 1, 'z': 3}, {'z': 4, 'y': 2})))
```

```
list(c.keys()), list(c.values())
Out[26]: (['y', 'x', 'z'], [2, 1, 3])
```

3、《Python Cookbook 中文版，第3版》的笔记-第七章:函数

```
## 7.1 可接受任意数量参数的函数
**问题**: 接受任意数量参数的函数
**解决方案**: 使用\*, \*\*def anyargs(*args, **kwargs):
    print(args)
    print(kwargs)

# (1, 2, 3)
# {'a': 1, 'b': 2}
anyargs(1, 2, 3, a = 1, b = 2)

## 7.2 减少可调用对象的参数个数
**解决方案**: 使用functools.partial()def spam(a, b, c, d):
    print(a, b, c, d)
```

```
from functools import partial
s1 = partial(spam, 1, d = 42)
# 1 11 22 42
s1(11, 22)
## 7.3 将单方法的类转换为函数
**问题**: 一个类只有一个方法, 将此方法转换为类.
**解决方案**: 使用闭包class C:
    def __init__(self, n):
        self.n = n
    def show(self):
        print(list(range(self.n)))
```

```
c = C(10)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
c.show()
```

```
def C1(n):
    def show():
        print(list(range(n)))
    return show
c1 = C1(10)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
c1()
```

7.4 带额外状态信息的回调函数

****问题**:** 代码中需要依赖到回调函数的使用, 并且还需要让回调函数拥有额外的状态值, 以便在它的内


```
def apply_async(func, args, *, callback):
    result = func(*args)
    callback(result)
```

```
def print_result(result):
    print('Got:', result)
```

```
def add(x, y):
    return x + y
```

```
apply_async(add, (2, 3), callback=print_result)
apply_async(add, ('hello', 'world'), callback=print_result)
```

****解决方案**:**

- 绑定方法来代替一个简单函数.

```
class ResultHandler:
    def __init__(self):
        self.sequence = 0
    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

```
r = ResultHandler()
# [1] Got: 5
apply_async(add, (2, 3), callback=r.handler)
# [2] Got: helloworld
apply_async(add, ('hello', 'world'), callback=r.handler)
```

- 使用闭包

```
def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler
```

```
handler = make_handler()
# [1] Got: 5
apply_async(add, (2, 3), callback=handler)
# [2] Got: helloworld
apply_async(add, ('hello', 'world'), callback=handler)
```

- 使用协程

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

```
handler = make_handler()
```

```

next(handler)
# [1] Got: 5
apply_async(add, (2, 3), callback=handler.send)
# [2] Got: helloworld
apply_async(add, ('hello', 'world'), callback=handler.send)
## 7.5 内联回调函数
**问题**: 当你编写回调函数的代码时, 担心很多小函数的扩张会弄乱程序控制流. 你希望找到某个方法来让代码看上去更像是一个普通的执行序列.
**解决方案**: 使用生成器和协程
def apply_async(func, args, *, callback):
    result = func(*args)
    callback(result)

from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
        self.args = args

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
        return wrapper

def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')

```

```
test()
输出:
5
helloworld
0
2
4
6
8
10
12
14
16
18
Goodbye
```

4、《Python Cookbook 中文版 , 第 3 版》的笔记-第三章: 数字日期和时间

3.1 数字的四舍五入

****问题**:** 对浮点数执行指定精度的舍入运算

****解决方案**:** 使用round(values, ndigits)round(1.234, 2)

Out[68]: 1.23

```
round(12345, -2)
```

Out[69]: 12300

3.2 执行精确的浮点数运算

****问题**:** 执行浮点数运算时,不允许出现误差

****解决方案**:** 使用decimal模块from decimal import Decimal

```
a = Decimal('4.2')
```

```
b = Decimal('2.1')
```

```
a + b
```

Out[73]: Decimal('6.3')

```
a + b == 6.3
```

Out[74]: False

```
a + b == Decimal('6.3')
```

Out[75]: True

3.3 数字的格式化输出

****问题**:** 数字的格式化输出

****解决方案**:** 使用formatformat(x, '0.2f')

Out[77]: '1234.57'

```
format(x, '>10.2f')
Out[78]: ' 1234.57'
```

```
format(x, '<10.2f')
Out[79]: '1234.57 '
```

```
format(x, ',')
Out[80]: '1,234.56789'
## 3.4 二八十六进制整数
```

****问题**:** 转换或输出二进制,八进制或十六进制
****解决方案**:** 使用bin(), oct()或hex())x = 1234

```
bin(x)
Out[86]: '0b10011010010'
```

```
oct(x)
Out[87]: '0o2322'
```

```
hex(x)
Out[88]: '0x4d2'
```

```
format(x, 'b')
Out[89]: '10011010010'
```

```
format(x, 'o')
Out[90]: '2322'
```

```
format(x, 'x')
Out[91]: '4d2'
```

```
int(oct(x), 8)
Out[92]: 1234
而Python的八进制语法特殊,需要输入:0o(数字0,字母o)0o755
Out[93]: 493
```

3.5 字节到大整数的打包与解包

****问题**:** 将字节字符串解压成一个整数, 或者将大整数转换为一个字节字符串.
****解决方案**:** 使用int.from.bytes()/int.to.bytes()data =
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'

```
len(data)
Out[95]: 16
```

```
int.from_bytes(data, 'little')
Out[96]: 69120565665751139577663547927094891008
```

```
int.from_bytes(data, 'little').to_bytes(16, 'little')
Out[97]: b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

3.7 无穷大与NaN

****问题**:** 创建无穷或NaN数值

****解决方案****: 使用float()来完成import math

```
a = float('inf')
```

```
b = float('-inf')
```

```
c = float('nan')
```

```
math.isinf(a)
```

```
Out[106]: True
```

```
math.isnan(c)
```

```
Out[107]: True
```

3.11 随机选择

****问题****: 生成随机数

****解决方案****: 使用random模块

- random.choice(): 从一个序列中随机的抽取一个元素

```
import random
```

```
random.choice(values)
```

```
Out[112]: 6
```

- random.sample(): 取出N个不同元素

```
random.sample(values, 3)
```

```
Out[113]: [7, 6, 4]
```

- random.shuffle(): 打乱顺序

```
values
```

```
Out[118]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
random.shuffle(values)
```

```
values
```

```
Out[120]: [2, 5, 9, 3, 4, 1, 8, 7, 6]
```

- random.randint(): 随机生成整数

```
random.randint(0, 10)
```

```
Out[121]: 5
```

3.12 基本的日期与时间转换

```
import time
```

```
t = '2016-04-23 12:12:12'
```

```
t1 = time.strptime(t, '%Y-%m-%d %H:%M:%S')
```

```
t2 = time.mktime(t1)
```

```
print(t, t1, t2, sep='\n')
```

```
t3 = time.localtime(t2)
```

```
t4 = time.strftime('%Y-%m-%d %H:%M:%S', t3)
```

```
print(t3, t4, sep='\n')
```

****output****:

```
2016-04-23 12:12:12
```

```
time.struct_time(tm_year=2016, tm_mon=4, tm_mday=23, tm_hour=12, tm_min=12, tm_sec=12, tm_wday=5, tm_yday=114, tm_isdst=-1)
```

1461384732.0

time.struct_time(tm_year=2016, tm_mon=4, tm_mday=23, tm_hour=12, tm_min=12, tm_sec=12, tm_wday=5, tm_yday=114, tm_isdst=0)

2016-04-23 12:12:12

5、《Python Cookbook 中文版 , 第 3 版》的笔记-第六章: 数据编码和处理

6.1 读写CSV数据

```
import csv
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
        ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
        ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000)
]
```

with open('stocks.csv', 'w') as f:

```
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

```
# ['AA', '39.48', '6/11/2007', '9:36am', '-0.18', '181800']
# ['AIG', '71.38', '6/11/2007', '9:36am', '-0.15', '195500']
# ['AXP', '62.58', '6/11/2007', '9:36am', '-0.46', '935000']
```

with open('stocks.csv') as f:

```
    f_csv = csv.reader(f)
    headings = next(f_csv)
    for r in f_csv:
        print(r)
```

```
# {'Change': '-0.18', 'Symbol': 'AA', 'Time': '9:36am', 'Date': '6/11/2007', 'Price': '39.48', 'Volume': '181800'}
# {'Change': '-0.15', 'Symbol': 'AIG', 'Time': '9:36am', 'Date': '6/11/2007', 'Price': '71.38', 'Volume': '195500'}
# {'Change': '-0.46', 'Symbol': 'AXP', 'Time': '9:36am', 'Date': '6/11/2007', 'Price': '62.58', 'Volume': '935000'}
```

with open('stocks.csv') as f:

```
    f_csv = csv.DictReader(f)
    for row in f_csv:
        print(row)
```

6.2 读写JSON数据

****问题**:** 读写JSON数据

****解决方案**:** 使用json.dumps/json.loadsimport json

```
data = {
    'name': 'ACME',
    'shares': 100,
    'price': 542.23
}
json_str = json.dumps(data)
# <class 'str'>: {"name": "ACME", "shares": 100, "price": 542.23}
print(type(json_str), json_str)
```

```
data = json.loads(json_str)
```

```
# <class 'dict'> {'name': 'ACME', 'shares': 100, 'price': 542.23}
print(type(data), data)
```

6、《Python Cookbook 中文版 , 第3版》的笔记-第五章: 文件与IO

5.1 打印输出至文件中

****问题**:** 使用print()函数的输出重定向到一个文件中

****解决方案**:** 使用print函数中指定的file关键字参数with open('./test.txt', 'wt') as f:

```
print("hello world", file=f)
```

5.2 使用其他分隔符或行终止符打印

****问题**:** 使用print时候, 改变默认的分隔符或行尾符

****解决方案**:** 使用sep/end关键字参数

```
# hello|1|2|3
```

```
print("hello", 1, 2, 3, sep='|')
```

```
s = [1, 2, 3]
```

```
# 1|2|3|
```

```
for i in s:
```

```
    print(i, end='|')
```

5.3 读写字节数据

****问题**:** 读写二进制文件

****解决方案**:** 使用rb/wb

with open('somefile.bin', 'wb') as f:

```
    f.write(b'Hello world!')
```

with open('somefile.bin', 'rb') as f:

```
    data = f.read()
```

```
    text = data.decode('utf-8')
```

```
# 72,101,108,108,111,32,119,111,114,108,100,33,
```

```
for c in data:
```

```
    print(c, end=',')
```

```
print('\n')
```

```
# Hello world!
```

```
print(text)
```

5.4 文件不存在才能写入

****问题**:** 在文件中写入,但是前提是这个文件在系统上不存在

****解决方案**:**

- 使用x模式

with open('somefile', 'wt') as f:

```
    f.write("hello\n")
```

抛出异常

with open('somefile', 'xt') as f:

```
    f.write("world\n")
```

- 使用os.path.exists判断文件是否存在

with open('somefile', 'wt') as f:

```
    f.write("hello\n")
```

```
import os
```

```
if not os.path.exists('./somefile'):
```

```
with open('somefile', 'xt') as f:
    f.write("world\n")
```

```
with open('somefile', 'rt') as f:
    print(f.read())
```

5.6 字符串的I/O操作

****问题**:** 使用操作类文件对象的程序来操作文本或二进制字符串

****解决方案**:** 使用io.StringIO()和io.BytesIO()类来创建类文件对象操作字符串数据.import io

```
s = io.StringIO()
s.write("Hello world\n")
print("this is a test", file=s)
# Hello world
# this is a test
print(s.getvalue())
```

5.8 固定大小记录的文件迭代

****问题**:** 在一个固定长度记录或者数据块的集合上迭代,而不是在一个文件中一行一行的迭代.

****解决方案**:** 使用iter和functools.partialfrom functools import partial

```
RECORD_SIZE = 32
```

```
with open('test.txt', 'rb') as f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for r in records:
        print(r)
```

****iter的一个特性**:** 给它传递一个可调用对象和一个标记值, 它会创建一个迭代器. 这个迭代器会一直调用传入的可调用对象直到它返回标记值为止, 这时候迭代终止.

5.9 读取二进制数据到可变缓冲区中

****问题**:** 读取二进制数据到一个可变缓冲区中, 而不需要做任何中间复制操作.

****解决方案**:** 使用readinto()方法

```
import os.path
```

```
def read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with open(filename, 'rb') as f:
        f.readinto(buf)
    return buf
```

```
with open('sample.bin', 'wb') as f:
    f.write(b'Hello World')
```

```
buf = read_into_buffer('sample.bin')
# bytearray(b'Hello World')
print(buf)
```

5.10 文件的一些基本操作

- 文件名的操作

```
import os
```



```

path = '/Users/lgt/Data/data.csv'
# data.csv
print(os.path.basename(path))
# /Users/lgt/Data
print(os.path.dirname(path))
# tmp/data/data.csv
print(os.path.join('tmp', 'data', os.path.basename(path)))
# False
print(os.path.exists('./no_exist.txt'))
# True
print(os.path.isfile('./test.py'))
- 获取目录下的文件列表
import os

pwd = os.getcwd()

names = [name for name in os.listdir(pwd)
         if os.path.isfile(os.path.join(pwd, name))]
# ['sample.bin', 'test.py', 'test.txt']
print(names)
- 向文件中写入字节数据
import sys

# hello world
sys.stdout.buffer.write(b'hello world\n')
## 5.11 创建临时文件和文件夹
**解决方案**: 使用tempfile模块
- tempfile.TemporaryFile: 创建一个匿名的临时文件
- tempfile.NamedTemporaryFile: 创建一个有名字的 临时文件
- tempfile.TemporaryDirectory: 创建一个临时文件夹
from tempfile import TemporaryFile, NamedTemporaryFile, TemporaryDirectory

with TemporaryFile('w+t') as f:
    print(f.name)
    f.write('hello world')
    f.seek(0)
    data = f.read()
    print(data)

# /var/folders/z5/4l6129g92l33x8br4kzw_m3c0000gn/T/tmpdpdzi8i0w
with NamedTemporaryFile('w+t') as f:
    print(f.name)

# dirname is: /var/folders/z5/4l6129g92l33x8br4kzw_m3c0000gn/T/tmpxf0dkajd
with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)

```

7、《Python Cookbook 中文版，第3版》的笔记-第八章: 类与对象

8.1 改变对象的字符串显示

****问题****: 改变实例的打印或显示输出, 让它们更具有可读性.

****解决方案****: 重新定义__str__和__repr__

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)

p = Pair(3, 4)
# Pair(3, 4)
p
# (3, 4)
print(p)- {0.x}代表的是第一个参数的x属性.
- !r格式化代码指明输出使用\_\_repr\_\_()来代替默认的\_\_str\_\_()
```

8.2 自定义字符串的格式化

****问题****: 通过format()函数和字符串方法使得一个对象能支持自定义的格式化.

****解决方案****: 自定义__format__()方法 _formats = {

```
'ymd': '{d.year}-{d.month}-{d.day}',
'mdy': '{d.month}/{d.day}/{d.year}',
'dmy': '{d.day}/{d.month}/{d.year}'
}
```

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d = self)
```

```
d = Date(2012, 12, 21)
# 2012-12-21
print(format(d))
# 12/21/2012
print(format(d, 'mdy'))
# 21/12/2012
print(format(d, 'dmy'))
```

8.3 让对象支持上下文管理协议

****问题**:** 让对象支持上下文管理协议

****解决方案**:** 自定义`__enter__()`和`__exit__()`class C:

```
def __init__(self, values):
    print("__init__")
    self.values = values
def __enter__(self):
    print('__enter__')
    return self
def show(self):
    print(self.values)
def __exit__(self, exc_type, exc_val, exc_tb):
    print('__exit__')
```

```
c = C([1, 2, 3])
```

```
# __init__
```

```
# __enter__
```

```
# [1, 2, 3]
```

```
# __exit__
```

```
with c as f:
```

```
    f.show()
```

在with语句出现时, `__enter__()`方法被触发, 它返回的值会被赋值给as声明的变量. 然后, with语句块里面的代码开始执行. 最后, `__exit__()`方法被触发进行清理工作.

8.4 创建大量对象时节省内存方法

****问题**:** 程序创建大量的对象时候, 占用很大的内存.

****解决方案**:** 添加`__slots__`属性class Date:

```
__slots__ = ['year', 'month', 'day']
def __init__(self, year, month, day):
    self.year = year
    self.month = month
    self.day = day
def __str__(self):
    return "{}-{}-{}".format(self.year, self.month, self.day)
```

```
d = Date(2012, 12, 12)
```

```
print(d)
```

```
# exception
```

```
d.m = '111' - 添加__slots__属性后, 再也不能给对象添加特性.
```

8.5 在类中封装属性名

****问题**:** 你想封装类的实例上面的"私有"数据, 但是Python语言并没有访问控制.

****解决方案**:**

- 以单下划线命名私有变量
- 在继承中, 以双下划线命名变量

8.6 创建可管理的属性

****问题**:** 给实例增加除访问与修改外的其他处理逻辑.

****解决方案**:** 使用装饰器.class Person:

```
def __init__(self, name):
```

```
self._name = name
```

```
@property
def name(self):
    return self._name
@name.setter
def name(self, name):
    self._name = name
@name.deleter
def name(self):
    del self._name
```

```
p = Person("leicj")
# leicj
print(p.name)
p.name = "lgt"
# lgt
print(p.name)
del p.name
- @property定义getter函数
- @name.setter定义setter函数
- @name.deleter定义del函数
```

或者可以定义property函数class Person:

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

def set(self, name):
    self._name = name
def get(self):
    return self._name
def delete(self):
    del self._name
name = property(get, set, delete)
```

```
p = Person("leicj")
# leicj
print(p.name)
p.name = "lgt"
# lgt
print(p.name)
del p.name
## 8.7 调用父类方法
**问题**: 在子类中调用父类的某个已经被覆盖的方法
**解决方案**: 使用super()函数class A:
    def __init__(self):
        self.x = 0
```

```
class B(A):
```

```
def __init__(self):
    super().__init__()
    self.y = 1
```

- 在没有使用super情况下的意外情况

```
class Base:
    def __init__(self):
        print('Base.__init__')
```

```
class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

```
class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')
```

```
class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

```
# Base.__init__
```

```
# A.__init__
```

```
# Base.__init__
```

```
# B.__init__
```

```
# C.__init__
```

```
c = C()
```

- 使用super的情况

```
class Base:
    def __init__(self):
        print('Base.__init__')
```

```
class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')
```

```
class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')
```

```
class C(A, B):
    def __init__(self):
        super().__init__()
        print('C.__init__')
```

```
# Base.__init__
# B.__init__
# A.__init__
# C.__init__
c = C()
```

Python一般使用MRO列表实现继承.在使用super()函数时, Python会在MRO列表上继承搜索下一个类. 只要重定义的方法统一使用super()并只调用它一次, 那么控制流最终会遍历完整MRO列表, 每个方法也只會被调用一次.

8.8 子类中扩展property

****问题**:** 在子类中, 扩展定义在父类中的property的功能

****解决方案**:** 使用super()函数来实现扩展

```
class Person:
```

```
    def __init__(self, name):
```

```
        self._name = name
```

```
    @property
```

```
    def name(self):
```

```
        return self._name
```

```
    @name.setter
```

```
    def name(self, value):
```

```
        self._name = value
```

```
    @name.deleter
```

```
    def name(self):
```

```
        del self._name
```

```
class SubPerson(Person):
```

```
    @property
```

```
    def name(self):
```

```
        print('Getting name')
```

```
        return super().name
```

```
    @name.setter
```

```
    def name(self, value):
```

```
        # 这里需要通过类变量来读取name, 否则name会被当做一个临时变量
```

```
        # 而类变量必须通过super(SubPerson, SubPerson)来读取
```

```
        super(SubPerson, SubPerson).name.__set__(self, value)
```

```
    @name.deleter
```

```
    def name(self):
```

```
        super().name.__delete__(self)
```

```
s = SubPerson("leicj")
```

```
print(s.name)
```

```
s.name = 'lgt'
```

```
print(s.name)
```

8.9 创建新的类或实例属性

****问题**:** 创建一个新的拥有一些额外功能的实例属性类型.

****解决方案**:** 自定义__get__、__set__和__delete__ class Integer:

```
    def __init__(self, name):
```

```
        self.name = name
```

```
def __get__(self, instance, owner):
    if instance is None:
        return self
    else:
        print(instance.__dict__)
        return instance.__dict__[self.name]
def __set__(self, instance, value):
    if not isinstance(value, int):
        raise TypeError('Excepted an int')
    instance.__dict__[self.name] = value
def __delete__(self, instance):
    del instance.__dict__[self.name]
```

```
class Point:
    x = Integer('x1')
    y = Integer('y1')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
p = Point(2, 3)
# {'y1': 3, 'x1': 2}
# 2
```

```
print(p.x)
```

我们先看看`__get__()`函数, 如果描述器被当做一个类变量来访问, 那么instance参数被设置成None. 这种情况下, 就简单的返回描述器本身即可.

```
class Point:
    x = Integer('x1')
    y = Integer('y1')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
p = Point(2, 3)
# {'x1': 2, 'y1': 3}
# 2
print(p.x)
# <__main__.Integer object at 0x101979cc0>
print(Point.x)
```

描述器只能在类级别被定义, 而不能为每个实例单独定义. 如下面的代码的输出可能有点意外的:

```
class Point:
    def __init__(self, x, y):
        self.x = Integer('x1')
        self.y = Integer('y1')
        self.x = x
        self.y = y
```

```
p = Point(2, 3)
# 2
print(p.x) 这里Integer完全不起作用, 毕竟self.x/self.y被重新赋值.
```

以下是一个稍微复杂的描述器使用:

```
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]

def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate
```

```
@typeassert(name=str, shares=int, price=float)
```

```
class Stock:
```

```
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

8.10 使用延迟计算属性

****问题**:** 你想将一个只读属性定义成一个property, 并且只在访问的时候才会计算结果. 但是一旦被访问, 你希望结果被缓存起来, 不用每次都去计算.

****解决方案**:** 自定义`__get__`方法即可.

```
class lazyproperty:
    def __init__(self, func):
        self.func = func
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

```
import math
```

```
class Circle:
```

```
    def __init__(self, radius):
        self.radius = radius
    @lazyproperty
```



```
def area(self):
    print('Computing area')
    return math.pi * self.radius ** 2
```

```
@lazyproperty
def perimeter(self):
    print('Computing perimeter')
    return 2 * math.pi * self.radius
```

```
c = Circle(4.0)
# Computing area
# 50.26548245743669
print(c.area)
# 50.26548245743669
print(c.area)这里实现使用了小技巧, 使用\_\_get\_\_()方法在实例中存储计算出来的值, 这个实例使用相
同的名字作为它的property. 我们可以通过以下代码进行检查:c = Circle(4.0)
```

```
# {'radius': 4.0}
print(vars(c))
# Computing area
c.area
# {'radius': 4.0, 'area': 50.26548245743669}
print(vars(c))
```

8.12 定义接口或者抽象基类

****问题****: 定义一个接口或抽象类, 并且通过执行类型检查来确保子类实现了某些特定的方法.

****解决方案****: 使用abc模块可以很轻松的定义抽象基类.from abc import ABCMeta, abstractmethod

```
class IStream(metaclass=ABCMeta)
    @abstractmethod
    def read(self, maxbytes=-1):
        pass
    @abstractmethod
    def write(self, data):
        pass
```

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        pass
    def write(self, data):
        pass
```

8、《Python Cookbook 中文版，第3版》的笔记-第十一章: 网络与Web编程

11.1 作为客户端与HTTP服务交互

9、《Python Cookbook 中文版，第3版》的笔记-第二章: 字符串和文本

2.1 使用多个界定符分割字符串

****问题**:** 将一个字符串分割成多个字段, 但分隔符并不固定

****解决方案**:** 使用正则表达式 `re.splitline = 'asdf fjdk; afed, fjek,asdf, foo'`

```
import re
```

```
re.split(r'[:,\s]\s*', line)
```

Out[29]: ['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo'] 使用 `re.split()` 函数时候, 需要特别注意的是正则表达式中是否包含一个括号捕获分组. 如果使用了捕获分组, 那么被匹配的文本也将出现在结果列表中: `fields =`

```
re.split(r'(:|,|\s)\s*', line)
```

```
fields
```

```
Out[31]: ['asdf', '', 'fjdk', '', 'afed', '', 'fjek', '', 'asdf', '', 'foo']
```

2.2 字符串开头或结尾匹配

****问题**:** 匹配字符串的开头或结尾

****解决方案**:** 使用 `str.startswith()/str.endswith()`

****备注**:** 所传递的参数为单元素, 或者为元祖 `s = ['foo.c', 'bar.py', 'spam.c', 'spam.h']`

```
[name for name in s if name.endswith(('c', 'h'))]
```

```
Out[33]: ['foo.c', 'spam.c', 'spam.h']
```

2.3 用Shell通配符匹配字符串

****问题**:** 使用Unix Shell中常用的通配符(`*.py...`)去匹配文本字符串

****解决方案**:** 使用 `fnmatch` 的两个函数--`fnmatch()` (对大小写敏感) 和 `fnmatchcase()` (对大小写不敏感) from `fnmatch import fnmatch, fnmatchcase`

```
names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
```

```
[name for name in names if fnmatch(name, 'Dat*.csv')]
```

```
Out[40]: ['Dat1.csv', 'Dat2.csv']
```

2.4 字符串匹配和搜索

****问题**:** 匹配或者搜索特定模式的文本

****解决方案**:** 使用正则表达式的三个基本方法--`match`, `findall`, `finditer` `text1 = '2016-04-23'`

```
re.match(r'\d+-\d+-\d+', text1)
Out[42]: <_sre.SRE_Match object; span=(0, 10), match='2016-04-23'>
我们可以预编译正则表达式,并引入分组的概念:datepat = re.compile(r'(\d+)-(\d+)-(\d+)')
```

```
m = datepat.match('2016-04-23')
```

```
m.group()
Out[45]: '2016-04-23'
```

```
m.groups()
Out[46]: ('2016', '04', '23')
```

2.5 字符串搜索和替换

****问题****: 在字符串中搜索和替换

****解决方案****: 简单的使用str.replace,复杂的使用re模块中的sub函数.text = 'yeah, but no, but yeah, but no, but yeah'

```
text.replace('yeah', 'yep')
Out[48]: 'yep, but no, but yep, but no, but yep'
```

```
import re
```

```
datepat = re.compile(r'(\d+)-(\d+)-(\d+)')
```

```
datepat.sub(r'\2/\3/\1', 'today is 2016-04-23')
Out[51]: 'today is 04/23/2016'
```

2.15 字符串中插入变量

****问题****: 创建一个内嵌变量的字符串,变量被它的值所表示的字符串替换掉

****解决方案****: 通常使用format来解决.s = '{name} has {n} message'

```
s.format(name='guido', n=37)
Out[54]: 'guido has 37 message'
如果要被替换的变量能在变量域中找到,则需要结合使用format_map()和vars():name = 'guido'
```

```
n = 37
```

```
s.format_map(vars())
Out[57]: 'guido has 37 message'
```

10、《Python Cookbook 中文版 , 第 3 版》的笔记-第四章: 迭代器与生成器

4.1 迭代器基础知识

****问题****: 关于迭代器的基础知识

****解决方案****: 定义一个迭代器

- 一般一个可迭代的对象实现如下,而next函数调用的是__next__():

```
class C:
    def __init__(self, values):
        self.values = values
    def __iter__(self):
```

```

    return iter(self.values)
def __next__(self):
    if 0 == len(self.values):
        raise StopIteration
    return self.values.pop()
def __len__(self):
    return len(self.values)

```

```

c = C([1, 2, 3, 4, 5])
try:
    for i in range(len(c) + 1):
        print(next(c))
except StopIteration:
    pass

```

- 通常可以使用yield创建一个迭代模式

```

def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment

```

```

for n in frange(0, 10, 3):
    print(n)

```

- 定义__reversed__可以反向迭代一个序列

```

class C:
    def __init__(self, values):
        self.values = values
    def __iter__(self):
        return iter(self.values)
    def __reversed__(self):
        while len(self.values) > 0:
            yield self.values.pop()

```

```

c = C([1, 2, 3])
for i in reversed(c):
    print(i)

```

- 使用itertools.islice()来对迭代器进行切片操作

```

def count(n):
    while True:
        yield n
        n += 1

```

```

import itertools
c = count(0)
for x in itertools.islice(c, 10, 14):
    print(x)

```

- 当遍历一个可迭代对象时, 跳过开头的某些元素. 一般我们可以使用列表推导式的方式, 但是这样会过滤所有的元素(非开头的元素)

```

from itertools import dropwhile, islice

```

```

c = iter(['a', 'b', 'c', 1, 2, 3, 4, 'a', 'b', 'c'])
# 过滤开头的元素
# 1, 2, 3, 4, 'a', 'b', 'c'
for i in dropwhile(lambda key: key in ['a', 'b', 'c'], c):
    print(i)

# 知道过滤的个数, 则使用islice
c = iter(['a', 'b', 'c', 1, 2, 3, 4, 'a', 'b', 'c'])
# 1, 2, 3, 4, 'a', 'b', 'c'
for i in islice(c, 3, None):
    print(i)
## 4.2 排列组合的迭代
**问题**: 迭代遍历一个集合中元素的所有可能的排列或组合
**解决方案**: 使用itertools模块
- itertools.permutations(items, len): 接受一个集合和长度(默认长度为集合的长度)产生一个所有元素的可排列组合
from itertools import permutations
items = ['a', 'b', 'c']
#('a', 'b')
#('a', 'c')
#('b', 'a')
#('b', 'c')
#('c', 'a')
#('c', 'b')
for p in permutations(items, 2):
    print(p)
- itertools.combinations()可得到集合元素的所有组合(顺序不重要)
from itertools import combinations
items = ['a', 'b', 'c']
#('a', 'b')
#('a', 'c')
#('b', 'c')
for p in combinations(items, 2):
    print(p)
- itertools.combinations_with_replacement(): 允许元素被选择多次
from itertools import combinations_with_replacement
items = ['a', 'b', 'c']
#('a', 'a')
#('a', 'b')
#('a', 'c')
#('b', 'b')
#('b', 'c')
#('c', 'c')
for p in combinations_with_replacement(items, 2):
    print(p)
## 4.3 不同集合上元素的迭代
**问题**: 在多个对象上执行相同的操作
**解决方案**: 使用itertools.chainfrom itertools import chain

```

```

a = range(4)
b = ['a', 'b']
# 0 1 2 3 a b
for x in chain(a, b):
    print(x)
## 4.4 展开嵌套的序列
**问题**: 将多层嵌套的序列展开成单层嵌套
**解决方案**: yield + yield from
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, ['hello', 'world']]]
# 1, 2, 3, 4, 'hello', 'world'
for x in flatten(items):
    print(x)
## 4.5 顺序迭代合并后的排序迭代对象
**问题**: 将多个顺序集合进行迭代排序
**解决方案**: 使用heapq.merge()
import heapq
a = [1, 5, 3, 2]
b = [2, 8, 2, 1]
# 1 2 5 3 2 8 2 1
for c in heapq.merge(a, b):
    print(c)

a, b = sorted(a), sorted(b)
# 1 1 2 2 2 3 5 8
for c in heapq.merge(a, b):
    print(c)

```

11、《Python Cookbook 中文版，第 3 版》的笔记-第十三章: 脚本编程与系统管理

13.1 通过重定向/管道/文件接受输入

12、《Python Cookbook 中文版，第 3 版》的笔记-第十二章: 并发编程

12.1 启动与停止线程

13、《Python Cookbook 中文版，第 3 版》的笔记-第十四章: 测试,调试和异常

14.1 测试stdout输出

14、《Python Cookbook 中文版，第 3 版》的笔记-第九章: 元编程

9.1 在函数上添加包装器

版权说明

本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问:www.tushu000.com