《Python Cookbook》第三版

Release 3.0.0

熊能

Dec 09, 2017

目录

版權

書名：《Python Cookbook》3rd Edition
作者：David Beazley, Brian K. Jones
譯者：熊能
版本：第 3 版
出版社：O’Reilly Media, Inc.
出版日期：2013 年 5 月 08 日

更多發佈信息請參考 http://oreilly.com/catalog/errata.csp?isbn=9781449340377

前言

項目主頁

https://github.com/yidao620c/python3-cookbook

譯者的話

人生苦短，我用 Python！

譯者一直堅持使用 Python 3，因爲它代表了 Python 的未來。雖然向後兼容是它的硬傷，但是這個局面遲早會改變的，而且 Python 3 的未來需要每個人的幫助和支持。目前市面上的教程書籍，網上的手冊大部分基本都是 2.x 系列的，專門基於 3.x 系列的書籍少的可憐。

最近看到一本《Python Cookbook》3rd Edition，完全基於 Python 3，寫的也很不錯。爲了 Python 3 的普及，我也不自量力，想做點什麼事情。於是乎，就有了翻譯這本書的衝動了！這不是一項輕鬆的工作，卻是一件值得做的工作：不僅方便了別人，而且對自己翻譯能力也是一種鍛鍊和提升。

譯者會堅持對自己每一句的翻譯負責，力求高質量。但受能力限制，也難免有疏漏或者表意不當的地方。如果譯文中有什麼錯漏的地方請大家見諒，也歡迎大家隨時指正：yidao620@gmail.com

作者的話

自從 2008 年以來，Python 3 橫空出世並慢慢進化。Python 3 的流行一直被認爲需要很長一段時間。事實上，到我寫這本書的 2013 年，絕大部分的 Python 程序員仍然在生產環境中使用的是版本 2 系列，最主要是因爲 Python 3 不向後兼容。毫無疑問，對於工作在遺留代碼上的每個程序員來講，向後兼容是不得不考慮的問題。但是放眼未來，你就會發現 Python 3 給你帶來不一樣的驚喜。

正如 Python 3 代表未來一樣，新的《Python Cookbook》版本相比較之前的版本有了一個全新的改變。首先，也是最重要的，這意味着本書是一本非常前沿的參考書。書中所有代碼都是在 Python 3.3 版本下面編寫和測試的，並沒有考慮之前老版本的兼容性，也沒有標註舊版本下的解決方案。這樣子可能會有爭議，但是我們最終的目的是寫一本完全基於現代工具和語言的書籍。我們希望本書能夠指導人們使用 Python 3 編寫新的代碼或者升級之前的遺留代碼。

毫無疑問，編寫一本這樣的書給編輯工作帶來一定的挑戰。如果在網上搜索 Python 祕籍的話，會在諸如 ActiveState’s Python recipes 或者 Stack Overflow 的網站上搜到數以千計的有用的祕籍，但是其中絕大部分都已經是過時的了。這些祕籍除了是基於 Python 2 編寫之外，可能還有很多解決方案在不同的版本之間是不一樣的（比如 2.3 和 2.4 版本）。另外，它們還會經常使用一些過時的技術，這些可能已經內置到 Python 3.3 裏面去了。尋找完全基於 Python 3 的祕籍真的難上加難啊。

這本書的所有主題都是基於已經存在的代碼和技術，而不是專門去尋找 Python 3 特有的祕籍。在原有代碼基礎上，我們完全使用最新的 Python 技術去改造。所以，任

何想使用最新技術編寫代碼的程序員，都可以將本書當做一本很好的參考書籍。

在選擇要包含哪些祕籍方面，很明顯不可能編寫一本書囊括 Python 領域所有的東西。因此，我們優先選擇了 Python 語言核心部分，以及那些有着廣泛應用領域的問題。另外，其中有很多祕籍用來展示 Python 3 的新特性，這對於很多人來說是比較陌生的，哪怕是使用 Python 老版本的經驗豐富的程序員。這些示例程序也會偏向於展示一些有着廣泛應用的編程技術（即編程模式），而不是僅僅定位在一些具體的問題上。儘管也提及到了一些第三方包，但是本書主要定位在 Python 語言核心和標準庫。

這本書適合誰

這本書的目標讀者是那些想深入理解 Python 語言機制和現代編程風格的有經驗的 Python 程序員。本書大部分內容集中於在標準庫，框架和應用程序中廣泛使用的高級技術。本書所有示例均假設讀者具有一定的編程背景並且可以讀懂相關主題（比如基本的計算機科學知識，數據結構知識，算法複雜度，系統編程，並行，C 語言編程等）。另外，每個示例都只是一個入門指導，如果讀者想深入研究，需要自己去查閱更多資料。我們假定讀者可以很熟練的使用搜索引擎以及知道怎樣查詢在線的 Python 文檔。

有一些更加高級的祕籍，如果耐心閱讀，將有助於理解 Python 底層的工作原理。從中你將學到一些新的技巧和技術，並應用到你自己的代碼中去。

這本書不適合誰

這本書不適合 Python 的初學者。事實上，本書假定讀者具有 Python 教程或入門書籍中所教授的基礎知識。本書也不是那種快速參考手冊（例如快速查詢某個模塊下的某個函數）。本書旨在聚焦幾個最重要的主題，演示幾種可能的解決方案，提供一個跳板引導讀者進入一些更高級的內容（這些可以在網上或者參考手冊中找到）。

在線示例代碼

本書幾乎所有源代碼均可以在 http://github.com/dabeaz/python-cookbook 上面找到。作者歡迎各位讀者修正 bug，改進代碼和評論。

使用示例代碼

本書就是幫助你完成你的工作的。一般來講，只要是本書上面的示例代碼，你都可以隨時拿過去在你的源代碼和文檔中使用。除非你使用了大量的代碼，否則不需要向我們申請許可。例如，使用幾個代碼片段去完成一個程序不需要許可，販賣或者分發示例代碼的光盤則需要許可。引用本書和示例代碼去網上回答一個問題不需要許可，但是合併大量的代碼到你的正式產品文檔中去則需要許可。

我們不會要求你添加代碼的出處，但是如果你這麼做了，我們會很感激的。引用通常包含標題，作者，出版社，ISBN。例如：*Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

如果你覺得你對示例代碼的使用超出了合理使用或者上述列出的許可範圍，請隨時聯繫我們，我們的郵箱是 permissions@oreilly.com。

聯繫我們

請將關於本書的評論和問題發送給出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

我們爲本書建立了一個網頁，其中包含勘誤表，示例和一些其他信息。可以通過鏈接 http://oreil.ly/python_cookbook_3e 訪問。

關於本書的建議和技術性問題，請發送郵件至：bookquestions@oreilly.com

關於我們的書籍，討論會，新聞的更多信息，請訪問我們的網站：http://www.oreilly.com

在 Facebook 上找到我們：http://facebook.com/oreilly

在 Twitter 上關注我們：http://twitter.com/oreillymedia

在 YouTube 上觀看我們：http://www.youtube.com/oreillymedia

致謝

我們衷心感謝本書的技術校審人員 Jake Vanderplas，Robert Kern 和 Andrea Crotti 非常有用的評論和建議，還有 Python 社區的幫助和鼓勵。我們同樣感謝上一個版本的編輯 Alex Martelli，Anna Ravenscroft 和 David Ascher。儘管這個版本是新創作的，但是前一個版本爲本書提供了一個挑選主題和祕籍的初始框架。最後也是最重要的，我們要感謝所有早期預覽版本的讀者，感謝你們爲本書的改進提出的建議和意見。

第一章：數據結構和算法

Python 提供了大量的內置數據結構，包括列表，集合以及字典。大多數情況下使用這些數據結構是很簡單的。但是，我們也會經常碰到到諸如查詢，排序和過濾等等這些普遍存在的問題。因此，這一章的目的就是討論這些比較常見的問題和算法。另外，我們也會給出在集合模塊 `collections` 當中操作這些數據結構的方法。

1.1 解壓序列賦值給多個變量

問題

現在有一個包含 N 個元素的元組或者是序列，怎樣將它裏面的值解壓後同時賦值給 N 個變量？

解決方案

任何的序列（或者是可迭代對象）可以通過一個簡單的賦值語句解壓並賦值給多個變量。唯一的前提就是變量的數量必須跟序列元素的數量是一樣的。

代碼示例：

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)
>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

如果變量個數和序列元素的個數不匹配，會產生一個異常。

代碼示例：

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

討論

實際上，這種解壓賦值可以用在任何可迭代對象上面，而不僅僅是列表或者元組。包括字符串，文件對象，迭代器和生成器。

代碼示例：

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>
```

有時候，你可能只想解壓一部分，丟棄其他的值。對於這種情況 Python 並沒有提供特殊的語法。但是你可以使用任意變量名去佔位，到時候丟掉這些變量就行了。

代碼示例：

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

你必須保證你選用的那些佔位變量名在其他地方沒被使用到。

1.2 解壓可迭代對象賦值給多個變量

問題

如果一個可迭代對象的元素個數超過變量個數時，會拋出一個 `ValueError` 。那麼怎樣才能從這個可迭代對象中解壓出 N 個元素出來？

解決方案

Python 的星號表達式可以用來解決這個問題。比如，你在學習一門課程，在學期末的時候，你想統計下家庭作業的平均成績，但是排除掉第一個和最後一個分數。如果只有四個分數，你可能就直接去簡單的手動賦值，但如果有 24 個呢？這時候星號表達式就派上用場了：

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

另外一種情況，假設你現在有一些用戶的記錄列表，每條記錄包含一個名字、郵件，接着就是不確定數量的電話號碼。你可以像下面這樣分解這些記錄：

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

值得注意的是上面解壓出的 phone_numbers 變量永遠都是列表類型，不管解壓的電話號碼數量是多少（包括 0 個）。所以，任何使用到 phone_numbers 變量的代碼就不需要做多餘的類型檢查去確認它是否是列表類型了。

星號表達式也能用在列表的開始部分。比如，你有一個公司前 8 個月銷售數據的序列，但是你想看下最近一個月數據和前面 7 個月的平均值的對比。你可以這樣做：

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

下面是在 Python 解釋器中執行的結果：

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

討論

擴展的迭代解壓語法是專門爲解壓不確定個數或任意個數元素的可迭代對象而設計的。通常，這些可迭代對象的元素結構有確定的規則（比如第 1 個元素後面都是電話號碼），星號表達式讓開發人員可以很容易的利用這些規則來解壓出元素來。而不是通過一些比較複雜的手段去獲取這些關聯的元素值。

值得注意的是，星號表達式在迭代元素爲可變長元組的序列時是很有用的。比如，下面是一個帶有標籤的元組序列：

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

星號解壓語法在字符串操作的時候也會很有用，比如字符串的分割。

代碼示例：

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

有時候，你想解壓一些元素後丟棄它們，你不能簡單就使用 * ，但是你可以使用一個普通的廢棄名稱，比如 _ 或者 ign （ ignore ）。

代碼示例：

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

在很多函數式語言中，星號解壓語法跟列表處理有許多相似之處。比如，如果你有一個列表，你可以很容易的將它分割成前後兩部分：

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

如果你夠聰明的話，還能用這種分割語法去巧妙的實現遞歸算法。比如：

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

然後，由於語言層面的限制，遞歸併不是 Python 擅長的。因此，最後那個遞歸演示僅僅是個好奇的探索罷了，對這個不要太認真了。

1.3 保留最後 N 個元素

問題

在迭代操作或者其他操作的時候，怎樣只保留最後有限幾個元素的歷史記錄？

解決方案

保留有限歷史記錄正是 collections.deque 大顯身手的時候。比如，下面的代碼在多行上面做簡單的文本匹配，並返回匹配所在行的最後 N 行：

```
from collections import deque


def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open(r'../../cookbook/somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
```

```
            print(line, end='')
            print('-' * 20)
```

討論

我們在寫查詢元素的代碼時，通常會使用包含 `yield` 表達式的生成器函數，也就是我們上面示例代碼中的那樣。這樣可以將搜索過程代碼和使用搜索結果代碼解耦。如果你還不清楚什麼是生成器，請參看 4.3 節。

使用 `deque(maxlen=N)` 構造函數會新建一個固定大小的隊列。當新的元素加入並且這個隊列已滿的時候，最老的元素會自動被移除掉。

代碼示例：

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

儘管你也可以手動在一個列表上實現這一的操作（比如增加、刪除等等）。但是這裏的隊列方案會更加優雅並且運行得更快些。

更一般的，`deque` 類可以被用在任何你只需要一個簡單隊列數據結構的場合。如果你不設置最大隊列大小，那麼就會得到一個無限大小隊列，你可以在隊列的兩端執行添加和彈出元素的操作。

代碼示例：

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
```

```
>>> q.popleft()
4
```

在隊列兩端插入或刪除元素時間複雜度都是 O(1)，而在列表的開頭插入或刪除元素的時間複雜度爲 O(N)。

1.4 查找最大或最小的 N 個元素

問題

怎樣從一個集合中獲得最大或者最小的 N 個元素列表？

解決方案

heapq 模塊有兩個函數：nlargest() 和 nsmallest() 可以完美解決這個問題。

```
import heapq
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

兩個函數都能接受一個關鍵字參數，用於更復雜的數據結構中：

```
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]
cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

譯者注：上面代碼在對每個元素進行對比的時候，會以 price 的值進行比較。

討論

如果你想在一個集合中查找最小或最大的 N 個元素，並且 N 小於集合元素數量，那麼這些函數提供了很好的性能。因爲在底層實現裏面，首先會先將集合數據進行堆排序後放入一個列表中：

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
```

```
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

堆數據結構最重要的特徵是 heap[0] 永遠是最小的元素。並且剩餘的元素可以很容易的通過調用 heapq.heappop() 方法得到，該方法會先將第一個元素彈出來，然後用下一個最小的元素來取代被彈出元素（這種操作時間複雜度僅僅是 O(log N)，N 是堆大小）。比如，如果想要查找最小的 3 個元素，你可以這樣做：

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

當要查找的元素個數相對比較小的時候，函數 nlargest() 和 nsmallest() 是很合適的。如果你僅僅想查找唯一的最小或最大（N=1）的元素的話，那麼使用 min() 和 max() 函數會更快些。類似的，如果 N 的大小和集合大小接近的時候，通常先排序這個集合然後再使用切片操作會更快點（sorted(items)[:N] 或者是 sorted(items)[-N:]）。需要在正確場合使用函數 nlargest() 和 nsmallest() 才能發揮它們的優勢（如果 N 快接近集合大小了，那麼使用排序操作會更好些）。

儘管你沒有必要一定使用這裏的方法，但是堆數據結構的實現是一個很有趣並且值得你深入學習的東西。基本上只要是數據結構和算法書籍裏面都會有提及到。heapq 模塊的官方文檔裏面也詳細的介紹了堆數據結構底層的實現細節。

1.5 實現一個優先級隊列

問題

怎樣實現一個按優先級排序的隊列？並且在這個隊列上面每次 pop 操作總是返回優先級最高的那個元素

解決方案

下面的類利用 heapq 模塊實現了一個簡單的優先級隊列：

```
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]
```

下面是它的使用方式：

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

仔細觀察可以發現，第一個 pop() 操作返回優先級最高的元素。另外注意到如果兩個有着相同優先級的元素（foo 和 grok ），pop 操作按照它們被插入到隊列的順序返回的。

討論

這一小節我們主要關注 heapq 模塊的使用。函數 heapq.heappush() 和 heapq.heappop() 分別在隊列 _queue 上插入和刪除第一個元素，並且隊列 _queue 保證第一個元素擁有最高優先級（1.4 節已經討論過這個問題）。heappop() 函數總是返回”最小的”的元素，這就是保證隊列 pop 操作返回正確元素的關鍵。另外，由於 push 和 pop 操作時間複雜度爲 O(log N)，其中 N 是堆的大小，因此就算是 N 很大的時候它們運行速度也依舊很快。

在上面代碼中，隊列包含了一個 (-priority, index, item) 的元組。優先級爲負數的目的是使得元素按照優先級從高到低排序。這個跟普通的按優先級從低到高排序的堆排序恰巧相反。

index 變量的作用是保證同等優先級元素的正確排序。通過保存一個不斷增加的 index 下標變量，可以確保元素按照它們插入的順序排序。而且，index 變量也在相同優先級元素比較的時候起到重要作用。

爲了闡明這些，先假定 Item 實例是不支持排序的：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果你使用元組 (priority, item)，只要兩個元素的優先級不同就能比較。但是如果兩個元素優先級一樣的話，那麼比較操作就會跟之前一樣出錯：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通過引入另外的 index 變量組成三元組 (priority, index, item)，就能很好的避免上面的錯誤，因爲不可能有兩個元素有相同的 index 值。Python 在做元組比較時候，如果前面的比較已經可以確定結果了，後面的比較操作就不會發生了：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果你想在多個線程中使用同一個隊列，那麼你需要增加適當的鎖和信號量機制。可以查看 12.3 小節的例子演示是怎樣做的。

heapq 模塊的官方文檔有更詳細的例子程序以及對於堆理論及其實現的詳細說明。

1.6 字典中的鍵映射多個值

問題

怎樣實現一個鍵對應多個值的字典 (也叫 multidict)?

解决方案

一個字典就是一個鍵對應一個單值的映射。如果你想要一個鍵映射多個值，那麼你就需要將這多個值放到另外的容器中，比如列表或者集合裏面。比如，你可以像下面這樣構造這樣的字典：

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}
e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

選擇使用列表還是集合取決於你的實際需求。如果你想保持元素的插入順序就應該使用列表，如果想去掉重複元素就使用集合（並且不關心元素的順序問題）。

你可以很方便的使用 `collections` 模塊中的 `defaultdict` 來構造這樣的字典。`defaultdict` 的一個特徵是它會自動初始化每個 `key` 剛開始對應的值，所以你只需要關注添加元素操作了。比如：

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
```

需要注意的是，`defaultdict` 會自動爲將要訪問的鍵（就算目前字典中並不存在這樣的鍵）創建映射實體。如果你並不需要這樣的特性，你可以在一個普通的字典上使用 `setdefault()` 方法來代替。比如：

```
d = {} # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
```

但是很多程序員覺得 `setdefault()` 用起來有點彆扭。因爲每次調用都得創建一個新的初始值的實例（例子程序中的空列表 `[]`）。

討論

一般來講，創建一個多值映射字典是很簡單的。但是，如果你選擇自己實現的話，那麼對於值的初始化可能會有點麻煩，你可能會像下面這樣來實現：

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

如果使用 defaultdict 的話代碼就更加簡潔了：

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

這一小節所討論的問題跟數據處理中的記錄歸類問題有大的關聯。可以參考 1.15 小節的例子。

1.7 字典排序

問題

你想創建一個字典，並且在迭代或序列化這個字典的時候能夠控制元素的順序。

解決方案

爲了能控制一個字典中元素的順序，你可以使用 collections 模塊中的 OrderedDict 類。在迭代操作的時候它會保持元素被插入時的順序，示例如下：

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4
# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

當你想要構建一個將來需要序列化或編碼成其他格式的映射的時候，OrderedDict 是非常有用的。比如，你想精確控制以 JSON 編碼後字段的順序，你可以先使用 OrderedDict 來構建這樣的數據：

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

討論

OrderedDict 內部維護着一個根據鍵插入順序排序的雙向鏈表。每次當一個新的元素插入進來的時候，它會被放到鏈表的尾部。對於一個已經存在的鍵的重複賦值不會改變鍵的順序。

需要注意的是，一個 OrderedDict 的大小是一個普通字典的兩倍，因爲它內部維護着另外一個鏈表。所以如果你要構建一個需要大量 OrderedDict 實例的數據結構的時候（比如讀取 100,000 行 CSV 數據到一個 OrderedDict 列表中去），那麼你就得仔細權衡一下是否使用 OrderedDict 帶來的好處要大過額外內存消耗的影響。

1.8 字典的運算

問題

怎樣在數據字典中執行一些計算操作（比如求最小值、最大值、排序等等）?

解決方案

考慮下面的股票名和價格映射字典：

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

爲了對字典值執行計算操作，通常需要使用 zip() 函數先將鍵和值反轉過來。比如，下面是查找最小和最大股票價格和股票值的代碼：

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')
max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

類似的，可以使用 zip() 和 sorted() 函數來排列字典數據：

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                   (45.23, 'ACME'), (205.55, 'IBM'),
#                   (612.78, 'AAPL')]
```

執行這些計算的時候，需要注意的是 zip() 函數創建的是一個只能訪問一次的迭代器。比如，下面的代碼就會產生錯誤：

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names)) # OK
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

討論

如果你在一個字典上執行普通的數學運算，你會發現它們僅僅作用於鍵，而不是值。比如：

```
min(prices) # Returns 'AAPL'
max(prices) # Returns 'IBM'
```

這個結果並不是你想要的，因爲你想要在字典的值集合上執行這些計算。或許你會嘗試着使用字典的 values() 方法來解決這個問題：

```
min(prices.values()) # Returns 10.75
max(prices.values()) # Returns 612.78
```

不幸的是，通常這個結果同樣也不是你想要的。你可能還想要知道對應的鍵的信息 (比如那種股票價格是最低的 ?)。

你可以在 min() 和 max() 函數中提供 key 函數參數來獲取最小值或最大值對應的鍵的信息。比如：

```
min(prices, key=lambda k: prices[k]) # Returns 'FB'
max(prices, key=lambda k: prices[k]) # Returns 'AAPL'
```

但是，如果還想要得到最小值，你又得執行一次查找操作。比如：

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

前面的 zip() 函數方案通過將字典”反轉”爲 (值，鍵) 元組序列來解決了上述問題。當比較兩個元組的時候，值會先進行比較，然後纔是鍵。這樣的話你就能通過一條簡單的語句就能很輕鬆的實現在字典上的求最值和排序操作了。

需要注意的是在計算操作中使用到了 (值，鍵) 對。當多個實體擁有相同的值的時候，鍵會決定返回結果。比如，在執行 min() 和 max() 操作的時候，如果恰巧最小或最大值有重複的，那麼擁有最小或最大鍵的實體會返回：

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9 查找兩字典的相同點

問題

怎樣在兩個字典中尋尋找相同點（比如相同的鍵、相同的值等等）?

解決方案

考慮下面兩個字典：

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

b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

爲了尋找兩個字典的相同點，可以簡單的在兩字典的 `keys()` 或者 `items()` 方法返回結果上執行集合操作。比如：

```
# Find keys in common
a.keys() & b.keys() # { 'x', 'y' }
# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }
# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

這些操作也可以用於修改或者過濾字典元素。比如，假如你想以現有字典構造一個排除幾個指定鍵的新字典。下面利用字典推導來實現這樣的需求：

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

討論

一個字典就是一個鍵集合與值集合的映射關係。字典的 `keys()` 方法返回一個展現鍵集合的鍵視圖對象。鍵視圖的一個很少被瞭解的特性就是它們也支持集合操作，比如集合並、交、差運算。所以，如果你想對集合的鍵執行一些普通的集合操作，可以直接使用鍵視圖對象而不用先將它們轉換成一個 set。

字典的 `items()` 方法返回一個包含 (鍵，值) 對的元素視圖對象。這個對象同樣也支持集合操作，並且可以被用來查找兩個字典有哪些相同的鍵值對。

儘管字典的 `values()` 方法也是類似，但是它並不支持這裏介紹的集合操作。某種程度上是因爲值視圖不能保證所有的值互不相同，這樣會導致某些集合操作會出現問

題。不過，如果你硬要在值上面執行這些集合操作的話，你可以先將值集合轉換成 set，然後再執行集合運算就行了。

1.10 刪除序列相同元素並保持順序

問題

怎樣在一個序列上面保持元素順序的同時消除重複的值？

解決方案

如果序列上的值都是 hashable 類型，那麼可以很簡單的利用集合或者生成器來解決這個問題。比如：

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

下面是使用上述函數的例子：

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

這個方法僅僅在序列中元素爲 hashable 的時候才管用。如果你想消除元素不可哈希（比如 dict 類型）的序列中重複元素的話，你需要將上述代碼稍微改變一下，就像這樣：

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

這裏的 key 參數指定了一個函數，將序列元素轉換成 hashable 類型。下面是它的用法示例：

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

如果你想基於單個字段、屬性或者某個更大的數據結構來消除重複元素，第二種方案同樣可以勝任。

討論

如果你僅僅就是想消除重複元素，通常可以簡單的構造一個集合。比如：

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

然而，這種方法不能維護元素的順序，生成的結果中的元素位置被打亂。而上面的方法可以避免這種情況。

在本節中我們使用了生成器函數讓我們的函數更加通用，不僅僅是侷限於列表處理。比如，如果如果你想讀取一個文件，消除重複行，你可以很容易像這樣做：

```
with open(somefile,'r') as f:
for line in dedupe(f):
    ...
```

上述 key 函數參數模仿了 sorted() , min() 和 max() 等內置函數的相似功能。可以參考 1.8 和 1.13 小節瞭解更多。

1.11 命名切片

問題

你的程序已經出現一大堆已無法直視的硬編碼切片下標，然後你想清理下代碼。

解決方案

假定你有一段代碼要從一個記錄字符串中幾個固定位置提取出特定的數據字段 (比如文件或類似格式):

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '.....................100 .......513.25 ..........'
cost = int(record[20:23]) * float(record[31:37])
```

與其那樣寫，爲什麼不想這樣命名切片呢：

```
SHARES = slice(20, 23)
PRICE = slice(31, 37)
cost = int(record[SHARES]) * float(record[PRICE])
```

第二種版本中，你避免了大量無法理解的硬編碼下標，使得你的代碼更加清晰可讀了。

討論

一般來講，代碼中如果出現大量的硬編碼下標值會使得可讀性和可維護性大大降低。比如，如果你回過來看看一年前你寫的代碼，你會摸着腦袋想那時候自己到底想幹嘛啊。這裏的解決方案是一個很簡單的方法讓你更加清晰的表達代碼到底要做什麼。

內置的 `slice()` 函數創建了一個切片對象，可以被用在任何切片允許使用的地方。比如：

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

如果你有一個切片對象 `a`，你可以分別調用它的 `a.start` , `a.stop` , `a.step` 屬性來獲取更多的信息。比如：

```
>>> a = slice(5, 50, 2)
>>> a.start
5
>>> a.stop
50
>>> a.step
2
>>>
```

另外，你還能通過調用切片的 `indices(size)` 方法將它映射到一個確定大小的序列上，這個方法返回一個三元組（`start`，`stop`，`step`），所有值都會被合適的縮小以滿足邊界限制，從而使用的時候避免出現 `IndexError` 異常。比如：

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
```

```
d
>>>
```

1.12 序列中出現次數最多的元素

問題

怎樣找出一個序列中出現次數最多的元素呢？

解決方案

collections.Counter 類就是專門爲這類問題而設計的，它甚至有一個有用的 most_common() 方法直接給了你答案。

爲了演示，先假設你有一個單詞列表並且想找出哪個單詞出現頻率最高。你可以這樣做：

```
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)
# 出現頻率最高的 3 個單詞
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

討論

作爲輸入，Counter 對象可以接受任意的由可哈希（hashable）元素構成的序列對象。在底層實現上，一個 Counter 對象就是一個字典，將元素映射到它出現的次數上。比如：

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

如果你想手動增加計數，可以簡單的用加法：

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
```

```
...     word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

或者你可以使用 update() 方法：

```
>>> word_counts.update(morewords)
>>>
```

Counter 實例一個鮮爲人知的特性是它們可以很容易的跟數學運算操作相結合。比如：

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
"you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
'my': 1, 'why': 1})
>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
'looking': 1, 'are': 1, 'under': 1, 'you': 1})
>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
"you're": 1, "don't": 1, 'under': 1})
>>>
```

毫無疑問，Counter 對象在幾乎所有需要製表或者計數數據的場合是非常有用的工具。在解決這類問題的時候你應該優先選擇它，而不是手動的利用字典去實現。

1.13 通過某個關鍵字排序一個字典列表

問題

你有一個字典列表，你想根據某個或某幾個字典字段來排序這個列表。

解決方案

通過使用 operator 模塊的 itemgetter 函數，可以非常容易的排序這樣的數據結構。假設你從數據庫中檢索出來網站會員信息列表，並且以下列的數據結構返回：

```
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}
]
```

根據任意的字典字段來排序輸入結果行是很容易實現的，代碼示例：

```
from operator import itemgetter
rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))
print(rows_by_fname)
print(rows_by_uid)
```

代碼的輸出如下：

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]
[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

itemgetter() 函數也支持多個 keys，比如下面的代碼

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

會產生如下的輸出：

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
{'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

討論

在上面例子中，rows 被傳遞給接受一個關鍵字參數的 sorted() 內置函數。這個參數是 callable 類型，並且從 rows 中接受一個單一元素，然後返回被用來排序的值。itemgetter() 函數就是負責創建這個 callable 對象的。

operator.itemgetter() 函數有一個被 rows 中的記錄用來查找值的索引參數。可以是一個字典鍵名稱，一個整形值或者任何能夠傳入一個對象的 __getitem__() 方法的值。如果你傳入多個索引參數給 itemgetter()，它生成的 callable 對象會返回一個包含所有元素值的元組，並且 sorted() 函數會根據這個元組中元素順序去排序。但你想要同時在幾個字段上面進行排序（比如通過姓和名來排序，也就是例子中的那樣）的時候這種方法是很有用的。

itemgetter() 有時候也可以用 lambda 表達式代替，比如：

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

這種方案也不錯。但是，使用 itemgetter() 方式會運行的稍微快點。因此，如果你對性能要求比較高的話就使用 itemgetter() 方式。

最後，不要忘了這節中展示的技術也同樣適用於 min() 和 max() 等函數。比如：

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14 排序不支持原生比較的對象

問題

你想排序類型相同的對象，但是他們不支持原生的比較操作。

解決方案

內置的 sorted() 函數有一個關鍵字參數 key ，可以傳入一個 callable 對象給它，這個 callable 對象對每個傳入的對象返回一個值，這個值會被 sorted 用來排序這些對象。比如，如果你在應用程序裏面有一個 User 實例序列，並且你希望通過他們的 user_id 屬性進行排序，你可以提供一個以 User 實例作爲輸入並輸出對應 user_id 值的 callable 對象。比如：

```
class User:
    def __init__(self, user_id):
        self.user_id = user_id

    def __repr__(self):
        return 'User({})'.format(self.user_id)


def sort_notcompare():
    users = [User(23), User(3), User(99)]
    print(users)
    print(sorted(users, key=lambda u: u.user_id))
```

另外一種方式是使用 operator.attrgetter() 來代替 lambda 函數：

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

討論

選擇使用 lambda 函數或者是 attrgetter() 可能取決於個人喜好。但是，attrgetter() 函數通常會運行的快點，並且還能同時允許多個字段進行比較。這個跟 operator.itemgetter() 函數作用於字典類型很類似（參考 1.13 小節）。例如，如果 User 實例還有一個 first_name 和 last_name 屬性，那麼可以向下面這樣排序：

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

同樣需要注意的是，這一小節用到的技術同樣適用於像 min() 和 max() 之類的函數。比如：

```
>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>
```

1.15 通過某個字段將記錄分組

問題

你有一個字典或者實例的序列，然後你想根據某個特定的字段比如 date 來分組迭代訪問。

解決方案

itertools.groupby() 函數對於這樣的數據分組操作非常實用。爲了演示，假設你已經有了下列的字典列表：

```
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'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

現在假設你想在按 date 分組後的數據塊上進行迭代。爲了這樣做，你首先需要按照指定的字段（這裏就是 date）排序，然後調用 itertools.groupby() 函數：

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))
# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

運行結果：

```
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'}
  {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

討論

groupby() 函數掃描整個序列並且查找連續相同值（或者根據指定 key 函數返回值相同）的元素序列。在每次迭代的時候，它會返回一個值和一個迭代器對象，這個迭代器對象可以生成元素值全部等於上面那個值的組中所有對象。

一個非常重要的準備步驟是要根據指定的字段將數據排序。因爲 groupby() 僅僅檢查連續的元素，如果事先並沒有排序完成的話，分組函數將得不到想要的結果。

如果你僅僅只是想根據 date 字段將數據分組到一個大的數據結構中去，並且允許隨機訪問，那麼你最好使用 defaultdict() 來構建一個多值字典，關於多值字典已經在 1.6 小節有過詳細的介紹。比如：

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

這樣的話你可以很輕鬆的就能對每個指定日期訪問對應的記錄：

```
>>> for r in rows_by_date['07/01/2012']:
... print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
```

```
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

在上面這個例子中，我們沒有必要先將記錄排序。因此，如果對內存佔用不是很關心，這種方式會比先排序然後再通過 groupby() 函數迭代的方式運行得快一些。

1.16 過濾序列元素

問題

你有一個數據序列，想利用一些規則從中提取出需要的值或者是縮短序列

解決方案

最簡單的過濾序列元素的方法就是使用列表推導。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

使用列表推導的一個潛在缺陷就是如果輸入非常大的時候會產生一個非常大的結果集，佔用大量內存。如果你對內存比較敏感，那麼你可以使用生成器表達式迭代產生過濾的元素。比如：

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
... print(x)
...
1
4
10
2
3
>>>
```

有時候，過濾規則比較複雜，不能簡單的在列表推導或者生成器表達式中表達出來。比如，假設過濾的時候需要處理一些異常或者其他複雜情況。這時候你可以將過濾代碼放到一個函數中，然後使用內建的 filter() 函數。示例如下：

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']
def is_int(val):
    try:
        x = int(val)
```

```
        return True
    except ValueError:
        return False
ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

filter() 函數創建了一個迭代器，因此如果你想得到一個列表的話，就得像示例那樣使用 list() 去轉換。

討論

列表推導和生成器表達式通常情況下是過濾數據最簡單的方式。其實它們還能在過濾的時候轉換數據。比如：

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

過濾操作的一個變種就是將不符合條件的值用新的值代替，而不是丟棄它們。比如，在一列數據中你可能不僅想找到正數，而且還想將不是正數的數替換成指定的數。通過將過濾條件放到條件表達式中去，可以很容易的解決這個問題，就像這樣：

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

另外一個值得關注的過濾工具就是 itertools.compress() ，它以一個 iterable 對象和一個相對應的 Boolean 選擇器序列作爲輸入參數。然後輸出 iterable 對象中對應選擇器爲 True 的元素。當你需要用另外一個相關聯的序列來過濾某個序列的時候，這個函數是非常有用的。比如，假如現在你有下面兩列數據：

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]
counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

現在你想將那些對應 count 值大於 5 的地址全部輸出，那麼你可以這樣做：

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '1060 W ADDISON', '4801 N BROADWAY']
>>>
```

這裏的關鍵點在於先創建一個 Boolean 序列，指示哪些元素符合條件。然後 compress() 函數根據這個序列去選擇輸出對應位置爲 True 的元素。

和 filter() 函數類似，compress() 也是返回的一個迭代器。因此，如果你需要得到一個列表，那麼你需要使用 list() 來將結果轉換爲列表類型。

1.17 從字典中提取子集

問題

你想構造一個字典，它是另外一個字典的子集。

解決方案

最簡單的方式是使用字典推導。比如：

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
# Make a dictionary of all prices over 200
p1 = {key: value for key, value in prices.items() if value > 200}
# Make a dictionary of tech stocks
tech_names = {'AAPL', 'IBM', 'HPQ', 'MSFT'}
p2 = {key: value for key, value in prices.items() if key in tech_names}
```

討論

大多數情況下字典推導能做到的，通過創建一個元組序列然後把它傳給 dict() 函數也能實現。比如：

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

但是，字典推導方式表意更清晰，並且實際上也會運行的更快些（在這個例子中，實際測試幾乎比 dcit() 函數方式快整整一倍）。

有時候完成同一件事會有多種方式。比如，第二個例子程序也可以像這樣重寫：

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

但是，運行時間測試結果顯示這種方案大概比第一種方案慢 1.6 倍。如果對程序運行性能要求比較高的話，需要花點時間去做計時測試。關於更多計時和性能測試，可以參考 14.13 小節。

1.18 映射名稱到序列元素

問題

你有一段通過下標訪問列表或者元組中元素的代碼，但是這樣有時候會使得你的代碼難以閱讀，於是你想通過名稱來訪問元素。

解決方案

collections.namedtuple() 函數通過使用一個普通的元組對象來幫你解決這個問題。這個函數實際上是一個返回 Python 中標準元組類型子類的一個工廠方法。你需要傳遞一個類型名和你需要的字段給它，然後它就會返回一個類，你可以初始化這個類，爲你定義的字段傳遞值等。代碼示例：

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

儘管 namedtuple 的實例看起來像一個普通的類實例，但是它跟元組類型是可交換的，支持所有的普通元組操作，比如索引和解壓。比如：

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

命名元組的一個主要用途是將你的代碼從下標操作中解脫出來。因此，如果你從數據庫調用中返回了一個很大的元組列表，通過下標去操作其中的元素，當你在表中添加了新的列的時候你的代碼可能就會出錯了。但是如果你使用了命名元組，那麼就不會有這樣的顧慮。

爲了說明清楚，下面是使用普通元組的代碼：

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

下標操作通常會讓代碼表意不清晰，並且非常依賴記錄的結構。下面是使用命名元組的版本：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

討論

命名元組另一個用途就是作爲字典的替代，因爲字典存儲需要更多的內存空間。如果你需要構建一個非常大的包含字典的數據結構，那麼使用命名元組會更加高效。但是需要注意的是，不像字典那樣，一個命名元組是不可更改的。比如：

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

如果你真的需要改變屬性的值，那麼可以使用命名元組實例的 `_replace()` 方法，它會創建一個全新的命名元組並將對應的字段用新的值取代。比如：

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

`_replace()` 方法還有一個很有用的特性就是當你的命名元組擁有可選或者缺失字

段時候，它是一個非常方便的填充數據的方法。你可以先創建一個包含缺省值的原型元組，然後使用 _replace() 方法創建新的值被更新過的實例。比如：

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

下面是它的使用方法：

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

最後要說的是，如果你的目標是定義一個需要更新很多實例屬性的高效數據結構，那麼命名元組並不是你的最佳選擇。這時候你應該考慮定義一個包含 __slots__ 方法的類（參考 8.4 小節）。

1.19 轉換並同時計算數據

問題

你需要在數據序列上執行聚集函數（比如 sum() , min() , max() ），但是首先你需要先轉換或者過濾數據

解決方案

一個非常優雅的方式去結合數據計算與轉換就是使用一個生成器表達式參數。比如，如果你想計算平方和，可以像下面這樣做：

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

下面是更多的例子：

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
```

```
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')
# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))
# Data reduction across fields of a data structure
portfolio = [
    {'name':'GOOG', 'shares': 50},
    {'name':'YHOO', 'shares': 75},
    {'name':'AOL', 'shares': 20},
    {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

討論

上面的示例向你演示了當生成器表達式作爲一個單獨參數傳遞給函數時候的巧妙語法（你並不需要多加一個括號）。比如，下面這些語句是等效的：

```
s = sum((x * x for x in nums)) # 顯示的傳遞一個生成器表達式對象
s = sum(x * x for x in nums) # 更加優雅的實現方式，省略了括號
```

使用一個生成器表達式作爲參數會比先創建一個臨時列表更加高效和優雅。比如，如果你不使用生成器表達式的話，你可能會考慮使用下面的實現方式：

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

這種方式同樣可以達到想要的效果，但是它會多一個步驟，先創建一個額外的列表。對於小型列表可能沒什麼關係，但是如果元素數量非常大的時候，它會創建一個巨大的僅僅被使用一次就被丟棄的臨時數據結構。而生成器方案會以迭代的方式轉換數據，因此更省內存。

在使用一些聚集函數比如 `min()` 和 `max()` 的時候你可能更加傾向於使用生成器版本，它們接受的一個 key 關鍵字參數或許對你很有幫助。比如，在上面的證券例子中，你可能會考慮下面的實現版本：

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)
# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20 合併多個字典或映射

問題

現在有多個字典或者映射，你想將它們從邏輯上合併爲一個單一的映射後執行某些操作，比如查找值或者檢查某些鍵是否存在。

解決方案

假如你有如下兩個字典:

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

現在假設你必須在兩個字典中執行查找操作（比如先從 a 中找，如果找不到再在 b 中找）。一個非常簡單的解決方案就是使用 collections 模塊中的 ChainMap 類。比如：

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x']) # Outputs 1 (from a)
print(c['y']) # Outputs 2 (from b)
print(c['z']) # Outputs 3 (from a)
```

討論

一個 ChainMap 接受多個字典並將它們在邏輯上變爲一個字典。然後，這些字典並不是真的合併在一起了，ChainMap 類只是在內部創建了一個容納這些字典的列表並重新定義了一些常見的字典操作來遍歷這個列表。大部分字典操作都是可以正常使用的，比如：

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

如果出現重複鍵，那麼第一次出現的映射值會被返回。因此，例子程序中的 c['z'] 總是會返回字典 a 中對應的值，而不是 b 中對應的值。

對於字典的更新或刪除操作總是影響的是列表中第一個字典。比如：

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
```

```
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap 對於編程語言中的作用範圍變量（比如 globals , locals 等）是非常有用的。事實上，有一些方法可以使它變得簡單：

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

作爲 ChainMap 的替代，你可能會考慮使用 update() 方法將兩個字典合併。比如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

這樣也能行得通，但是它需要你創建一個完全不同的字典對象（或者是破壞現有字典結構）。同時，如果原字典做了更新，這種改變不會反應到新的合併字典中去。比如：

```
>>> a['x'] = 13
>>> merged['x']
1
```

ChainMap 使用原來的字典，它自己不創建新的字典。所以它並不會產生上面所說的結果，比如：

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x'] # Notice change to merged dicts
42
>>>
```

第二章：字符串和文本

幾乎所有有用的程序都會涉及到某些文本處理，不管是解析數據還是產生輸出。這一章將重點關注文本的操作處理，比如提取字符串，搜索，替換以及解析等。大部分的問題都能簡單的調用字符串的內建方法完成。但是，一些更爲複雜的操作可能需要正則表達式或者強大的解析器，所有這些主題我們都會詳細講解。並且在操作 Unicode 時候碰到的一些棘手的問題在這裏也會被提及到。

2.1 使用多個界定符分割字符串

問題

你需要將一個字符串分割爲多個字段，但是分隔符 (還有周圍的空格) 並不是固定的。

解決方案

string 對象的 split() 方法只適應於非常簡單的字符串分割情形，它並不允許有多個分隔符或者是分隔符周圍不確定的空格。當你需要更加靈活的切割字符串的時候，最好使用 re.split() 方法：

```
>>> line = 'asdf fjdk; afed, fjek,asdf, foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

討論

函數 re.split() 是非常實用的，因爲它允許你爲分隔符指定多個正則模式。比如，在上面的例子中，分隔符可以是逗號，分號或者是空格，並且後面緊跟着任意個的空格。只要這個模式被找到，那麼匹配的分隔符兩邊的實體都會被當成是結果中的元素返回。返回結果爲一個字段列表，這個跟 str.split() 返回值類型是一樣的。

當你使用 re.split() 函數時候，需要特別注意的是正則表達式中是否包含一個括號捕獲分組。如果使用了捕獲分組，那麼被匹配的文本也將出現在結果列表中。比如，觀察一下這段代碼運行後的結果：

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```

獲取分割字符在某些情況下也是有用的。比如，你可能想保留分割字符串，用來在後面重新構造一個新的輸出字符串：

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']
>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

如果你不想保留分割字符串到結果列表中去，但仍然需要使用到括號來分組正則表達式的話，確保你的分組是非捕獲分組，形如 `(?:...)` 。比如：

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

2.2 字符串開頭或結尾匹配

問題

你需要通過指定的文本模式去檢查字符串的開頭或者結尾，比如文件名後綴，URL Scheme 等等。

解決方案

檢查字符串開頭或結尾的一個簡單方法是使用 `str.startswith()` 或者是 `str.endswith()` 方法。比如：

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

如果你想檢查多種匹配可能，只需要將所有的匹配項放入到一個元組中去，然後傳給 `startswith()` 或者 `endswith()` 方法：

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
```

```
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

下面是另一個例子：

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
            return f.read()
```

奇怪的是，這個方法中必須要輸入一個元組作爲參數。如果你恰巧有一個 `list` 或者 `set` 類型的選擇項，要確保傳遞參數前先調用 `tuple()` 將其轉換爲元組類型。比如：

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

討論

`startswith()` 和 `endswith()` 方法提供了一個非常方便的方式去做字符串開頭和結尾的檢查。類似的操作也可以使用切片來實現，但是代碼看起來沒有那麼優雅。比如：

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

你可以能還想使用正則表達式去實現，比如：

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
```

```
<_sre.SRE_Match object at 0x101253098>
>>>
```

這種方式也行得通，但是對於簡單的匹配實在是有點小材大用了，本節中的方法更加簡單並且運行會更快些。

最後提一下，當和其他操作比如普通數據聚合相結合的時候 `startswith()` 和 `endswith()` 方法是很不錯的。比如，下面這個語句檢查某個文件夾中是否存在指定的文件類型：

```
if any(name.endswith(('.c', '.h')) for name in listdir(dirname)):
...
```

2.3 用 Shell 通配符匹配字符串

問題

你想使用 **Unix Shell** 中常用的通配符 (比如 `*.py` , `Dat[0-9]*.csv` 等) 去匹配文本字符串

解決方案

`fnmatch` 模塊提供了兩個函數——`fnmatch()` 和 `fnmatchcase()` ，可以用來實現這樣的匹配。用法如下：

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

`fnmatch()` 函數使用底層操作系統的大小寫敏感規則 (不同的系統是不一樣的) 來匹配模式。比如：

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False
>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

如果你對這個區別很在意，可以使用 fnmatchcase() 來代替。它完全使用你的模式大小寫匹配。比如：

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

這兩個函數通常會被忽略的一個特性是在處理非文件名的字符串時候它們也是很有用的。比如，假設你有一個街道地址的列表數據：

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

你可以像這樣寫列表推導：

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

討論

fnmatch() 函數匹配能力介於簡單的字符串方法和強大的正則表達式之間。如果在數據處理操作中只需要簡單的通配符就能完成的時候，這通常是一個比較合理的方案。

如果你的代碼需要做文件名的匹配，最好使用 glob 模塊。參考 5.13 小節。

2.4 字符串匹配和搜索

問題

你想匹配或者搜索特定模式的文本

解決方案

如果你想匹配的是字面字符串，那麼你通常只需要調用基本字符串方法就行，比如 str.find() , str.endswith() , str.startswith() 或者類似的方法：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Exact match
>>> text == 'yeah'
```

```
False
>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

對於複雜的匹配需要使用正則表達式和 re 模塊。爲了解釋正則表達式的基本原理，假設你想匹配數字格式的日期字符串比如 11/27/2012 ，你可以這樣做：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

如果你想使用同一個模式去做多次匹配，你應該先將模式字符串預編譯爲模式對象。比如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if datepat.match(text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

match() 總是從字符串開始去匹配，如果你想查找字符串任意部分的模式出現位置，使用 findall() 方法去代替。比如：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

在定義正則式的時候，通常會利用括號去捕獲分組。比如：

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

捕獲分組可以使得後面的處理更加簡單，因爲可以分別將每個組的內容提取出來。比如：

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>
>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

findall() 方法會搜索文本並以列表形式返回所有的匹配。如果你想以迭代方式返回匹配，可以使用 finditer() 方法來代替，比如：

```
>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
```

```
('3', '13', '2013')
>>>
```

討論

關於正則表達式理論的教程已經超出了本書的範圍。不過，這一節闡述了使用 re 模塊進行匹配和搜索文本的最基本方法。核心步驟就是先使用 re.compile() 編譯正則表達式字符串，然後使用 match() , findall() 或者 finditer() 等方法。

當寫正則式字符串的時候，相對普遍的做法是使用原始字符串比如 r'(\d+)/(\d+)/(\d+)' 。這種字符串將不去解析反斜槓，這在正則表達式中是很有用的。如果不這樣做的話，你必須使用兩個反斜槓，類似 '(\\d+)/(\\d+)/(\\d+)' 。

需要注意的是 match() 方法僅僅檢查字符串的開始部分。它的匹配結果有可能並不是你期望的那樣。比如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

如果你想精確匹配，確保你的正則表達式以 $ 結尾，就像這麼這樣：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最後，如果你僅僅是做一次簡單的文本匹配/搜索操作的話，可以略過編譯部分，直接使用 re 模塊級別的函數。比如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

但是需要注意的是，如果你打算做大量的匹配和搜索操作的話，最好先編譯正則表達式，然後再重複使用它。模塊級別的函數會將最近編譯過的模式緩存起來，因此並不會消耗太多的性能，但是如果使用預編譯模式的話，你將會減少查找和一些額外的處理損耗。

2.5 字符串搜索和替換

問題

你想在字符串中搜索和匹配指定的文本模式

解決方案

對於簡單的字面模式，直接使用 str.replace() 方法即可，比如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

對於複雜的模式，請使用 re 模塊中的 sub() 函數。爲了說明這個，假設你想將形式爲 11/27/2012 的日期字符串改成 2012-11-27 。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

sub() 函數中的第一個參數是被匹配的模式，第二個參數是替換模式。反斜槓數字比如 \3 指向前面模式的捕獲組號。

如果你打算用相同的模式做多次替換，考慮先編譯它來提升性能。比如：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

對於更加複雜的替換，可以傳遞一個替換回調函數來代替，比如：

```
>>> from calendar import month_abbr
>>> def change_date(m):
... mon_name = month_abbr[int(m.group(1))]
... return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

一個替換回調函數的參數是一個 match 對象，也就是 match() 或者 find() 返回的對象。使用 group() 方法來提取特定的匹配部分。回調函數最後返回替換字符串。

如果除了替換後的結果外，你還想知道有多少替換髮生了，可以使用 re.subn() 來代替。比如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

討論

關於正則表達式搜索和替換，上面演示的 sub() 方法基本已經涵蓋了所有。其實最難的部分就是編寫正則表達式模式，這個最好是留給讀者自己去練習了。

2.6 字符串忽略大小寫的搜索替換

問題

你需要以忽略大小寫的方式搜索與替換文本字符串

解決方案

爲了在文本操作時忽略大小寫，你需要在使用 re 模塊的時候給這些操作提供 re.IGNORECASE 標誌參數。比如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

最後的那個例子揭示了一個小缺陷，替換字符串並不會自動跟被匹配字符串的大小寫保持一致。爲了修復這個，你可能需要一個輔助函數，就像下面的這樣：

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

下面是使用上述函數的方法：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

譯者注：matchcase('snake') 返回了一個回調函數 (參數必須是 match 對象)，前面一節提到過，sub() 函數除了接受替換字符串外，還能接受一個回調函數。

討論

對於一般的忽略大小寫的匹配操作，簡單的傳遞一個 `re.IGNORECASE` 標誌參數就已經足夠了。但是需要注意的是，這個對於某些需要大小寫轉換的 Unicode 匹配可能還不夠，參考 2.10 小節瞭解更多細節。

2.7 最短匹配模式

問題

你正在試着用正則表達式匹配某個文本模式，但是它找到的是模式的最長可能匹配。而你想修改它變成查找最短的可能匹配。

解決方案

這個問題一般出現在需要匹配一對分隔符之間的文本的時候 (比如引號包含的字符串)。爲了說明清楚，考慮如下的例子：

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

在這個例子中，模式 `r'\"(.*)\"'` 的意圖是匹配被雙引號包含的文本。但是在正則表達式中 * 操作符是貪婪的，因此匹配操作會查找最長的可能匹配。於是在第二個例子中搜索 `text2` 的時候返回結果並不是我們想要的。

爲了修正這個問題，可以在模式中的 * 操作符後面加上? 修飾符，就像這樣：

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

這樣就使得匹配變成非貪婪模式，從而得到最短的匹配，也就是我們想要的結果。

討論

這一節展示了在寫包含點 (.) 字符的正則表達式的時候遇到的一些常見問題。在一個模式字符串中，點 (.) 匹配除了換行外的任何字符。然而，如果你將點 (.) 號放在開始與結束符 (比如引號) 之間的時候，那麼匹配操作會查找符合模式的最長可能匹配。這樣通常會導致很多中間的被開始與結束符包含的文本被忽略掉，並最終被包含在匹配結果字符串中返回。通過在 * 或者 + 這樣的操作符後面添加一個 ? 可以強制匹配算法改成尋找最短的可能匹配。

2.8 多行匹配模式

問題

你正在試着使用正則表達式去匹配一大塊的文本，而你需要跨越多行去匹配。

解決方案

這個問題很典型的出現在當你用點 (.) 去匹配任意字符的時候，忘記了點 (.) 不能匹配換行符的事實。比如，假設你想試着去匹配 C 語言分割的註釋：

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
... multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

爲了修正這個問題，你可以修改模式字符串，增加對換行的支持。比如：

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n multiline comment ']
>>>
```

在這個模式中，(?:.|\n) 指定了一個非捕獲組 (也就是它定義了一個僅僅用來做匹配，而不能通過單獨捕獲或者編號的組)。

討論

re.compile() 函數接受一個標誌參數叫 re.DOTALL ，在這裏非常有用。它可以讓正則表達式中的點 (.) 匹配包括換行符在內的任意字符。比如：

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n multiline comment ']
```

對於簡單的情況使用 re.DOTALL 標記參數工作的很好，但是如果模式非常複雜或者是爲了構造字符串令牌而將多個模式合併起來 (2.18 節有詳細描述)，這時候使用這個標記參數就可能出現一些問題。如果讓你選擇的話，最好還是定義自己的正則表達式模式，這樣它可以在不需要額外的標記參數下也能工作的很好。

2.9 將 Unicode 文本標準化

問題

你正在處理 Unicode 字符串，需要確保所有字符串在底層有相同的表示。

解決方案

在 Unicode 中，某些字符能夠用多個合法的編碼表示。爲了說明，考慮下面的這個例子：

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

這裏的文本”Spicy Jalapeño”使用了兩種形式來表示。第一種使用整體字符”ñ”(U+00F1)，第二種使用拉丁字母”n”後面跟一個”~”的組合字符 (U+0303)。

在需要比較字符串的程序中使用字符的多種表示會產生問題。爲了修正這個問題，你可以使用 unicodedata 模塊先將文本標準化：

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'
>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

normalize() 第一個參數指定字符串標準化的方式。NFC 表示字符應該是整體組成 (比如可能的話就使用單一編碼)，而 NFD 表示字符應該分解爲多個組合字符表示。

Python 同樣支持擴展的標準化形式 NFKC 和 NFKD，它們在處理某些字符的時候增加了額外的兼容特性。比如：

```
>>> s = '\ufb01' # A single character
>>> s
'ﬁ'
>>> unicodedata.normalize('NFD', s)
'ﬁ'
# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

討論

標準化對於任何需要以一致的方式處理 Unicode 文本的程序都是非常重要的。當處理來自用戶輸入的字符串而你很難去控制編碼的時候尤其如此。

在清理和過濾文本的時候字符的標準化也是很重要的。比如，假設你想清除掉一些文本上面的變音符的時候 (可能是爲了搜索和匹配)：

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

最後一個例子展示了 `unicodedata` 模塊的另一個重要方面，也就是測試字符類的工具函數。`combining()` 函數可以測試一個字符是否爲和音字符。在這個模塊中還有其他函數用於查找字符類別，測試是否爲數字字符等等。

Unicode 顯然是一個很大的主題。如果想更深入的瞭解關於標準化方面的信息，請看考 Unicode 官網中關於這部分的說明 Ned Batchelder 在 他的網站 上對 Python 的 Unicode 處理問題也有一個很好的介紹。

2.10 在正則式中使用 Unicode

問題

你正在使用正則表達式處理文本，但是關注的是 Unicode 字符處理。

解決方案

默認情況下 `re` 模塊已經對一些 Unicode 字符類有了基本的支持。比如，`\\d` 已經匹配任意的 unicode 數字字符了：

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>
>>> # Arabic digits
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

如果你想在模式中包含指定的 Unicode 字符，你可以使用 Unicode 字符對應的轉義序列 (比如 \uFFF 或者 \UFFFFFFF)。比如，下面是一個匹配幾個不同阿拉伯編碼頁面中所有字符的正則表達式：

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
>>>
```

當執行匹配和搜索操作的時候，最好是先標準化並且清理所有文本爲標準化格式 (參考 2.9 小節)。但是同樣也應該注意一些特殊情況，比如在忽略大小寫匹配和大小寫轉換時的行爲。

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s) # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper()) # Doesn't match
>>> s.upper() # Case folds
'STRASSE'
>>>
```

討論

混合使用 Unicode 和正則表達式通常會讓你抓狂。如果你真的打算這樣做的話，最好考慮下安裝第三方正則式庫，它們會爲 Unicode 的大小寫轉換和其他大量有趣特性提供全面的支持，包括模糊匹配。

2.11 刪除字符串中不需要的字符

問題

你想去掉文本字符串開頭，結尾或者中間不想要的字符，比如空白。

解決方案

strip() 方法能用於刪除開始或結尾的字符。lstrip() 和 rstrip() 分別從左和從右執行刪除操作。默認情況下，這些方法會去除空白字符，但是你也可以指定其他字

符。比如：

```
>>> # Whitespace stripping
>>> s = ' hello world \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>
>>> # Character stripping
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

討論

這些 strip() 方法在讀取和清理數據以備後續處理的時候是經常會被用到的。比如，你可以用它們來去掉空格，引號和完成其他任務。

但是需要注意的是去除操作不會對字符串的中間的文本產生任何影響。比如：

```
>>> s = ' hello      world \n'
>>> s = s.strip()
>>> s
'hello      world'
>>>
```

如果你想處理中間的空格，那麼你需要求助其他技術。比如使用 replace() 方法或者是用正則表達式替換。示例如下：

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

通常情況下你想將字符串 strip 操作和其他迭代操作相結合，比如從文件中讀取多行數據。如果是這樣的話，那麼生成器表達式就可以大顯身手了。比如：

```
with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:
        print(line)
```

在這裏，表達式 lines = (line.strip() for line in f) 執行數據轉換操作。這種方式非常高效，因爲它不需要預先讀取所有數據放到一個臨時的列表中去。它僅僅只是創建一個生成器，並且每次返回行之前會先執行 strip 操作。

對於更高階的 strip，你可能需要使用 translate() 方法。請參閱下一節瞭解更多關於字符串清理的內容。

2.12 審查清理文本字符串

問題

一些無聊的幼稚黑客在你的網站頁面表單中輸入文本”pýt̂hön”，然後你想將這些字符清理掉。

解決方案

文本清理問題會涉及到包括文本解析與數據處理等一系列問題。在非常簡單的情形下，你可能會選擇使用字符串函數 (比如 str.upper() 和 str.lower()) 將文本轉爲標準格式。使用 str.replace() 或者 re.sub() 的簡單替換操作能刪除或者改變指定的字符序列。你同樣還可以使用 2.9 小節的 unicodedata.normalize() 函數將 unicode 文本標準化。

然後，有時候你可能還想在清理操作上更進一步。比如，你可能想消除整個區間上的字符或者去除變音符。爲了這樣做，你可以使用經常會被忽視的 str.translate() 方法。爲了演示，假設你現在有下面這個凌亂的字符串：

```
>>> s = 'pýt̂höñ\fis\tawesome\r\n'
>>> s
'pýt̂höñ\x0cis\tawesome\r\n'
>>>
```

第一步是清理空白字符。爲了這樣做，先創建一個小的轉換表格然後使用 translate() 方法：

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýt̂höñ is awesome\n'
>>>
```

正如你看的那樣，空白字符 \t 和 \f 已經被重新映射到一個空格。回車字符 r 直接被刪除。

你可以以這個表格爲基礎進一步構建更大的表格。比如，讓我們刪除所有的和音符：

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýt̂ön is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

上面例子中，通過使用 `dict.fromkeys()` 方法構造一個字典，每個 Unicode 和音符作爲鍵，對應的值全部爲 `None` 。

然後使用 `unicodedata.normalize()` 將原始輸入標準化爲分解形式字符。然後再調用 `translate` 函數刪除所有重音符。同樣的技術也可以被用來刪除其他類型的字符(比如控制字符等)。

作爲另一個例子，這裏構造一個將所有 Unicode 數字字符映射到對應的 ASCII 字符上的表格：

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...          for c in range(sys.maxunicode)
...          if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>
```

另一種清理文本的技術涉及到 I/O 解碼與編碼函數。這裏的思路是先對文本做一些初步的清理，然後再結合 `encode()` 或者 `decode()` 操作來清除或修改它。比如：

```
>>> a
'pýt̂ön is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

這裏的標準化操作將原來的文本分解爲單獨的和音符。接下來的 ASCII 編碼/解碼只是簡單的一下子丟棄掉那些字符。當然，這種方法僅僅只在最後的目標就是獲取到文本對應 ACSII 表示的時候生效。

討論

文本字符清理一個最主要的問題應該是運行的性能。一般來講，代碼越簡單運行越快。對於簡單的替換操作，`str.replace()` 方法通常是最快的，甚至在你需要多次調用的時候。比如，爲了清理空白字符，你可以這樣做：

```
def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s
```

如果你去測試的話，你就會發現這種方式會比使用 `translate()` 或者正則表達式要快很多。

另一方面，如果你需要執行任何複雜字符對字符的重新映射或者刪除操作的話，`tanslate()` 方法會非常的快。

從大的方面來講，對於你的應用程序來說性能是你不得不去自己研究的東西。不幸的是，我們不可能給你建議一個特定的技術，使它能夠適應所有的情況。因此實際情況中需要你自己去嘗試不同的方法並評估它。

儘管這一節集中討論的是文本，但是類似的技術也可以適用於字節，包括簡單的替換，轉換和正則表達式。

2.13 字符串對齊

問題

你想通過某種對齊方式來格式化字符串

解決方案

對於基本的字符串對齊操作，可以使用字符串的 `ljust()` , `rjust()` 和 `center()` 方法。比如：

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World         '
>>> text.rjust(20)
'         Hello World'
>>> text.center(20)
'    Hello World     '
>>>
```

所有這些方法都能接受一個可選的填充字符。比如：

```
>>> text.rjust(20,'=')
'=========Hello World'
```

```
>>> text.center(20,'*')
'****Hello World*****'
>>>
```

函數 format() 同樣可以用來很容易的對齊字符串。你要做的就是使用 <,> 或者 ^ 字符後面緊跟一個指定的寬度。比如：

```
>>> format(text, '>20')
'         Hello World'
>>> format(text, '<20')
'Hello World         '
>>> format(text, '^20')
'    Hello World     '
>>>
```

如果你想指定一個非空格的填充字符，將它寫到對齊字符的前面即可：

```
>>> format(text, '=>20s')
'==========Hello World'
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

當格式化多個值的時候，這些格式代碼也可以被用在 format() 方法中。比如：

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'     Hello      World'
>>>
```

format() 函數的一個好處是它不僅適用於字符串。它可以用來格式化任何值，使得它非常的通用。比如，你可以用它來格式化數字：

```
>>> x = 1.2345
>>> format(x, '>10')
'    1.2345'
>>> format(x, '^10.2f')
'   1.23   '
>>>
```

討論

在老的代碼中，你經常會看到被用來格式化文本的 % 操作符。比如：

```
>>> '%-20s' % text
'Hello World         '
>>> '%20s' % text
'         Hello World'
>>>
```

但是，在新版本代碼中，你應該優先選擇 format() 函數或者方法。format() 要比 % 操作符的功能更爲強大。並且 format() 也比使用 ljust() , rjust() 或 center() 方法更通用，因爲它可以用來格式化任意對象，而不僅僅是字符串。

如果想要完全瞭解 format() 函數的有用特性，請參考 在線 Python 文檔

2.14 合併拼接字符串

問題

你想將幾個小的字符串合併爲一個大的字符串

解決方案

如果你想要合併的字符串是在一個序列或者 iterable 中，那麼最快的方式就是使用 join() 方法。比如：

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

初看起來，這種語法看上去會比較怪，但是 join() 被指定爲字符串的一個方法。這樣做的部分原因是你想去連接的對象可能來自各種不同的數據序列 (比如列表，元組，字典，文件，集合或生成器等)，如果在所有這些對象上都定義一個 join() 方法明顯是冗餘的。因此你只需要指定你想要的分割字符串並調用他的 join() 方法去將文本片段組合起來。

如果你僅僅只是合併少數幾個字符串，使用加號 (+) 通常已經足夠了：

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

加號 (+) 操作符在作爲一些複雜字符串格式化的替代方案的時候通常也工作的很好，比如：

```
>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

如果你想在源碼中將兩個字面字符串合併起來，你只需要簡單的將它們放到一起，不需要用加號 (+)。比如：

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

討論

字符串合併可能看上去並不需要用一整節來討論。但是不應該小看這個問題，程序員通常在字符串格式化的時候因爲選擇不當而給應用程序帶來嚴重性能損失。

最重要的需要引起注意的是，當我們使用加號 (+) 操作符去連接大量的字符串的時候是非常低效率的，因爲加號連接會引起內存複製以及垃圾回收操作。特別的，你永遠都不應像下面這樣寫字符串連接代碼：

```
s = ''
for p in parts:
    s += p
```

這種寫法會比使用 join() 方法運行的要慢一些，因爲每一次執行 += 操作的時候會創建一個新的字符串對象。你最好是先收集所有的字符串片段然後再將它們連接起來。

一個相對比較聰明的技巧是利用生成器表達式 (參考 1.19 小節) 轉換數據爲字符串的同時合併字符串，比如：

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

同樣還得注意不必要的字符串連接操作。有時候程序員在沒有必要做連接操作的時候仍然多此一舉。比如在打印的時候：

```
print(a + ':' + b + ':' + c) # Ugly
print(':'.join([a, b, c])) # Still ugly
print(a, b, c, sep=':') # Better
```

當混合使用 I/O 操作和字符串連接操作的時候，有時候需要仔細研究你的程序。比如，考慮下面的兩端代碼片段：

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)

# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

如果兩個字符串很小，那麼第一個版本性能會更好些，因爲 I/O 系統調用天生就慢。另外一方面，如果兩個字符串很大，那麼第二個版本可能會更加高效，因爲它避免了創建一個很大的臨時結果並且要複製大量的內存塊數據。還是那句話，有時候是需要根據你的應用程序特點來決定應該使用哪種方案。

最後談一下，如果你準備編寫構建大量小字符串的輸出代碼，你最好考慮下使用生成器函數，利用 yield 語句產生輸出片段。比如：

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

這種方法一個有趣的方面是它並沒有對輸出片段到底要怎樣組織做出假設。例如，你可以簡單的使用 join() 方法將這些片段合併起來：

```
text = ''.join(sample())
```

或者你也可以將字符串片段重定向到 I/O：

```
for part in sample():
    f.write(part)
```

再或者你還可以寫出一些結合 I/O 操作的混合方案：

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
        yield ''.join(parts)

# 結合文件操作
with open('filename', 'w') as f:
    for part in combine(sample(), 32768):
        f.write(part)
```

這裏的關鍵點在於原始的生成器函數並不需要知道使用細節，它只負責生成字符串片段就行了。

2.15 字符串中插入變量

問題

你想創建一個內嵌變量的字符串，變量被它的值所表示的字符串替換掉。

解決方案

Python 並沒有對在字符串中簡單替換變量值提供直接的支持。但是通過使用字符串的 format() 方法來解決這個問題。比如：

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

或者，如果要被替換的變量能在變量域中找到，那麼你可以結合使用 format_map() 和 vars() 。就像下面這樣：

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

vars() 還有一個有意思的特性就是它也適用於對象實例。比如：

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido',37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

format 和 format_map() 的一個缺陷就是它們並不能很好的處理變量缺失的情況，比如：

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

一種避免這種錯誤的方法是另外定義一個含有 __missing__() 方法的字典對象，就像下面這樣：

```
class safesub(dict):
""" 防止 key 找不到"""
```

```
def __missing__(self, key):
    return '{' + key + '}'
```

現在你可以利用這個類包裝輸入後傳遞給 format_map() ：

```
>>> del n # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

如果你發現自己在代碼中頻繁的執行這些步驟，你可以將變量替換步驟用一個工具函數封裝起來。就像下面這樣：

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

現在你可以像下面這樣寫了：

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

討論

多年以來由於 Python 缺乏對變量替換的內置支持而導致了各種不同的解決方案。作爲本節中展示的一個可能的解決方案，你可以有時候會看到像下面這樣的字符串格式化代碼：

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

你可能還會看到字符串模板的使用：

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

然而，format() 和 format_map() 相比較上面這些方案而已更加先進，因此應該被優先選擇。使用 format() 方法還有一個好處就是你可以獲得對字符串格式化的所有支持 (對齊，填充，數字格式化等待)，而這些特性是使用像模板字符串之類的方案不可能獲得的。

本機還部分介紹了一些高級特性。映射或者字典類中鮮爲人知的 __missing__() 方法可以讓你定義如何處理缺失的值。在 SafeSub 類中，這個方法被定義爲對缺失的值返回一個佔位符。你可以發現缺失的值會出現在結果字符串中 (在調試的時候可能很有用)，而不是產生一個 KeyError 異常。

sub() 函數使用 sys._getframe(1) 返回調用者的棧幀。可以從中訪問屬性 f_locals 來獲得局部變量。毫無疑問絕大部分情況下在代碼中去直接操作棧幀應該是不推薦的。但是，對於像字符串替換工具函數而言它是非常有用的。另外，值得注意的是 f_locals 是一個複製調用函數的本地變量的字典。儘管你可以改變 f_locals 的內容，但是這個修改對於後面的變量訪問沒有任何影響。所以，雖說訪問一個棧幀看上去很邪惡，但是對它的任何操作不會覆蓋和改變調用者本地變量的值。

2.16 以指定列寬格式化字符串

問題

你有一些長字符串，想以指定的列寬將它們重新格式化。

解決方案

使用 textwrap 模塊來格式化字符串的輸出。比如，假如你有下列的長字符串：

```
s = "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."
```

下面演示使用 textwrap 格式化字符串的多種方式：

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
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.

>>> print(textwrap.fill(s, 40))
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.

>>> print(textwrap.fill(s, 40, initial_indent='    '))
    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.

>>> print(textwrap.fill(s, 40, subsequent_indent='    '))
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.
```

討論

textwrap 模塊對於字符串打印是非常有用的，特別是當你希望輸出自動匹配終端大小的時候。你可以使用 os.get_terminal_size() 方法來獲取終端的大小尺寸。比如：

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

fill() 方法接受一些其他可選參數來控制 tab，語句結尾等。參閱 textwrap.TextWrapper 文檔 獲取更多內容。

2.17 在字符串中處理 html 和 xml

問題

你想將 HTML 或者 XML 實體如 &entity; 或 &#code; 替換爲對應的文本。再者，你需要轉換文本中特定的字符 (比如 <, >, 或 &)。

解決方案

如果你想替換文本字符串中的‘<’或者‘>’，使用 html.escape() 函數可以很容易的完成。比如：

```
>>> s = 'Elements are written as "<tag>text</tag>".'
>>> import html
>>> print(s)
Elements are written as "<tag>text</tag>".
>>> print(html.escape(s))
Elements are written as &quot;&lt;tag&gt;text&lt;/tag&gt;&quot;.

>>> # Disable escaping of quotes
>>> print(html.escape(s, quote=False))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
>>>
```

如果你正在處理的是 ASCII 文本，並且想將非 ASCII 文本對應的編碼實體嵌入進去，可以給某些 I/O 函數傳遞參數 `errors='xmlcharrefreplace'` 來達到這個目。比如：

```
>>> s = 'Spicy Jalapeño'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

爲了替換文本中的編碼實體，你需要使用另外一種方法。如果你正在處理 HTML 或者 XML 文本，試着先使用一個合適的 HTML 或者 XML 解析器。通常情況下，這些工具會自動替換這些編碼值，你無需擔心。

有時候，如果你接收到了一些含有編碼值的原始文本，需要手動去做替換，通常你只需要使用 HTML 或者 XML 解析器的一些相關工具函數/方法即可。比如：

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapeño".'
>>>
>>> t = 'The prompt is &gt;&gt;&gt;'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

討論

在生成 HTML 或者 XML 文本的時候，如果正確的轉換特殊標記字符是一個很容易被忽視的細節。特別是當你使用 `print()` 函數或者其他字符串格式化來產生輸出的時候。使用像 `html.escape()` 的工具函數可以很容易的解決這類問題。

如果你想以其他方式處理文本，還有一些其他的工具函數比如 `xml.sax.saxutils.unescapge()` 可以幫助你。然而，你應該先調研清楚怎樣使用一個合適的解析器。比如，如果你在處理 HTML 或 XML 文本，使用某個解析模塊比如 `html.parse` 或 `xml.etree.ElementTree` 已經幫你自動處理了相關的替換細節。

2.18 字符串令牌解析

問題

你有一個字符串，想從左至右將其解析爲一個令牌流。

解決方案

假如你有下面這樣一個文本字符串：

```
text = 'foo = 23 + 42 * 10'
```

爲了令牌化字符串，你不僅需要匹配模式，還得指定模式的類型。比如，你可能想將字符串像下面這樣轉換爲序列對：

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', '10')]
```

爲了執行這樣的切分，第一步就是像下面這樣利用命名捕獲組的正則表達式來定義所有可能的令牌，包括空格：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

在上面的模式中，`?P<TOKENNAME>` 用於給一個模式命名，供後面使用。

下一步，爲了令牌化，使用模式對象很少被人知道的 `scanner()` 方法。這個方法會創建一個 `scanner` 對象，在這個對象上不斷的調用 `match()` 方法會一步步的掃描目標文本，每步一個匹配。下面是演示一個 `scanner` 對象如何工作的交互式例子：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

實際使用這種技術的時候，可以很容易的像下面這樣將上述代碼打包到一個生成器中：

```
def generate_tokens(pat, text):
    Token = namedtuple('Token', ['type', 'value'])
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)
# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

如果你想過濾令牌流，你可以定義更多的生成器函數或者使用一個生成器表達式。比如，下面演示怎樣過濾所有的空白令牌：

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

討論

通常來講令牌化是很多高級文本解析與處理的第一步。爲了使用上面的掃描方法，你需要記住這裏一些重要的幾點。第一點就是你必須確認你使用正則表達式指定了所有輸入中可能出現的文本序列。如果有任何不可匹配的文本出現了，掃描就會直接停止。這也是爲什麼上面例子中必須指定空白字符令牌的原因。

令牌的順序也是有影響的。re 模塊會按照指定好的順序去做匹配。因此，如果一個模式恰好是另一個更長模式的子字符串，那麼你需要確定長模式寫在前面。比如：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

第二個模式是錯的，因爲它會將文本 <= 匹配爲令牌 LT 緊跟着 EQ，而不是單獨的令牌 LE，這個並不是我們想要的結果。

最後，你需要留意下子字符串形式的模式。比如，假設你有如下兩個模式：

```
PRINT = r'(?P<PRINT>print)'
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :
# Token(type='PRINT', value='print')
# Token(type='NAME', value='er')
```

關於更高階的令牌化技術，你可能需要查看 PyParsing 或者 PLY 包。一個調用 PLY 的例子在下一節會有演示。

2.19 實現一個簡單的遞歸下降分析器

問題

你想根據一組語法規則解析文本並執行命令，或者構造一個代表輸入的抽象語法樹。如果語法非常簡單，你可以自己寫這個解析器，而不是使用一些框架。

解決方案

在這個問題中，我們集中討論根據特殊語法去解析文本的問題。爲了這樣做，你首先要以 BNF 或者 EBNF 形式指定一個標準語法。比如，一個簡單數學表達式語法可能像下面這樣：

```
expr ::= expr + term
    |   expr - term
    |   term

term ::= term * factor
    |   term / factor
    |   factor

factor ::= ( expr )
    |   NUM
```

或者，以 EBNF 形式：

```
expr ::= term { (+|-) term }*

term ::= factor { (*|/) factor }*

factor ::= ( expr )
    |   NUM
```

在 EBNF 中，被包含在 `{...}*` 中的規則是可選的。*代表 0 次或多次重複 (跟正則表達式中意義是一樣的)。

現在，如果你對 BNF 的工作機制還不是很明白的話，就把它當做是一組左右符號可相互替換的規則。一般來講，解析的原理就是你利用 BNF 完成多個替換和擴展以匹配輸入文本和語法規則。爲了演示，假設你正在解析形如 `3 + 4 * 5` 的表達式。這個表達式先要通過使用 2.18 節中介紹的技術分解爲一組令牌流。結果可能是像下列這樣的令牌序列：

```
NUM + NUM * NUM
```

在此基礎上，解析動作會試着去通過替換操作匹配語法到輸入令牌：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

下面所有的解析步驟可能需要花點時間弄明白，但是它們原理都是查找輸入並試着去匹配語法規則。第一個輸入令牌是 NUM，因此替換首先會匹配那個部分。一旦匹配成功，就會進入下一個令牌 +，以此類推。當已經確定不能匹配下一個令牌的時候，右邊的部分 (比如 `{ (*/) factor }*`) 就會被清理掉。在一個成功的解析中，整個右邊部分會完全展開來匹配輸入令牌流。

有了前面的知識背景，下面我們舉一個簡單示例來展示如何構建一個遞歸下降表達式求值程序：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 下降解析器
Desc :
"""
import re
import collections

# Token specification
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
MINUS = r'(?P<MINUS>-)'
TIMES = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
```

```
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer
Token = collections.namedtuple('Token', ['type', 'value'])


def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok


# Parser
class ExpressionEvaluator:
    '''
    Implementation of a recursive descent parser. Each method
    implements a single grammar rule. Use the ._accept() method
    to test and accept the current lookahead token. Use the ._expect()
    method to exactly match and discard the next token on on the input
    (or raise a SyntaxError if it doesn't match).
    '''

    def parse(self, text):
        self.tokens = generate_tokens(text)
        self.tok = None  # Last symbol consumed
        self.nexttok = None  # Next symbol tokenized
        self._advance()  # Load first lookahead token
        return self.expr()

    def _advance(self):
        'Advance one token ahead'
        self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

    def _accept(self, toktype):
        'Test and consume the next token if it matches toktype'
        if self.nexttok and self.nexttok.type == toktype:
            self._advance()
            return True
        else:
            return False

    def _expect(self, toktype):
        'Consume next token if it matches toktype or raise SyntaxError'
        if not self._accept(toktype):
            raise SyntaxError('Expected ' + toktype)
```

```
    # Grammar rules follow
    def expr(self):
        "expression ::= term { ('+'|'-') term }*"
        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval += right
            elif op == 'MINUS':
                exprval -= right
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }*"
        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval *= right
            elif op == 'DIVIDE':
                termval /= right
        return termval

    def factor(self):
        "factor ::= NUM | ( expr )"
        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')


def descent_parser():
    e = ExpressionEvaluator()
    print(e.parse('2'))
    print(e.parse('2 + 3'))
    print(e.parse('2 + 3 * 4'))
    print(e.parse('2 + (3 + 4) * 5'))
    # print(e.parse('2 + (3 + * 4)'))
    # Traceback (most recent call last):
    #   File "<stdin>", line 1, in <module>
    #   File "exprparse.py", line 40, in parse
    #   return self.expr()
    #   File "exprparse.py", line 67, in expr
    #   right = self.term()
```

```
    #    File "exprparse.py", line 77, in term
    #    termval = self.factor()
    #    File "exprparse.py", line 93, in factor
    #    exprval = self.expr()
    #    File "exprparse.py", line 67, in expr
    #    right = self.term()
    #    File "exprparse.py", line 77, in term
    #    termval = self.factor()
    #    File "exprparse.py", line 97, in factor
    #    raise SyntaxError("Expected NUMBER or LPAREN")
    #    SyntaxError: Expected NUMBER or LPAREN


if __name__ == '__main__':
    descent_parser()
```

討論

文本解析是一個很大的主題，一般會佔用學生學習編譯課程時剛開始的三週時間。如果你在找尋關於語法，解析算法等相關的背景知識的話，你應該去看一下編譯器書籍。很顯然，關於這方面的內容太多，不可能在這裏全部展開。

儘管如此，編寫一個遞歸下降解析器的整體思路是比較簡單的。開始的時候，你先獲得所有的語法規則，然後將其轉換爲一個函數或者方法。因此如果你的語法類似這樣：

```
expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
    | NUM
```

你應該首先將它們轉換成一組像下面這樣的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
    ...
    def term(self):
    ...
    def factor(self):
    ...
```

每個方法要完成的任務很簡單 - 它必須從左至右遍歷語法規則的每一部分，處理每個令牌。從某種意義上講，方法的目的就是要麼處理完語法規則，要麼產生一個語法錯誤。爲了這樣做，需採用下面的這些實現方法：

- 如果規則中的下個符號是另外一個語法規則的名字 (比如 term 或 factor)，就簡單的調用同名的方法即可。這就是該算法中”下降”的由來 - 控制下降到另一個語

法規則中去。有時候規則會調用已經執行的方法 (比如，在 `factor ::= '('expr ')'` 中對 expr 的調用)。這就是算法中”遞歸”的由來。

- 如果規則中下一個符號是個特殊符號 (比如 ()，你得查找下一個令牌並確認是一個精確匹配)。如果不匹配，就產生一個語法錯誤。這一節中的 `_expect()` 方法就是用來做這一步的。
- 如果規則中下一個符號爲一些可能的選擇項 (比如 + 或 -)，你必須對每一種可能情況檢查下一個令牌，只有當它匹配一個的時候才能繼續。這也是本節示例中 `_accept()` 方法的目的。它相當於 _expect() 方法的弱化版本，因爲如果一個匹配找到了它會繼續，但是如果沒找到，它不會產生錯誤而是回滾 (允許後續的檢查繼續進行)。
- 對於有重複部分的規則 (比如在規則表達式 `::= term { ('+'|'-') term }*` 中)，重複動作通過一個 while 循環來實現。循環主體會收集或處理所有的重複元素直到沒有其他元素可以找到。
- 一旦整個語法規則處理完成，每個方法會返回某種結果給調用者。這就是在解析過程中值是怎樣累加的原理。比如，在表達式求值程序中，返回值代表表達式解析後的部分結果。最後所有值會在最頂層的語法規則方法中合併起來。

儘管向你演示的是一個簡單的例子，遞歸下降解析器可以用來實現非常複雜的解析。比如，Python 語言本身就是通過一個遞歸下降解析器去解釋的。如果你對此感興趣，你可以通過查看 Python 源碼文件 Grammar/Grammar 來研究下底層語法機制。看完你會發現，通過手動方式去實現一個解析器其實會有很多的侷限和不足之處。

其中一個侷限就是它們不能被用於包含任何左遞歸的語法規則中。比如，加入你需要翻譯下面這樣一個規則：

```
items ::= items ',' item
    | item
```

爲了這樣做，你可能會像下面這樣使用 `items()` 方法：

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

唯一的問題是這個方法根本不能工作，事實上，它會產生一個無限遞歸錯誤。

關於語法規則本身你可能也會碰到一些棘手的問題。比如，你可能想知道下面這個簡單扼語法是否表述得當：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
    | NUM
```

這個語法看上去沒啥問題，但是它卻不能察覺到標準四則運算中的運算符優先級。比如，表達式 `"3 + 4 * 5"` 會得到 35 而不是期望的 23. 分開使用”expr”和”term”

規則可以讓它正確的工作。

對於複雜的語法，你最好是選擇某個解析工具比如 PyParsing 或者是 PLY。下面是使用 PLY 來重寫表達式求值程序的代碼：

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]
# Ignored characters
t_ignore = ' \t\n'
# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
         | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]


def p_expr_term(p):
    '''
    expr : term
    '''
    p[0] = p[1]
```

```
def p_term(p):
    '''
    term : term TIMES factor
    | term DIVIDE factor
    '''
    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''
    term : factor
    '''
    p[0] = p[1]

def p_factor(p):
    '''
    factor : NUM
    '''
    p[0] = p[1]

def p_factor_group(p):
    '''
    factor : LPAREN expr RPAREN
    '''
    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()
```

這個程序中，所有代碼都位於一個比較高的層次。你只需要爲令牌寫正則表達式和規則匹配時的高階處理函數即可。而實際的運行解析器，接受令牌等等底層動作已經被庫函數實現了。

下面是一個怎樣使用得到的解析對象的例子：

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果你想在你的編程過程中來點挑戰和刺激，編寫解析器和編譯器是個不錯的選擇。再次，一本編譯器的書籍會包含很多底層的理論知識。不過很多好的資源也可以在

網上找到。Python 自己的 ast 模塊也值得去看一下。

2.20 字節字符串上的字符串操作

問題

你想在字節字符串上執行普通的文本操作 (比如移除，搜索和替換)。

解決方案

字節字符串同樣也支持大部分和文本字符串一樣的內置操作。比如：

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

這些操作同樣也適用於字節數組。比如：

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

你可以使用正則表達式匹配字節字符串，但是正則表達式本身必須也是字節串。比如：

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/lib/python3.3/re.py", line 191, in split
return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object
>>> re.split(b'[:,]',data) # Notice: pattern as bytes
```

```
[b'FOO', b'BAR', b'SPAM']
>>>
```

討論

大多數情況下，在文本字符串上的操作均可用於字節字符串。然而，這裏也有一些需要注意的不同點。首先，字節字符串的索引操作返回整數而不是單獨字符。比如：

```
>>> a = 'Hello World' # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World' # Byte string
>>> b[0]
72
>>> b[1]
101
>>>
```

這種語義上的區別會對於處理面向字節的字符數據有影響。

第二點，字節字符串不會提供一個美觀的字符串表示，也不能很好的打印出來，除非它們先被解碼爲一個文本字符串。比如：

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World' # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

類似的，也不存在任何適用於字節字符串的格式化操作：

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'
>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

如果你想格式化字節字符串，你得先使用標準的文本字符串，然後將其編碼爲字節字符串。比如：

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME 100 490.10'
>>>
```

最後需要注意的是，使用字節字符串可能會改變一些操作的語義，特別是那些跟文件系統有關的操作。比如，如果你使用一個編碼爲字節的文件名，而不是一個普通的文本字符串，會禁用文件名的編碼/解碼。比如：

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('spicy')
...
>>> # Get a directory listing
>>> import os
>>> os.listdir('.') # Text string (names are decoded)
['jalapeño.txt']
>>> os.listdir(b'.') # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

注意例子中的最後部分給目錄名傳遞一個字節字符串是怎樣導致結果中文件名以未解碼字節返回的。在目錄中的文件名包含原始的 UTF-8 編碼。參考 5.15 小節獲取更多文件名相關的內容。

最後提一點，一些程序員爲了提升程序執行的速度會傾向於使用字節字符串而不是文本字符串。儘管操作字節字符串確實會比文本更加高效 (因爲處理文本固有的 Unicode 相關開銷)。這樣做通常會導致非常雜亂的代碼。你會經常發現字節字符串並不能和 Python 的其他部分工作的很好，並且你還得手動處理所有的編碼/解碼操作。坦白講，如果你在處理文本的話，就直接在程序中使用普通的文本字符串而不是字節字符串。不做死就不會死！

第三章：數字日期和時間

在 Python 中執行整數和浮點數的數學運算時很簡單的。儘管如此，如果你需要執行分數、數組或者是日期和時間的運算的話，就得做更多的工作了。本章集中討論的就是這些主題。

3.1 數字的四捨五入

問題

你想對浮點數執行指定精度的舍入運算。

解決方案

對於簡單的舍入運算，使用內置的 round(value, ndigits) 函數即可。比如：

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361,3)
1.254
>>>
```

當一個值剛好在兩個邊界的中間的時候，round 函數返回離它最近的偶數。也就是說，對 1.5 或者 2.5 的舍入運算都會得到 2。

傳給 round() 函數的 ndigits 參數可以是負數，這種情況下，舍入運算會作用在十位、百位、千位等上面。比如：

```
>>> a = 1627731
>>> round(a, -1)
1627730
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

討論

不要將舍入和格式化輸出搞混淆了。如果你的目的只是簡單的輸出一定寬度的數，你不需要使用 round() 函數。而僅僅只需要在格式化的時候指定精度即可。比如：

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:0.3f}'.format(x)
'value is 1.235'
>>>
```

同樣，不要試着去舍入浮點值來”修正”表面上看起來正確的問題。比如，你可能傾向於這樣做：

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2) # "Fix" result (???)
>>> c
6.3
>>>
```

對於大多數使用到浮點的程序，沒有必要也不推薦這樣做。儘管在計算的時候會有一點點小的誤差，但是這些小的誤差是能被理解與容忍的。如果不能允許這樣的小誤差(比如涉及到金融領域)，那麼就得考慮使用 `decimal` 模塊了，下一節我們會詳細討論。

3.2 執行精確的浮點數運算

問題

你需要對浮點數執行精確的計算操作，並且不希望有任何小誤差的出現。

解決方案

浮點數的一個普遍問題是它們並不能精確的表示十進制數。並且，即使是最簡單的數學運算也會產生小的誤差，比如：

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

這些錯誤是由底層 CPU 和 IEEE 754 標準通過自己的浮點單位去執行算術時的特徵。由於 Python 的浮點數據類型使用底層表示存儲數據，因此你沒辦法去避免這樣的誤差。

如果你想更加精確 (並能容忍一定的性能損耗)，你可以使用 decimal 模塊：

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
```

初看起來，上面的代碼好像有點奇怪，比如我們用字符串來表示數字。然而，Decimal 對象會像普通浮點數一樣的工作 (支持所有的常用數學運算)。如果你打印它們或者在字符串格式化函數中使用它們，看起來跟普通數字沒什麼兩樣。

decimal 模塊的一個主要特徵是允許你控制計算的每一方面，包括數字位數和四捨五入運算。爲了這樣做，你先得創建一個本地上下文並更改它的設置，比如：

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.76470588235294117647058823529
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

討論

decimal 模塊實現了 IBM 的”通用小數運算規範”。不用說，有很多的配置選項這本書沒有提到。

Python 新手會傾向於使用 decimal 模塊來處理浮點數的精確運算。然而，先理解你的應用程序目的是非常重要的。如果你是在做科學計算或工程領域的計算、電腦繪圖，或者是科學領域的大多數運算，那麼使用普通的浮點類型是比較普遍的做法。其中一個原因是，在真實世界中很少會要求精確到普通浮點數能提供的 17 位精度。因此，計算過程中的那麼一點點的誤差是被允許的。第二點就是，原生的浮點數計算要快的多-有時候你在執行大量運算的時候速度也是非常重要的。

即便如此，你卻不能完全忽略誤差。數學家花了大量時間去研究各類算法，有些處理誤差會比其他方法更好。你也得注意下減法刪除以及大數和小數的加分運算所帶來

的影響。比如：

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums) # Notice how 1 disappears
0.0
>>>
```

上面的錯誤可以利用 `math.fsum()` 所提供的更精確計算能力來解決：

```
>>> import math
>>> math.fsum(nums)
1.0
>>>
```

然而，對於其他的算法，你應該仔細研究它並理解它的誤差產生來源。

總的來說，`decimal` 模塊主要用在涉及到金融的領域。在這類程序中，哪怕是一點小小的誤差在計算過程中蔓延都是不允許的。因此，`decimal` 模塊爲解決這類問題提供了方法。當 Python 和數據庫打交道的時候也通常會遇到 `Decimal` 對象，並且，通常也是在處理金融數據的時候。

3.3 數字的格式化輸出

問題

你需要將數字格式化後輸出，並控制數字的位數、對齊、千位分隔符和其他的細節。

解決方案

格式化輸出單個數字的時候，可以使用內置的 `format()` 函數，比如：

```
>>> x = 1234.56789

>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
'    1234.6'

>>> # Left justified
>>> format(x, '<10.1f')
'1234.6    '

>>> # Centered
>>> format(x, '^10.1f')
'  1234.6  '
```

```
>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

如果你想使用指數記法，將 f 改成 e 或者 E(取決於指數輸出的大小寫形式)。比如：

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

同時指定寬度和精度的一般形式是 '[<>^]?width[,]?(.digits)?' ，其中 width 和 digits 爲整數，? 代表可選部分。同樣的格式也被用在字符串的 format() 方法中。比如：

```
>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>
```

討論

數字格式化輸出通常是比較簡單的。上面演示的技術同時適用於浮點數和 decimal 模塊中的 Decimal 數字對象。

當指定數字的位數後，結果值會根據 round() 函數同樣的規則進行四捨五入後返回。比如：

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

包含千位符的格式化跟本地化沒有關係。如果你需要根據地區來顯示千位符，你需要自己去調查下 locale 模塊中的函數了。你同樣也可以使用字符串的 translate() 方法來交換千位符。比如：

```
>>> swap_separators = { ord('.'):',', ord(','):'.' }
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

在很多 Python 代碼中會看到使用% 來格式化數字的，比如：

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'    1234.6'
>>> '%-10.1f' % x
'1234.6    '
>>>
```

這種格式化方法也是可行的，不過比更加先進的 format() 要差一點。比如，在使用% 操作符格式化數字的時候，一些特性 (添加千位符) 並不能被支持。

3.4 二八十六進制整數

問題

你需要轉換或者輸出使用二進制，八進制或十六進制表示的整數。

解決方案

爲了將整數轉換爲二進制、八進制或十六進制的文本串，可以分別使用 bin() , oct() 或 hex() 函數：

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

另外，如果你不想輸出 0b , 0o 或者 0x 的前綴的話，可以使用 format() 函數。比如：

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

整數是有符號的，所以如果你在處理負數的話，輸出結果會包含一個負號。比如：

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
```

```
'-4d2'
>>>
```

如果你想產生一個無符號值，你需要增加一個指示最大位長度的值。比如爲了顯示 32 位的值，可以像下面這樣寫：

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'11111111111111111111101100101110'
>>> format(2**32 + x, 'x')
'fffffb2e'
>>>
```

爲了以不同的進制轉換整數字符串，簡單的使用帶有進制的 `int()` 函數即可：

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

討論

大多數情況下處理二進制、八進制和十六進制整數是很簡單的。只要記住這些轉換屬於整數和其對應的文本表示之間的轉換即可。永遠只有一種整數類型。

最後，使用八進制的程序員有一點需要注意下。Python 指定八進制數的語法跟其他語言稍有不同。比如，如果你像下面這樣指定八進制，會出現語法錯誤：

```
>>> import os
>>> os.chmod('script.py', 0755)
    File "<stdin>", line 1
        os.chmod('script.py', 0755)
                              ^
SyntaxError: invalid token
>>>
```

需確保八進制數的前綴是 `0o` ，就像下面這樣：

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5 字節到大整數的打包與解包

問題

你有一個字節字符串並想將它解壓成一個整數。或者，你需要將一個大整數轉換爲一個字節字符串。

解決方案

假設你的程序需要處理一個擁有 128 位長的 16 個元素的字節字符串。比如：

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

爲了將 bytes 解析爲整數，使用 int.from_bytes() 方法，並像下面這樣指定字節順序：

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

爲了將一個大整數轉換爲一個字節字符串，使用 int.to_bytes() 方法，並像下面這樣指定字節數和字節順序：

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

討論

大整數和字節字符串之間的轉換操作並不常見。然而，在一些應用領域有時候也會出現，比如密碼學或者網絡。例如，IPv6 網絡地址使用一個 128 位的整數表示。如果你要從一個數據記錄中提取這樣的值的時候，你就會面對這樣的問題。

作爲一種替代方案，你可能想使用 6.11 小節中所介紹的 struct 模塊來解壓字節。這樣也行得通，不過利用 struct 模塊來解壓對於整數的大小是有限制的。因此，你可能想解壓多個字節串並將結果合併爲最終的結果，就像下面這樣：

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

字節順序規則 (little 或 big) 僅僅指定了構建整數時的字節的低位高位排列方式。我們從下面精心構造的 16 進制數的表示中可以很容易的看出來：

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
```

```
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
>>>
```

如果你試着將一個整數打包爲字節字符串，那麼它就不合適了，你會得到一個錯誤。如果需要的話，你可以使用 `int.bit_length()` 方法來決定需要多少字節位來存儲這個值。

```
>>> x = 523 ** 23
>>> x
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
... nbytes += 1
...
>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6 複數的數學運算

問題

你寫的最新的網絡認證方案代碼遇到了一個難題，並且你唯一的解決辦法就是使用複數空間。再或者是你僅僅需要使用複數來執行一些計算操作。

解決方案

複數可以用使用函數 `complex(real, imag)` 或者是帶有後綴 `j` 的浮點數來指定。比如：

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

對應的實部、虛部和共軛複數可以很容易的獲取。就像下面這樣：

```
>>> a.real
2.0
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

另外，所有常見的數學運算都可以工作：

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

如果要執行其他的複數函數比如正弦、餘弦或平方根，使用 cmath 模塊：

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

討論

Python 中大部分與數學相關的模塊都能處理複數。比如如果你使用 numpy ，可以很容易的構造一個複數數組並在這個數組上執行各種操作：

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j, 4.+5.j, 6.-7.j, 8.+9.j])
>>> a + 2
array([ 4.+3.j, 6.+5.j, 8.-7.j, 10.+9.j])
>>> np.sin(a)
array([ 9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
        -153.20827755-526.47684926j, 4008.42651446-589.49948373j])
>>>
```

Python 的標準數學函數確實情況下並不能產生複數值，因此你的代碼中不可能會出現複數返回值。比如：

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

如果你想生成一個複數返回結果，你必須顯示的使用 cmath 模塊，或者在某個支持複數的庫中聲明覆數類型的使用。比如：

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7 無窮大與 NaN

問題

你想創建或測試正無窮、負無窮或 NaN(非數字) 的浮點數。

解決方案

Python 並沒有特殊的語法來表示這些特殊的浮點值，但是可以使用 float() 來創建它們。比如：

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

爲了測試這些值的存在，使用 math.isinf() 和 math.isnan() 函數。比如：

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

討論

想了解更多這些特殊浮點值的信息，可以參考 IEEE 754 規範。然而，也有一些地方需要你特別注意，特別是跟比較和操作符相關的時候。

無窮大數在執行數學計算的時候會傳播，比如：

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

但是有些操作時未定義的並會返回一個 NaN 結果。比如：

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN 值會在所有操作中傳播，而不會產生異常。比如：

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
>>>
```

NaN 值的一個特別的地方時它們之間的比較操作總是返回 False。比如：

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

由於這個原因，測試一個 NaN 值得唯一安全的方法就是使用 `math.isnan()` ，也就是上面演示的那樣。

有時候程序員想改變 Python 默認行爲，在返回無窮大或 NaN 結果的操作中拋出異常。fpectl 模塊可以用來改變這種行爲，但是它在標準的 Python 構建中並沒有被啓用，它是平臺相關的，並且針對的是專家級程序員。可以參考在線的 Python 文檔獲取更多的細節。

3.8 分數運算

問題

你進入時間機器，突然發現你正在做小學家庭作業，並涉及到分數計算問題。或者你可能需要寫代碼去計算在你的木工工廠中的測量值。

解決方案

fractions 模塊可以被用來執行包含分數的數學運算。比如：

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

討論

在大多數程序中一般不會出現分數的計算問題，但是有時候還是需要用到的。比如，在一個允許接受分數形式的測試單位並以分數形式執行運算的程序中，直接使用分數可以減少手動轉換爲小數或浮點數的工作。

3.9 大型數組運算

問題

你需要在大數據集 (比如數組或網格) 上面執行計算。

解決方案

涉及到數組的重量級運算操作，可以使用 NumPy 庫。NumPy 的一個主要特徵是它會給 Python 提供一個數組對象，相比標準的 Python 列表而已更適合用來做數學運算。下面是一個簡單的小例子，向你展示標準列表對象和 NumPy 數組對象之間的差別：

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6, 8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

正如所見，兩種方案中數組的基本數學運算結果並不相同。特別的，NumPy 中的標量運算 (比如 `ax * 2` 或 `ax + 10`) 會作用在每一個元素上。另外，當兩個操作數都是數組的時候執行元素對等位置計算，並最終生成一個新的數組。

對整個數組中所有元素同時執行數學運算可以使得作用在整個數組上的函數運算簡單而又快速。比如，如果你想計算多項式的值，可以這樣做：

```
>>> def f(x):
... return 3*x**2 - 2*x + 7
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy 還爲數組操作提供了大量的通用函數，這些函數可以作爲 math 模塊中類似函數的替代。比如：

```
>>> np.sqrt(ax)
array([ 1. , 1.41421356, 1.73205081, 2. ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

使用這些通用函數要比循環數組並使用 math 模塊中的函數執行計算要快的多。因此，只要有可能的話儘量選擇 NumPy 的數組方案。

底層實現中，NumPy 數組使用了 C 或者 Fortran 語言的機制分配內存。也就是說，它們是一個非常大的連續的並由同類型數據組成的內存區域。所以，你可以構造一個比普通 Python 列表大的多的數組。比如，如果你想構造一個 10,000*10,000 的浮點數二維網格，很輕鬆：

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
    array([[ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    ...,
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.]])
>>>
```

所有的普通操作還是會同時作用在所有元素上：

```
>>> grid += 10
>>> grid
array([[ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    ...,
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.]])
>>> np.sin(grid)
array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
```

```
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    ...,
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
>>>
```

關於 NumPy 有一點需要特別的主意，那就是它擴展 Python 列表的索引功能 - 特別是對於多維數組。爲了說明清楚，先構造一個簡單的二維數組並試着做些試驗：

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1, 2, 3, 4],
[ 5, 6, 7, 8],
[ 9, 10, 11, 12]])

>>> # Select row 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1
>>> a[:,1]
array([ 2, 6, 10])

>>> # Select a subregion and change it
>>> a[1:3, 1:3]
array([[ 6, 7],
        [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1, 2, 3, 4],
        [ 5, 16, 17, 8],
        [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
        [105, 117, 119, 111],
        [109, 121, 123, 115]])
>>> a
array([[ 1, 2, 3, 4],
        [ 5, 16, 17, 8],
        [ 9, 20, 21, 12]])
```

```
>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1, 2, 3, 4],
       [ 5, 10, 10, 8],
       [ 9, 10, 10, 10]])
>>>
```

討論

NumPy 是 Python 領域中很多科學與工程庫的基礎，同時也是被廣泛使用的最大最複雜的模塊。即便如此，在剛開始的時候通過一些簡單的例子和玩具程序也能幫我們完成一些有趣的事情。

通常我們導入 NumPy 模塊的時候會使用語句 `import numpy as np` 。這樣的話你就不用再你的程序裏面一遍遍的敲入 `numpy` ，只需要輸入 `np` 就行了，節省了不少時間。

如果想獲取更多的信息，你當然得去 NumPy 官網逛逛了，網址是：http://www.numpy.org

3.10 矩陣與線性代數運算

問題

你需要執行矩陣和線性代數運算，比如矩陣乘法、尋找行列式、求解線性方程組等等。

解決方案

NumPy 庫有一個矩陣對象可以用來解決這個問題。

矩陣類似於 3.9 小節中數組對象，但是遵循線性代數的計算規則。下面的一個例子展示了矩陣的一些基本特性：

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0, 4, 5],
        [ 7, 8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1, 0, 7],
        [-2, 4, 8],
        [ 3, 5, -9]])
```

```
>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391, 0.13043478, 0.02173913],
        [ 0.12173913, 0.09565217, -0.0173913 ]])

>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>
```

可以在 numpy.linalg 子包中找到更多的操作函數，比如：

```
>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312, 2.75956154, 6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
>>>
```

討論

很顯然線性代數是個非常大的主題，已經超出了本書能討論的範圍。但是，如果你需要操作數組和向量的話，NumPy 是一個不錯的入口點。可以訪問 NumPy 官網 http://www.numpy.org 獲取更多信息。

3.11 隨機選擇

問題

你想從一個序列中隨機抽取若干元素，或者想生成幾個隨機數。

解決方案

random 模塊有大量的函數用來產生隨機數和隨機選擇元素。比如，要想從一個序列中隨機的抽取一個元素，可以使用 random.choice() ：

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

爲了提取出 N 個不同元素的樣本用來做進一步的操作，可以使用 random.sample() ：

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

如果你僅僅只是想打亂序列中元素的順序，可以使用 random.shuffle() ：

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

```
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

生成隨機整數，請使用 `random.randint()`：

```
>>> random.randint(0,10)
2
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
>>> random.randint(0,10)
3
>>>
```

爲了生成 0 到 1 範圍內均勻分佈的浮點數，使用 `random.random()`：

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

如果要獲取 N 位隨機位 (二進制) 的整數，使用 `random.getrandbits()`：

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

討論

random 模塊使用 *Mersenne Twister* 算法來計算生成隨機數。這是一個確定性算法，但是你可以通過 `random.seed()` 函數修改初始化種子。比如：

```
random.seed() # Seed based on system time or os.urandom()
random.seed(12345) # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data
```

除了上述介紹的功能，random 模塊還包含基於均勻分佈、高斯分佈和其他分佈的隨機數生成函數。比如，`random.uniform()` 計算均勻分佈隨機數，`random.gauss()` 計

算正態分佈隨機數。對於其他的分佈情況請參考在線文檔。

在 random 模塊中的函數不應該用在和密碼學相關的程序中。如果你確實需要類似的功能，可以使用 ssl 模塊中相應的函數。比如，ssl.RAND_bytes() 可以用來生成一個安全的隨機字節序列。

3.12 基本的日期與時間轉換

問題

你需要執行簡單的時間轉換，比如天到秒，小時到分鐘等的轉換。

解決方案

爲了執行不同時間單位的轉換和計算，請使用 datetime 模塊。比如，爲了表示一個時間段，可以創建一個 timedelta 實例，就像下面這樣：

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

如果你想表示指定的日期和時間，先創建一個 datetime 實例然後使用標準的數學運算來操作它們。比如：

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

在計算的時候，需要注意的是 datetime 會自動處理閏年。比如：

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

討論

對大多數基本的日期和時間處理問題，datetime 模塊已經足夠了。如果你需要執行更加複雜的日期操作，比如處理時區，模糊時間範圍，節假日計算等等，可以考慮使用 dateutil 模塊

許多類似的時間計算可以使用 dateutil.relativedelta() 函數代替。但是，有一點需要注意的就是，它會在處理月份 (還有它們的天數差距) 的時候填充間隙。看例子最清楚：

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>
>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>
>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13 計算最後一個週五的日期

問題

你需要查找星期中某一天最後出現的日期，比如星期五。

解決方案

Python 的 datetime 模塊中有工具函數和類可以幫助你執行這樣的計算。下面是對類似這樣的問題的一個通用解決方案：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 最後的週五
Desc :
"""
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']


def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

在交互式解釋器中使用如下：

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

可選的 start_date 參數可以由另外一個 datetime 實例來提供。比如：

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

討論

上面的算法原理是這樣的：先將開始日期和目標日期映射到星期數組的位置上 (星期一索引爲 0)，然後通過模運算計算出目標日期要經過多少天才能到達開始日期。然後用開始日期減去那個時間差即得到結果日期。

如果你要像這樣執行大量的日期計算的話，你最好安裝第三方包 python-dateutil 來代替。比如，下面是是使用 dateutil 模塊中的 relativedelta() 函數執行同樣的計算：

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14 計算當前月份的日期範圍

問題

你的代碼需要在當前月份中循環每一天，想找到一個計算這個日期範圍的高效方法。

解決方案

在這樣的日期上循環並需要事先構造一個包含所有日期的列表。你可以先計算出開始日期和結束日期，然後在你步進的時候使用 datetime.timedelta 對象遞增這個日期變量即可。

下面是一個接受任意 datetime 對象並返回一個由當前月份開始日和下個月開始日組成的元組對象。

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

有了這個就可以很容易的在返回的日期範圍上面做循環操作了：

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...
```

討論

上面的代碼先計算出一個對應月份第一天的日期。一個快速的方法就是使用 `date` 或 `datetime` 對象的 `replace()` 方法簡單的將 `days` 屬性設置成 1 即可。`replace()` 方法一個好處就是它會創建和你開始傳入對象類型相同的對象。所以，如果輸入參數是一個 `date` 實例，那麼結果也是一個 `date` 實例。同樣的，如果輸入是一個 `datetime` 實例，那麼你得到的就是一個 `datetime` 實例。

然後，使用 `calendar.monthrange()` 函數來找出該月的總天數。任何時候只要你想獲得日曆信息，那麼 `calendar` 模塊就非常有用了。`monthrange()` 函數會返回包含星期和該月天數的元組。

一旦該月的天數已知了，那麼結束日期就可以通過在開始日期上面加上這個天數獲得。有個需要注意的是結束日期並不包含在這個日期範圍內 (事實上它是下個月的開始日期)。這個和 Python 的 `slice` 與 `range` 操作行爲保持一致，同樣也不包含結尾。

爲了在日期範圍上循環，要使用到標準的數學和比較操作。比如，可以利用 `timedelta` 實例來遞增日期，小於號 < 用來檢查一個日期是否在結束日期之前。

理想情況下，如果能爲日期迭代創建一個同內置的 `range()` 函數一樣的函數就好了。幸運的是，可以使用一個生成器來很容易的實現這個目標：

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

下面是使用這個生成器的例子：

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1),
                        timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

這種實現之所以這麼簡單，還得歸功於 Python 中的日期和時間能夠使用標準的數學和比較操作符來進行運算。

3.15 字符串轉換爲日期

問題

你的應用程序接受字符串格式的輸入，但是你想將它們轉換爲 datetime 對象以便在上面執行非字符串操作。

解決方案

使用 Python 的標準模塊 datetime 可以很容易的解決這個問題。比如：

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

討論

datetime.strptime() 方法支持很多的格式化代碼，比如 %Y 代表 4 位數年份，%m 代表兩位數月份。還有一點值得注意的是這些格式化佔位符也可以反過來使用，將日期

輸出爲指定的格式字符串形式。

比如，假設你的代碼中生成了一個 datetime 對象，你想將它格式化爲漂亮易讀形式後放在自動生成的信件或者報告的頂部：

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

還有一點需要注意的是，strptime() 的性能要比你想象中的差很多，因爲它是使用純 Python 實現，並且必須處理所有的系統本地設置。如果你要在代碼中需要解析大量的日期並且已經知道了日期字符串的確切格式，可以自己實現一套解析方案來獲取更好的性能。比如，如果你已經知道所以日期格式是 YYYY-MM-DD ，你可以像下面這樣實現一個解析函數：

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

實際測試中，這個函數比 datetime.strptime() 快 7 倍多。如果你要處理大量的涉及到日期的數據的話，那麼最好考慮下這個方案！

3.16 結合時區的日期操作

問題

你有一個安排在 2012 年 12 月 21 日早上 9:30 的電話會議，地點在芝加哥。而你的朋友在印度的班加羅爾，那麼他應該在當地時間幾點參加這個會議呢？

解決方案

對幾乎所有涉及到時區的問題，你都應該使用 pytz 模塊。這個包提供了 Olson 時區數據庫，它是時區信息的事實上的標準，在很多語言和操作系統裏面都可以找到。

pytz 模塊一個主要用途是將 datetime 庫創建的簡單日期對象本地化。比如，下面如何表示一個芝加哥時間的示例：

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
```

```
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

一旦日期被本地化了，它就可以轉換爲其他時區的時間了。爲了得到班加羅爾對應的時間，你可以這樣做：

```
>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

如果你打算在本地化日期上執行計算，你需要特別注意夏令時轉換和其他細節。比如，在 2013 年，美國標準夏令時時間開始於本地時間 3 月 13 日凌晨 2:00(在那時，時間向前跳過一小時)。如果你正在執行本地計算，你會得到一個錯誤。比如：

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00 # WRONG! WRONG!
>>>
```

結果錯誤是因爲它並沒有考慮在本地時間中有一小時的跳躍。爲了修正這個錯誤，可以使用時區對象 `normalize()` 方法。比如：

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

討論

爲了不讓你被這些東東弄的暈頭轉向，處理本地化日期的通常的策略先將所有日期轉換爲 UTC 時間，並用它來執行所有的中間存儲和操作。比如：

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

一旦轉換爲 UTC，你就不用去擔心跟夏令時相關的問題了。因此，你可以跟之前一樣放心的執行常見的日期計算。當你想將輸出變爲本地時間的時候，使用合適的時區去轉換下就行了。比如：

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

當涉及到時區操作的時候，有個問題就是我們如何得到時區的名稱。比如，在這個例子中，我們如何知道“Asia/Kolkata”就是印度對應的時區名呢？爲了查找，可以使用 ISO 3166 國家代碼作爲關鍵字去查閱字典 `pytz.country_timezones` 。比如：

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

注：當你閱讀到這裏的時候，有可能 pytz 模塊已經不再建議使用了，因爲 PEP431 提出了更先進的時區支持。但是這裏談到的很多問題還是有參考價值的 (比如使用 UTC 日期的建議等)。

第四章：迭代器與生成器

迭代是 Python 最強大的功能之一。初看起來，你可能會簡單的認爲迭代只不過是處理序列中元素的一種方法。然而，絕非僅僅就是如此，還有很多你可能不知道的，比如創建你自己的迭代器對象，在 itertools 模塊中使用有用的迭代模式，構造生成器函數等等。這一章目的就是向你展示跟迭代有關的各種常見問題。

4.1 手動遍歷迭代器

問題

你想遍歷一個可迭代對象中的所有元素，但是卻不想使用 for 循環。

解決方案

爲了手動的遍歷可迭代對象，使用 `next()` 函數並在代碼中捕獲 `StopIteration` 異常。比如，下面的例子手動讀取一個文件中的所有行：

```
def manual_iter():
    with open('/etc/passwd') as f:
        try:
            while True:
                line = next(f)
                print(line, end='')
        except StopIteration:
            pass
```

通常來講，`StopIteration` 用來指示迭代的結尾。然而，如果你手動使用上面演示的 `next()` 函數的話，你還可以通過返回一個指定值來標記結尾，比如 `None` 。下面是示例：

```
with open('/etc/passwd') as f:
    while True:
        line = next(f, None)
        if line is None:
            break
        print(line, end='')
```

討論

大多數情況下，我們會使用 `for` 循環語句用來遍歷一個可迭代對象。但是，偶爾也需要對迭代做更加精確的控制，這時候瞭解底層迭代機制就顯得尤爲重要了。

下面的交互示例向我們演示了迭代期間所發生的基本細節：

```
>>> items = [1, 2, 3]
>>> # Get the iterator
>>> it = iter(items) # Invokes items.__iter__()
>>> # Run the iterator
>>> next(it) # Invokes it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

本章接下來幾小節會更深入的講解迭代相關技術，前提是你先要理解基本的迭代協議機制。所以確保你已經把這章的內容牢牢記在心中。

4.2 代理迭代

問題

你構建了一個自定義容器對象，裏面包含有列表、元組或其他可迭代對象。你想直接在你的這個新容器對象上執行迭代操作。

解決方案

實際上你只需要定義一個 `__iter__()` 方法，將迭代操作代理到容器內部的對象上去。比如：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

# Example
if __name__ == '__main__':
    root = Node(0)
```

```
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    # Outputs Node(1), Node(2)
    for ch in root:
        print(ch)
```

在上面代碼中，__iter__() 方法只是簡單的將迭代請求傳遞給內部的 _children 屬性。

討論

Python 的迭代器協議需要 __iter__() 方法返回一個實現了 __next__() 方法的迭代器對象。如果你只是迭代遍歷其他容器的內容，你無須擔心底層是怎樣實現的。你所要做的只是傳遞迭代請求既可。

這裏的 iter() 函數的使用簡化了代碼，iter(s) 只是簡單的通過調用 s.__iter__() 方法來返回對應的迭代器對象，就跟 len(s) 會調用 s.__len__() 原理是一樣的。

4.3 使用生成器創建新的迭代模式

問題

你想實現一個自定義迭代模式，跟普通的內置函數比如 range() , reversed() 不一樣。

解決方案

如果你想實現一種新的迭代模式，使用一個生成器函數來定義它。下面是一個生產某個範圍內浮點數的生成器：

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

爲了使用這個函數，你可以用 for 循環迭代它或者使用其他接受一個可迭代對象的函數 (比如 sum() , list() 等)。示例如下：

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
```

```
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

討論

一個函數中需要有一個 `yield` 語句即可將其轉換爲一個生成器。跟普通函數不同的是，生成器只能用於迭代操作。下面是一個實驗，向你展示這樣的函數底層工作機制：

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
>>> next(c)
Starting to count from 3
3

>>> # Run to the next yield
>>> next(c)
2

>>> # Run to next yield
>>> next(c)
1

>>> # Run to next yield (iteration stops)
>>> next(c)
Done!
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

一個生成器函數主要特徵是它只會迴應在迭代中使用到的 *next* 操作。一旦生成器函數返回退出，迭代終止。我們在迭代中通常使用的 for 語句會自動處理這些細節，所以你無需擔心。

4.4 實現迭代器協議

問題

你想構建一個能支持迭代操作的自定義對象，並希望找到一個能實現迭代協議的簡單方法。

解決方案

目前爲止，在一個對象上實現迭代最簡單的方式是使用一個生成器函數。在 4.2 小節中，使用 Node 類來表示樹形數據結構。你可能想實現一個以深度優先方式遍歷樹形節點的生成器。下面是代碼示例：

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
```

```
        print(ch)
    # Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)
```

在這段代碼中，depth_first() 方法簡單直觀。它首先返回自己本身並迭代每一個子節點並通過調用子節點的 depth_first() 方法 (使用 yield from 語句) 返回對應元素。

討論

Python 的迭代協議要求一個 __iter__() 方法返回一個特殊的迭代器對象，這個迭代器對象實現了 __next__() 方法並通過 StopIteration 異常標識迭代的完成。但是，實現這些通常會比較繁瑣。下面我們演示下這種方式，如何使用一個關聯迭代器類重新實現 depth_first() 方法：

```
class Node2:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)


class DepthFirstIterator(object):
    '''
    Depth-first traversal
    '''

    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Return myself if just started; create an iterator for children
        if self._children_iter is None:
            self._children_iter = iter(self._node)
```

```
            return self._node
        # If processing a child, return its next item
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None
                return next(self)
        # Advance to the next child and start its iteration
        else:
            self._child_iter = next(self._children_iter).depth_first()
            return next(self)
```

DepthFirstIterator 類和上面使用生成器的版本工作原理類似，但是它寫起來很繁瑣，因爲迭代器必須在迭代處理過程中維護大量的狀態信息。坦白來講，沒人願意寫這麼晦澀的代碼。將你的迭代器定義爲一個生成器後一切迎刃而解。

4.5 反向迭代

問題

你想反方向迭代一個序列

解決方案

使用內置的 reversed() 函數，比如：

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

反向迭代僅僅當對象的大小可預先確定或者對象實現了 __reversed__() 的特殊方法時才能生效。如果兩者都不符合，那你必須先將對象轉換爲一個列表才行，比如：

```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

要注意的是如果可迭代對象元素很多的話，將其預先轉換爲一個列表要消耗大量的內存。

討論

很多程序員並不知道可以通過在自定義類上實現 `__reversed__()` 方法來實現反向迭代。比如：

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1

for rr in reversed(Countdown(30)):
    print(rr)
for rr in Countdown(30):
    print(rr)
```

定義一個反向迭代器可以使得代碼非常的高效，因爲它不再需要將數據填充到一個列表中然後再去反向迭代這個列表。

4.6 帶有外部狀態的生成器函數

問題

你想定義一個生成器函數，但是它會調用某個你想暴露給用戶使用的外部狀態值。

解決方案

如果你想讓你的生成器暴露外部狀態給用戶，別忘了你可以簡單的將它實現爲一個類，然後把生成器函數放到 `__iter__()` 方法中過去。比如：

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)
```

```
    def __iter__(self):
        for lineno, line in enumerate(self.lines, 1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

爲了使用這個類，你可以將它當做是一個普通的生成器函數。然而，由於可以創建一個實例對象，於是你可以訪問內部屬性值，比如 history 屬性或者是 clear() 方法。代碼示例如下：

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno, hline), end='')
```

討論

關於生成器，很容易掉進函數無所不能的陷阱。如果生成器函數需要跟你的程序其他部分打交道的話 (比如暴露屬性值，允許通過方法調用來控制等等)，可能會導致你的代碼異常的複雜。如果是這種情況的話，可以考慮使用上面介紹的定義類的方式。在 __iter__() 方法中定義你的生成器不會改變你任何的算法邏輯。由於它是類的一部分，所以允許你定義各種屬性和方法來供用戶使用。

一個需要注意的小地方是，如果你在迭代操作時不使用 for 循環語句，那麼你得先調用 iter() 函數。比如：

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

4.7 迭代器切片

問題

你想得到一個由迭代器生成的切片對象，但是標準切片操作並不能做到。

解決方案

函數 itertools.islice() 正好適用於在迭代器和生成器上做切片操作。比如：

```
>>> def count(n):
...     while True:
...         yield n
...         n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>
```

討論

迭代器和生成器不能使用標準的切片操作，因爲它們的長度事先我們並不知道 (並且也沒有實現索引)。函數 islice() 返回一個可以生成指定元素的迭代器，它通過遍歷並丢棄直到切片開始索引位置的所有元素。然後纔開始一個個的返回元素，並直到切片結束索引位置。

這裏要着重強調的一點是 islice() 會消耗掉傳入的迭代器中的數據。必須考慮到迭代器是不可逆的這個事實。所以如果你需要之後再次訪問這個迭代器的話，那你就得先將它裏面的數據放入一個列表中。

4.8 跳過可迭代對象的開始部分

問題

你想遍歷一個可迭代對象，但是它開始的某些元素你並不感興趣，想跳過它們。

解決方案

itertools 模塊中有一些函數可以完成這個任務。首先介紹的是 itertools.dropwhile() 函數。使用時，你給它傳遞一個函數對象和一個可迭代對象。它會返回一個迭代器對象，丟棄原有序列中直到函數返回 Flase 之前的所有元素，然後返回後面所有元素。

爲了演示，假定你在讀取一個開始部分是幾行註釋的源文件。比如：

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times, this information is provided by
# Open Directory.
...
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

如果你想跳過開始部分的註釋行的話，可以這樣做：

```
>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: line.startswith('#'), f):
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

這個例子是基於根據某個測試函數跳過開始的元素。如果你已經明確知道了要跳過的元素的個數的話，那麼可以使用 itertools.islice() 來代替。比如：

```
>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
```

```
...     print(x)
...
1
4
10
15
>>>
```

在這個例子中，islice() 函數最後那個 None 參數指定了你要獲取從第 3 個到最後的所有元素，如果 None 和 3 的位置對調，意思就是僅僅獲取前三個元素恰恰相反，(這個跟切片的相反操作 [3:] 和 [:3] 原理是一樣的)。

討論

函數 dropwhile() 和 islice() 其實就是兩個幫助函數，爲的就是避免寫出下面這種冗餘代碼：

```
with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)
```

跳過一個可迭代對象的開始部分跟通常的過濾是不同的。比如，上述代碼的第一個部分可能會這樣重寫：

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end='')
```

這樣寫確實可以跳過開始部分的註釋行，但是同樣也會跳過文件中其他所有的註釋行。換句話講，我們的解決方案是僅僅跳過開始部分滿足測試條件的行，在那以後，所有的元素不再進行測試和過濾了。

最後需要着重強調的一點是，本節的方案適用於所有可迭代對象，包括那些事先不能確定大小的，比如生成器，文件及其類似的對象。

4.9 排列組合的迭代

問題

你想迭代遍歷一個集合中元素的所有可能的排列或組合

解決方案

itertools 模塊提供了三個函數來解決這類問題。其中一個是 itertools.permutations()，它接受一個集合併產生一個元組序列，每個元組由集合中所有元素的一個可能排列組成。也就是說通過打亂集合中元素排列順序生成一個元組，比如：

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

如果你想得到指定長度的所有排列，你可以傳遞一個可選的長度參數。就像這樣：

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

使用 itertools.combinations() 可得到輸入集合中元素的所有的組合。比如：

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')

>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
```

```
('b', 'c')

>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

對於 combinations() 來講，元素的順序已經不重要了。也就是說，組合 ('a', 'b') 跟 ('b', 'a') 其實是一樣的 (最終只會輸出其中一個)。

在計算組合的時候，一旦元素被選取就會從候選中剔除掉 (比如如果元素'a'已經被選取了，那麼接下來就不會再考慮它了)。而函數 itertools.combinations_with_replacement() 允許同一個元素被選擇多次，比如：

```
>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

討論

這一小節我們向你展示的僅僅是 itertools 模塊的一部分功能。儘管你也可以自己手動實現排列組合算法，但是這樣做得要花點腦力。當我們碰到看上去有些複雜的迭代問題時，最好可以先去看看 itertools 模塊。如果這個問題很普遍，那麼很有可能會在裏面找到解決方案！

4.10 序列上索引值迭代

問題

你想在迭代一個序列的同時跟蹤正在被處理的元素索引。

解决方案

内置的 enumerate() 函數可以很好的解決這個問題：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

爲了按傳統行號輸出 (行號從 1 開始)，你可以傳遞一個開始參數：

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

這種情況在你遍歷文件時想在錯誤消息中使用行號定位時候非常有用：

```
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f, 1):
            fields = line.split()
            try:
                count = int(fields[1])
                ...
            except ValueError as e:
                print('Line {}: Parse error: {}'.format(lineno, e))
```

enumerate() 對於跟蹤某些值在列表中出現的位置是很有用的。所以，如果你想將一個文件中出現的單詞映射到它出現的行號上去，可以很容易的利用 enumerate() 來完成：

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Create a list of words in current line
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

如果你處理完文件後打印 word_summary ，會發現它是一個字典 (準確來講是一個 defaultdict)，對於每個單詞有一個 key ，每個 key 對應的值是一個由這個單詞出現

的行號組成的列表。如果某個單詞在一行中出現過兩次，那麼這個行號也會出現兩次，同時也可以作爲文本的一個簡單統計。

討論

當你想額外定義一個計數變量的時候，使用 enumerate() 函數會更加簡單。你可能會像下面這樣寫代碼：

```
lineno = 1
for line in f:
    # Process line
    ...
    lineno += 1
```

但是如果使用 enumerate() 函數來代替就顯得更加優雅了：

```
for lineno, line in enumerate(f):
    # Process line
    ...
```

enumerate() 函數返回的是一個 enumerate 對象實例，它是一個迭代器，返回連續的包含一個計數和一個值的元組，元組中的值通過在傳入序列上調用 next() 返回。

還有一點可能並不很重要，但是也值得注意，有時候當你在一個已經解壓後的元組序列上使用 enumerate() 函數時很容易調入陷阱。你得像下面正確的方式這樣寫：

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Correct!
for n, (x, y) in enumerate(data):
    ...
# Error!
for n, x, y in enumerate(data):
    ...
```

4.11 同時迭代多個序列

問題

你想同時迭代多個序列，每次分別從一個序列中取一個元素。

解決方案

爲了同時迭代多個序列，使用 zip() 函數。比如：

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
```

```
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>
```

zip(a, b) 會生成一個可返回元組 (x, y) 的迭代器，其中 x 來自 a，y 來自 b。一旦其中某個序列到底結尾，迭代宣告結束。因此迭代長度跟參數中最短序列長度一致。

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
>>>
```

如果這個不是你想要的效果，那麼還可以使用 itertools.zip_longest() 函數來代替。比如：

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')

>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

討論

當你想成對處理數據的時候 zip() 函數是很有用的。比如，假設你頭列表和一個值列表，就像下面這樣：

```
headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]
```

使用 zip() 可以讓你將它們打包並生成一個字典：

```
s = dict(zip(headers,values))
```

或者你也可以像下面這樣產生輸出：

```
for name, val in zip(headers, values):
    print(name, '=', val)
```

雖然不常見，但是 zip() 可以接受多於兩個的序列的參數。這時候所生成的結果元組中元素個數跟輸入序列個數一樣。比如;

```
>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x','y','z']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>
```

最後強調一點就是，zip() 會創建一個迭代器來作爲結果返回。如果你需要將結對的值存儲在列表中，要使用 list() 函數。比如：

```
>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>
```

4.12 不同集合上元素的迭代

問題

你想在多個對象執行相同的操作，但是這些對象在不同的容器中，你希望代碼在不失可讀性的情況下避免寫重複的循環。

解決方案

itertools.chain() 方法可以用來簡化這個任務。它接受一個可迭代對象列表作爲輸入，並返回一個迭代器，有效的屏蔽掉在多個容器中迭代細節。爲了演示清楚，考慮下面這個例子：

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
... print(x)
...
1
2
3
4
x
y
z
>>>
```

使用 chain() 的一個常見場景是當你想對不同的集合中所有元素執行某些操作的時候。比如：

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for item in chain(active_items, inactive_items):
    # Process item
```

這種解決方案要比像下面這樣使用兩個單獨的循環更加優雅，

```
for item in active_items:
    # Process item
    ...

for item in inactive_items:
    # Process item
    ...
```

討論

itertools.chain() 接受一個或多個可迭代對象最爲輸入參數。然後創建一個迭代器，依次連續的返回每個可迭代對象中的元素。這種方式要比先將序列合併再迭代要高效的多。比如：

```
# Inefficent
for x in a + b:
    ...

# Better
for x in chain(a, b):
    ...
```

第一種方案中，a + b 操作會創建一個全新的序列並要求 a 和 b 的類型一致。chian() 不會有這一步，所以如果輸入序列非常大的時候會很省內存。並且當可迭代對象類型不一樣的時候 chain() 同樣可以很好的工作。

4.13 創建數據處理管道

問題

你想以數據管道 (類似 Unix 管道) 的方式迭代處理數據。比如，你有個大量的數據需要處理，但是不能將它們一次性放入內存中。

解決方案

生成器函數是一個實現管道機制的好辦法。爲了演示，假定你要處理一個非常大的日誌文件目錄：

```
foo/
    access-log-012007.gz
    access-log-022007.gz
    access-log-032007.gz
    ...
    access-log-012008
bar/
    access-log-092007.bz2
    ...
    access-log-022008
```

假設每個日誌文件包含這樣的數據：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

爲了處理這些文件，你可以定義一個由多個執行特定任務獨立任務的簡單生成器函數組成的容器。就像這樣：

```
import os
import fnmatch
import gzip
import bz2
import re

def gen_find(filepat, top):
    '''
    Find all filenames in a directory tree that match a shell wildcard pattern
    '''
```

```
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path,name)

def gen_opener(filenames):
    '''
    Open a sequence of filenames one at a time producing a file object.
    The file is closed immediately when proceeding to the next iteration.
    '''
    for filename in filenames:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    '''
    Chain a sequence of iterators together into a single sequence.
    '''
    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    '''
    Look for a regex pattern in a sequence of lines
    '''
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line):
            yield line
```

現在你可以很容易的將這些函數連起來創建一個處理管道。比如，爲了查找包含單詞 python 的所有日誌行，你可以這樣做：

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for line in pylines:
    print(line)
```

如果將來的時候你想擴展管道，你甚至可以在生成器表達式中包裝數據。比如，下面這個版本計算出傳輸的字節數並計算其總和。

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
```

```
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

討論

以管道方式處理數據可以用來解決各類其他問題，包括解析，讀取實時數據，定時輪詢等。

爲了理解上述代碼，重點是要明白 `yield` 語句作爲數據的生產者而 `for` 循環語句作爲數據的消費者。當這些生成器被連在一起後，每個 `yield` 會將一個單獨的數據元素傳遞給迭代處理管道的下一階段。在例子最後部分，`sum()` 函數是最終的程序驅動者，每次從生成器管道中提取出一個元素。

這種方式一個非常好的特點是每個生成器函數很小並且都是獨立的。這樣的話就很容易編寫和維護它們了。很多時候，這些函數如果比較通用的話可以在其他場景重複使用。並且最終將這些組件組合起來的代碼看上去非常簡單，也很容易理解。

使用這種方式的內存效率也不得不提。上述代碼即便是在一個超大型文件目錄中也能工作的很好。事實上，由於使用了迭代方式處理，代碼運行過程中只需要很小很小的內存。

在調用 `gen_concatenate()` 函數的時候你可能會有些不太明白。這個函數的目的是將輸入序列拼接成一個很長的行序列。`itertools.chain()` 函數同樣有類似的功能，但是它需要將所有可迭代對象最爲參數傳入。在上面這個例子中，你可能會寫類似這樣的語句 `lines = itertools.chain(*files)` ，這將導致 `gen_opener()` 生成器被提前全部消費掉。但由於 `gen_opener()` 生成器每次生成一個打開過的文件，等到下一個迭代步驟時文件就關閉了，因此 `chain()` 在這裏不能這樣使用。上面的方案可以避免這種情況。

`gen_concatenate()` 函數中出現過 `yield from` 語句，它將 `yield` 操作代理到父生成器上去。語句 `yield from it` 簡單的返回生成器 `it` 所產生的所有值。關於這個我們在 4.14 小節會有更進一步的描述。

最後還有一點需要注意的是，管道方式並不是萬能的。有時候你想立即處理所有數據。然而，即便是這種情況，使用生成器管道也可以將這類問題從邏輯上變爲工作流的處理方式。

David Beazley 在他的 Generator Tricks for Systems Programmers 教程中對於這種技術有非常深入的講解。可以參考這個教程獲取更多的信息。

4.14 展開嵌套的序列

問題

你想將一個多層嵌套的序列展開成一個單層列表

解決方案

可以寫一個包含 yield from 語句的遞歸生成器來輕鬆解決這個問題。比如：

```
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, [5, 6], 7], 8]
# Produces 1 2 3 4 5 6 7 8
for x in flatten(items):
    print(x)
```

在上面代碼中，isinstance(x, Iterable) 檢查某個元素是否是可迭代的。如果是的話，yield from 就會返回所有子例程的值。最終返回結果就是一個沒有嵌套的簡單序列了。

額外的參數 ignore_types 和檢測語句 isinstance(x, ignore_types) 用來將字符串和字節排除在可迭代對象外，防止將它們再展開成單個的字符。這樣的話字符串數組就能最終返回我們所期望的結果了。比如：

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

討論

語句 yield from 在你想在生成器中調用其他生成器作爲子例程的時候非常有用。如果你不使用它的話，那麼就必須寫額外的 for 循環了。比如：

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

儘管只改了一點點，但是 yield from 語句看上去感覺更好，並且也使得代碼更簡潔清爽。

之前提到的對於字符串和字節的額外檢查是爲了防止將它們再展開成單個字符。如果還有其他你不想展開的類型，修改參數 ignore_types 即可。

最後要注意的一點是，yield from 在涉及到基於協程和生成器的併發編程中扮演着更加重要的角色。可以參考 12.12 小節查看另外一個例子。

4.15 順序迭代合併後的排序迭代對象

問題

你有一系列排序序列，想將它們合併後得到一個排序序列並在上面迭代遍歷。

解決方案

heapq.merge() 函數可以幫你解決這個問題。比如：

```
>>> import heapq
>>> a = [1, 4, 7, 10]
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

討論

heapq.merge 可迭代特性意味着它不會立馬讀取所有序列。這就意味着你可以在非常長的序列中使用它，而不會有太大的開銷。比如，下面是一個例子來演示如何合併兩個排序文件：

```
with open('sorted_file_1', 'rt') as file1, \
    open('sorted_file_2', 'rt') as file2, \
    open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

有一點要強調的是 heapq.merge() 需要所有輸入序列必須是排過序的。特別的，它並不會預先讀取所有數據到堆棧中或者預先排序，也不會對輸入做任何的排序檢測。它僅僅是檢查所有序列的開始部分並返回最小的那個，這個過程一直會持續直到所有輸入序列中的元素都被遍歷完。

4.16 迭代器代替 while 無限循環

問題

你在代碼中使用 while 循環來迭代處理數據，因爲它需要調用某個函數或者和一般迭代模式不同的測試條件。能不能用迭代器來重寫這個循環呢？

解決方案

一個常見的 IO 操作程序可能會想下面這樣：

```
CHUNKSIZE = 8192

def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)
```

這種代碼通常可以使用 iter() 來代替，如下所示：

```
def reader2(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        pass
        # process_data(data)
```

如果你懷疑它到底能不能正常工作，可以試驗下一個簡單的例子。比如：

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

討論

iter 函數一個鮮爲人知的特性是它接受一個可選的 callable 對象和一個標記 (結尾) 值作爲輸入參數。當以這種方式使用的時候，它會創建一個迭代器，這個迭代器會不斷調用 callable 對象直到返回值和標記值相等爲止。

這種特殊的方法對於一些特定的會被重複調用的函數很有效果，比如涉及到 I/O 調用的函數。舉例來講，如果你想從套接字或文件中以數據塊的方式讀取數據，通常你得要不斷重複的執行 read() 或 recv() ，並在後面緊跟一個文件結尾測試來決定是否終止。這節中的方案使用一個簡單的 iter() 調用就可以將兩者結合起來了。其中 lambda 函數參數是爲了創建一個無參的 callable 對象，併爲 recv 或 read() 方法提供了 size 參數。

第五章：文件與 IO

所有程序都要處理輸入和輸出。這一章將涵蓋處理不同類型的文件，包括文本和二進制文件，文件編碼和其他相關的內容。對文件名和目錄的操作也會涉及到。

5.1 讀寫文本數據

問題

你需要讀寫各種不同編碼的文本數據，比如 ASCII，UTF-8 或 UTF-16 編碼等。

解決方案

使用帶有 `rt` 模式的 `open()` 函數讀取文本文件。如下所示：

```
# Read the entire file as a single string
with open('somefile.txt', 'rt') as f:
    data = f.read()

# Iterate over the lines of the file
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
        ...
```

類似的，爲了寫入一個文本文件，使用帶有 `wt` 模式的 `open()` 函數，如果之前文件內容存在則清除並覆蓋掉。如下所示：

```
# Write chunks of text data
with open('somefile.txt', 'wt') as f:
    f.write(text1)
    f.write(text2)
    ...

# Redirected print statement
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
    ...
```

如果是在已存在文件中添加內容，使用模式爲 `at` 的 `open()` 函數。

文件的讀寫操作默認使用系統編碼，可以通過調用 `sys.getdefaultencoding()` 來得到。在大多數機器上面都是 utf-8 編碼。如果你已經知道你要讀寫的文本是其他編碼方式，那麼可以通過傳遞一個可選的 `encoding` 參數給 open() 函數。如下所示：

```
with open('somefile.txt', 'rt', encoding='latin-1') as f:
    ...
```

Python 支持非常多的文本編碼。幾個常見的編碼是 ascii, latin-1, utf-8 和 utf-16。在 web 應用程序中通常都使用的是 UTF-8。ascii 對應從 U+0000 到 U+007F 範圍內的 7 位字符。latin-1 是字節 0-255 到 U+0000 至 U+00FF 範圍內 Unicode 字符的直接映射。當讀取一個未知編碼的文本時使用 latin-1 編碼永遠不會產生解碼錯誤。使用 latin-1 編碼讀取一個文件的時候也許不能產生完全正確的文本解碼數據，但是它也能從中提取出足夠多的有用數據。同時，如果你之後將數據回寫回去，原先的數據還是會保留的。

討論

讀寫文本文件一般來講是比較簡單的。但是也幾點是需要注意的。首先，在例子程序中的 with 語句給被使用到的文件創建了一個上下文環境，但 with 控制塊結束時，文件會自動關閉。你也可以不使用 with 語句，但是這時候你就必須記得手動關閉文件：

```
f = open('somefile.txt', 'rt')
data = f.read()
f.close()
```

另外一個問題是關於換行符的識別問題，在 Unix 和 Windows 中是不一樣的 (分別是 \n 和 \r\n)。默認情況下，Python 會以統一模式處理換行符。這種模式下，在讀取文本的時候，Python 可以識別所有的普通換行符並將其轉換爲單個 \n 字符。類似的，在輸出時會將換行符 \n 轉換爲系統默認的換行符。如果你不希望這種默認的處理方式，可以給 open() 函數傳入參數 newline='' ，就像下面這樣：

```
# Read with disabled newline translation
with open('somefile.txt', 'rt', newline='') as f:
    ...
```

爲了說明兩者之間的差異，下面我在 Unix 機器上面讀取一個 Windows 上面的文本文件，裏面的內容是 hello world!\r\n ：

```
>>> # Newline translation enabled (the default)
>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'

>>> # Newline translation disabled
>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
'hello world!\r\n'
>>>
```

最後一個問題就是文本文件中可能出現的編碼錯誤。但你讀取或者寫入一個文本文件時，你可能會遇到一個編碼或者解碼錯誤。比如：

```
>>> f = open('sample.txt', 'rt', encoding='ascii')
>>> f.read()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
    File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in decode
        return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
12: ordinal not in range(128)
>>>
```

如果出現這個錯誤，通常表示你讀取文本時指定的編碼不正確。你最好仔細閱讀說明並確認你的文件編碼是正確的 (比如使用 UTF-8 而不是 Latin-1 編碼或其他)。如果編碼錯誤還是存在的話，你可以給 `open()` 函數傳遞一個可選的 `errors` 參數來處理這些錯誤。下面是一些處理常見錯誤的方法：

```
>>> # Replace bad chars with Unicode U+fffd replacement char
>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replace')
>>> f.read()
'Spicy Jalape?o!'
>>> # Ignore bad chars entirely
>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore')
>>> g.read()
'Spicy Jalapeo!'
>>>
```

如果你經常使用 `errors` 參數來處理編碼錯誤，可能會讓你的生活變得很糟糕。對於文本處理的首要原則是確保你總是使用的是正確編碼。當模棱兩可的時候，就使用默認的設置 (通常都是 UTF-8)。

5.2 打印輸出至文件中

問題

你想將 `print()` 函數的輸出重定向到一個文件中去。

解決方案

在 `print()` 函數中指定 `file` 關鍵字參數，像下面這樣：

```
with open('d:/work/test.txt', 'wt') as f:
    print('Hello World!', file=f)
```

討論

關於輸出重定向到文件中就這些了。但是有一點要注意的就是文件必須是以文本模式打開。如果文件是二進制模式的話，打印就會出錯。

5.3 使用其他分隔符或行終止符打印

問題

你想使用 print() 函數輸出數據，但是想改變默認的分隔符或者行尾符。

解決方案

可以使用在 print() 函數中使用 sep 和 end 關鍵字參數，以你想要的方式輸出。比如：

```
>>> print('ACME', 50, 91.5)
ACME 50 91.5
>>> print('ACME', 50, 91.5, sep=',')
ACME,50,91.5
>>> print('ACME', 50, 91.5, sep=',', end='!!\n')
ACME,50,91.5!!
>>>
```

使用 end 參數也可以在輸出中禁止換行。比如：

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(5):
...     print(i, end=' ')
...
0 1 2 3 4 >>>
```

討論

當你想使用非空格分隔符來輸出數據的時候，給 print() 函數傳遞一個 sep 參數是最簡單的方案。有時候你會看到一些程序員會使用 str.join() 來完成同樣的事情。比如：

```
>>> print(','.join(('ACME','50','91.5')))
ACME,50,91.5
>>>
```

str.join() 的問題在於它僅僅適用於字符串。這意味着你通常需要執行另外一些轉換才能讓它正常工作。比如：

```
>>> row = ('ACME', 50, 91.5)
>>> print(','.join(row))
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
TypeError: sequence item 1: expected str instance, int found
>>> print(','.join(str(x) for x in row))
ACME,50,91.5
>>>
```

你當然可以不用那麼麻煩，只需要像下面這樣寫：

```
>>> print(*row, sep=',')
ACME,50,91.5
>>>
```

5.4 讀寫字節數據

問題

你想讀寫二進制文件，比如圖片，聲音文件等等。

解決方案

使用模式爲 rb 或 wb 的 open() 函數來讀取或寫入二進制數據。比如：

```
# Read the entire file as a single byte string
with open('somefile.bin', 'rb') as f:
    data = f.read()

# Write binary data to a file
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

在讀取二進制數據時，需要指明的是所有返回的數據都是字節字符串格式的，而不是文本字符串。類似的，在寫入的時候，必須保證參數是以字節形式對外暴露數據的對象 (比如字節字符串，字節數組對象等)。

討論

在讀取二進制數據的時候，字節字符串和文本字符串的語義差異可能會導致一個潛在的陷阱。特別需要注意的是，索引和迭代動作返回的是字節的值而不是字節字符串。比如：

```
>>> # Text string
>>> t = 'Hello World'
>>> t[0]
'H'
>>> for c in t:
...     print(c)
...
```

```
H
e
l
l
o
...
>>> # Byte string
>>> b = b'Hello World'
>>> b[0]
72
>>> for c in b:
...     print(c)
...
72
101
108
108
111
...
>>>
```

如果你想從二進制模式的文件中讀取或寫入文本數據，必須確保要進行解碼和編碼操作。比如：

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

二進制 I/O 還有一個鮮爲人知的特性就是數組和 C 結構體類型能直接被寫入，而不需要中間轉換爲自己對象。比如：

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin','wb') as f:
    f.write(nums)
```

這個適用於任何實現了被稱之爲”緩衝接口”的對象，這種對象會直接暴露其底層的內存緩衝區給能處理它的操作。二進制數據的寫入就是這類操作之一。

很多對象還允許通過使用文件對象的 `readinto()` 方法直接讀取二進制數據到其底層的內存中去。比如：

```
>>> import array
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with open('data.bin', 'rb') as f:
...     f.readinto(a)
...
```

```
16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

但是使用這種技術的時候需要格外小心，因爲它通常具有平臺相關性，並且可能會依賴字長和字節順序 (高位優先和低位優先)。可以查看 5.9 小節中另外一個讀取二進制數據到可修改緩衝區的例子。

5.5 文件不存在才能寫入

問題

你想像一個文件中寫入數據，但是前提必須是這個文件在文件系統上不存在。也就是不允許覆蓋已存在的文件內容。

解決方案

可以在 `open()` 函數中使用 `x` 模式來代替 `w` 模式的方法來解決這個問題。比如：

```
>>> with open('somefile', 'wt') as f:
...     f.write('Hello\n')
...
>>> with open('somefile', 'xt') as f:
...     f.write('Hello\n')
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

如果文件是二進制的，使用 `xb` 來代替 `xt`

討論

這一小節演示了在寫文件時通常會遇到的一個問題的完美解決方案 (不小心覆蓋一個已存在的文件)。一個替代方案是先測試這個文件是否存在，像下面這樣：

```
>>> import os
>>> if not os.path.exists('somefile'):
...     with open('somefile', 'wt') as f:
...         f.write('Hello\n')
... else:
...     print('File already exists!')
...
File already exists!
>>>
```

顯而易見，使用 x 文件模式更加簡單。要注意的是 x 模式是一個 Python3 對 open() 函數特有的擴展。在 Python 的舊版本或者是 Python 實現的底層 C 函數庫中都是沒有這個模式的。

5.6 字符串的 I/O 操作

問題

你想使用操作類文件對象的程序來操作文本或二進制字符串。

解決方案

使用 io.StringIO() 和 io.BytesIO() 類來創建類文件對象操作字符串數據。比如：

```
>>> s = io.StringIO()
>>> s.write('Hello World\n')
12
>>> print('This is a test', file=s)
15
>>> # Get all of the data written so far
>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>

>>> # Wrap a file interface around an existing string
>>> s = io.StringIO('Hello\nWorld\n')
>>> s.read(4)
'Hell'
>>> s.read()
'o\nWorld\n'
>>>
```

io.StringIO 只能用於文本。如果你要操作二進制數據，要使用 io.BytesIO 類來代替。比如：

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

討論

當你想模擬一個普通的文件的時候 StringIO 和 BytesIO 類是很有用的。比如，在單元測試中，你可以使用 StringIO 來創建一個包含測試數據的類文件對象，這個對象可以被傳給某個參數爲普通文件對象的函數。

需要注意的是，StringIO 和 BytesIO 實例並沒有正確的整數類型的文件描述符。因此，它們不能在那些需要使用真實的系統級文件如文件，管道或者是套接字的程序中使用。

5.7 讀寫壓縮文件

問題

你想讀寫一個 gzip 或 bz2 格式的壓縮文件。

解決方案

gzip 和 bz2 模塊可以很容易的處理這些文件。兩個模塊都爲 open() 函數提供了另外的實現來解決這個問題。比如，爲了以文本形式讀取壓縮文件，可以這樣做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'rt') as f:
    text = f.read()

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
    text = f.read()
```

類似的，爲了寫入壓縮數據，可以這樣做：

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'wt') as f:
    f.write(text)

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'wt') as f:
    f.write(text)
```

如上，所有的 I/O 操作都使用文本模式並執行 Unicode 的編碼/解碼。類似的，如果你想操作二進制數據，使用 rb 或者 wb 文件模式即可。

討論

大部分情況下讀寫壓縮數據都是很簡單的。但是要注意的是選擇一個正確的文件模式是非常重要的。如果你不指定模式，那麼默認的就是二進制模式，如果這時候程序想要接受的是文本數據，那麼就會出錯。gzip.open() 和 bz2.open() 接受跟內置的 open() 函數一樣的參數，包括 encoding，errors，newline 等等。

當寫入壓縮數據時，可以使用 compresslevel 這個可選的關鍵字參數來指定一個壓縮級別。比如：

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

默認的等級是 9，也是最高的壓縮等級。等級越低性能越好，但是數據壓縮程度也越低。

最後一點，gzip.open() 和 bz2.open() 還有一個很少被知道的特性，它們可以作用在一個已存在並以二進制模式打開的文件上。比如，下面代碼是可行的：

```
import gzip
f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
    text = g.read()
```

這樣就允許 gzip 和 bz2 模塊可以工作在許多類文件對象上，比如套接字，管道和內存中文件等。

5.8 固定大小記錄的文件迭代

問題

你想在一個固定長度記錄或者數據塊的集合上迭代，而不是在一個文件中一行一行的迭代。

解決方案

通過下面這個小技巧使用 iter 和 functools.partial() 函數：

```
from functools import partial

RECORD_SIZE = 32

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

這個例子中的 records 對象是一個可迭代對象，它會不斷的產生固定大小的數據塊，直到文件末尾。要注意的是如果總記錄大小不是塊大小的整數倍的話，最後一個返回元素的字節數會比期望值少。

討論

iter() 函數有一個鮮爲人知的特性就是，如果你給它傳遞一個可調用對象和一個標記值，它會創建一個迭代器。這個迭代器會一直調用傳入的可調用對象直到它返回標

記值爲止，這時候迭代終止。

在例子中，`functools.partial` 用來創建一個每次被調用時從文件中讀取固定數目字節的可調用對象。標記值 `b''` 就是當到達文件結尾時的返回值。

最後再提一點，上面的例子中的文件時以二進制模式打開的。如果是讀取固定大小的記錄，這通常是最普遍的情況。而對於文本文件，一行一行的讀取 (默認的迭代行爲) 更普遍點。

5.9 讀取二進制數據到可變緩衝區中

問題

你想直接讀取二進制數據到一個可變緩衝區中，而不需要做任何的中間複製操作。或者你想原地修改數據並將它寫回到一個文件中去。

解決方案

爲了讀取數據到一個可變數組中，使用文件對象的 `readinto()` 方法。比如：

```
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
```

下面是一個演示這個函數使用方法的例子：

```
>>> # Write a sample file
>>> with open('sample.bin', 'wb') as f:
...     f.write(b'Hello World')
...
>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hallo'
>>> buf
bytearray(b'Hallo World')
>>> with open('newsample.bin', 'wb') as f:
...     f.write(buf)
...
11
>>>
```

討論

文件對象的 readinto() 方法能被用來爲預先分配內存的數組填充數據，甚至包括由 array 模塊或 numpy 庫創建的數組。和普通 read() 方法不同的是，readinto() 填充已存在的緩衝區而不是爲新對象重新分配內存再返回它們。因此，你可以使用它來避免大量的內存分配操作。比如，如果你讀取一個由相同大小的記錄組成的二進制文件時，你可以像下面這樣寫：

```
record_size = 32 # Size of each record (adjust value)

buf = bytearray(record_size)
with open('somefile', 'rb') as f:
    while True:
        n = f.readinto(buf)
        if n < record_size:
            break
        # Use the contents of buf
        ...
```

另外有一個有趣特性就是 memoryview ，它可以通過零複製的方式對已存在的緩衝區執行切片操作，甚至還能修改它的內容。比如：

```
>>> buf
bytearray(b'Hello World')
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>
```

使用 f.readinto() 時需要注意的是，你必須檢查它的返回值，也就是實際讀取的字節數。

如果字節數小於緩衝區大小，表明數據被截斷或者被破壞了 (比如你期望每次讀取指定數量的字節)。

最後，留心觀察其他函數庫和模塊中和 into 相關的函數 (比如 recv_into() ，pack_into() 等)。Python 的很多其他部分已經能支持直接的 I/O 或數據訪問操作，這些操作可被用來填充或修改數組和緩衝區內容。

關於解析二進制結構和 memoryviews 使用方法的更高級例子，請參考 6.12 小節。

5.10 內存映射的二進制文件

問題

你想內存映射一個二進制文件到一個可變字節數組中，目的可能是爲了隨機訪問它的內容或者是原地做些修改。

解決方案

使用 mmap 模塊來內存映射文件。下面是一個工具函數，向你演示瞭如何打開一個文件並以一種便捷方式內存映射這個文件。

```
import os
import mmap

def memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return mmap.mmap(fd, size, access=access)
```

爲了使用這個函數，你需要有一個已創建並且內容不爲空的文件。下面是一個例子，教你怎樣初始創建一個文件並將其內容擴充到指定大小：

```
>>> size = 1000000
>>> with open('data', 'wb') as f:
...     f.seek(size-1)
...     f.write(b'\x00')
...
>>>
```

下面是一個利用 memory_map() 函數類內存映射文件內容的例子：

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Reassign a slice
>>> m[0:11] = b'Hello World'
>>> m.close()

>>> # Verify that changes were made
>>> with open('data', 'rb') as f:
... print(f.read(11))
...
b'Hello World'
>>>
```

mmap() 返回的 mmap 對象同樣也可以作爲一個上下文管理器來使用，這時候底層的文件會被自動關閉。比如：

```
>>> with memory_map('data') as m:
...     print(len(m))
...     print(m[0:10])
...
1000000
b'Hello World'
>>> m.closed
True
>>>
```

默認情況下，memeory_map() 函數打開的文件同時支持讀和寫操作。任何的修改內容都會複製回原來的文件中。如果需要只讀的訪問模式，可以給參數 access 賦值爲 mmap.ACCESS_READ 。比如：

```
m = memory_map(filename, mmap.ACCESS_READ)
```

如果你想在本地修改數據，但是又不想將修改寫回到原始文件中，可以使用 mmap.ACCESS_COPY ：

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

討論

爲了隨機訪問文件的內容，使用 mmap 將文件映射到內存中是一個高效和優雅的方法。例如，你無需打開一個文件並執行大量的 seek() ，read() ，write() 調用，只需要簡單的映射文件並使用切片操作訪問數據即可。

一般來講，mmap() 所暴露的內存看上去就是一個二進制數組對象。但是，你可以使用一個內存視圖來解析其中的數據。比如：

```
>>> m = memory_map('data')
>>> # Memoryview of unsigned integers
>>> v = memoryview(m).cast('I')
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

需要強調的一點是，內存映射一個文件並不會導致整個文件被讀取到內存中。也就是說，文件並沒有被複制到內存緩存或數組中。相反，操作系統僅僅爲文件內容保留了一段虛擬內存。當你訪問文件的不同區域時，這些區域的內容才根據需要被讀取並映射到內存區域中。而那些從沒被訪問到的部分還是留在磁盤上。所有這些過程是透明的，在幕後完成！

如果多個 Python 解釋器內存映射同一個文件，得到的 mmap 對象能夠被用來在解釋器直接交換數據。也就是說，所有解釋器都能同時讀寫數據，並且其中一個解釋器所

做的修改會自動呈現在其他解釋器中。很明顯，這裏需要考慮同步的問題。但是這種方法有時候可以用來在管道或套接字間傳遞數據。

這一小節中函數儘量寫得很通用，同時適用於 Unix 和 Windows 平臺。要注意的是使用 mmap() 函數時會在底層有一些平臺的差異性。另外，還有一些選項可以用來創建匿名的內存映射區域。如果你對這個感興趣，確保你仔細研讀了 Python 文檔中 這方面的內容 。

5.11 文件路徑名的操作

問題

你需要使用路徑名來獲取文件名，目錄名，絕對路徑等等。

解決方案

使用 os.path 模塊中的函數來操作路徑名。下面是一個交互式例子來演示一些關鍵的特性：

```
>>> import os
>>> path = '/Users/beazley/Data/data.csv'

>>> # Get the last component of the path
>>> os.path.basename(path)
'data.csv'

>>> # Get the directory name
>>> os.path.dirname(path)
'/Users/beazley/Data'

>>> # Join path components together
>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Expand the user's home directory
>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Split the file extension
>>> os.path.splitext(path)
('~/Data/data', '.csv')
>>>
```

討論

對於任何的文件名的操作，你都應該使用 os.path 模塊，而不是使用標準字符串操作來構造自己的代碼。特別是爲了可移植性考慮的時候更應如此，因爲 os.path 模塊知道 Unix 和 Windows 系統之間的差異並且能夠可靠地處理類似 Data/data.csv 和 Data\data.csv 這樣的文件名。其次，你真的不應該浪費時間去重複造輪子。通常最好是直接使用已經爲你準備好的功能。

要注意的是 os.path 還有更多的功能在這裏並沒有列舉出來。可以查閱官方文檔來獲取更多與文件測試，符號鏈接等相關的函數說明。

5.12 測試文件是否存在

問題

你想測試一個文件或目錄是否存在。

解決方案

使用 os.path 模塊來測試一個文件或目錄是否存在。比如：

```
>>> import os
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/spam')
False
>>>
```

你還能進一步測試這個文件時什麼類型的。在下面這些測試中，如果測試的文件不存在的時候，結果都會返回 False：

```
>>> # Is a regular file
>>> os.path.isfile('/etc/passwd')
True

>>> # Is a directory
>>> os.path.isdir('/etc/passwd')
False

>>> # Is a symbolic link
>>> os.path.islink('/usr/local/bin/python3')
True

>>> # Get the file linked to
>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.3'
>>>
```

如果你還想獲取元數據 (比如文件大小或者是修改日期)，也可以使用 os.path 模塊來解決：

```
>>> os.path.getsize('/etc/passwd')
3669
>>> os.path.getmtime('/etc/passwd')
1272478234.0
>>> import time
>>> time.ctime(os.path.getmtime('/etc/passwd'))
'Wed Apr 28 13:10:34 2010'
>>>
```

討論

使用 os.path 來進行文件測試是很簡單的。在寫這些腳本時，可能唯一需要注意的就是你需要考慮文件權限的問題，特別是在獲取元數據時候。比如：

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/genericpath.py", line 49, in getsize
        return os.stat(filename).st_size
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desktop/foo.txt'
>>>
```

5.13 獲取文件夾中的文件列表

問題

你想獲取文件系統中某個目錄下的所有文件列表。

解決方案

使用 os.listdir() 函數來獲取某個目錄中的文件列表：

```
import os
names = os.listdir('somedir')
```

結果會返回目錄中所有文件列表，包括所有文件，子目錄，符號鏈接等等。如果你需要通過某種方式過濾數據，可以考慮結合 os.path 庫中的一些函數來使用列表推導。比如：

```
import os.path

# Get all regular files
names = [name for name in os.listdir('somedir')
        if os.path.isfile(os.path.join('somedir', name))]
```

```
# Get all dirs
dirnames = [name for name in os.listdir('somedir')
        if os.path.isdir(os.path.join('somedir', name))]
```

字符串的 startswith() 和 endswith() 方法對於過濾一個目錄的內容也是很有用的。比如：

```
pyfiles = [name for name in os.listdir('somedir')
            if name.endswith('.py')]
```

對於文件名的匹配，你可能會考慮使用 glob 或 fnmatch 模塊。比如：

```
import glob
pyfiles = glob.glob('somedir/*.py')

from fnmatch import fnmatch
pyfiles = [name for name in os.listdir('somedir')
            if fnmatch(name, '*.py')]
```

討論

獲取目錄中的列表是很容易的，但是其返回結果只是目錄中實體名列表而已。如果你還想獲取其他的元信息，比如文件大小，修改時間等等，你或許還需要使用到 os.path 模塊中的函數或着 os.stat() 函數來收集數據。比如：

```
# Example of getting a directory listing

import os
import os.path
import glob

pyfiles = glob.glob('*.py')

# Get file sizes and modification dates
name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(name))
                for name in pyfiles]
for name, size, mtime in name_sz_date:
    print(name, size, mtime)

# Alternative: Get file metadata
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, meta.st_size, meta.st_mtime)
```

最後還有一點要注意的就是，有時候在處理文件名編碼問題時候可能會出現一些問題。通常來講，函數 os.listdir() 返回的實體列表會根據系統默認的文件名編碼來解碼。但是有時候也會碰到一些不能正常解碼的文件名。關於文件名的處理問題，在 5.14 和 5.15 小節有更詳細的講解。

5.14 忽略文件名編碼

問題

你想使用原始文件名執行文件的 I/O 操作，也就是說文件名並沒有經過系統默認編碼去解碼或編碼過。

解決方案

默認情況下，所有的文件名都會根據 `sys.getfilesystemencoding()` 返回的文本編碼來編碼或解碼。比如：

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

如果因爲某種原因你想忽略這種編碼，可以使用一個原始字節字符串來指定一個文件名即可。比如：

```
>>> # Wrte a file using a unicode filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('Spicy!')
...
6
>>> # Directory listing (decoded)
>>> import os
>>> os.listdir('.')
['jalapeño.txt']

>>> # Directory listing (raw)
>>> os.listdir(b'.') # Note: byte string
[b'jalapen\xcc\x83o.txt']

>>> # Open file with raw filename
>>> with open(b'jalapen\xcc\x83o.txt') as f:
...     print(f.read())
...
Spicy!
>>>
```

正如你所見，在最後兩個操作中，當你給文件相關函數如 `open()` 和 `os.listdir()` 傳遞字節字符串時，文件名的處理方式會稍有不同。

討論

通常來講，你不需要擔心文件名的編碼和解碼，普通的文件名操作應該就沒問題了。但是，有些操作系統允許用戶通過偶然或惡意方式去創建名字不符合默認編碼的文件。這些文件名可能會神祕地中斷那些需要處理大量文件的 Python 程序。

讀取目錄並通過原始未解碼方式處理文件名可以有效的避免這樣的問題，儘管這樣會帶來一定的編程難度。

關於打印不可解碼的文件名，請參考 5.15 小節。

5.15 打印不合法的文件名

問題

你的程序獲取了一個目錄中的文件名列表，但是當它試着去打印文件名的時候程序崩潰，出現了 UnicodeEncodeError 異常和一條奇怪的消息——surrogates not allowed 。

解決方案

當打印未知的文件名時，使用下面的方法可以避免這樣的錯誤：

```
def bad_filename(filename):
    return repr(filename)[1:-1]

try:
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

討論

這一小節討論的是在編寫必須處理文件系統的程序時一個不太常見但又很棘手的問題。默認情況下，Python 假定所有文件名都已經根據 sys.getfilesystemencoding() 的值編碼過了。但是，有一些文件系統並沒有強制要求這樣做，因此允許創建文件名沒有正確編碼的文件。這種情況不太常見，但是總會有些用戶冒險這樣做或者是無意之中這樣做了（可能是在一個有缺陷的代碼中給 open() 函數傳遞了一個不合規範的文件名）。

當執行類似 os.listdir() 這樣的函數時，這些不合規範的文件名就會讓 Python 陷入困境。一方面，它不能僅僅只是丟棄這些不合格的名字。而另一方面，它又不能將這些文件名轉換爲正確的文本字符串。Python 對這個問題的解決方案是從文件名中獲取未解碼的字節值比如 \xhh 並將它映射成 Unicode 字符 \udchh 表示的所謂的”代理編碼”。下面一個例子演示了當一個不合格目錄列表中含有一個文件名爲 bäd.txt(使用 Latin-1 而不是 UTF-8 編碼) 時的樣子：

```
>>> import os
>>> files = os.listdir('.')
>>> files
['spam.py', 'b\udce4d.txt', 'foo.txt']
>>>
```

如果你有代碼需要操作文件名或者將文件名傳遞給 open() 這樣的函數，一切都能正常工作。只有當你想要輸出文件名時纔會碰到些麻煩 (比如打印輸出到屏幕或日誌文件等)。特別的，當你想打印上面的文件名列表時，你的程序就會崩潰：

```
>>> for name in files:
...     print(name)
...
spam.py
Traceback (most recent call last):
    File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udce4' in
position 1: surrogates not allowed
>>>
```

程序崩潰的原因就是字符 \udce4 是一個非法的 Unicode 字符。它其實是一個被稱爲代理字符對的雙字符組合的後半部分。由於缺少了前半部分，因此它是個非法的 Unicode。所以，唯一能成功輸出的方法就是當遇到不合法文件名時採取相應的補救措施。比如可以將上述代碼修改如下：

```
>>> for name in files:
... try:
...     print(name)
... except UnicodeEncodeError:
...     print(bad_filename(name))
...
spam.py
b\udce4d.txt
foo.txt
>>>
```

在 bad_filename() 函數中怎樣處置取決於你自己。另外一個選擇就是通過某種方式重新編碼，示例如下：

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors=
↪'surrogateescape')
    return temp.decode('latin-1')
```

譯者注:

```
surrogateescape:
這種是 Python 在絕大部分面向 OS 的 API 中所使用的錯誤處理器，
它能以一種優雅的方式處理由操作系統提供的數據的編碼問題。
在解碼出錯時會將出錯字節存儲到一個很少被使用到的 Unicode 編碼範圍內。
在編碼時將那些隱藏值又還原回原先解碼失敗的字節序列。
它不僅對於 OS API 非常有用，也能很容易的處理其他情況下的編碼錯誤。
```

使用這個版本產生的輸出如下：

```
>>> for name in files:
...     try:
```

```
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
bäd.txt
foo.txt
>>>
```

這一小節主題可能會被大部分讀者所忽略。但是如果你在編寫依賴文件名和文件系統的關鍵任務程序時，就必須得考慮到這個。否則你可能會在某個週末被叫到辦公室去調試一些令人費解的錯誤。

5.16 增加或改變已打開文件的編碼

問題

你想在不關閉一個已打開的文件前提下增加或改變它的 Unicode 編碼。

解決方案

如果你想給一個以二進制模式打開的文件添加 Unicode 編碼/解碼方式，可以使用 `io.TextIOWrapper()` 對象包裝它。比如：

```
import urllib.request
import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u, encoding='utf-8')
text = f.read()
```

如果你想修改一個已經打開的文本模式的文件的編碼方式，可以先使用 `detach()` 方法移除掉已存在的文本編碼層，並使用新的編碼方式代替。下面是一個在 `sys.stdout` 上修改編碼方式的例子：

```
>>> import sys
>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='latin-1')
>>> sys.stdout.encoding
'latin-1'
>>>
```

這樣做可能會中斷你的終端，這裏僅僅是爲了演示而已。

討論

I/O 系統由一系列的層次構建而成。你可以試着運行下面這個操作一個文本文件的例子來查看這種層次：

```
>>> f = open('sample.txt','w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

在這個例子中，io.TextIOWrapper 是一個編碼和解碼 Unicode 的文本處理層，io.BufferedWriter 是一個處理二進制數據的帶緩衝的 I/O 層，io.FileIO 是一個表示操作系統底層文件描述符的原始文件。增加或改變文本編碼會涉及增加或改變最上面的 io.TextIOWrapper 層。

一般來講，像上面例子這樣通過訪問屬性值來直接操作不同的層是很不安全的。例如，如果你試着使用下面這樣的技術改變編碼看看會發生什麼：

```
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

結果出錯了，因爲 f 的原始值已經被破壞了並關閉了底層的文件。

detach() 方法會斷開文件的最頂層並返回第二層，之後最頂層就沒什麼用了。例如：

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>
```

一旦斷開最頂層後，你就可以給返回結果添加一個新的最頂層。比如：

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

儘管已經向你演示了改變編碼的方法，但是你還可以利用這種技術來改變文件行處理、錯誤機制以及文件處理的其他方面。例如：

```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='ascii',
...                               errors='xmlcharrefreplace')
>>> print('Jalape\u00f1o')
Jalape&#241;o
>>>
```

注意下最後輸出中的非 ASCII 字符 ñ 是如何被 ñ 取代的。

5.17 將字節寫入文本文件

問題

你想在文本模式打開的文件中寫入原始的字節數據。

解決方案

將字節數據直接寫入文件的緩衝區即可，例如：

```
>>> import sys
>>> sys.stdout.write(b'Hello\n')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n')
Hello
5
>>>
```

類似的，能夠通過讀取文本文件的 buffer 屬性來讀取二進制數據。

討論

I/O 系統以層級結構的形式構建而成。文本文件是通過在一個擁有緩衝的二進制模式文件上增加一個 Unicode 編碼/解碼層來創建。buffer 屬性指向對應的底層文件。如果你直接訪問它的話就會繞過文本編碼/解碼層。

本小節例子展示的 sys.stdout 可能看起來有點特殊。默認情況下，sys.stdout 總是以文本模式打開的。但是如果你在寫一個需要打印二進制數據到標準輸出的腳本的話，你可以使用上面演示的技術來繞過文本編碼層。

5.18 將文件描述符包裝成文件對象

問題

你有一個對應於操作系統上一個已打開的 I/O 通道 (比如文件、管道、套接字等) 的整型文件描述符，你想將它包裝成一個更高層的 Python 文件對象。

解決方案

一個文件描述符和一個打開的普通文件是不一樣的。文件描述符僅僅是一個由操作系統指定的整數，用來指代某個系統的 I/O 通道。如果你碰巧有這麼一個文件描述符，你可以通過使用 `open()` 函數來將其包裝爲一個 Python 的文件對象。你僅僅只需要使用這個整數值的文件描述符作爲第一個參數來代替文件名即可。例如：

```
# Open a low-level file descriptor
import os
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)

# Turn into a proper file
f = open(fd, 'wt')
f.write('hello world\n')
f.close()
```

當高層的文件對象被關閉或者破壞的時候，底層的文件描述符也會被關閉。如果這個並不是你想要的結果，你可以給 `open()` 函數傳遞一個可選的 `colsefd=False` 。比如：

```
# Create a file object, but don't close underlying fd when done
f = open(fd, 'wt', closefd=False)
...
```

討論

在 Unix 系統中，這種包裝文件描述符的技術可以很方便的將一個類文件接口作用於一個以不同方式打開的 I/O 通道上，如管道、套接字等。舉例來講，下面是一個操作管道的例子：

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Make text-mode file wrappers for socket reading/writing
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                closefd=False)

    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
                closefd=False)
```

```
    # Echo lines back to the client using file I/O
    for line in client_in:
        client_out.write(line)
        client_out.flush()

    client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)
```

需要重點強調的一點是，上面的例子僅僅是爲了演示內置的 open() 函數的一個特性，並且也只適用於基於 Unix 的系統。如果你想將一個類文件接口作用在一個套接字並希望你的代碼可以跨平臺，請使用套接字對象的 makefile() 方法。但是如果不考慮可移植性的話，那上面的解決方案會比使用 makefile() 性能更好一點。

你也可以使用這種技術來構造一個別名，允許以不同於第一次打開文件的方式使用它。例如，下面演示如何創建一個文件對象，它允許你輸出二進制數據到標準輸出 (通常以文本模式打開)：

```
import sys
# Create a binary-mode file for stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()
```

儘管可以將一個已存在的文件描述符包裝成一個正常的文件對象，但是要注意的是並不是所有的文件模式都被支持，並且某些類型的文件描述符可能會有副作用 (特別是涉及到錯誤處理、文件結尾條件等等的時候)。在不同的操作系統上這種行爲也是不一樣，特別的，上面的例子都不能在非 Unix 系統上運行。我說了這麼多，意思就是讓你充分測試自己的實現代碼，確保它能按照期望工作。

5.19 創建臨時文件和文件夾

問題

你需要在程序執行時創建一個臨時文件或目錄，並希望使用完之後可以自動銷燬掉。

解決方案

tempfile 模塊中有很多的函數可以完成這任務。爲了創建一個匿名的臨時文件，可以使用 tempfile.TemporaryFile：

```
from tempfile import TemporaryFile

with TemporaryFile('w+t') as f:
    # Read/write to the file
    f.write('Hello World\n')
    f.write('Testing\n')

    # Seek back to beginning and read the data
    f.seek(0)
    data = f.read()

# Temporary file is destroyed
```

或者，如果你喜歡，你還可以像這樣使用臨時文件：

```
f = TemporaryFile('w+t')
# Use the temporary file
...
f.close()
# File is destroyed
```

TemporaryFile() 的第一個參數是文件模式，通常來講文本模式使用 w+t ，二進制模式使用 w+b 。這個模式同時支持讀和寫操作，在這裏是很有用的，因爲當你關閉文件去改變模式的時候，文件實際上已經不存在了。TemporaryFile() 另外還支持跟內置的 open() 函數一樣的參數。比如：

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
    ...
```

在大多數 Unix 系統上，通過 TemporaryFile() 創建的文件都是匿名的，甚至連目錄都沒有。如果你想打破這個限制，可以使用 NamedTemporaryFile() 來代替。比如：

```
from tempfile import NamedTemporaryFile

with NamedTemporaryFile('w+t') as f:
    print('filename is:', f.name)
    ...

# File automatically destroyed
```

這裏，被打開文件的 f.name 屬性包含了該臨時文件的文件名。當你需要將文件名傳遞給其他代碼來打開這個文件的時候，這個就很有用了。和 TemporaryFile() 一樣，結果文件關閉時會被自動刪除掉。如果你不想這麼做，可以傳遞一個關鍵字參數 delete=False 即可。比如：

```
with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
    ...
```

爲了創建一個臨時目錄，可以使用 tempfile.TemporaryDirectory() 。比如：

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)
    # Use the directory
    ...
# Directory and all contents destroyed
```

討論

TemporaryFile() 、NamedTemporaryFile() 和 TemporaryDirectory() 函數應該是處理臨時文件目錄的最簡單的方式了，因爲它們會自動處理所有的創建和清理步驟。在一個更低的級別，你可以使用 mkstemp() 和 mkdtemp() 來創建臨時文件和目錄。比如：

```
>>> import tempfile
>>> tempfile.mkstemp()
(3, '/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv')
>>> tempfile.mkdtemp()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'
>>>
```

但是，這些函數並不會做進一步的管理了。例如，函數 mkstemp() 僅僅就返回一個原始的 OS 文件描述符，你需要自己將它轉換爲一個真正的文件對象。同樣你還需要自己清理這些文件。

通常來講，臨時文件在系統默認的位置被創建，比如 /var/tmp 或類似的地方。爲了獲取真實的位置，可以使用 tempfile.gettempdir() 函數。比如：

```
>>> tempfile.gettempdir()
'/var/folders/7W/7WZl5sfZEF0pljrEB1UMWE+++TI/-Tmp-'
>>>
```

所有和臨時文件相關的函數都允許你通過使用關鍵字參數 prefix 、suffix 和 dir 來自定義目錄以及命名規則。比如：

```
>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
>>> f.name
'/tmp/mytemp8ee899.txt'
>>>
```

最後還有一點，儘可能以最安全的方式使用 tempfile 模塊來創建臨時文件。包括僅給當前用戶授權訪問以及在文件創建過程中採取措施避免競態條件。要注意的是不同的平臺可能會不一樣。因此你最好閱讀 官方文檔 來了解更多的細節。

5.20 與串行端口的數據通信

問題

你想通過串行端口讀寫數據，典型場景就是和一些硬件設備打交道 (比如一個機器人或傳感器)。

解決方案

儘管你可以通過使用 Python 內置的 I/O 模塊來完成這個任務，但對於串行通信最好的選擇是使用 pySerial 包 。這個包的使用非常簡單，先安裝 pySerial，使用類似下面這樣的代碼就能很容易的打開一個串行端口：

```
import serial
ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies
                    baudrate=9600,
                    bytesize=8,
                    parity='N',
                    stopbits=1)
```

設備名對於不同的設備和操作系統是不一樣的。比如，在 Windows 系統上，你可以使用 0, 1 等表示的一個設備來打開通信端口”COM0”和”COM1”。一旦端口打開，那就可以使用 read()，readline() 和 write() 函數讀寫數據了。例如：

```
ser.write(b'G1 X50 Y50\r\n')
resp = ser.readline()
```

大多數情況下，簡單的串口通信從此變得十分簡單。

討論

儘管表面上看起來很簡單，其實串口通信有時候也是挺麻煩的。推薦你使用第三方包如 pySerial 的一個原因是它提供了對高級特性的支持 (比如超時，控制流，緩衝區刷新，握手協議等等)。舉個例子，如果你想啓用 RTS-CTS 握手協議，你只需要給 Serial() 傳遞一個 rtscts=True 的參數即可。其官方文檔非常完善，因此我在這裏極力推薦這個包。

時刻記住所有涉及到串口的 I/O 都是二進制模式的。因此，確保你的代碼使用的是字節而不是文本 (或有時候執行文本的編碼/解碼操作)。另外當你需要創建二進制編碼的指令或數據包的時候，struct 模塊也是非常有用的。

5.21 序列化 Python 對象

問題

你需要將一個 Python 對象序列化爲一個字節流，以便將它保存到一個文件、存儲到數據庫或者通過網絡傳輸它。

解決方案

對於序列化最普遍的做法就是使用 pickle 模塊。爲了將一個對象保存到一個文件中，可以這樣做：

```
import pickle

data = ... # Some Python object
f = open('somefile', 'wb')
pickle.dump(data, f)
```

爲了將一個對象轉儲爲一個字符串，可以使用 pickle.dumps() ：

```
s = pickle.dumps(data)
```

爲了從字節流中恢復一個對象，使用 picle.load() 或 pickle.loads() 函數。比如：

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

討論

對於大多數應用程序來講，dump() 和 load() 函數的使用就是你有效使用 pickle 模塊所需的全部了。它可適用於絕大部分 Python 數據類型和用戶自定義類的對象實例。如果你碰到某個庫可以讓你在數據庫中保存/恢復 Python 對象或者是通過網絡傳輸對象的話，那麼很有可能這個庫的底層就使用了 pickle 模塊。

pickle 是一種 Python 特有的自描述的數據編碼。通過自描述，被序列化後的數據包含每個對象開始和結束以及它的類型信息。因此，你無需擔心對象記錄的定義，它總是能工作。舉個例子，如果要處理多個對象，你可以這樣做：

```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

你還能序列化函數，類，還有接口，但是結果數據僅僅將它們的名稱編碼成對應的代碼對象。例如：

```
>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

當數據反序列化回來的時候，會先假定所有的源數據時可用的。模塊、類和函數會自動按需導入進來。對於 Python 數據被不同機器上的解析器所共享的應用程序而言，數據的保存可能會有問題，因爲所有的機器都必須訪問同一個源代碼。

注

```
千萬不要對不信任的數據使用 pickle.load()。
pickle 在加載時有一個副作用就是它會自動加載相應模塊並構造實例對象。
但是某個壞人如果知道 pickle 的工作原理，
他就可以創建一個惡意的數據導致 Python 執行隨意指定的系統命令。
因此，一定要保證 pickle 只在相互之間可以認證對方的解析器的內部使用。
```

有些類型的對象是不能被序列化的。這些通常是那些依賴外部系統狀態的對象，比如打開的文件，網絡連接，線程，進程，棧帧等等。用戶自定義類可以通過提供 __getstate__() 和 __setstate__() 方法來繞過這些限制。如果定義了這兩個方法，pickle.dump() 就會調用 __getstate__() 獲取序列化的對象。類似的，__setstate__() 在反序列化時被調用。爲了演示這個工作原理，下面是一個在內部定義了一個線程但仍然可以序列化和反序列化的類：

```
# countdown.py
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

    def __getstate__(self):
        return self.n

    def __setstate__(self, n):
        self.__init__(n)
```

試着運行下面的序列化試驗代碼：

```
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...

>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()
```

然後退出 Python 解析器並重啓後再試驗下：

```
>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

你可以看到線程又奇蹟般的重生了，從你第一次序列化它的地方又恢復過來。

pickle 對於大型的數據結構比如使用 array 或 numpy 模塊創建的二進制數組效率並不是一個高效的編碼方式。如果你需要移動大量的數組數據，你最好是先在一個文件中將其保存爲數組數據塊或使用更高級的標準編碼方式如 HDF5 (需要第三方庫的支持)。

由於 pickle 是 Python 特有的並且附着在源碼上，所有如果需要長期存儲數據的時候不應該選用它。例如，如果源碼變動了，你所有的存儲數據可能會被破壞並且變得不可讀取。坦白來講，對於在數據庫和存檔文件中存儲數據時，你最好使用更加標準的數據編碼格式如 XML，CSV 或 JSON。這些編碼格式更標準，可以被不同的語言支持，並且也能很好的適應源碼變更。

最後一點要注意的是 pickle 有大量的配置選項和一些棘手的問題。對於最常見的使用場景，你不需要去擔心這個，但是如果你要在一個重要的程序中使用 pickle 去做序列化的話，最好去查閱一下 官方文檔 。

第六章：數據編碼和處理

這一章主要討論使用 Python 處理各種不同方式編碼的數據，比如 CSV 文件，JSON，XML 和二進制包裝記錄。和數據結構那一章不同的是，這章不會討論特殊的算法問題，而是關注於怎樣獲取和存儲這些格式的數據。

6.1 讀寫 CSV 數據

問題

你想讀寫一個 CSV 格式的文件。

解決方案

對於大多數的 CSV 格式的數據讀寫問題，都可以使用 `csv` 庫。例如：假設你在一個名叫 stocks.csv 文件中有一些股票市場數據，就像這樣：

```
Symbol,Price,Date,Time,Change,Volume
"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
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

下面向你展示如何將這些數據讀取爲一個元組的序列：

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Process row
        ...
```

在上面的代碼中，`row` 會是一個列表。因此，爲了訪問某個字段，你需要使用下標，如 `row[0]` 訪問 Symbol，`row[4]` 訪問 Change。

由於這種下標訪問通常會引起混淆，你可以考慮使用命名元組。例如：

```
from collections import namedtuple
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headings = next(f_csv)
    Row = namedtuple('Row', headings)
    for r in f_csv:
        row = Row(*r)
```

```
        # Process row
        ...
```

它允許你使用列名如 row.Symbol 和 row.Change 代替下標訪問。需要注意的是這個只有在列名是合法的 Python 標識符的時候才生效。如果不是的話，你可能需要修改下原始的列名 (如將非標識符字符替換成下劃線之類的)。

另外一個選擇就是將數據讀取到一個字典序列中去。可以這樣做：

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.DictReader(f)
    for row in f_csv:
        # process row
        ...
```

在這個版本中，你可以使用列名去訪問每一行的數據了。比如，row['Symbol'] 或者 row['Change']

爲了寫入 CSV 數據，你仍然可以使用 csv 模塊，不過這時候先創建一個 writer 對象。例如:

```
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)
```

如果你有一個字典序列的數據，可以像這樣做：

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

with open('stocks.csv','w') as f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)
```

討論

你應該總是優先選擇 csv 模塊分割或解析 CSV 數據。例如，你可能會像編寫類似下面這樣的代碼：

```
with open('stocks.csv') as f:
for line in f:
    row = line.split(',')
    # process row
    ...
```

使用這種方式的一個缺點就是你仍然需要去處理一些棘手的細節問題。比如，如果某些字段值被引號包圍，你不得不去除這些引號。另外，如果一個被引號包圍的字段碰巧含有一個逗號，那麼程序就會因爲產生一個錯誤大小的行而出錯。

默認情況下，csv 庫可識別 Microsoft Excel 所使用的 CSV 編碼規則。這或許也是最常見的形式，並且也會給你帶來最好的兼容性。然而，如果你查看 csv 的文檔，就會發現有很多種方法將它應用到其他編碼格式上 (如修改分割字符等)。例如，如果你想讀取以 tab 分割的數據，可以這樣做：

```
# Example of reading tab-separated values
with open('stock.tsv') as f:
    f_tsv = csv.reader(f, delimiter='\t')
    for row in f_tsv:
        # Process row
        ...
```

如果你正在讀取 CSV 數據並將它們轉換爲命名元組，需要注意對列名進行合法性認證。例如，一個 CSV 格式文件有一個包含非法標識符的列頭行，類似下面這樣：

```
Street Address,Num-Premises,Latitude,Longitude 5412 N CLARK,10,41.980262,-87.
↪668452
```

這樣最終會導致在創建一個命名元組時產生一個 ValueError 異常而失敗。爲了解決這問題，你可能不得不先去修正列標題。例如，可以像下面這樣在非法標識符上使用一個正則表達式替換：

```
import re
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headers = [ re.sub('[^a-zA-Z_]', '_', h) for h in next(f_csv) ]
    Row = namedtuple('Row', headers)
    for r in f_csv:
        row = Row(*r)
        # Process row
        ...
```

還有重要的一點需要強調的是，csv 產生的數據都是字符串類型的，它不會做任何其他類型的轉換。如果你需要做這樣的類型轉換，你必須自己手動去實現。下面是一個在 CSV 數據上執行其他類型轉換的例子：

```
col_types = [str, float, str, str, float, int]
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Apply conversions to the row items
        row = tuple(convert(value) for convert, value in zip(col_types, row))
        ...
```

另外，下面是一個轉換字典中特定字段的例子：

```
print('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                ('Change', float),
                ('Volume', int) ]

with open('stocks.csv') as f:
    for row in csv.DictReader(f):
        row.update((key, conversion(row[key]))
                for key, conversion in field_types)
        print(row)
```

通常來講，你可能並不想過多去考慮這些轉換問題。在實際情況中，CSV 文件都或多或少有些缺失的數據，被破壞的數據以及其它一些讓轉換失敗的問題。因此，除非你的數據確實有保障是準確無誤的，否則你必須考慮這些問題 (你可能需要增加合適的錯誤處理機制)。

最後，如果你讀取 CSV 數據的目的是做數據分析和統計的話，你可能需要看一看 Pandas 包。Pandas 包含了一個非常方便的函數叫 `pandas.read_csv()` ，它可以加載 CSV 數據到一個 `DataFrame` 對象中去。然後利用這個對象你就可以生成各種形式的統計、過濾數據以及執行其他高級操作了。在 6.13 小節中會有這樣一個例子。

6.2 讀寫 JSON 數據

問題

你想讀寫 JSON(JavaScript Object Notation) 編碼格式的數據。

解決方案

`json` 模塊提供了一種很簡單的方式來編碼和解碼 JSON 數據。其中兩個主要的函數是 `json.dumps()` 和 `json.loads()` ，要比其他序列化函數庫如 pickle 的接口少得多。下面演示如何將一個 Python 數據結構轉換爲 JSON：

```
import json

data = {
    'name' : 'ACME',
```

```
    'shares' : 100,
    'price' : 542.23
}

json_str = json.dumps(data)
```

下面演示如何將一個 JSON 編碼的字符串轉換回一個 Python 數據結構：

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

如果你要處理的是文件而不是字符串，你可以使用 `json.dump()` 和 `json.load()` 來編碼和解碼 JSON 數據。例如：

```
# Writing JSON data
with open('data.json', 'w') as f:
    json.dump(data, f)

# Reading data back
with open('data.json', 'r') as f:
    data = json.load(f)
```

討論

JSON 編碼支持的基本數據類型爲 `None` ，`bool` ，`int` ，`float` 和 `str` ，以及包含這些類型數據的 lists，tuples 和 dictionaries。對於 dictionaries，keys 需要是字符串類型 (字典中任何非字符串類型的 key 在編碼時會先轉換爲字符串)。爲了遵循 JSON 規範，你應該只編碼 Python 的 lists 和 dictionaries。而且，在 web 應用程序中，頂層對象被編碼爲一個字典是一個標準做法。

JSON 編碼的格式對於 Python 語法而已幾乎是完全一樣的，除了一些小的差異之外。比如，True 會被映射爲 true，False 被映射爲 false，而 None 會被映射爲 null。下面是一個例子，演示了編碼後的字符串效果：

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...      'b': 'Hello',
...      'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

如果你試着去檢查 JSON 解碼後的數據，你通常很難通過簡單的打印來確定它的結構，特別是當數據的嵌套結構層次很深或者包含大量的字段時。爲了解決這個問題，可以考慮使用 pprint 模塊的 `pprint()` 函數來代替普通的 `print()` 函數。它會按照 key 的字母順序並以一種更加美觀的方式輸出。下面是一個演示如何漂亮的打印輸出 Twitter 上搜索結果的例子：

```
>>> from urllib.request import urlopen
>>> import json
>>> u = urlopen('http://search.twitter.com/search.json?q=python&rpp=5')
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import pprint
>>> pprint(resp)
{'completed_in': 0.074,
'max_id': 264043230692245504,
'max_id_str': '264043230692245504',
'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
'page': 1,
'query': 'python',
'refresh_url': '?since_id=264043230692245504&q=python',
'results': [{'created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
             'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
             'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
             'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',
             'from_user': ...
             }
             {'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',
             'from_user': ...
             }],
'results_per_page': 5,
'since_id': 0,
'since_id_str': '0'}
>>>
```

一般來講，JSON 解碼會根據提供的數據創建 dicts 或 lists。如果你想要創建其他類型的對象，可以給 `json.loads()` 傳遞 object_pairs_hook 或 object_hook 參數。例如，下面是演示如何解碼 JSON 數據並在一個 OrderedDict 中保留其順序的例子：

```
>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'
>>> from collections import OrderedDict
>>> data = json.loads(s, object_pairs_hook=OrderedDict)
>>> data
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])
>>>
```

下面是如何將一個 JSON 字典轉換爲一個 Python 對象例子：

```
>>> class JSONObject:
...     def __init__(self, d):
...         self.__dict__ = d
...
```

```
>>>
>>> data = json.loads(s, object_hook=JSONObject)
>>> data.name
'ACME'
>>> data.shares
50
>>> data.price
490.1
>>>
```

最後一個例子中，JSON 解碼後的字典作爲一個單個參數傳遞給 `__init__()` 。然後，你就可以隨心所欲的使用它了，比如作爲一個實例字典來直接使用它。

在編碼 JSON 的時候，還有一些選項很有用。如果你想獲得漂亮的格式化字符串後輸出，可以使用 `json.dumps()` 的 indent 參數。它會使得輸出和 pprint() 函數效果類似。比如：

```
>>> print(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
}
>>>
```

對象實例通常並不是 JSON 可序列化的。例如：

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/json/__init__.py", line 226, in dumps
        return _default_encoder.encode(obj)
    File "/usr/local/lib/python3.3/json/encoder.py", line 187, in encode
        chunks = self.iterencode(o, _one_shot=True)
    File "/usr/local/lib/python3.3/json/encoder.py", line 245, in iterencode
        return _iterencode(o, 0)
    File "/usr/local/lib/python3.3/json/encoder.py", line 169, in default
        raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON serializable
>>>
```

如果你想序列化對象實例，你可以提供一個函數，它的輸入是一個實例，返回一個可序列化的字典。例如：

```
def serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d
```

如果你想反過來獲取這個實例，可以這樣做：

```
# Dictionary mapping names to known classes
classes = {
    'Point' : Point
}

def unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if clsname:
        cls = classes[clsname]
        obj = cls.__new__(cls) # Make instance without calling __init__
        for key, value in d.items():
            setattr(obj, key, value)
        return obj
    else:
        return d
```

下面是如何使用這些函數的例子：

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

json 模塊還有很多其他選項來控制更低級別的數字、特殊值如 NaN 等的解析。可以參考官方文檔獲取更多細節。

6.3 解析簡單的 XML 數據

問題

你想從一個簡單的 XML 文檔中提取數據。

解決方案

可以使用 xml.etree.ElementTree 模塊從簡單的 XML 文檔中提取數據。爲了演示，假設你想解析 Planet Python 上的 RSS 源。下面是相應的代碼：

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Download the RSS feed and parse it
u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)

# Extract and output tags of interest
for item in doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print(title)
    print(date)
    print(link)
    print()
```

運行上面的代碼，輸出結果類似這樣：

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.html

Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html

Python Diary: Been playing around with Object Databases
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been-...-object-databases.html

Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cloud.html

Jesse Jiryu Davis: Toro: synchronization primitives for Tornado coroutines
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_DOZT2Kd0hQ/
```

很顯然，如果你想做進一步的處理，你需要替換 print() 語句來完成其他有趣的事。

討論

在很多應用程序中處理 XML 編碼格式的數據是很常見的。不僅因爲 XML 在 Internet 上面已經被廣泛應用於數據交換，同時它也是一種存儲應用程序數據的常用格式 (比如字處理，音樂庫等)。接下來的討論會先假定讀者已經對 XML 基礎比較熟悉了。

在很多情況下，當使用 XML 來僅僅存儲數據的時候，對應的文檔結構非常緊湊並且直觀。例如，上面例子中的 RSS 訂閱源類似於下面的格式：

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
    <channel>
        <title>Planet Python</title>
        <link>http://planet.python.org/</link>
        <language>en</language>
        <description>Planet Python - http://planet.python.org/</description>
        <item>
            <title>Steve Holden: Python for Data Analysis</title>
            <guid>http://holdenweb.blogspot.com/...-data-analysis.html</guid>
            <link>http://holdenweb.blogspot.com/...-data-analysis.html</link>
            <description>...</description>
            <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
        </item>
        <item>
            <title>Vasudev Ram: The Python Data model (for v2 and v3)</title>
            <guid>http://jugad2.blogspot.com/...-data-model.html</guid>
            <link>http://jugad2.blogspot.com/...-data-model.html</link>
            <description>...</description>
            <pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
        </item>
        <item>
            <title>Python Diary: Been playing around with Object Databases</
↪title>
            <guid>http://www.pythondiary.com/...-object-databases.html</guid>
            <link>http://www.pythondiary.com/...-object-databases.html</link>
            <description>...</description>
            <pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
        </item>
        ...
    </channel>
</rss>
```

`xml.etree.ElementTree.parse()` 函數解析整個 XML 文檔並將其轉換成一個文檔對象。然後，你就能使用 `find()`、`iterfind()` 和 `findtext()` 等方法來搜索特定的 XML 元素了。這些函數的參數就是某個指定的標籤名，例如 `channel/item` 或 `title`。

每次指定某個標籤時，你需要遍歷整個文檔結構。每次搜索操作會從一個起始元素開始進行。同樣，每次操作所指定的標籤名也是起始元素的相對路徑。例如，執行 `doc.iterfind('channel/item')` 來搜索所有在 `channel` 元素下面的 `item` 元素。`doc` 代表文檔的最頂層 (也就是第一級的 `rss` 元素)。然後接下來的調用 `item.findtext()` 會從已找到的 `item` 元素位置開始搜索。

ElementTree 模塊中的每個元素有一些重要的屬性和方法，在解析的時候非常有用。tag 屬性包含了標籤的名字，text 屬性包含了內部的文本，而 get() 方法能獲取屬性值。例如：

```
>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>
```

有一點要強調的是 xml.etree.ElementTree 並不是 XML 解析的唯一方法。對於更高級的應用程序，你需要考慮使用 lxml 。它使用了和 ElementTree 同樣的編程接口，因此上面的例子同樣也適用於 lxml。你只需要將剛開始的 import 語句換成 from lxml.etree import parse 就行了。lxml 完全遵循 XML 標準，並且速度也非常快，同時還支持驗證，XSLT，和 XPath 等特性。

6.4 增量式解析大型 XML 文件

問題

你想使用盡可能少的內存從一個超大的 XML 文檔中提取數據。

解決方案

任何時候只要你遇到增量式的數據處理時，第一時間就應該想到迭代器和生成器。下面是一個很簡單的函數，只使用很少的內存就能增量式的處理一個大型 XML 文件：

```
from xml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
```

```
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
            try:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError:
                pass
```

爲了測試這個函數，你需要先有一個大型的 XML 文件。通常你可以在政府網站或公共數據網站上找到這樣的文件。例如，你可以下載 XML 格式的芝加哥城市道路坑窪數據庫。在寫這本書的時候，下載文件已經包含超過 100,000 行數據，編碼格式類似於下面這樣：

```
<response>
    <row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906549</service_request_number>
            <type_of_service_request>Pot Hole in Street</type_of_service_
↪request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_
↪action>
            <street_address>4714 S TALMAN AVE</street_address>
            <zip>60632</zip>
            <x_coordinate>1159494.68618856</x_coordinate>
            <y_coordinate>1873313.83503384</y_coordinate>
            <ward>14</ward>
            <police_district>9</police_district>
            <community_area>58</community_area>
            <latitude>41.808090232127896</latitude>
            <longitude>-87.69053684711305</longitude>
            <location latitude="41.808090232127896"
            longitude="-87.69053684711305" />
        </row>
        <row ...>
            <creation_date>2012-11-18T00:00:00</creation_date>
            <status>Completed</status>
            <completion_date>2012-11-18T00:00:00</completion_date>
            <service_request_number>12-01906695</service_request_number>
            <type_of_service_request>Pot Hole in Street</type_of_service_
↪request>
            <current_activity>Final Outcome</current_activity>
            <most_recent_action>CDOT Street Cut ... Outcome</most_recent_
↪action>
            <street_address>3510 W NORTH AVE</street_address>
            <zip>60647</zip>
```

```
            <x_coordinate>1152732.14127696</x_coordinate>
            <y_coordinate>1910409.38979075</y_coordinate>
            <ward>26</ward>
            <police_district>14</police_district>
            <community_area>23</community_area>
            <latitude>41.91002084292946</latitude>
            <longitude>-87.71435952353961</longitude>
            <location latitude="41.91002084292946"
            longitude="-87.71435952353961" />
        </row>
    </row>
</response>
```

假設你想寫一個腳本來按照坑窪報告數量排列郵編號碼。你可以像這樣做：

```
from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

這個腳本唯一的問題是它會先將整個 XML 文件加載到內存中然後解析。在我的機器上，爲了運行這個程序需要用到 450MB 左右的內存空間。如果使用如下代碼，程序只需要修改一點點：

```
from collections import Counter

potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

結果是：這個版本的代碼運行時只需要 7MB 的內存–大大節約了內存資源。

討論

這一節的技術會依賴 ElementTree 模塊中的兩個核心功能。第一，iterparse() 方法允許對 XML 文檔進行增量操作。使用時，你需要提供文件名和一個包含下面一種或多種類型的事件列表：start , end, start-ns 和 end-ns 。由 iterparse() 創建的迭代器會產生形如 (event, elem) 的元組，其中 event 是上述事件列表中的某一個，而 elem 是相應的 XML 元素。例如：

```
>>> data = iterparse('potholes.xml',('start','end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
>>>
```

start 事件在某個元素第一次被創建並且還沒有被插入其他數據 (如子元素) 時被創建。而 end 事件在某個元素已經完成時被創建。儘管沒有在例子中演示，start-ns 和 end-ns 事件被用來處理 XML 文檔命名空間的聲明。

這本節例子中，start 和 end 事件被用來管理元素和標籤棧。棧代表了文檔被解析時的層次結構，還被用來判斷某個元素是否匹配傳給函數 parse_and_remove() 的路徑。如果匹配，就利用 yield 語句向調用者返回這個元素。

在 yield 之後的下面這個語句纔是使得程序佔用極少內存的 ElementTree 的核心特性：

```
elem_stack[-2].remove(elem)
```

這個語句使得之前由 yield 產生的元素從它的父節點中刪除掉。假設已經沒有其它的地方引用這個元素了，那麼這個元素就被銷燬並回收內存。

對節點的迭代式解析和刪除的最終效果就是一個在文檔上高效的增量式清掃過程。文檔樹結構從始自終沒被完整的創建過。儘管如此，還是能通過上述簡單的方式來處理這個 XML 數據。

這種方案的主要缺陷就是它的運行性能了。我自己測試的結果是，讀取整個文檔到內存中的版本的運行速度差不多是增量式處理版本的兩倍快。但是它卻使用了超過後者 60 倍的內存。因此，如果你更關心內存使用量的話，那麼增量式的版本完勝。

6.5 將字典轉換爲 XML

問題

你想使用一個 Python 字典存儲數據，並將它轉換成 XML 格式。

解決方案

儘管 xml.etree.ElementTree 庫通常用來做解析工作，其實它也可以創建 XML 文檔。例如，考慮如下這個函數：

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
'''
Turn a simple dict of key/value pairs into XML
'''
elem = Element(tag)
for key, val in d.items():
    child = Element(key)
    child.text = str(val)
    elem.append(child)
return elem
```

下面是一個使用例子：

```
>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
<Element 'stock' at 0x1004b64c8>
>>>
```

轉換結果是一個 Element 實例。對於 I/O 操作，使用 xml.etree.ElementTree 中的 tostring() 函數很容易就能將它轉換成一個字節字符串。例如：

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
>>>
```

如果你想給某個元素添加屬性值，可以使用 set() 方法：

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name>GOOG</name>
</stock>'
>>>
```

如果你還想保持元素的順序，可以考慮構造一個 OrderedDict 來代替一個普通的字典。請參考 1.7 小節。

討論

當創建 XML 的時候，你被限制只能構造字符串類型的值。例如：

```
def dict_to_xml_str(tag, d):
    '''
    Turn a simple dict of key/value pairs into XML
    '''
    parts = ['<{}>'.format(tag)]
    for key, val in d.items():
        parts.append('<{0}>{1}</{0}>'.format(key,val))
    parts.append('</{}>'.format(tag))
    return ''.join(parts)
```

問題是如果你手動的去構造的時候可能會碰到一些麻煩。例如，當字典的值中包含一些特殊字符的時候會怎樣呢？

```
>>> d = { 'name' : '<spam>' }

>>> # String creation
>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'

>>> # Proper XML creation
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

注意到程序的後面那個例子中，字符‘<’和‘>’被替換成了 `<` 和 `>`

下面僅供參考，如果你需要手動去轉換這些字符，可以使用 `xml.sax.saxutils` 中的 `escape()` 和 `unescape()` 函數。例如：

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

除了能創建正確的輸出外，還有另外一個原因推薦你創建 `Element` 實例而不是字符串，那就是使用字符串組合構造一個更大的文檔並不是那麼容易。而 `Element` 實例可以不用考慮解析 XML 文本的情況下通過多種方式被處理。也就是說，你可以在一個高級數據結構上完成你所有的操作，並在最後以字符串的形式將其輸出。

6.6 解析和修改 XML

問題

你想讀取一個 XML 文檔，對它最一些修改，然後將結果寫回 XML 文檔。

解決方案

使用 xml.etree.ElementTree 模塊可以很容易的處理這些任務。第一步是以通常的方式來解析這個文檔。例如，假設你有一個名爲 pred.xml 的文檔，類似下面這樣：

```
<?xml version="1.0"?>
<stop>
    <id>14791</id>
    <nm>Clark &amp;amp; Balmoral</nm>
    <sri>
        <rt>22</rt>
        <d>North Bound</d>
        <dd>North Bound</dd>
    </sri>
    <cr>22</cr>
    <pre>
        <pt>5 MIN</pt>
        <fd>Howard</fd>
        <v>1378</v>
        <rn>22</rn>
    </pre>
    <pre>
        <pt>15 MIN</pt>
        <fd>Howard</fd>
        <v>1867</v>
        <rn>22</rn>
    </pre>
</stop>
```

下面是一個利用 ElementTree 來讀取這個文檔並對它做一些修改的例子：

```
>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>

>>> # Remove a few elements
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))
>>> # Insert a new element after <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'This is a test'
>>> root.insert(2, e)

>>> # Write back to a file
>>> doc.write('newpred.xml', xml_declaration=True)
>>>
```

處理結果是一個像下面這樣新的 XML 文件：

```
<?xml version='1.0' encoding='us-ascii'?>
<stop>
    <id>14791</id>
    <nm>Clark &amp; Balmoral</nm>
    <spam>This is a test</spam>
    <pre>
        <pt>5 MIN</pt>
        <fd>Howard</fd>
        <v>1378</v>
        <rn>22</rn>
    </pre>
    <pre>
        <pt>15 MIN</pt>
        <fd>Howard</fd>
        <v>1867</v>
        <rn>22</rn>
    </pre>
</stop>
```

討論

修改一個 XML 文檔結構是很容易的，但是你必須牢記的是所有的修改都是針對父節點元素，將它作爲一個列表來處理。例如，如果你刪除某個元素，通過調用父節點的 `remove()` 方法從它的直接父節點中刪除。如果你插入或增加新的元素，你同樣使用父節點元素的 `insert()` 和 `append()` 方法。還能對元素使用索引和切片操作，比如 `element[i]` 或 `element[i:j]`

如果你需要創建新的元素，可以使用本節方案中演示的 `Element` 類。我們在 6.5 小節已經詳細討論過了。

6.7 利用命名空間解析 XML 文檔

問題

你想解析某個 XML 文檔，文檔中使用了 XML 命名空間。

解決方案

考慮下面這個使用了命名空間的文檔：

```
<?xml version="1.0" encoding="utf-8"?>
<top>
    <author>David Beazley</author>
    <content>
        <html xmlns="http://www.w3.org/1999/xhtml">
```

```
            <head>
                <title>Hello World</title>
            </head>
            <body>
                <h1>Hello World!</h1>
            </body>
        </html>
    </content>
</top>
```

如果你解析這個文檔並執行普通的查詢，你會發現這個並不是那麼容易，因爲所有步驟都變得相當的繁瑣。

```
>>> # Some queries that work
>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')
<Element 'content' at 0x100776ec0>
>>> # A query involving a namespace (doesn't work)
>>> doc.find('content/html')
>>> # Works if fully qualified
>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> # Doesn't work
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head/title')
>>> # Fully qualified
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
... '{http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml}title')
'Hello World'
>>>
```

你可以通過將命名空間處理邏輯包裝爲一個工具類來簡化這個過程：

```
class XMLNamespaces:
    def __init__(self, **kwargs):
        self.namespaces = {}
        for name, uri in kwargs.items():
            self.register(name, uri)
    def register(self, name, uri):
        self.namespaces[name] = '{'+uri+'}'
    def __call__(self, path):
        return path.format_map(self.namespaces)
```

通過下面的方式使用這個類：

```
>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>
```

討論

解析含有命名空間的 XML 文檔會比較繁瑣。上面的 XMLNamespaces 僅僅是允許你使用縮略名代替完整的 URI 將其變得稍微簡潔一點。

很不幸的是，在基本的 ElementTree 解析中沒有任何途徑獲取命名空間的信息。但是，如果你使用 iterparse() 函數的話就可以獲取更多關於命名空間處理範圍的信息。例如：

```
>>> from xml.etree.ElementTree import iterparse
>>> for evt, elem in iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
... print(evt, elem)
...
end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0>
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8>
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260>
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70>
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
>>> elem # This is the topmost element
<Element 'top' at 0x10110dd60>
>>>
```

最後一點，如果你要處理的 XML 文本除了要使用到其他高級 XML 特性外，還要使用到命名空間，建議你最好是使用 lxml 函數庫來代替 ElementTree 。例如，lxml 對利用 DTD 驗證文檔、更好的 XPath 支持和一些其他高級 XML 特性等都提供了更好的支持。這一小節其實只是教你如何讓 XML 解析稍微簡單一點。

6.8 與關係型數據庫的交互

問題

你想在關係型數據庫中查詢、增加或刪除記錄。

解決方案

Python 中表示多行數據的標準方式是一個由元組構成的序列。例如：

```
stocks = [
    ('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
```

```
    ('HPQ', 75, 33.2),
]
```

依據 PEP249，通過這種形式提供數據，可以很容易的使用 Python 標準數據庫 API 和關係型數據庫進行交互。所有數據庫上的操作都通過 SQL 查詢語句來完成。每一行輸入輸出數據用一個元組來表示。

爲了演示說明，你可以使用 Python 標準庫中的 sqlite3 模塊。如果你使用的是一個不同的數據庫 (比如 MySql、Postgresql 或者 ODBC)，還得安裝相應的第三方模塊來提供支持。不過相應的編程接口幾乎都是一樣的，除了一點點細微差別外。

第一步是連接到數據庫。通常你要執行 connect() 函數，給它提供一些數據庫名、主機、用戶名、密碼和其他必要的一些參數。例如：

```
>>> import sqlite3
>>> db = sqlite3.connect('database.db')
>>>
```

爲了處理數據，下一步你需要創建一個遊標。一旦你有了遊標，那麼你就可以執行 SQL 查詢語句了。比如：

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares integer, price␣
↪real)')
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

爲了向數據庫表中插入多條記錄，使用類似下面這樣的語句：

```
>>> c.executemany('insert into portfolio values (?,?,?)', stocks)
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

爲了執行某個查詢，使用像下面這樣的語句：

```
>>> for row in db.execute('select * from portfolio'):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

如果你想接受用戶輸入作爲參數來執行查詢操作，必須確保你使用下面這樣的佔位符 “?“來進行引用參數：

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
```

```
                        (min_price,)):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

討論

在比較低的級別上和數據庫交互是非常簡單的。你只需提供 SQL 語句並調用相應的模塊就可以更新或提取數據了。雖說如此，還是有一些比較棘手的細節問題需要你逐個列出去解決。

一個難點是數據庫中的數據和 Python 類型直接的映射。對於日期類型，通常可以使用 `datetime` 模塊中的 `datetime` 實例，或者可能是 `time` 模塊中的系統時間戳。對於數字類型，特別是使用到小數的金融數據，可以用 `decimal` 模塊中的 `Decimal` 實例來表示。不幸的是，對於不同的數據庫而言具體映射規則是不一樣的，你必須參考相應的文檔。

另外一個更加複雜的問題就是 SQL 語句字符串的構造。你千萬不要使用 Python 字符串格式化操作符 (如%) 或者 `.format()` 方法來創建這樣的字符串。如果傳遞給這些格式化操作符的值來自於用戶的輸入，那麼你的程序就很有可能遭受 SQL 注入攻擊 (參考 http://xkcd.com/327)。查詢語句中的通配符 `?` 指示後臺數據庫使用它自己的字符串替換機制，這樣更加的安全。

不幸的是，不同的數據庫後臺對於通配符的使用是不一樣的。大部分模塊使用 `?` 或 `%s` ，還有其他一些使用了不同的符號，比如:0 或:1 來指示參數。同樣的，你還是得去參考你使用的數據庫模塊相應的文檔。一個數據庫模塊的 `paramstyle` 屬性包含了參數引用風格的信息。

對於簡單的數據庫數據的讀寫問題，使用數據庫 API 通常非常簡單。如果你要處理更加複雜的問題，建議你使用更加高級的接口，比如一個對象關係映射 ORM 所提供的接口。類似 `SQLAlchemy` 這樣的庫允許你使用 Python 類來表示一個數據庫表，並且能在隱藏底層 SQL 的情況下實現各種數據庫的操作。

6.9 編碼和解碼十六進制數

問題

你想將一個十六進制字符串解碼成一個字節字符串或者將一個字節字符串編碼成一個十六進制字符串。

解決方案

如果你只是簡單的解碼或編碼一個十六進制的原始字符串，可以使用 `binascii` 模塊。例如：

```
>>> # Initial byte string
>>> s = b'hello'
>>> # Encode as hex
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
>>> # Decode back to bytes
>>> binascii.a2b_hex(h)
b'hello'
>>>
```

類似的功能同樣可以在 base64 模塊中找到。例如：

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

討論

大部分情況下，通過使用上述的函數來轉換十六進制是很簡單的。上面兩種技術的主要不同在於大小寫的處理。函數 base64.b16decode() 和 base64.b16encode() 只能操作大寫形式的十六進制字母，而 binascii 模塊中的函數大小寫都能處理。

還有一點需要注意的是編碼函數所產生的輸出總是一個字節字符串。如果想強制以 Unicode 形式輸出，你需要增加一個額外的界面步驟。例如：

```
>>> h = base64.b16encode(s)
>>> print(h)
b'68656C6C6F'
>>> print(h.decode('ascii'))
68656C6C6F
>>>
```

在解碼十六進制數時，函數 b16decode() 和 a2b_hex() 可以接受字節或 unicode 字符串。但是，unicode 字符串必須僅僅只包含 ASCII 編碼的十六進制數。

6.10 編碼解碼 Base64 數據

問題

你需要使用 Base64 格式解碼或編碼二進制數據。

解決方案

base64 模塊中有兩個函數 b64encode() and b64decode() 可以幫你解決這個問題。例如;

```
>>> # Some byte data
>>> s = b'hello'
>>> import base64

>>> # Encode as Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Decode from Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

討論

Base64 編碼僅僅用於面向字節的數據比如字節字符串和字節數組。此外，編碼處理的輸出結果總是一個字節字符串。如果你想混合使用 Base64 編碼的數據和 Unicode 文本，你必須添加一個額外的解碼步驟。例如：

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

當解碼 Base64 的時候，字節字符串和 Unicode 文本都可以作爲參數。但是，Unicode 字符串只能包含 ASCII 字符。

6.11 讀寫二進制數組數據

問題

你想讀寫一個二進制數組的結構化數據到 Python 元組中。

解決方案

可以使用 struct 模塊處理二進制數據。下面是一段示例代碼將一個 Python 元組列表寫入一個二進制文件，並使用 struct 將每個元組編碼爲一個結構體。

```
from struct import Struct
def write_records(records, format, f):
    '''
```

```
    Write a sequence of tuples to a binary file of structures.
    '''
    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))

# Example
if __name__ == '__main__':
    records = [ (1, 2.3, 4.5),
                (6, 7.8, 9.0),
                (12, 13.4, 56.7) ]
    with open('data.b', 'wb') as f:
        write_records(records, '<idd', f)
```

有很多種方法來讀取這個文件並返回一個元組列表。首先，如果你打算以塊的形式增量讀取文件，你可以這樣做：

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)

# Example
if __name__ == '__main__':
    with open('data.b','rb') as f:
        for rec in read_records('<idd', f):
            # Process rec
            ...
```

如果你想將整個文件一次性讀取到一個字節字符串中，然後在分片解析。那麼你可以這樣做：

```
from struct import Struct

def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack_from(data, offset)
            for offset in range(0, len(data), record_struct.size))

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        data = f.read()
    for rec in unpack_records('<idd', data):
        # Process rec
        ...
```

兩種情況下的結果都是一個可返回用來創建該文件的原始元組的可迭代對象。

討論

對於需要編碼和解碼二進制數據的程序而言，通常會使用 struct 模塊。爲了聲明一個新的結構體，只需要像這樣創建一個 Struct 實例即可：

```
# Little endian 32-bit integer, two double precision floats
record_struct = Struct('<idd')
```

結構體通常會使用一些結構碼值 i, d, f 等 [參考 Python 文檔]。這些代碼分別代表某個特定的二進制數據類型如 32 位整數，64 位浮點數，32 位浮點數等。第一個字符 < 指定了字節順序。在這個例子中，它表示”低位在前”。更改這個字符爲 > 表示高位在前，或者是 ！表示網絡字節順序。

產生的 Struct 實例有很多屬性和方法用來操作相應類型的結構。size 屬性包含了結構的字節數，這在 I/O 操作時非常有用。pack() 和 unpack() 方法被用來打包和解包數據。比如：

```
>>> from struct import Struct
>>> record_struct = Struct('<idd')
>>> record_struct.size
20
>>> record_struct.pack(1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'
>>> record_struct.unpack(_)
(1, 2.0, 3.0)
>>>
```

有時候你還會看到 pack() 和 unpack() 操作以模塊級別函數被調用，類似下面這樣：

```
>>> import struct
>>> struct.pack('<idd', 1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'
>>> struct.unpack('<idd', _)
(1, 2.0, 3.0)
>>>
```

這樣可以工作，但是感覺沒有實例方法那麼優雅，特別是在你代碼中同樣的結構出現在多個地方的時候。通過創建一個 Struct 實例，格式代碼只會指定一次並且所有的操作被集中處理。這樣一來代碼維護就變得更加簡單了 (因爲你只需要改變一處代碼即可)。

讀取二進制結構的代碼要用到一些非常有趣而優美的編程技巧。在函數 read_records 中，iter() 被用來創建一個返回固定大小數據塊的迭代器，參考 5.8 小節。這個迭代器會不斷的調用一個用戶提供的可調用對象 (比如 lambda: f.read(record_struct.size))，直到它返回一個特殊的值 (如 b’‘)，這時候迭代停止。例如：

```
>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda: f.read(20), b'')
>>> chunks
```

```
<callable_iterator object at 0x10069e6d0>
>>> for chk in chunks:
... print(chk)
...
b'\x01\x00\x00\x00ffffff\x02@\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00333333\x1f@\x00\x00\x00\x00\x00\x00"@'
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc\xcc*@\x9a\x99\x99\x99\x99YL@'
>>>
```

如你所見，創建一個可迭代對象的一個原因是它能允許使用一個生成器推導來創建記錄。如果你不使用這種技術，那麼代碼可能會像下面這樣：

```
def read_records(format, f):
    record_struct = Struct(format)
    while True:
        chk = f.read(record_struct.size)
        if chk == b'':
            break
        yield record_struct.unpack(chk)
```

在函數 unpack_records() 中使用了另外一種方法 unpack_from() 。unpack_from() 對於從一個大型二進制數組中提取二進制數據非常有用，因爲它不會產生任何的臨時對象或者進行內存複製操作。你只需要給它一個字節字符串 (或數組) 和一個字節偏移量，它會從那個位置開始直接解包數據。

如果你使用 unpack() 來代替 unpack_from() ，你需要修改代碼來構造大量的小的切片以及進行偏移量的計算。比如：

```
def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack(data[offset:offset + record_struct.size])
            for offset in range(0, len(data), record_struct.size))
```

這種方案除了代碼看上去很複雜外，還得做很多額外的工作，因爲它執行了大量的偏移量計算，複製數據以及構造小的切片對象。如果你準備從讀取到的一個大型字節字符串中解包大量的結構體的話，unpack_from() 會表現的更出色。

在解包的時候，collections 模塊中的命名元組對象或許是你想要用到的。它可以讓你給返回元組設置屬性名稱。例如：

```
from collections import namedtuple

Record = namedtuple('Record', ['kind','x','y'])

with open('data.p', 'rb') as f:
    records = (Record(*r) for r in read_records('<idd', f))

for r in records:
    print(r.kind, r.x, r.y)
```

如果你的程序需要處理大量的二進制數據，你最好使用 numpy 模塊。例如，你可

以將一個二進制數據讀取到一個結構化數組中而不是一個元組列表中。就像下面這樣：

```
>>> import numpy as np
>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>
```

最後提一點，如果你需要從已知的文件格式 (如圖片格式，圖形文件，HDF5 等) 中讀取二進制數據時，先檢查看看 Python 是不是已經提供了現存的模塊。因爲不到萬不得已沒有必要去重複造輪子。

6.12 讀取嵌套和可變長二進制數據

問題

你需要讀取包含嵌套或者可變長記錄集合的複雜二進制格式的數據。這些數據可能包含圖片、視頻、電子地圖文件等。

解決方案

struct 模塊可被用來編碼/解碼幾乎所有類型的二進制的數據結構。爲了解釋清楚這種數據，假設你用下面的 Python 數據結構來表示一個組成一系列多邊形的點的集合：

```
polys = [
    [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
    [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
    [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
]
```

現在假設這個數據被編碼到一個以下列頭部開始的二進制文件中去了：

```
+------+--------+-------------------------------------+
|Byte  | Type   |  Description                        |
+======+========+=====================================+
|0     | int    |  文件代碼 ( 0x1234, 小端 )          |
+------+--------+-------------------------------------+
|4     | double |  x 的最小值 ( 小端 )                |
+------+--------+-------------------------------------+
|12    | double |  y 的最小值 ( 小端 )                |
+------+--------+-------------------------------------+
```

```
|20    | double |  x 的最大值 ( 小端 )                  |
+------+--------+-------------------------------------+
|28    | double |  y 的最大值 ( 小端 )                  |
+------+--------+-------------------------------------+
|36    | int    |  三角形數量 ( 小端 )                  |
+------+--------+-------------------------------------+
```

緊跟着頭部是一系列的多邊形記錄，編碼格式如下：

```
+------+--------+-------------------------------------------+
|Byte  | Type   |  Description                               |
+======+========+===========================================+
|0     | int    |  記錄長度 ( N 字節 )                        |
+------+--------+-------------------------------------------+
|4-N   | Points |  (X,Y) 座標，以浮點數表示                    |
+------+--------+-------------------------------------------+
```

爲了寫這樣的文件，你可以使用如下的 Python 代碼：

```
import struct
import itertools

def write_polys(filename, polys):
    # Determine bounding box
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)
    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi', 0x1234,
                            min_x, min_y,
                            max_x, max_y,
                            len(polys)))
        for poly in polys:
            size = len(poly) * struct.calcsize('<dd')
            f.write(struct.pack('<i', size + 4))
            for pt in poly:
                f.write(struct.pack('<dd', *pt))
```

將數據讀取回來的時候，可以利用函數 `struct.unpack()` ，代碼很相似，基本就是上面寫操作的逆序。如下：

```
def read_polys(filename):
    with open(filename, 'rb') as f:
        # Read the header
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iddddi', header)
        polys = []
        for n in range(num_polys):
```

```
            pbytes, = struct.unpack('<i', f.read(4))
            poly = []
            for m in range(pbytes // 16):
                pt = struct.unpack('<dd', f.read(16))
                poly.append(pt)
            polys.append(poly)
    return polys
```

儘管這個代碼可以工作，但是裏面混雜了很多讀取、解包數據結構和其他細節的代碼。如果用這樣的代碼來處理真實的數據文件，那未免也太繁雜了點。因此很顯然應該有另一種解決方法可以簡化這些步驟，讓程序員只關注自最重要的事情。

在本小節接下來的部分，我會逐步演示一個更加優秀的解析字節數據的方案。目標是可以給程序員提供一個高級的文件格式化方法，並簡化讀取和解包數據的細節。但是我要先提醒你，本小節接下來的部分代碼應該是整本書中最複雜最高級的例子，使用了大量的面向對象編程和元編程技術。一定要仔細的閱讀我們的討論部分，另外也要參考下其他章節內容。

首先，當讀取字節數據的時候，通常在文件開始部分會包含文件頭和其他的數據結構。儘管 struct 模塊可以解包這些數據到一個元組中去，另外一種表示這種信息的方式就是使用一個類。就像下面這樣：

```
import struct

class StructField:
    '''
    Descriptor representing a simple structure field
    '''
    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            r = struct.unpack_from(self.format, instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)
```

這裏我們使用了一個描述器來表示每個結構字段，每個描述器包含一個結構兼容格式的代碼以及一個字節偏移量，存儲在內部的內存緩衝中。在 `__get__()` 方法中，`struct.unpack_from()` 函數被用來從緩衝中解包一個值，省去了額外的分片或複製操作步驟。

`Structure` 類就是一個基礎類，接受字節數據並存儲在內部的內存緩衝中，並被 `StructField` 描述器使用。這裏使用了 `memoryview()`，我們會在後面詳細講解它是用來幹嘛的。

使用這個代碼，你現在就能定義一個高層次的結構對象來表示上面表格信息所期望的文件格式。例如：

```
class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)
```

下面的例子利用這個類來讀取之前我們寫入的多邊形數據的頭部數據：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

這個很有趣，不過這種方式還是有一些煩人的地方。首先，儘管你獲得了一個類接口的便利，但是這個代碼還是有點臃腫，還需要使用者指定很多底層的細節 (比如重複使用 StructField ，指定偏移量等)。另外，返回的結果類同樣確實一些便利的方法來計算結構的總數。

任何時候只要你遇到了像這樣冗餘的類定義，你應該考慮下使用類裝飾器或元類。元類有一個特性就是它能夠被用來填充許多低層的實現細節，從而釋放使用者的負擔。下面我來舉個例子，使用元類稍微改造下我們的 Structure 類：

```
class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith(('<','>','!','@')):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
```

```
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

class Structure(metaclass=StructureMeta):
    def __init__(self, bytedata):
        self._buffer = bytedata

    @classmethod
    def from_file(cls, f):
        return cls(f.read(cls.struct_size))
```

使用新的 Structure 類，你可以像下面這樣定義一個結構：

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
    ]
```

正如你所見，這樣寫就簡單多了。我們添加的類方法 from_file() 讓我們在不需要知道任何數據的大小和結構的情況下就能輕鬆的從文件中讀取數據。比如：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

一旦你開始使用了元類，你就可以讓它變得更加智能。例如，假設你還想支持嵌套的字節結構，下面是對前面元類的一個小的改進，提供了一個新的輔助描述器來達到想要的效果：

```
class NestedStruct:
    '''
    Descriptor representing a nested structure
    '''
```

```
    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                                    self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Save resulting structure back on instance to avoid
            # further recomputation of this step
            setattr(instance, self.name, result)
            return result

class StructureMeta(type):
    '''
    Metaclass that automatically creates StructField descriptors
    '''
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if isinstance(format, StructureMeta):
                setattr(self, fieldname,
                        NestedStruct(fieldname, format, offset))
                offset += format.struct_size
            else:
                if format.startswith(('<','>','!','@')):
                    byte_order = format[0]
                    format = format[1:]
                format = byte_order + format
                setattr(self, fieldname, StructField(format, offset))
                offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)
```

在這段代碼中，NestedStruct 描述器被用來疊加另外一個定義在某個內存區域上的結構。它通過將原始內存緩衝進行切片操作後實例化給定的結構類型。由於底層的內存緩衝區是通過一個內存視圖初始化的，所以這種切片操作不會引發任何的額外的內存複製。相反，它僅僅就是之前的內存的一個疊加而已。另外，爲了防止重複實例化，通過使用和 8.10 小節同樣的技術，描述器保存了該實例中的內部結構對象。

使用這個新的修正版，你就可以像下面這樣編寫：

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
```

```
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'), # nested struct
        (Point, 'max'), # nested struct
        ('i', 'num_polys')
    ]
```

令人驚訝的是，它也能按照預期的正常工作，我們實際操作下：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min # Nested structure
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>
```

到目前爲止，一個處理定長記錄的框架已經寫好了。但是如果組件記錄是變長的呢？比如，多邊形文件包含變長的部分。

一種方案是寫一個類來表示字節數據，同時寫一個工具函數來通過多少方式解析內容。跟 6.11 小節的代碼很類似：

```
class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)

    def iter_as(self, code):
        if isinstance(code, str):
            s = struct.Struct(code)
```

```
            for off in range(0, len(self._buffer), s.size):
                yield s.unpack_from(self._buffer, off)
        elif isinstance(code, StructureMeta):
            size = code.struct_size
            for off in range(0, len(self._buffer), size):
                data = self._buffer[off:off+size]
                yield code(data)
```

類方法 SizedRecord.from_file() 是一個工具，用來從一個文件中讀取帶大小前綴的數據塊，這也是很多文件格式常用的方式。作爲輸入，它接受一個包含大小編碼的結構格式編碼，並且也是自己形式。可選的 includes_size 參數指定了字節數是否包含頭部大小。下面是一個例子教你怎樣使用從多邊形文件中讀取單獨的多邊形數據：

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...              for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
<__main__.SizedRecord object at 0x1006a4f50>,
<__main__.SizedRecord object at 0x10070da90>]
>>>
```

可以看出，SizedRecord 實例的內容還沒有被解析出來。可以使用 iter_as() 方法來達到目的，這個方法接受一個結構格式化編碼或者是 Structure 類作爲輸入。這樣子可以很靈活的去解析數據，例如：

```
>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd'):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for n, poly in enumerate(polydata):
```

```
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>
```

將所有這些結合起來，下面是一個 `read_polys()` 函數的另外一個修正版：

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):
    polys = []
    with open(filename, 'rb') as f:
        phead = PolyHeader.from_file(f)
        for n in range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y) for p in rec.iter_as(Point) ]
            polys.append(poly)
    return polys
```

討論

這一節向你展示了許多高級的編程技術，包括描述器，延遲計算，元類，類變量和內存視圖。然而，它們都爲了同一個特定的目標服務。

上面的實現的一個主要特徵是它是基於懶解包的思想。當一個 Structure 實例被創建時，__init__() 僅僅只是創建一個字節數據的內存視圖，沒有做其他任何事。特別的，這時候並沒有任何的解包或者其他與結構相關的操作發生。這樣做的一個動機是你可能僅僅只對一個字節記錄的某一小部分感興趣。我們只需要解包你需要訪問的部分，而不是整個文件。

爲了實現懶解包和打包，需要使用 StructField 描述器類。用戶在 _fields_ 中列出來的每個屬性都會被轉化成一個 StructField 描述器，它將相關結構格式碼和偏移值保存到存儲緩存中。元類 StructureMeta 在多個結構類被定義時自動創建了這些描述器。我們使用元類的一個主要原因是它使得用戶非常方便的通過一個高層描述就能指定結構格式，而無需考慮低層的細節問題。

StructureMeta 的一個很微妙的地方就是它會固定字節數據順序。也就是說，如果任意的屬性指定了一個字節順序 (< 表示低位優先或者 > 表示高位優先)，那後面所有字段的順序都以這個順序爲準。這麼做可以幫助避免額外輸入，但是在定義的中間我們仍然可能切換順序的。比如，你可能有一些比較複雜的結構，就像下面這樣：

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'), # Big endian
        ('20s', 'unused'),
        ('i', 'file_length'),
        ('<i', 'version'), # Little endian
        ('i', 'shape_type'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('d', 'min_z'),
        ('d', 'max_z'),
        ('d', 'min_m'),
        ('d', 'max_m') ]
```

之前我們提到過，memoryview() 的使用可以幫助我們避免內存的複製。當結構存在嵌套的時候，memoryviews 可以疊加同一內存區域上定義的機構的不同部分。這個特性比較微妙，但是它關注的是內存視圖與普通字節數組的切片操作行爲。如果你在一個字節字符串或字節數組上執行切片操作，你通常會得到一個數據的拷貝。而內存視圖切片不是這樣的，它僅僅是在已存在的內存上面疊加而已。因此，這種方式更加高效。

還有很多相關的章節可以幫助我們擴展這裏討論的方案。參考 8.13 小節使用描述器構建一個類型系統。8.10 小節有更多關於延遲計算屬性值的討論，並且跟 NestedStruct 描述器的實現也有關。9.19 小節有一個使用元類來初始化類成員的例子，和 StructureMeta 類非常相似。Python 的 ctypes 源碼同樣也很有趣，它提供了對定義數據結構、數據結構嵌套這些相似功能的支持。

6.13 數據的累加與統計操作

問題

你需要處理一個很大的數據集並需要計算數據總和或其他統計量。

解決方案

對於任何涉及到統計、時間序列以及其他相關技術的數據分析問題，都可以考慮使用 Pandas 庫 。

爲了讓你先體驗下，下面是一個使用 Pandas 來分析芝加哥城市的 老鼠和齧齒類動物數據庫 的例子。在我寫這篇文章的時候，這個數據庫是一個擁有大概 74,000 行數據的 CSV 文件。

```
>>> import pandas

>>> # Read a CSV file, skipping last line
>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date 74055 non-null values
Status 74055 non-null values
Completion Date 72154 non-null values
Service Request Number 74055 non-null values
Type of Service Request 74055 non-null values
Number of Premises Baited 65804 non-null values
Number of Premises with Garbage 65600 non-null values
Number of Premises with Rats 65752 non-null values
Current Activity 66041 non-null values
Most Recent Action 66023 non-null values
Street Address 74055 non-null values
ZIP Code 73584 non-null values
X Coordinate 74043 non-null values
Y Coordinate 74043 non-null values
Ward 74044 non-null values
Police District 74044 non-null values
Community Area 74044 non-null values
Latitude 74043 non-null values
Longitude 74043 non-null values
Location 74043 non-null values
dtypes: float64(11), object(9)

>>> # Investigate range of values for a certain field
>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=object)
>>> # Filter the data
>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch Crew']
>>> len(crew_dispatched)
65676
>>>

>>> # Find 10 most rat-infested ZIP codes in Chicago
>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647 3837
```

```
60618 3530
60614 3284
60629 3251
60636 2801
60657 2465
60641 2238
60609 2206
60651 2152
60632 2071
>>>

>>> # Group by completion date
>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)
472
>>>

>>> # Determine counts on each day
>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011 4
01/03/2012 125
01/04/2011 54
01/04/2012 38
01/05/2011 78
01/05/2012 100
01/06/2011 100
01/06/2012 58
01/07/2011 1
01/09/2012 12
>>>

>>> # Sort the counts
>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
10/12/2012 313
10/21/2011 314
09/20/2011 316
10/26/2011 319
02/22/2011 325
10/26/2012 333
03/17/2011 336
10/13/2011 378
10/14/2011 391
10/07/2011 457
>>>
```

嗯，看樣子 2011 年 10 月 7 日對老鼠們來說是個很忙碌的日子啊！^_^

討論

Pandas 是一個擁有很多特性的大型函數庫，我在這裏不可能介紹完。但是隻要你需要去分析大型數據集合、對數據分組、計算各種統計量或其他類似任務的話，這個函數庫真的值得你去看一看。

第七章：函數

使用 def 語句定義函數是所有程序的基礎。本章的目標是講解一些更加高級和不常見的函數定義與使用模式。涉及到的內容包括默認參數、任意數量參數、強制關鍵字參數、註解和閉包。另外，一些高級的控制流和利用回調函數傳遞數據的技術在這裏也會講解到。

7.1 可接受任意數量參數的函數

問題

你想構造一個可接受任意數量參數的函數。

解決方案

爲了能讓一個函數接受任意數量的位置參數，可以使用一個 * 參數。例如：

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))

# Sample use
avg(1, 2) # 1.5
avg(1, 2, 3, 4) # 2.5
```

在這個例子中，rest 是由所有其他位置參數組成的元組。然後我們在代碼中把它當成了一個序列來進行後續的計算。

爲了接受任意數量的關鍵字參數，使用一個以 ** 開頭的參數。比如：

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
                name=name,
                attrs=attr_str,
                value=html.escape(value))
    return element

# Example
# Creates '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Creates '<p>&lt;spam&gt;</p>'
make_element('p', '<spam>')
```

在這裏，attrs 是一個包含所有被傳入進來的關鍵字參數的字典。

如果你還希望某個函數能同時接受任意數量的位置參數和關鍵字參數，可以同時使用 * 和 **。比如：

```
def anyargs(*args, **kwargs):
    print(args) # A tuple
    print(kwargs) # A dict
```

使用這個函數時，所有位置參數會被放到 args 元組中，所有關鍵字參數會被放到字典 kwargs 中。

討論

一個 * 參數只能出現在函數定義中最後一個位置參數後面，而**參數只能出現在最後一個參數。有一點要注意的是，在 * 參數後面仍然可以定義其他參數。

```
def a(x, *args, y):
    pass

def b(x, *args, y, **kwargs):
    pass
```

這種參數就是我們所說的強制關鍵字參數，在後面 7.2 小節還會詳細講解到。

7.2 只接受關鍵字參數的函數

問題

你希望函數的某些參數強制使用關鍵字參數傳遞

解決方案

將強制關鍵字參數放到某個 * 參數或者單個 * 後面就能達到這種效果。比如：

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True) # TypeError
recv(1024, block=True) # Ok
```

利用這種技術，我們還能在接受任意多個位置參數的函數中指定關鍵字參數。比如：

```
def mininum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m
```

```
minimum(1, 5, 2, -5, 10) # Returns -5
minimum(1, 5, 2, -5, 10, clip=0) # Returns 0
```

討論

很多情況下，使用強制關鍵字參數會比使用位置參數表意更加清晰，程序也更加具有可讀性。例如，考慮下如下一個函數調用：

```
msg = recv(1024, False)
```

如果調用者對 recv 函數並不是很熟悉，那他肯定不明白那個 False 參數到底來幹嘛用的。但是，如果代碼變成下面這樣子的話就清楚多了：

```
msg = recv(1024, block=False)
```

另外，使用強制關鍵字參數也會比使用 **kwargs 參數更好，因爲在使用函數 help 的時候輸出也會更容易理解：

```
>>> help(recv)
Help on function recv in module __main__:
recv(maxsize, *, block)
    Receives a message
```

強制關鍵字參數在一些更高級場合同樣也很有用。例如，它們可以被用來在使用 *args 和 **kwargs 參數作爲輸入的函數中插入參數，9.11 小節有一個這樣的例子。

7.3 給函數參數增加元信息

問題

你寫好了一個函數，然後想爲這個函數的參數增加一些額外的信息，這樣的話其他使用者就能清楚的知道這個函數應該怎麼使用。

解決方案

使用函數參數註解是一個很好的辦法，它能提示程序員應該怎樣正確使用這個函數。例如，下面有一個被註解了的函數：

```
def add(x:int, y:int) -> int:
    return x + y
```

python 解釋器不會對這些註解添加任何的語義。它們不會被類型檢查，運行時跟沒有加註解之前的效果也沒有任何差距。然而，對於那些閱讀源碼的人來講就很有幫助啦。第三方工具和框架可能會對這些註解添加語義。同時它們也會出現在文檔中。

```
>>> help(add)
Help on function add in module __main__:
add(x: int, y: int) -> int
>>>
```

儘管你可以使用任意類型的對象給函數添加註解 (例如數字，字符串，對象實例等等)，不過通常來講使用類或者字符串會比較好點。

討論

函數註解只存儲在函數的 __annotations__ 屬性中。例如：

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

儘管註解的使用方法可能有很多種，但是它們的主要用途還是文檔。因爲 python 並沒有類型聲明，通常來講僅僅通過閱讀源碼很難知道應該傳遞什麼樣的參數給這個函數。這時候使用註解就能給程序員更多的提示，讓他們可以正確的使用函數。

參考 9.20 小節的一個更加高級的例子，演示瞭如何利用註解來實現多分派 (比如重載函數)。

7.4 返回多個值的函數

問題

你希望構造一個可以返回多個值的函數

解決方案

爲了能返回多個值，函數直接 return 一個元組就行了。例如：

```
>>> def myfun():
... return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

討論

儘管 myfun() 看上去返回了多個值，實際上是先創建了一個元組然後返回的。這個語法看上去比較奇怪，實際上我們使用的是逗號來生成一個元組，而不是用括號。比如下面的：

```
>>> a = (1, 2) # With parentheses
>>> a
(1, 2)
>>> b = 1, 2 # Without parentheses
>>> b
(1, 2)
>>>
```

當我們調用返回一個元組的函數的時候，通常我們會將結果賦值給多個變量，就像上面的那樣。其實這就是 1.1 小節中我們所說的元組解包。返回結果也可以賦值給單個變量，這時候這個變量值就是函數返回的那個元組本身了：

```
>>> x = myfun()
>>> x
(1, 2, 3)
>>>
```

7.5 定義有默認參數的函數

問題

你想定義一個函數或者方法，它的一個或多個參數是可選的並且有一個默認值。

解決方案

定義一個有可選參數的函數是非常簡單的，直接在函數定義中給參數指定一個默認值，並放到參數列表最後就行了。例如：

```
def spam(a, b=42):
    print(a, b)

spam(1) # Ok. a=1, b=42
spam(1, 2) # Ok. a=1, b=2
```

如果默認參數是一個可修改的容器比如一個列表、集合或者字典，可以使用 None 作爲默認值，就像下面這樣：

```
# Using a list as a default value
def spam(a, b=None):
    if b is None:
        b = []
    ...
```

如果你並不想提供一個默認值，而是想僅僅測試下某個默認參數是不是有傳遞進來，可以像下面這樣寫：

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...
```

我們測試下這個函數：

```
>>> spam(1)
No b value supplied
>>> spam(1, 2) # b = 2
>>> spam(1, None) # b = None
>>>
```

仔細觀察可以發現到傳遞一個 None 值和不傳值兩種情況是有差別的。

討論

定義帶默認值參數的函數是很簡單的，但絕不僅僅只是這個，還有一些東西在這裏也深入討論下。

首先，默認參數的值僅僅在函數定義的時候賦值一次。試着運行下面這個例子：

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42
>>> x = 23 # Has no effect
>>> spam(1)
1 42
>>>
```

注意到當我們改變 x 的值的時候對默認參數值並沒有影響，這是因爲在函數定義的時候就已經確定了它的默認值了。

其次，默認參數的值應該是不可變的對象，比如 None、True、False、數字或字符串。特別的，千萬不要像下面這樣寫代碼：

```
def spam(a, b=[]): # NO!
    ...
```

如果你這麼做了，當默認值在其他地方被修改後你將會遇到各種麻煩。這些修改會影響到下次調用這個函數時的默認值。比如：

```
>>> def spam(a, b=[]):
...     print(b)
...     return b
...
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1) # Modified list gets returned!
[99, 'Yow!']
>>>
```

這種結果應該不是你想要的。爲了避免這種情況的發生，最好是將默認值設爲 None，然後在函數裏面檢查它，前面的例子就是這樣做的。

在測試 None 值時使用 is 操作符是很重要的，也是這種方案的關鍵點。有時候大家會犯下下面這樣的錯誤：

```
def spam(a, b=None):
    if not b: # NO! Use 'b is None' instead
        b = []
    ...
```

這麼寫的問題在於儘管 None 值確實是被當成 False，但是還有其他的對象 (比如長度爲 0 的字符串、列表、元組、字典等) 都會被當做 False。因此，上面的代碼會誤將一些其他輸入也當成是沒有輸入。比如：

```
>>> spam(1) # OK
>>> x = []
>>> spam(1, x) # Silent error. x value overwritten by default
>>> spam(1, 0) # Silent error. 0 ignored
>>> spam(1, '') # Silent error. '' ignored
>>>
```

最後一個問題比較微妙，那就是一個函數需要測試某個可選參數是否被使用者傳遞進來。這時候需要小心的是你不能用某個默認值比如 None、0 或者 False 值來測試用戶提供的值 (因爲這些值都是合法的值，是可能被用戶傳遞進來的)。因此，你需要其他的解決方案了。

爲了解決這個問題，你可以創建一個獨一無二的私有對象實例，就像上面的 _no_value 變量那樣。在函數裏面，你可以通過檢查被傳遞參數值跟這個實例是否一樣來判斷。這裏的思路是用戶不可能去傳遞這個 _no_value 實例作爲輸入。因此，這裏通過檢查這個值就能確定某個參數是否被傳遞進來了。

這裏對 object() 的使用看上去有點不太常見。object 是 python 中所有類的基類。你可以創建 object 類的實例，但是這些實例沒什麼實際用處，因爲它並沒有任何有用的方法，也沒有任何實例數據 (因爲它沒有任何的實例字典，你甚至都不能設置任何屬性值)。你唯一能做的就是測試同一性。這個剛好符合我的要求，因爲我在函數中就只

是需要一個同一性的測試而已。

7.6 定義匿名或內聯函數

問題

你想爲 sort() 操作創建一個很短的回調函數，但又不想用 def 去寫一個單行函數，而是希望通過某個快捷方式以內聯方式來創建這個函數。

解決方案

當一些函數很簡單，僅僅只是計算一個表達式的值的時候，就可以使用 lambda 表達式來代替了。比如：

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

這裏使用的 lambda 表達式跟下面的效果是一樣的：

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

lambda 表達式典型的使用場景是排序或數據 reduce 等：

```
>>> names = ['David Beazley', 'Brian Jones',
...          'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

討論

儘管 lambda 表達式允許你定義簡單函數，但是它的使用是有限制的。你只能指定單個表達式，它的值就是最後的返回值。也就是說不能包含其他的語言特性了，包括多個語句、條件表達式、迭代以及異常處理等等。

你可以不使用 lambda 表達式就能編寫大部分 python 代碼。但是，當有人編寫大量計算表達式值的短小函數或者需要用戶提供回調函數的程序的時候，你就會看到 lambda 表達式的身影了。

7.7 匿名函數捕獲變量值

問題

你用 lambda 定義了一個匿名函數，並想在定義時捕獲到某些變量的值。

解決方案

先看下下面代碼的效果：

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

現在我問你，a(10) 和 b(10) 返回的結果是什麼？如果你認爲結果是 20 和 30，那麼你就錯了：

```
>>> a(10)
30
>>> b(10)
30
>>>
```

這其中的奧妙在於 lambda 表達式中的 x 是一個自由變量，在運行時綁定值，而不是定義時就綁定，這跟函數的默認值參數定義是不同的。因此，在調用這個 lambda 表達式的時候，x 的值是執行時的值。例如：

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

如果你想讓某個匿名函數在定義時就捕獲到值，可以將那個參數值定義成默認參數即可，就像下面這樣：

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

討論

在這裏列出來的問題是新手很容易犯的錯誤，有些新手可能會不恰當的使用 lambda 表達式。比如，通過在一個循環或列表推導中創建一個 lambda 表達式列表，並期望函數能在定義時就記住每次的迭代值。例如：

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
... print(f(0))
...
4
4
4
4
4
>>>
```

但是實際效果是運行是 n 的值爲迭代的最後一個值。現在我們用另一種方式修改一下：

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
... print(f(0))
...
0
1
2
3
4
>>>
```

通過使用函數默認值參數形式，lambda 函數在定義時就能綁定到值。

7.8 減少可調用對象的參數個數

問題

你有一個被其他 python 代碼使用的 callable 對象，可能是一個回調函數或者是一個處理器，但是它的參數太多了，導致調用時出錯。

解決方案

如果需要減少某個函數的參數個數，你可以使用 `functools.partial()`。`partial()` 函數允許你給一個或多個參數設置固定的值，減少接下來被調用時的參數個數。爲了演示清楚，假設你有下面這樣的函數：

```
def spam(a, b, c, d):
    print(a, b, c, d)
```

現在我們使用 partial() 函數來固定某些參數值：

```
>>> from functools import partial
>>> s1 = partial(spam, 1) # a = 1
>>> s1(2, 3, 4)
1 2 3 4
>>> s1(4, 5, 6)
1 4 5 6
>>> s2 = partial(spam, d=42) # d = 42
>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42
>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
>>>
```

可以看出 partial() 固定某些參數並返回一個新的 callable 對象。這個新的 callable 接受未賦值的參數，然後跟之前已經賦值過的參數合併起來，最後將所有參數傳遞給原始函數。

討論

本節要解決的問題是讓原本不兼容的代碼可以一起工作。下面我會列舉一系列的例子。

第一個例子是，假設你有一個點的列表來表示 (x,y) 座標元組。你可以使用下面的函數來計算兩點之間的距離：

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

現在假設你想以某個點爲基點，根據點和基點之間的距離來排序所有的這些點。列表的 sort() 方法接受一個關鍵字參數來自定義排序邏輯，但是它只能接受一個單個參數的函數 (distance() 很明顯是不符合條件的)。現在我們可以通過使用 partial() 來解決這個問題：

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
```

```
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

更進一步，partial() 通常被用來微調其他庫函數所使用的回調函數的參數。例如，下面是一段代碼，使用 multiprocessing 來異步計算一個結果值，然後這個值被傳遞給一個接受一個 result 值和一個可選 logging 參數的回調函數：

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# A sample function
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

當給 apply_async() 提供回調函數時，通過使用 partial() 傳遞額外的 logging 參數。而 multiprocessing 對這些一無所知——它僅僅只是使用單個值來調用回調函數。

作爲一個類似的例子，考慮下編寫網絡服務器的問題，socketserver 模塊讓它變得很容易。下面是個簡單的 echo 服務器：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

不過，假設你想給 EchoHandler 增加一個可以接受其他配置選項的 __init__ 方法。比如：

```
class EchoHandler(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are
    # any normal parameters supplied (which are passed on)
```

```
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)

    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)
```

這麼修改後，我們就不需要顯式地在 TCPServer 類中添加前綴了。但是你再次運行程序後會報類似下面的錯誤：

```
Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'
```

初看起來好像很難修正這個錯誤，除了修改 socketserver 模塊源代碼或者使用某些奇怪的方法之外。但是，如果使用 partial() 就能很輕鬆的解決——給它傳遞 ack 參數的值來初始化即可，如下：

```
from functools import partial
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVED:'))
serv.serve_forever()
```

在這個例子中，__init__() 方法中的 ack 參數聲明方式看上去很有趣，其實就是聲明 ack 爲一個強制關鍵字參數。關於強制關鍵字參數問題我們在 7.2 小節我們已經討論過了，讀者可以再去回顧一下。

很多時候 partial() 能實現的效果，lambda 表達式也能實現。比如，之前的幾個例子可以使用下面這樣的表達式：

```
points.sort(key=lambda p: distance(pt, p))
p.apply_async(add, (3, 4), callback=lambda result: output_result(result,log))
serv = TCPServer(('', 15000),
        lambda *args, **kwargs: EchoHandler(*args, ack=b'RECEIVED:',␣
↪**kwargs))
```

這樣寫也能實現同樣的效果，不過相比而已會顯得比較臃腫，對於閱讀代碼的人來講也更加難懂。這時候使用 partial() 可以更加直觀的表達你的意圖 (給某些參數預先賦值)。

7.9 將單方法的類轉換爲函數

問題

你有一個除 __init__() 方法外只定義了一個方法的類。爲了簡化代碼，你想將它轉換成一個函數。

解決方案

大多數情況下，可以使用閉包來將單個方法的類轉換成函數。舉個例子，下面示例中的類允許使用者根據某個模板方案來獲取到 URL 鏈接地址。

```
from urllib.request import urlopen

class UrlTemplate:
    def __init__(self, template):
        self.template = template

    def open(self, **kwargs):
        return urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f=
↪{fields}')
for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

這個類可以被一個更簡單的函數來代替：

```
def urltemplate(template):
    def opener(**kwargs):
        return urlopen(template.format_map(kwargs))
    return opener

# Example use
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f=
↪{fields}')
for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

討論

大部分情況下，你擁有一個單方法類的原因是需要存儲某些額外的狀態來給方法使用。比如，定義 UrlTemplate 類的唯一目的就是先在某個地方存儲模板值，以便將來可以在 open() 方法中使用。

使用一個內部函數或者閉包的方案通常會更優雅一些。簡單來講，一個閉包就是一個函數，只不過在函數內部帶上了一個額外的變量環境。閉包關鍵特點就是它會記住自己被定義時的環境。因此，在我們的解決方案中，opener() 函數記住了 template 參數的值，並在接下來的調用中使用它。

任何時候只要你碰到需要給某個函數增加額外的狀態信息的問題，都可以考慮使用閉包。相比將你的函數轉換成一個類而言，閉包通常是一種更加簡潔和優雅的方案。

7.10 帶額外狀態信息的回調函數

問題

你的代碼中需要依賴到回調函數的使用 (比如事件處理器、等待後臺任務完成後的回調等)，並且你還需要讓回調函數擁有額外的狀態值，以便在它的內部使用到。

解決方案

這一小節主要討論的是那些出現在很多函數庫和框架中的回調函數的使用——特別是跟異步處理有關的。爲了演示與測試，我們先定義如下一個需要調用回調函數的函數：

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

    # Invoke the callback with the result
    callback(result)
```

實際上，這段代碼可以做任何更高級的處理，包括線程、進程和定時器，但是這些都不是我們要關心的。我們僅僅只需要關注回調函數的調用。下面是一個演示怎樣使用上述代碼的例子：

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

注意到 `print_result()` 函數僅僅只接受一個參數 `result` 。不能再傳入其他信息。而當你想讓回調函數訪問其他變量或者特定環境的變量值的時候就會遇到麻煩。

爲了讓回調函數訪問外部信息，一種方法是使用一個綁定方法來代替一個簡單函數。比如，下面這個類會保存一個內部序列號，每次接收到一個 `result` 的時候序列號加 1：

```
class ResultHandler:

    def __init__(self):
        self.sequence = 0

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

使用這個類的時候，你先創建一個類的實例，然後用它的 handler() 綁定方法來做爲回調函數：

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

第二種方式，作爲類的替代，可以使用一個閉包捕獲狀態值，例如：

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

下面是使用閉包方式的一個例子：

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

還有另外一個更高級的方法，可以使用協程來完成同樣的事情：

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

對於協程，你需要使用它的 send() 方法作爲回調函數，如下所示：

```
>>> handler = make_handler()
>>> next(handler) # Advance to the yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

討論

基於回調函數的軟件通常都有可能變得非常複雜。一部分原因是回調函數通常會跟請求執行代碼斷開。因此，請求執行和處理結果之間的執行環境實際上已經丟失了。如果你想讓回調函數連續執行多步操作，那你就必須去解決如何保存和恢復相關的狀態信息了。

至少有兩種主要方式來捕獲和保存狀態信息，你可以在一個對象實例 (通過一個綁定方法) 或者在一個閉包中保存它。兩種方式相比，閉包或許是更加輕量級和自然一點，因爲它們可以很簡單的通過函數來構造。它們還能自動捕獲所有被使用到的變量。因此，你無需去擔心如何去存儲額外的狀態信息 (代碼中自動判定)。

如果使用閉包，你需要注意對那些可修改變量的操作。在上面的方案中，nonlocal 聲明語句用來指示接下來的變量會在回調函數中被修改。如果沒有這個聲明，代碼會報錯。

而使用一個協程來作爲一個回調函數就更有趣了，它跟閉包方法密切相關。某種意義上來講，它顯得更加簡潔，因爲總共就一個函數而已。並且，你可以很自由的修改變量而無需去使用 nonlocal 聲明。這種方式唯一缺點就是相對於其他 Python 技術而言或許比較難以理解。另外還有一些比較難懂的部分，比如使用之前需要調用 next() ，實際使用時這個步驟很容易被忘記。儘管如此，協程還有其他用處，比如作爲一個內聯回調函數的定義 (下一節會講到)。

如果你僅僅只需要給回調函數傳遞額外的值的話，還有一種使用 partial() 的方式也很有用。在沒有使用 partial() 的時候，你可能經常看到下面這種使用 lambda 表達式的複雜代碼：

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

可以參考 7.8 小節的幾個示例，教你如何使用 partial() 來更改參數簽名來簡化上述代碼。

7.11 內聯回調函數

問題

當你編寫使用回調函數的代碼的時候，擔心很多小函數的擴張可能會弄亂程序控制流。你希望找到某個方法來讓代碼看上去更像是一個普通的執行序列。

解決方案

通過使用生成器和協程可以使得回調函數內聯在某個函數中。爲了演示說明，假設你有如下所示的一個執行某種計算任務然後調用一個回調函數的函數 (參考 7.10 小節)：

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

```
    # Invoke the callback with the result
    callback(result)
```

接下來讓我們看一下下面的代碼，它包含了一個 Async 類和一個 inlined_async 裝飾器：

```
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
```

這兩個代碼片段允許你使用 yield 語句內聯回調步驟。比如：

```
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')
```

如果你調用 test() ，你會得到類似如下的輸出：

```
5
helloworld
0
```

```
2
4
6
8
10
12
14
16
18
Goodbye
```

你會發現，除了那個特別的裝飾器和 yield 語句外，其他地方並沒有出現任何的回調函數 (其實是在後臺定義的)。

討論

本小節會實實在在的測試你關於回調函數、生成器和控制流的知識。

首先，在需要使用到回調的代碼中，關鍵點在於當前計算工作會掛起並在將來的某個時候重啓 (比如異步執行)。當計算重啓時，回調函數被調用來繼續處理結果。apply_async() 函數演示了執行回調的實際邏輯，儘管實際情況中它可能會更加複雜 (包括線程、進程、事件處理器等等)。

計算的暫停與重啓思路跟生成器函數的執行模型不謀而合。具體來講，yield 操作會使一個生成器函數產生一個值並暫停。接下來調用生成器的 __next__() 或 send() 方法又會讓它從暫停處繼續執行。

根據這個思路，這一小節的核心就在 inline_async() 裝飾器函數中了。關鍵點就是，裝飾器會逐步遍歷生成器函數的所有 yield 語句，每一次一個。爲了這樣做，剛開始的時候創建了一個 result 隊列並向裏面放入一個 None 值。然後開始一個循環操作，從隊列中取出結果值併發送給生成器，它會持續到下一個 yield 語句，在這裏一個 Async 的實例被接受到。然後循環開始檢查函數和參數，並開始進行異步計算 apply_async() 。然而，這個計算有個最詭異部分是它並沒有使用一個普通的回調函數，而是用隊列的 put() 方法來回調。

這時候，是時候詳細解釋下到底發生了什麼了。主循環立即返回頂部並在隊列上執行 get() 操作。如果數據存在，它一定是 put() 回調存放的結果。如果沒有數據，那麼先暫停操作並等待結果的到來。這個具體怎樣實現是由 apply_async() 函數來決定的。如果你不相信會有這麼神奇的事情，你可以使用 multiprocessing 庫來試一下，在單獨的進程中執行異步計算操作，如下所示：

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

實際上你會發現這個真的就是這樣的，但是要解釋清楚具體的控制流得需要點時間了。

將複雜的控制流隱藏到生成器函數背後的例子在標準庫和第三方包中都能看到。比如，在 contextlib 中的 @contextmanager 裝飾器使用了一個令人費解的技巧，通過一個 yield 語句將進入和離開上下文管理器粘合在一起。另外非常流行的 Twisted 包中也包含了非常類似的內聯回調。

7.12 訪問閉包中定義的變量

問題

你想要擴展函數中的某個閉包，允許它能訪問和修改函數的內部變量。

解決方案

通常來講，閉包的內部變量對於外界來講是完全隱藏的。但是，你可以通過編寫訪問函數並將其作爲函數屬性綁定到閉包上來實現這個目的。例如：

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

下面是使用的例子:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```

討論

爲了說明清楚它如何工作的，有兩點需要解釋一下。首先，nonlocal 聲明可以讓我們編寫函數來修改內部變量的值。其次，函數屬性允許我們用一種很簡單的方式將訪問方法綁定到閉包函數上，這個跟實例方法很像 (儘管並沒有定義任何類)。

還可以進一步的擴展，讓閉包模擬類的實例。你要做的僅僅是複製上面的內部函數到一個字典實例中並返回它即可。例如：

```
import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )
    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()

# Example use
def Stack():
    items = []
    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)

    return ClosureInstance()
```

下面是一個交互式會話來演示它是如何工作的：

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
```

```
10
>>>
```

有趣的是，這個代碼運行起來會比一個普通的類定義要快很多。你可能會像下面這樣測試它跟一個類的性能對比：

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __len__(self):
        return len(self.items)
```

如果這樣做，你會得到類似如下的結果：

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

結果顯示，閉包的方案運行起來要快大概 8%，大部分原因是因爲對實例變量的簡化訪問，閉包更快是因爲不會涉及到額外的 self 變量。

Raymond Hettinger 對於這個問題設計出了更加難以理解的改進方案。不過，你得考慮下是否真的需要在你代碼中這樣做，而且它只是真實類的一個奇怪的替換而已，例如，類的主要特性如繼承、屬性、描述器或類方法都是不能用的。並且你要做一些其他的工作才能讓一些特殊方法生效 (比如上面 `ClosureInstance` 中重寫過的 `__len__()` 實現。)

最後，你可能還會讓其他閱讀你代碼的人感到疑惑，爲什麼它看起來不像一個普通的類定義呢？(當然，他們也想知道爲什麼它運行起來會更快)。儘管如此，這對於怎樣訪問閉包的內部變量也不失爲一個有趣的例子。

總體上講，在配置的時候給閉包添加方法會有更多的實用功能，比如你需要重置內部狀態、刷新緩衝區、清除緩存或其他的反饋機制的時候。

第八章：類與對象

本章主要關注點的是和類定義有關的常見編程模型。包括讓對象支持常見的 Python 特性、特殊方法的使用、類封裝技術、繼承、內存管理以及有用的設計模式。

8.1 改變對象的字符串顯示

問題

你想改變對象實例的打印或顯示輸出，讓它們更具可讀性。

解決方案

要改變一個實例的字符串表示，可重新定義它的 `__str__()` 和 `__repr__()` 方法。例如：

```
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)
```

`__repr__()` 方法返回一個實例的代碼表示形式，通常用來重新構造這個實例。內置的 `repr()` 函數返回這個字符串，跟我們使用交互式解釋器顯示的值是一樣的。`__str__()` 方法將實例轉換爲一個字符串，使用 `str()` 或 `print()` 函數會輸出這個字符串。比如：

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4) # __repr__() output
>>> print(p)
(3, 4) # __str__() output
>>>
```

我們在這裏還演示了在格式化的時候怎樣使用不同的字符串表現形式。特別來講，`!r` 格式化代碼指明輸出使用 `__repr__()` 來代替默認的 `__str__()`。你可以用前面的類來試着測試下：

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
```

```
p is (3, 4)
>>>
```

討論

自定義 `__repr__()` 和 `__str__()` 通常是很好的習慣，因爲它能簡化調試和實例輸出。例如，如果僅僅只是打印輸出或日誌輸出某個實例，那麼程序員會看到實例更加詳細與有用的信息。

`__repr__()` 生成的文本字符串標準做法是需要讓 `eval(repr(x)) == x` 爲真。如果實在不能這樣子做，應該創建一個有用的文本表示，並使用 $<$ 和 $>$ 括起來。比如：

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

如果 `__str__()` 沒有被定義，那麼就會使用 `__repr__()` 來代替輸出。

上面的 `format()` 方法的使用看上去很有趣，格式化代碼 `{0.x}` 對應的是第 1 個參數的 `x` 屬性。因此，在下面的函數中，0 實際上指的就是 `self` 本身：

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

作爲這種實現的一個替代，你也可以使用 `%` 操作符，就像下面這樣：

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.2 自定義字符串的格式化

問題

你想通過 `format()` 函數和字符串方法使得一個對象能支持自定義的格式化。

解決方案

爲了自定義字符串的格式化，我們需要在類上面定義 `__format__()` 方法。例如：

```
_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)
```

現在 Date 類的實例可以支持格式化操作了，如同下面這樣：

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

討論

__format__() 方法給 Python 的字符串格式化功能提供了一個鉤子。這裏需要着重強調的是格式化代碼的解析工作完全由類自己決定。因此，格式化代碼可以是任何值。例如，參考下面來自 datetime 模塊中的代碼：

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

對於內置類型的格式化有一些標準的約定。可以參考 string 模塊文檔 說明。

8.3 讓對象支持上下文管理協議

問題

你想讓你的對象支持上下文管理協議 (with 語句)。

解決方案

爲了讓一個對象兼容 with 語句，你需要實現 __enter__() 和 __exit__() 方法。例如，考慮如下的一個類，它能爲我們創建一個網絡連接：

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

這個類的關鍵特點在於它表示了一個網絡連接，但是初始化的時候並不會做任何事情 (比如它並沒有建立一個連接)。連接的建立和關閉是使用 with 語句自動完成的，例如：

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed
with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

討論

編寫上下文管理器的主要原理是你的代碼會放到 with 語句塊中執行。當出現 with 語句的時候，對象的 __enter__() 方法被觸發，它返回的值 (如果有的話) 會被賦值給 as 聲明的變量。然後，with 語句塊裏面的代碼開始執行。最後，__exit__() 方法被觸發進行清理工作。

不管 with 代碼塊中發生什麼，上面的控制流都會執行完，就算代碼塊中發生了異常也是一樣的。事實上，__exit__() 方法的第三個參數包含了異常類型、異常值和追

溯信息 (如果有的話)。__exit__() 方法能自己決定怎樣利用這個異常信息，或者忽略它並返回一個 None 值。如果 __exit__() 返回 True ，那麼異常會被清空，就好像什麼都沒發生一樣，with 語句後面的程序繼續在正常執行。

還有一個細節問題就是 LazyConnection 類是否允許多個 with 語句來嵌套使用連接。很顯然，上面的定義中一次只能允許一個 socket 連接，如果正在使用一個 socket 的時候又重複使用 with 語句，就會產生一個異常了。不過你可以像下面這樣修改下上面的實現來解決這個問題：

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Example use
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
    pass
    with conn as s2:
        pass
        # s1 and s2 are independent sockets
```

在第二個版本中，LazyConnection 類可以被看做是某個連接工廠。在內部，一個列表被用來構造一個棧。每次 __enter__() 方法執行的時候，它複製創建一個新的連接並將其加入到棧裏面。__exit__() 方法簡單的從棧中彈出最後一個連接並關閉它。這裏稍微有點難理解，不過它能允許嵌套使用 with 語句創建多個連接，就如上面演示的那樣。

在需要管理一些資源比如文件、網絡連接和鎖的編程環境中，使用上下文管理器是很普遍的。這些資源的一個主要特徵是它們必須被手動的關閉或釋放來確保程序的正確運行。例如，如果你請求了一個鎖，那麼你必須確保之後釋放了它，否則就可能產生死鎖。通過實現 __enter__() 和 __exit__() 方法並使用 with 語句可以很容易的避免這些問題，因爲 __exit__() 方法可以讓你無需擔心這些了。

在 contextmanager 模塊中有一個標準的上下文管理方案模板，可參考 9.22 小節。同時在 12.6 小節中還有一個對本節示例程序的線程安全的修改版。

8.4 創建大量對象時節省內存方法

問題

你的程序要創建大量 (可能上百萬) 的對象，導致佔用很大的內存。

解決方案

對於主要是用來當成簡單的數據結構的類而言，你可以通過給類添加 `__slots__` 屬性來極大的減少實例所佔的內存。比如：

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

當你定義 `__slots__` 後，Python 就會爲實例使用一種更加緊湊的內部表示。實例通過一個很小的固定大小的數組來構建，而不是爲每個實例定義一個字典，這跟元組或列表很類似。在 `__slots__` 中列出的屬性名在內部被映射到這個數組的指定小標上。使用 slots 一個不好的地方就是我們不能再給實例添加新的屬性了，只能使用在 `__slots__` 中定義的那些屬性名。

討論

使用 slots 後節省的內存會跟存儲屬性的數量和類型有關。不過，一般來講，使用到的內存總量和將數據存儲在一個元組中差不多。爲了給你一個直觀認識，假設你不使用 slots 直接存儲一個 Date 實例，在 64 位的 Python 上面要佔用 428 字節，而如果使用了 slots，內存佔用下降到 156 字節。如果程序中需要同時創建大量的日期實例，那麼這個就能極大的減小內存使用量了。

儘管 slots 看上去是一個很有用的特性，很多時候你還是得減少對它的使用衝動。Python 的很多特性都依賴於普通的基於字典的實現。另外，定義了 slots 後的類不再支持一些普通類特性了，比如多繼承。大多數情況下，你應該只在那些經常被使用到的用作數據結構的類上定義 slots (比如在程序中需要創建某個類的幾百萬個實例對象)。

關於 `__slots__` 的一個常見誤區是它可以作爲一個封裝工具來防止用戶給實例增加新的屬性。儘管使用 slots 可以達到這樣的目的，但是這個並不是它的初衷。`__slots__` 更多的是用來作爲一個內存優化工具。

8.5 在類中封裝屬性名

問題

你想封裝類的實例上面的“私有”數據，但是 Python 語言並沒有訪問控制。

解決方案

Python 程序員不去依賴語言特性去封裝數據，而是通過遵循一定的屬性和方法命名規約來達到這個效果。第一個約定是任何以單下劃線 _ 開頭的名字都應該是內部實現。比如：

```
class A:
    def __init__(self):
        self._internal = 0 # An internal attribute
        self.public = 1 # A public attribute

    def public_method(self):
        '''
        A public method
        '''
        pass

    def _internal_method(self):
        pass
```

Python 並不會真的阻止別人訪問內部名稱。但是如果你這麼做肯定是不好的，可能會導致脆弱的代碼。同時還要注意到，使用下劃線開頭的約定同樣適用於模塊名和模塊級別函數。例如，如果你看到某個模塊名以單下劃線開頭 (比如 _socket)，那它就是內部實現。類似的，模塊級別函數比如 sys._getframe() 在使用的時候就得加倍小心了。

你還可能會遇到在類定義中使用兩個下劃線 (__) 開頭的命名。比如：

```
class B:
    def __init__(self):
        self.__private = 0

    def __private_method(self):
        pass

    def public_method(self):
        pass
        self.__private_method()
```

使用雙下劃線開始會導致訪問名稱變成其他形式。比如，在前面的類 B 中，私有屬性會被分別重命名爲 _B__private 和 _B__private_method 。這時候你可能會問這樣重命名的目的是什麼，答案就是繼承——這種屬性通過繼承是無法被覆蓋的。比如：

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1 # Does not override B.__private

    # Does not override B.__private_method()
    def __private_method(self):
        pass
```

這裏，私有名稱 `__private` 和 `__private_method` 被重命名爲 `_C__private` 和 `_C__private_method`，這個跟父類 B 中的名稱是完全不同的。

討論

上面提到有兩種不同的編碼約定 (單下劃線和雙下劃線) 來命名私有屬性，那麼問題就來了：到底哪種方式好呢？大多數而言，你應該讓你的非公共名稱以單下劃線開頭。但是，如果你清楚你的代碼會涉及到子類，並且有些內部屬性應該在子類中隱藏起來，那麼才考慮使用雙下劃線方案。

還有一點要注意的是，有時候你定義的一個變量和某個保留關鍵字衝突，這時候可以使用單下劃線作爲後綴，例如：

```
lambda_ = 2.0 # Trailing _ to avoid clash with lambda keyword
```

這裏我們並不使用單下劃線前綴的原因是它避免誤解它的使用初衷 (如使用單下劃線前綴的目的是爲了防止命名衝突而不是指明這個屬性是私有的)。通過使用單下劃線後綴可以解決這個問題。

8.6 創建可管理的屬性

問題

你想給某個實例 attribute 增加除訪問與修改之外的其他處理邏輯，比如類型檢查或合法性驗證。

解決方案

自定義某個屬性的一種簡單方法是將它定義爲一個 property。例如，下面的代碼定義了一個 property，增加對一個屬性簡單的類型檢查：

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

```
    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError("Can't delete attribute")
```

上述代碼中有三個相關聯的方法，這三個方法的名字都必須一樣。第一個方法是一個 getter 函數，它使得 first_name 成爲一個屬性。其他兩個方法給 first_name 屬性添加了 setter 和 deleter 函數。需要強調的是隻有在 first_name 屬性被創建後，後面的兩個裝飾器 @first_name.setter 和 @first_name.deleter 才能被定義。

property 的一個關鍵特徵是它看上去跟普通的 attribute 沒什麼兩樣，但是訪問它的時候會自動觸發 getter 、 setter 和 deleter 方法。例如：

```
>>> a = Person('Guido')
>>> a.first_name # Calls the getter
'Guido'
>>> a.first_name = 42 # Calls the setter
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "prop.py", line 14, in first_name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can`t delete attribute
>>>
```

在實現一個 property 的時候，底層數據 (如果有的話) 仍然需要存儲在某個地方。因此，在 get 和 set 方法中，你會看到對 _first_name 屬性的操作，這也是實際數據保存的地方。另外，你可能還會問爲什麼 __init__() 方法中設置了 self.first_name 而不是 self._first_name 。在這個例子中，我們創建一個 property 的目的就是在設置 attribute 的時候進行檢查。因此，你可能想在初始化的時候也進行這種類型檢查。通過設置 self.first_name ，自動調用 setter 方法，這個方法裏面會進行參數的檢查，否則就是直接訪問 self._first_name 了。

還能在已存在的 get 和 set 方法基礎上定義 property。例如：

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name

    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

```
    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

討論

一個 property 屬性其實就是一系列相關綁定方法的集合。如果你去查看擁有 property 的類，就會發現 property 本身的 fget、fset 和 fdel 屬性就是類裏面的普通方法。比如：

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

通常來講，你不會直接取調用 fget 或者 fset，它們會在訪問 property 的時候自動被觸發。

只有當你確實需要對 attribute 執行其他額外的操作的時候才應該使用到 property。有時候一些從其他編程語言 (比如 Java) 過來的程序員總認爲所有訪問都應該通過 getter 和 setter，所以他們認爲代碼應該像下面這樣寫：

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

不要寫這種沒有做任何其他額外操作的 property。首先，它會讓你的代碼變得很臃腫，並且還會迷惑閱讀者。其次，它還會讓你的程序運行起來變慢很多。最後，這樣的設計並沒有帶來任何的好處。特別是當你以後想給普通 attribute 訪問添加額外的處理邏輯的時候，你可以將它變成一個 property 而無需改變原來的代碼。因爲訪問 attribute 的代碼還是保持原樣。

Properties 還是一種定義動態計算 attribute 的方法。這種類型的 attributes 並不會被實際的存儲，而是在需要的時候計算出來。比如：

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * self.radius ** 2

    @property
    def diameter(self):
        return self.radius * 2

    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

在這裏，我們通過使用 properties，將所有的訪問接口形式統一起來，對半徑、直徑、周長和麪積的訪問都是通過屬性訪問，就跟訪問簡單的 attribute 是一樣的。如果不這樣做的話，那麼就要在代碼中混合使用簡單屬性訪問和方法調用。下面是使用的實例：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area  # Notice lack of ()
50.26548245743669
>>> c.perimeter  # Notice lack of ()
25.132741228718345
>>>
```

儘管 properties 可以實現優雅的編程接口，但有些時候你還是會想直接使用 getter 和 setter 函數。例如：

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

這種情況的出現通常是因爲 Python 代碼被集成到一個大型基礎平臺架構或程序中。例如，有可能是一個 Python 類準備加入到一個基於遠程過程調用的大型分佈式系統中。這種情況下，直接使用 get/set 方法 (普通方法調用) 而不是 property 或許會更容易兼容。

最後一點，不要像下面這樣寫有大量重複代碼的 property 定義：

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
```

```
    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)
    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._last_name = value
```

重複代碼會導致臃腫、易出錯和醜陋的程序。好消息是，通過使用裝飾器或閉包，有很多種更好的方法來完成同樣的事情。可以參考 8.9 和 9.21 小節的內容。

8.7 調用父類方法

問題

你想在子類中調用父類的某個已經被覆蓋的方法。

解決方案

爲了調用父類 (超類) 的一個方法，可以使用 `super()` 函數，比如：

```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()  # Call parent spam()
```

`super()` 函數的一個常見用法是在 `__init__()` 方法中確保父類被正確的初始化了：

```
class A:
    def __init__(self):
        self.x = 0

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

super() 的另外一個常見用法出現在覆蓋 Python 特殊方法的代碼中，比如：

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value) # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

在上面代碼中，__setattr__() 的實現包含一個名字檢查。如果某個屬性名以下劃線 (_) 開頭，就通過 super() 調用原始的 __setattr__()，否則的話就委派給內部的代理對象 self._obj 去處理。這看上去有點意思，因爲就算沒有顯式的指明某個類的父類，super() 仍然可以有效的工作。

討論

實際上，大家對於在 Python 中如何正確使用 super() 函數普遍知之甚少。你有時候會看到像下面這樣直接調用父類的一個方法：

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

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

儘管對於大部分代碼而言這麼做沒什麼問題，但是在更復雜的涉及到多繼承的代碼中就有可能導致很奇怪的問題發生。比如，考慮如下的情況：

```
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__')
```

如果你運行這段代碼就會發現 `Base.__init__()` 被調用兩次，如下所示：

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

可能兩次調用 `Base.__init__()` 沒什麼壞處，但有時候卻不是。另一方面，假設你在代碼中換成使用 `super()`，結果就很完美了：

```
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__()  # Only one call to super() here
        print('C.__init__')
```

運行這個新版本後，你會發現每個 `__init__()` 方法只會被調用一次了：

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

爲了弄清它的原理，我們需要花點時間解釋下 Python 是如何實現繼承的。對於你定義的每一個類，Python 會計算出一個所謂的方法解析順序 (MRO) 列表。這個 MRO 列表就是一個簡單的所有基類的線性順序表。例如：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>
```

爲了實現繼承，Python 會在 MRO 列表上從左到右開始查找基類，直到找到第一個匹配這個屬性的類爲止。

而這個 MRO 列表的構造是通過一個 C3 線性化算法來實現的。我們不去深究這個算法的數學原理，它實際上就是合併所有父類的 MRO 列表並遵循如下三條準則：

- 子類會先於父類被檢查
- 多個父類會根據它們在列表中的順序被檢查
- 如果對下一個類存在兩個合法的選擇，選擇第一個父類

老實說，你所要知道的就是 MRO 列表中的類順序會讓你定義的任意類層級關係變得有意義。

當你使用 `super()` 函數時，Python 會在 MRO 列表上繼續搜索下一個類。只要每個重定義的方法統一使用 `super()` 並只調用它一次，那麼控制流最終會遍歷完整個 MRO 列表，每個方法也只會被調用一次。這也是爲什麼在第二個例子中你不會調用兩次 `Base.__init__()` 的原因。

`super()` 有個令人吃驚的地方是它並不一定去查找某個類在 MRO 中下一個直接父類，你甚至可以在一個沒有直接父類的類中使用它。例如，考慮如下這個類：

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

如果你試着直接使用這個類就會出錯：

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
```

```
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

但是，如果你使用多繼承的話看看會發生什麼：

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

你可以看到在類 A 中使用 `super().spam()` 實際上調用的是跟類 A 毫無關係的類 B 中的 `spam()` 方法。這個用類 C 的 MRO 列表就可以完全解釋清楚了：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

在定義混入類的時候這樣使用 `super()` 是很普遍的。可以參考 8.13 和 8.18 小節。

然而，由於 `super()` 可能會調用不是你想要的方法，你應該遵循一些通用原則。首先，確保在繼承體系中所有相同名字的方法擁有可兼容的參數簽名 (比如相同的參數個數和參數名稱)。這樣可以確保 `super()` 調用一個非直接父類方法時不會出錯。其次，最好確保最頂層的類提供了這個方法的實現，這樣的話在 MRO 上面的查找鏈肯定可以找到某個確定的方法。

在 Python 社區中對於 `super()` 的使用有時候會引來一些爭議。儘管如此，如果一切順利的話，你應該在你最新代碼中使用它。Raymond Hettinger 爲此寫了一篇非常好的文章 “Python’s super() Considered Super!” ，通過大量的例子向我們解釋了爲什麼 `super()` 是極好的。

8.8 子類中擴展 property

問題

在子類中，你想要擴展定義在父類中的 property 的功能。

解決方案

考慮如下的代碼，它定義了一個 property：

```
class Person:
    def __init__(self, name):
        self.name = name

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Deleter function
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

下面是一個示例類，它繼承自 Person 並擴展了 name 屬性的功能：

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

接下來使用這個新類：

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
```

```
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

如果你僅僅只想擴展 property 的某一個方法，那麼可以像下面這樣寫：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

或者，你只想修改 setter 方法，就這麼寫：

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

討論

在子類中擴展一個 property 可能會引起很多不易察覺的問題，因爲一個 property 其實是 getter、setter 和 deleter 方法的集合，而不是單個方法。因此，當你擴展一個 property 的時候，你需要先確定你是否要重新定義所有的方法還是說只修改其中某一個。

在第一個例子中，所有的 property 方法都被重新定義。在每一個方法中，使用了 super() 來調用父類的實現。在 setter 函數中使用 super(SubPerson, SubPerson).name.__set__(self, value) 的語句是沒有錯的。爲了委託給之前定義的 setter 方法，需要將控制權傳遞給之前定義的 name 屬性的 __set__() 方法。不過，獲取這個方法的唯一途徑是使用類變量而不是實例變量來訪問它。這也是爲什麼我們要使用 super(SubPerson, SubPerson) 的原因。

如果你只想重定義其中一個方法，那隻使用 @property 本身是不夠的。比如，下面的代碼就無法工作：

```
class SubPerson(Person):
    @property  # Doesn't work
    def name(self):
        print('Getting name')
        return super().name
```

如果你試着運行會發現 setter 函數整個消失了：

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 5, in __init__
        self.name = name
AttributeError: can't set attribute
>>>
```

你應該像之前說過的那樣修改代碼：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

這麼寫後，property 之前已經定義過的方法會被複制過來，而 getter 函數被替換。然後它就能按照期望的工作了：

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

在這個特別的解決方案中，我們沒辦法使用更加通用的方式去替換硬編碼的 Person 類名。如果你不知道到底是哪個基類定義了 property，那你只能通過重新定義所有 property 並使用 super() 來將控制權傳遞給前面的實現。

值的注意的是上面演示的第一種技術還可以被用來擴展一個描述器 (在 8.9 小節我們有專門的介紹)。比如：

```
# A descriptor
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]
```

```
    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# A class with a descriptor
class Person:
    name = String('name')

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

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

最後值的注意的是，讀到這裏時，你應該會發現子類化 `setter` 和 `deleter` 方法其實是很簡單的。這裏演示的解決方案同樣適用，但是在 Python 的 issue 頁面 報告的一個 bug，或許會使得將來的 Python 版本中出現一個更加簡潔的方法。

8.9 創建新的類或實例屬性

問題

你想創建一個新的擁有一些額外功能的實例屬性類型，比如類型檢查。

解決方案

如果你想創建一個全新的實例屬性，可以通過一個描述器類的形式來定義它的功能。下面是一個例子：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
```

```
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

一個描述器就是一個實現了三個核心的屬性訪問操作 (get, set, delete) 的類，分別爲 __get__() 、__set__() 和 __delete__() 這三個特殊的方法。這些方法接受一個實例作爲輸入，之後相應的操作實例底層的字典。

爲了使用一個描述器，需將這個描述器的實例作爲類屬性放到一個類的定義中。例如：

```
class Point:
    x = Integer('x')
    y = Integer('y')

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

當你這樣做後，所有對描述器屬性 (比如 x 或 y) 的訪問會被 __get__() 、__set__() 和 __delete__() 方法捕獲到。例如：

```
>>> p = Point(2, 3)
>>> p.x # Calls Point.x.__get__(p,Point)
2
>>> p.y = 5 # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3 # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "descrip.py", line 12, in __set__
        raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

作爲輸入，描述器的每一個方法會接受一個操作實例。爲了實現請求操作，會相應的操作實例底層的字典 (__dict__ 屬性)。描述器的 self.name 屬性存儲了在實例字典中被實際使用到的 key。

討論

描述器可實現大部分 Python 類特性中的底層魔法，包括 `@classmethod` 、`@staticmethod` 、`@property` ，甚至是 `__slots__` 特性。

通過定義一個描述器，你可以在底層捕獲核心的實例操作 (get, set, delete)，並且可完全自定義它們的行爲。這是一個強大的工具，有了它你可以實現很多高級功能，並且它也是很多高級庫和框架中的重要工具之一。

描述器的一個比較困惑的地方是它只能在類級別被定義，而不能爲每個實例單獨定義。因此，下面的代碼是無法工作的：

```
# Does NOT work
class Point:
    def __init__(self, x, y):
        self.x = Integer('x') # No! Must be a class variable
        self.y = Integer('y')
        self.x = x
        self.y = y
```

同時，`__get__()` 方法實現起來比看上去要複雜得多：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
```

`__get__()` 看上去有點複雜的原因歸結於實例變量和類變量的不同。如果一個描述器被當做一個類變量來訪問，那麼 `instance` 參數被設置成 `None` 。這種情況下，標準做法就是簡單的返回這個描述器本身即可 (儘管你還可以添加其他的自定義操作)。例如：

```
>>> p = Point(2,3)
>>> p.x # Calls Point.x.__get__(p, Point)
2
>>> Point.x # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

描述器通常是那些使用到裝飾器或元類的大型框架中的一個組件。同時它們的使用也被隱藏在後面。舉個例子，下面是一些更高級的基於描述器的代碼，並涉及到一個類裝飾器：

```
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, cls):
```

```
        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]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate

# Example use
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

最後要指出的一點是，如果你只是想簡單的自定義某個類的單個屬性訪問的話就不用去寫描述器了。這種情況下使用 8.6 小節介紹的 property 技術會更加容易。當程序中有很多重複代碼的時候描述器就很有用了 (比如你想在你代碼的很多地方使用描述器提供的功能或者將它作爲一個函數庫特性)。

8.10 使用延遲計算屬性

問題

你想將一個只讀屬性定義成一個 property，並且只在訪問的時候纔會計算結果。但是一旦被訪問後，你希望結果值被緩存起來，不用每次都去計算。

解決方案

定義一個延遲屬性的一種高效方法是通過使用一個描述器類，如下所示：

```
class lazyproperty:
    def __init__(self, func):
        self.func = func
```

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

你需要像下面這樣在一個類中使用它：

```
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
```

下面在一個交互環境中演示它的使用：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
>>>
```

仔細觀察你會發現消息 `Computing area` 和 `Computing perimeter` 僅僅出現一次。

討論

很多時候，構造一個延遲計算屬性的主要目的是爲了提升性能。例如，你可以避免計算這些屬性值，除非你真的需要它們。這裏演示的方案就是用來實現這樣的效果的，

只不過它是通過以非常高效的方式使用描述器的一個精妙特性來達到這種效果的。

正如在其他小節 (如 8.9 小節) 所講的那樣，當一個描述器被放入一個類的定義時，每次訪問屬性時它的 `__get__()`、`__set__()` 和 `__delete__()` 方法就會被觸發。不過，如果一個描述器僅僅只定義了一個 `__get__()` 方法的話，它比通常的具有更弱的綁定。特別地，只有當被訪問屬性不在實例底層的字典中時 `__get__()` 方法纔會被觸發。

`lazyproperty` 類利用這一點，使用 `__get__()` 方法在實例中存儲計算出來的值，這個實例使用相同的名字作爲它的 property。這樣一來，結果值被存儲在實例字典中並且以後就不需要再去計算這個 property 了。你可以嘗試更深入的例子來觀察結果：

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

這種方案有一個小缺陷就是計算出的值被創建後是可以被修改的。例如：

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

如果你擔心這個問題，那麼可以使用一種稍微沒那麼高效的實現，就像下面這樣：

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
```

```
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

如果你使用這個版本，就會發現現在修改操作已經不被允許了：

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

然而，這種方案有一個缺點就是所有 get 操作都必須被定向到屬性的 getter 函數上去。這個跟之前簡單的在實例字典中查找值的方案相比效率要低一點。如果想獲取更多關於 property 和可管理屬性的信息，可以參考 8.6 小節。而描述器的相關內容可以在 8.9 小節找到。

8.11 簡化數據結構的初始化

問題

你寫了很多僅僅用作數據結構的類，不想寫太多煩人的 `__init__()` 函數

解決方案

可以在一個基類中寫一個公用的 `__init__()` 函數：

```
import math

class Structure1:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))
        # Set the arguments
```

```
        for name, value in zip(self._fields, args):
            setattr(self, name, value)
```

然後使你的類繼承自這個基類:

```
# Example class definitions
class Stock(Structure1):
    _fields = ['name', 'shares', 'price']

class Point(Structure1):
    _fields = ['x', 'y']

class Circle(Structure1):
    _fields = ['radius']

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

使用這些類的示例:

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "structure.py", line 6, in __init__
        raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

如果還想支持關鍵字參數，可以將關鍵字參數設置爲實例屬性:

```
class Structure2:
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.
↪join(kwargs)))
```

```
# Example use
if __name__ == '__main__':
    class Stock(Structure2):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
    # s3 = Stock('ACME', shares=50, price=91.1, aa=1)
```

你還能將不在 `_fields` 中的名稱加入到屬性中去：

```
class Structure3:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))

        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.
↪join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure3):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

討論

當你需要使用大量很小的數據結構類的時候，相比手工一個個定義 `__init__()` 方法而已，使用這種方式可以大大簡化代碼。

在上面的實現中我們使用了 `setattr()` 函數類設置屬性值，你可能不想用這種方式，而是想直接更新實例字典，就像下面這樣：

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments (alternate)
        self.__dict__.update(zip(self._fields,args))
```

儘管這也可以正常工作，但是當定義子類的時候問題就來了。當一個子類定義了 __slots__ 或者通過 property(或描述器) 來包裝某個屬性，那麼直接訪問實例字典就不起作用了。我們上面使用 setattr() 會顯得更通用些，因爲它也適用於子類情況。

這種方法唯一不好的地方就是對某些 IDE 而言，在顯示幫助函數時可能不太友好。比如：

```
>>> help(Stock)
Help on class Stock in module __main__:
class Stock(Structure)
...
| Methods inherited from Structure:
|
| __init__(self, *args, **kwargs)
|
...
>>>
```

可以參考 9.16 小節來強制在 __init__() 方法中指定參數的類型簽名。

8.12 定義接口或者抽象基類

問題

你想定義一個接口或抽象類，並且通過執行類型檢查來確保子類實現了某些特定的方法

解決方案

使用 abc 模塊可以很輕鬆的定義抽象基類：

```
from abc import ABCMeta, abstractmethod

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

```
    @abstractmethod
    def write(self, data):
        pass
```

抽象類的一個特點是它不能直接被實例化，比如你想像下面這樣做是不行的：

```
a = IStream() # TypeError: Can't instantiate abstract class
              # IStream with abstract methods read, write
```

抽象類的目的就是讓別的類繼承它並實現特定的抽象方法：

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

    def write(self, data):
        pass
```

抽象基類的一個主要用途是在代碼中檢查某些類是否爲特定類型，實現了特定接口：

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    pass
```

除了繼承這種方式外，還可以通過註冊方式來讓某個類實現抽象基類：

```
import io

# Register the built-in I/O classes as supporting our interface
IStream.register(io.IOBase)

# Open a normal file and type check
f = open('foo.txt')
isinstance(f, IStream) # Returns True
```

@abstractmethod 還能註解靜態方法、類方法和 properties 。你只需保證這個註解緊靠在函數定義前即可：

```
class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass
```

```
    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

討論

標準庫中有很多用到抽象基類的地方。collections 模塊定義了很多跟容器和迭代器 (序列、映射、集合等) 有關的抽象基類。numbers 庫定義了跟數字對象 (整數、浮點數、有理數等) 有關的基類。io 庫定義了很多跟 I/O 操作相關的基類。

你可以使用預定義的抽象類來執行更通用的類型檢查，例如：

```
import collections

# Check if x is a sequence
if isinstance(x, collections.Sequence):
...

# Check if x is iterable
if isinstance(x, collections.Iterable):
...

# Check if x has a size
if isinstance(x, collections.Sized):
...

# Check if x is a mapping
if isinstance(x, collections.Mapping):
```

儘管 ABCs 可以讓我們很方便的做類型檢查，但是我們在代碼中最好不要過多的使用它。因爲 Python 的本質是一門動態編程語言，其目的就是給你更多靈活性，強制類型檢查或讓你代碼變得更復雜，這樣做無異於捨本求末。

8.13 實現數據模型的類型約束

問題

你想定義某些在屬性賦值上面有限制的數據結構。

解決方案

在這個問題中，你需要在對某些實例屬性賦值時進行檢查。所以你要自定義屬性賦值函數，這種情況下最好使用描述器。

下面的代碼使用描述器實現了一個系統類型和賦值驗證框架：

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value


# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)


# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)


class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super().__set__(instance, value)
```

這些類就是你要創建的數據模型或類型系統的基礎構建模塊。下面就是我們實際定義的各種不同的數據類型：

```
class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass

class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass
```

然後使用這些自定義數據類型，我們定義一個類：

```
class Stock:
    # Specify constraints
    name = SizedString('name', size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')

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

然後測試這個類的屬性賦值約束，可發現對某些屬性的賦值違法了約束是不合法的：

```
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 23, in __set__
        raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in __set__
        raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
```

```
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 35, in __set__
        raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>
```

還有一些技術可以簡化上面的代碼，其中一種是使用類裝飾器：

```
# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls

    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

另外一種方式是使用元類：

```
# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

# Example
class Stock2(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()
```

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

討論

本節使用了很多高級技術，包括描述器、混入類、super() 的使用、類裝飾器和元類。不可能在這裏一一詳細展開來講，但是可以在 8.9、8.18、9.19 小節找到更多例子。但是，我在這裏還是要提一下幾個需要注意的點。

首先，在 Descriptor 基類中你會看到有個 __set__() 方法，卻沒有相應的 __get__() 方法。如果一個描述僅僅是從底層實例字典中獲取某個屬性值的話，那麼沒必要去定義 __get__() 方法。

所有描述器類都是基於混入類來實現的。比如 Unsigned 和 MaxSized 要跟其他繼承自 Typed 類混入。這裏利用多繼承來實現相應的功能。

混入類的一個比較難理解的地方是，調用 super() 函數時，你並不知道究竟要調用哪個具體類。你需要跟其他類結合後才能正確的使用，也就是必須合作才能產生效果。

使用類裝飾器和元類通常可以簡化代碼。上面兩個例子中你會發現你只需要輸入一次屬性名即可了。

```
# Normal
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

所有方法中，類裝飾器方案應該是最靈活和最高明的。首先，它並不依賴任何其他新的技術，比如元類。其次，裝飾器可以很容易的添加或刪除。

最後，裝飾器還能作爲混入類的替代技術來實現同樣的效果;

```
# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)
    super_set = cls.__set__

    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
        super_set(self, instance, value)
```

```
    cls.__set__ = __set__
    return cls


# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__

    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)

    cls.__init__ = __init__

    super_set = cls.__set__

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass


@Unsigned
class UnsignedInteger(Integer):
    pass
```

```
@Typed(float)
class Float(Descriptor):
    pass


@Unsigned
class UnsignedFloat(Float):
    pass


@Typed(str)
class String(Descriptor):
    pass


@MaxSized
class SizedString(String):
    pass
```

這種方式定義的類跟之前的效果一樣，而且執行速度會更快。設置一個簡單的類型屬性的值，裝飾器方式要比之前的混入類的方式幾乎快 100%。現在你應該慶幸自己讀完了本節全部內容了吧？^_^

8.14 實現自定義容器

問題

你想實現一個自定義的類來模擬內置的容器類功能，比如列表和字典。但是你不確定到底要實現哪些方法。

解決方案

collections 定義了很多抽象基類，當你想自定義容器類的時候它們會非常有用。比如你想讓你的類支持迭代，那就讓你的類繼承 collections.Iterable 即可：

```
import collections
class A(collections.Iterable):
    pass
```

不過你需要實現 collections.Iterable 所有的抽象方法，否則會報錯:

```
>>> a = A()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

你只要實現 `__iter__()` 方法就不會報錯了 (參考 4.2 和 4.7 小節)。

你可以先試着去實例化一個對象，在錯誤提示中可以找到需要實現哪些方法：

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
__getitem__, __len__
>>>
```

下面是一個簡單的示例，繼承自上面 Sequence 抽象類，並且實現元素按照順序存儲：

```
class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)


items = SortedItems([5, 1, 3])
print(list(items))
print(items[0], items[-1])
items.add(2)
print(list(items))
```

可以看到，SortedItems 跟普通的序列沒什麼兩樣，支持所有常用操作，包括索引、迭代、包含判斷，甚至是切片操作。

這裏面使用到了 `bisect` 模塊，它是一個在排序列表中插入元素的高效方式。可以保證元素插入後還保持順序。

討論

使用 `collections` 中的抽象基類可以確保你自定義的容器實現了所有必要的方法。並且還能簡化類型檢查。你的自定義容器會滿足大部分類型檢查需要，如下所示：

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
```

```
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>
```

collections 中很多抽象類會爲一些常見容器操作提供默認的實現，這樣一來你只需要實現那些你最感興趣的方法即可。假設你的類繼承自 collections.MutableSequence，如下：

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

如果你創建 Items 的實例，你會發現它支持幾乎所有的核心列表方法 (如 append()、remove()、count() 等)。下面是使用演示：

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
```

```
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

本小節只是對 Python 抽象類功能的拋磚引玉。numbers 模塊提供了一個類似的跟整數類型相關的抽象類型集合。可以參考 8.12 小節來構造更多自定義抽象基類。

8.15 屬性的代理訪問

問題

你想將某個實例的屬性訪問代理到內部另一個實例中去，目的可能是作爲繼承的一個替代方法或者實現代理模式。

解決方案

簡單來說，代理是一種編程模式，它將某個操作轉移給另外一個對象來實現。最簡單的形式可能是像下面這樣：

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass


class B1:
    """ 簡單的代理"""

    def __init__(self):
        self._a = A()

    def spam(self, x):
```

```
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass
```

如果僅僅就兩個方法需要代理，那麼像這樣寫就足夠了。但是，如果有大量的方法需要代理，那麼使用 __getattr__() 方法或許或更好些：

```
class B2:
    """ 使用__getattr__ 的代理，代理方法比較多時候"""

    def __init__(self):
        self._a = A()

    def bar(self):
        pass

    # Expose all of the methods defined on class A
    def __getattr__(self, name):
        """ 這個方法在訪問的 attribute 不存在的時候被調用
        the __getattr__() method is actually a fallback method
        that only gets called when an attribute is not found"""
        return getattr(self._a, name)
```

__getattr__ 方法是在訪問 attribute 不存在的時候被調用，使用演示：

```
b = B()
b.bar() # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam
```

另外一個代理例子是實現代理模式，例如：

```
# A proxy class that wraps around another object, but
# exposes its public attributes
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
```

```
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

使用這個代理類時，你只需要用它來包裝下其他類即可：

```
class Spam:
    def __init__(self, x):
        self.x = x

    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Create an instance
s = Spam(2)
# Create a proxy around it
p = Proxy(s)
# Access the proxy
print(p.x)  # Outputs 2
p.bar(3)  # Outputs "Spam.bar: 2 3"
p.x = 37  # Changes s.x to 37
```

通過自定義屬性訪問方法，你可以用不同方式自定義代理類行爲 (比如加入日誌功能、只讀訪問等)。

討論

代理類有時候可以作爲繼承的替代方案。例如，一個簡單的繼承如下：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)
    def bar(self):
        print('B.bar')
```

使用代理的話，就是下面這樣：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B:
    def __init__(self):
        self._a = A()
    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)
    def bar(self):
        print('B.bar')
    def __getattr__(self, name):
        return getattr(self._a, name)
```

當實現代理模式時，還有些細節需要注意。首先，__getattr__() 實際是一個後備方法，只有在屬性不存在時纔會調用。因此，如果代理類實例本身有這個屬性的話，那麼不會觸發這個方法的。另外，__setattr__() 和 __delattr__() 需要額外的魔法來區分代理實例和被代理實例 _obj 的屬性。一個通常的約定是隻代理那些不以下劃線 _ 開頭的屬性 (代理類只暴露被代理類的公共屬性)。

還有一點需要注意的是，__getattr__() 對於大部分以雙下劃線 (___) 開始和結尾的屬性並不適用。比如，考慮如下的類：

```
class ListLike:
    """__getattr__ 對於雙下劃線開始和結尾的方法是不能用的，需要一個個去重定義"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)
```

如果是創建一個 ListLike 對象，會發現它支持普通的列表方法，如 append() 和 insert()，但是卻不支持 len()、元素查找等。例如：

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
```

```
    File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

爲了讓它支持這些方法，你必須手動的實現這些方法代理：

```
class ListLike:
    """__getattr__ 對於雙下劃線開始和結尾的方法是不能用的，需要一個個去重定義"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __delitem__(self, index):
        del self._items[index]
```

11.8 小節還有一個在遠程方法調用環境中使用代理的例子。

8.16 在類中定義多個構造器

問題

你想實現一個類，除了使用 `__init__()` 方法外，還有其他方式可以初始化它。

解決方案

爲了實現多個構造器，你需要使用到類方法。例如：

```
import time

class Date:
    """ 方法一：使用類方法"""
    # Primary constructor
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
```

```
        self.day = day

    # Alternate constructor
    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

直接調用類方法即可，下面是使用示例：

```
a = Date(2012, 12, 21) # Primary
b = Date.today() # Alternate
```

討論

類方法的一個主要用途就是定義多個構造器。它接受一個 class 作爲第一個參數 (cls)。你應該注意到了這個類被用來創建並返回最終的實例。在繼承時也能工作的很好：

```
class NewDate(Date):
    pass

c = Date.today() # Creates an instance of Date (cls=Date)
d = NewDate.today() # Creates an instance of NewDate (cls=NewDate)
```

8.17 創建不調用 init 方法的實例

問題

你想創建一個實例，但是希望繞過執行 __init__() 方法。

解決方案

可以通過 __new__() 方法創建一個未初始化的實例。例如考慮如下這個類：

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

下面演示如何不調用 __init__() 方法來創建這個 Date 實例：

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
```

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

結果可以看到，這個 Date 實例的屬性 year 還不存在，所以你需要手動初始化：

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

討論

當我們在反序列對象或者實現某個類方法構造函數時需要繞過 `__init__()` 方法來創建對象。例如，對於上面的 Date 來講，有時候你可能會像下面這樣定義一個新的構造函數 `today()`：

```
from time import localtime

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

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d
```

同樣，在你反序列化 JSON 數據時產生一個如下的字典對象：

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

如果你想將它轉換成一個 Date 類型實例，可以使用上面的技術。

當你通過這種非常規方式來創建實例的時候，最好不要直接去訪問底層實例字典，除非你真的清楚所有細節。否則的話，如果這個類使用了 `__slots__`、properties、descriptors 或其他高級技術的時候代碼就會失效。而這時候使用 `setattr()` 方法會讓你的代碼變得更加通用。

8.18 利用 Mixins 擴展類功能

問題

你有很多有用的方法，想使用它們來擴展其他類的功能。但是這些類並沒有任何繼承的關係。因此你不能簡單的將這些方法放入一個基類，然後被其他類繼承。

解決方案

通常當你想自定義類的時候會碰上這些問題。可能是某個庫提供了一些基礎類，你可以利用它們來構造你自己的類。

假設你想擴展映射對象，給它們添加日誌、唯一性設置、類型檢查等等功能。下面是一些混入類：

```
class LoggedMappingMixin:
    """
    Add logging to get/set/delete operations for debugging.
    """
    __slots__ = ()  # 混入類都沒有實例變量，因爲直接實例化混入類沒有任何意義

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)


class SetOnceMappingMixin:
    '''
    Only allow a key to be set once.
    '''
    __slots__ = ()

    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)


class StringKeysMappingMixin:
    '''
    Restrict keys to strings only
    '''
```

```
    __slots__ = ()

    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)
```

這些類單獨使用起來沒有任何意義，事實上如果你去實例化任何一個類，除了產生異常外沒任何作用。它們是用來通過多繼承來和其他映射對象混入使用的。例如：

```
class LoggedDict(LoggedMappingMixin, dict):
    pass

d = LoggedDict()
d['x'] = 23
print(d['x'])
del d['x']

from collections import defaultdict

class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
    pass


d = SetOnceDefaultDict(list)
d['x'].append(2)
d['x'].append(3)
# d['x'] = 23  # KeyError: 'x already set'
```

這個例子中，可以看到混入類跟其他已存在的類 (比如 dict、defaultdict 和 OrderedDict) 結合起來使用，一個接一個。結合後就能發揮正常功效了。

討論

混入類在標準庫中很多地方都出現過，通常都是用來像上面那樣擴展某些類的功能。它們也是多繼承的一個主要用途。比如，當你編寫網絡代碼時候，你會經常使用 socketserver 模塊中的 ThreadingMixIn 來給其他網絡相關類增加多線程支持。例如，下面是一個多線程的 XML-RPC 服務：

```
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

同時在一些大型庫和框架中也會發現混入類的使用，用途同樣是增強已存在的類的功能和一些可選特徵。

對於混入類，有幾點需要記住。首先是，混入類不能直接被實例化使用。其次，混入類沒有自己的狀態信息，也就是說它們並沒有定義 `__init__()` 方法，並且沒有實例屬性。這也是爲什麼我們在上面明確定義了 `__slots__ = ()`。

還有一種實現混入類的方式就是使用類裝飾器，如下所示：

```
def LoggedMapping(cls):
    """ 第二種方式: 使用類裝飾器"""
    cls_getitem = cls.__getitem__
    cls_setitem = cls.__setitem__
    cls_delitem = cls.__delitem__

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return cls_getitem(self, key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return cls_setitem(self, key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return cls_delitem(self, key)

    cls.__getitem__ = __getitem__
    cls.__setitem__ = __setitem__
    cls.__delitem__ = __delitem__
    return cls


@LoggedMapping
class LoggedDict(dict):
    pass
```

這個效果跟之前的是一樣的，而且不再需要使用多繼承了。參考 9.12 小節獲取更多類裝飾器的信息，參考 8.13 小節查看更多混入類和類裝飾器的例子。

8.19 實現狀態對象或者狀態機

問題

你想實現一個狀態機或者是在不同狀態下執行操作的對象，但是又不想在代碼中出現太多的條件判斷語句。

解決方案

在很多程序中，有些對象會根據狀態的不同來執行不同的操作。比如考慮如下的一個連接對象：

```
class Connection:
    """ 普通方案，好多個判斷語句，效率低下~~"""
```

```
    def __init__(self):
        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')

    def write(self, data):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('writing')

    def open(self):
        if self.state == 'OPEN':
            raise RuntimeError('Already open')
        self.state = 'OPEN'

    def close(self):
        if self.state == 'CLOSED':
            raise RuntimeError('Already closed')
        self.state = 'CLOSED'
```

這樣寫有很多缺點，首先是代碼太複雜了，好多的條件判斷。其次是執行效率變低，因爲一些常見的操作比如 read()、write() 每次執行前都需要執行檢查。

一個更好的辦法是爲每個狀態定義一個對象：

```
class Connection1:
    """ 新方案—對每個狀態定義一個類"""

    def __init__(self):
        self.new_state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate
        # Delegate to the state class

    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)
```

```
# Connection state base class
class ConnectionState:
    @staticmethod
    def read(conn):
        raise NotImplementedError()

    @staticmethod
    def write(conn, data):
        raise NotImplementedError()

    @staticmethod
    def open(conn):
        raise NotImplementedError()

    @staticmethod
    def close(conn):
        raise NotImplementedError()


# Implementation of different states
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')


class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')
```

```
    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)
```

下面是使用演示：

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 10, in read
        return self._state.read(self)
    File "example.py", line 43, in read
        raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

討論

如果代碼中出現太多的條件判斷語句的話，代碼就會變得難以維護和閱讀。這裏的解決方案是將每個狀態抽取出來定義成一個類。

這裏看上去有點奇怪，每個狀態對象都只有靜態方法，並沒有存儲任何的實例屬性數據。實際上，所有狀態信息都只存儲在 Connection 實例中。在基類中定義的 NotImplementedError 是爲了確保子類實現了相應的方法。這裏你或許還想使用 8.12 小節講解的抽象基類方式。

設計模式中有一種模式叫狀態模式，這一小節算是一個初步入門！

8.20 通過字符串調用對象方法

問題

你有一個字符串形式的方法名稱，想通過它調用某個對象的對應方法。

解決方案

最簡單的情況，可以使用 getattr() ：

```
import math

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

    def __repr__(self):
        return 'Point({!r:},{!r:})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)


p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)  # Calls p.distance(0, 0)
```

另外一種方法是使用 operator.methodcaller() ，例如：

```
import operator
operator.methodcaller('distance', 0, 0)(p)
```

當你需要通過相同的參數多次調用某個方法時，使用 operator.methodcaller 就很方便了。比如你需要排序一系列的點，就可以這樣做：

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]
# Sort by distance from origin (0, 0)
points.sort(key=operator.methodcaller('distance', 0, 0))
```

討論

調用一個方法實際上是兩部獨立操作，第一步是查找屬性，第二步是函數調用。因此，爲了調用某個方法，你可以首先通過 getattr() 來查找到這個屬性，然後再去以函數方式調用它即可。

operator.methodcaller() 創建一個可調用對象，並同時提供所有必要參數，然後調用的時候只需要將實例對象傳遞給它即可，比如：

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
>>>
```

通過方法名稱字符串來調用方法通常出現在需要模擬 case 語句或實現訪問者模式的時候。參考下一小節獲取更多高級例子。

8.21 實現訪問者模式

問題

你要處理由大量不同類型的對象組成的複雜數據結構，每一個對象都需要需要進行不同的處理。比如，遍歷一個樹形結構，然後根據每個節點的相應狀態執行不同的操作。

解決方案

這裏遇到的問題在編程領域中是很普遍的，有時候會構建一個由大量不同對象組成的數據結構。假設你要寫一個表示數學表達式的程序，那麼你可能需要定義如下的類：

```
class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
```

```
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value
```

然後利用這些類構建嵌套數據結構，如下所示：

```
# Representation of 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)
```

這樣做的問題是對於每個表達式，每次都要重新定義一遍，有沒有一種更通用的方式讓它支持所有的數字和操作符呢。這裏我們使用訪問者模式可以達到這樣的目的：

```
class NodeVisitor:
    def visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name_
↪_))
```

爲了使用這個類，可以定義一個類繼承它並且實現各種 `visit_Name()` 方法，其中 Name 是 node 類型。例如，如果你想求表達式的值，可以這樣寫：

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -node.operand
```

使用示例：

```
>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>
```

作爲一個不同的例子，下面定義一個類在一個棧上面將一個表達式轉換成多個操作序列：

```
class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

使用示例：

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>
```

討論

剛開始的時候你可能會寫大量的 if/else 語句來實現，這裏訪問者模式的好處就是通過 getattr() 來獲取相應的方法，並利用遞歸來遍歷所有的節點：

```
def binop(self, node, instruction):
    self.visit(node.left)
    self.visit(node.right)
    self.instructions.append((instruction,))
```

還有一點需要指出的是，這種技術也是實現其他語言中 switch 或 case 語句的方式。比如，如果你正在寫一個 HTTP 框架，你可能會寫這樣一個請求分發的控制器：

```
class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)
    def do_GET(self, request):
        pass
    def do_POST(self, request):
        pass
    def do_HEAD(self, request):
        pass
```

訪問者模式一個缺點就是它嚴重依賴遞歸，如果數據結構嵌套層次太深可能會有問題，有時候會超過 Python 的遞歸深度限制 (參考 sys.getrecursionlimit())。

可以參照 8.22 小節，利用生成器或迭代器來實現非遞歸遍歷算法。

在跟解析和編譯相關的編程中使用訪問者模式是非常常見的。Python 本身的 ast 模塊值的關注下，可以去看看源碼。9.24 小節演示了一個利用 ast 模塊來處理 Python 源代碼的例子。

8.22 不用遞歸實現訪問者模式

問題

你使用訪問者模式遍歷一個很深的嵌套樹形數據結構，並且因爲超過嵌套層級限制而失敗。你想消除遞歸，並同時保持訪問者編程模式。

解決方案

通過巧妙的使用生成器可以在樹遍歷或搜索算法中消除遞歸。在 8.21 小節中，我們給出了一個訪問者類。下面我們利用一個棧和生成器重新實現這個類：

```
import types

class Node:
    pass
```

```
class NodeVisitor:
    def visit(self, node):
        stack = [node]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()

        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name_
↪_))
```

如果你使用這個類，也能達到相同的效果。事實上你完全可以將它作爲上一節中的訪問者模式的替代實現。考慮如下代碼，遍歷一個表達式的樹：

```
class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
```

```
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)
    # Evaluate it
    e = Evaluator()
    print(e.visit(t4))  # Outputs 0.6
```

如果嵌套層次太深那麼上述的 Evaluator 就會失效：

```
>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
```

```
Traceback (most recent call last):
...
    File "visitor.py", line 29, in _visit
return meth(node)
    File "visitor.py", line 67, in visit_Add
return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>
```

現在我們稍微修改下上面的 Evaluator：

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)

    def visit_Mul(self, node):
        yield (yield node.left) * (yield node.right)

    def visit_Div(self, node):
        yield (yield node.left) / (yield node.right)

    def visit_Negate(self, node):
        yield - (yield node.operand)
```

再次運行，就不會報錯了：

```
>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>
```

如果你還想添加其他自定義邏輯也沒問題：

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
```

```
        yield lhs + rhs
    ...
```

下面是簡單的測試：

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

討論

這一小節我們演示了生成器和協程在程序控制流方面的強大功能。避免遞歸的一個通常方法是使用一個棧或隊列的數據結構。例如，深度優先的遍歷算法，第一次碰到一個節點時將其壓入棧中，處理完後彈出棧。visit() 方法的核心思路就是這樣。

另外一個需要理解的就是生成器中 yield 語句。當碰到 yield 語句時，生成器會返回一個數據並暫時掛起。上面的例子使用這個技術來代替了遞歸。例如，之前我們是這樣寫遞歸：

```
value = self.visit(node.left)
```

現在換成 yield 語句：

```
value = yield node.left
```

它會將 node.left 返回給 visit() 方法，然後 visit() 方法調用那個節點相應的 visit_Name() 方法。yield 暫時將程序控制器讓出給調用者，當執行完後，結果會賦值給 value，

看完這一小節，你也許想去尋找其它沒有 yield 語句的方案。但是這麼做沒有必要，你必須處理很多棘手的問題。例如，爲了消除遞歸，你必須要維護一個棧結構，如果不使用生成器，代碼會變得很臃腫，到處都是棧操作語句、回調函數等。實際上，使用 yield 語句可以讓你寫出非常漂亮的代碼，它消除了遞歸但是看上去又很像遞歸實現，代碼很簡潔。

8.23 循環引用數據結構的內存管理

問題

你的程序創建了很多循環引用數據結構 (比如樹、圖、觀察者模式等)，你碰到了內存管理難題。

解決方案

一個簡單的循環引用數據結構例子就是一個樹形結構，雙親節點有指針指向孩子節點，孩子節點又返回來指向雙親節點。這種情況下，可以考慮使用 `weakref` 庫中的弱引用。例如：

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return None if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

這種是想方式允許 parent 靜默終止。例如：

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

討論

循環引用的數據結構在 Python 中是一個很棘手的問題，因爲正常的垃圾回收機制不能適用於這種情形。例如考慮如下代碼：

```
# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
```

```
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

下面我們使用這個代碼來做一些垃圾回收試驗：

```
>>> a = Data()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a # Not deleted (no message)
>>>
```

可以看到，最後一個的刪除時打印語句沒有出現。原因是 Python 的垃圾回收機制是基於簡單的引用計數。當一個對象的引用數變成 0 的時候纔會立即刪除掉。而對於循環引用這個條件永遠不會成立。因此，在上面例子中最後部分，父節點和孩子節點互相擁有對方的引用，導致每個對象的引用計數都不可能變成 0。

Python 有另外的垃圾回收器來專門針對循環引用的，但是你永遠不知道它什麼時候會觸發。另外你還可以手動的觸發它，但是代碼看上去很挫：

```
>>> import gc
>>> gc.collect() # Force collection
Data.__del__
Data.__del__
>>>
```

如果循環引用的對象自己還定義了自己的 `__del__()` 方法，那麼會讓情況變得更糟糕。假設你像下面這樣給 Node 定義自己的 `__del__()` 方法：

```
# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
```

```
        self.children.append(child)
        child.parent = self

    # NEVER DEFINE LIKE THIS.
    # Only here to illustrate pathological behavior
    def __del__(self):
        del self.data
        del.parent
        del.children
```

這種情況下，垃圾回收永遠都不會去回收這個對象的，還會導致內存泄露。如果你試着去運行它會發現，Data.__del__ 消息永遠不會出現了，甚至在你強制內存回收時：

```
>>> a = Node()
>>> a.add_child(Node()
>>> del a # No message (not collected)
>>> import gc
>>> gc.collect() # No message (not collected)
>>>
```

弱引用消除了引用循環的這個問題，本質來講，弱引用就是一個對象指針，它不會增加它的引用計數。你可以通過 weakref 來創建弱引用。例如：

```
>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>
```

爲了訪問弱引用所引用的對象，你可以像函數一樣去調用它即可。如果那個對象還存在就會返回它，否則就返回一個 None。由於原始對象的引用計數沒有增加，那麼就可以去刪除它了。例如;

```
>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
>>> print(a_ref())
None
>>>
```

通過這裏演示的弱引用技術，你會發現不再有循環引用問題了，一旦某個節點不被使用了，垃圾回收器立即回收它。你還能參考 8.25 小節關於弱引用的另外一個例子。

8.24 讓類支持比較操作

問題

你想讓某個類的實例支持標準的比較運算 (比如 >=,!=,<=,< 等)，但是又不想去實現那一大丟的特殊方法。

解決方案

Python 類對每個比較操作都需要實現一個特殊方法來支持。例如爲了支持 >= 操作符，你需要定義一個 __ge__() 方法。儘管定義一個方法沒什麼問題，但如果要你實現所有可能的比較方法那就有點煩人了。

裝飾器 functools.total_ordering 就是用來簡化這個處理的。使用它來裝飾一個來，你只需定義一個 __eq__() 方法，外加其他方法 (__lt__, __le__, __gt__, or __ge__) 中的一個即可。然後裝飾器會自動爲你填充其它比較方法。

作爲例子，我們構建一些房子，然後給它們增加一些房間，最後通過房子大小來比較它們：

```
from functools import total_ordering

class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
                self.living_space_footage,
                self.style)

    def __eq__(self, other):
        return self.living_space_footage == other.living_space_footage

    def __lt__(self, other):
```

```
        return self.living_space_footage < other.living_space_footage
```

這裏我們只是給 House 類定義了兩個方法：__eq__() 和 __lt__()，它就能支持所有的比較操作：

```
# Build a few houses, and add rooms to them
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))
h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))
h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]
print('Is h1 bigger than h2?', h1 > h2) # prints True
print('Is h2 smaller than h3?', h2 < h3) # prints True
print('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False
print('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-foot␣
↪Split'
print('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot Ranch'
```

討論

其實 total_ordering 裝飾器也沒那麼神祕。它就是定義了一個從每個比較支持方法到所有需要定義的其他方法的一個映射而已。比如你定義了 __le__() 方法，那麼它就被用來構建所有其他的需要定義的那些特殊方法。實際上就是在類裏面像下面這樣定義了一些特殊方法：

```
class House:
    def __eq__(self, other):
        pass
    def __lt__(self, other):
        pass
    # Methods created by @total_ordering
    __le__ = lambda self, other: self < other or self == other
    __gt__ = lambda self, other: not (self < other or self == other)
    __ge__ = lambda self, other: not (self < other)
    __ne__ = lambda self, other: not self == other
```

當然，你自己去寫也很容易，但是使用 @total_ordering 可以簡化代碼，何樂而不爲呢。

8.25 創建緩存實例

問題

在創建一個類的對象時，如果之前使用同樣參數創建過這個對象，你想返回它的緩存引用。

解決方案

這種通常是因爲你希望相同參數創建的對象時單例的。在很多庫中都有實際的例子，比如 `logging` 模塊，使用相同的名稱創建的 `logger` 實例永遠只有一個。例如：

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

爲了達到這樣的效果，你需要使用一個和類本身分開的工廠函數，例如：

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()
def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

然後做一個測試，你會發現跟之前那個日誌對象的創建行爲是一致的：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>
```

討論

編寫一個工廠函數來修改普通的實例創建行爲通常是一個比較簡單的方法。但是我們還能否找到更優雅的解決方案呢？

例如，你可能會考慮重新定義類的 `__new__()` 方法，就像下面這樣：

```
# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self
    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

初看起來好像可以達到預期效果，但是問題是 `__init__()` 每次都會被調用，不管這個實例是否被緩存了。例如：

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

這個或許不是你想要的效果，因此這種方法並不可取。

上面我們使用到了弱引用計數，對於垃圾回收來講是很有幫助的，關於這個我們在 8.23 小節已經講過了。當我們保持實例緩存時，你可能只想在程序中使用到它們時才保存。一個 `WeakValueDictionary` 實例只會保存那些在其它地方還在被使用的實例。否則的話，只要實例不再被使用了，它就從字典中被移除了。觀察下下面的測試結果：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
```

```
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

對於大部分程序而已，這裏代碼已經夠用了。不過還是有一些更高級的實現值得了解下。

首先是這裏使用到了一個全局變量，並且工廠函數跟類放在一塊。我們可以通過將緩存代碼放到一個單獨的緩存管理器中：

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
            self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

    def get_spam(name):
        return Spam.manager.get_spam(name)
```

這樣的話代碼更清晰，並且也更靈活，我們可以增加更多的緩存管理機制，只需要替代 manager 即可。

還有一點就是，我們暴露了類的實例化給用戶，用戶很容易去直接實例化這個類，而不是使用工廠方法，如：

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>
```

有幾種方式可以防止用戶這樣做，第一個是將類的名字修改爲以下劃線 (_) 開頭，提示用戶別直接調用它。第二種就是讓這個類的 __init__() 方法拋出一個異常，讓它

不能被初始化：

```
class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

然後修改緩存管理器代碼，使用 Spam._new() 來創建實例，而不是直接調用 Spam() 構造函數：

```
# -----------------------最後的修正方案-----------------------
class CachedSpamManager2:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            temp = Spam3._new(name)  # Modified creation
            self._cache[name] = temp
        else:
            temp = self._cache[name]
        return temp

    def clear(self):
            self._cache.clear()

class Spam3:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
        return self
```

最後這樣的方案就已經足夠好了。緩存和其他構造模式還可以使用 9.13 小節中的元類實現的更優雅一點 (使用了更高級的技術)。

第九章：元編程

軟件開發領域中最經典的口頭禪就是“don’t repeat yourself”。也就是說，任何時候當你的程序中存在高度重複 (或者是通過剪切複製) 的代碼時，都應該想想是否有更好的解決方案。在 Python 當中，通常都可以通過元編程來解決這類問題。簡而言之，元編程就是關於創建操作源代碼 (比如修改、生成或包裝原來的代碼) 的函數和類。主要技術是使用裝飾器、類裝飾器和元類。不過還有一些其他技術，包括簽名對象、使用 `exec()` 執行代碼以及對內部函數和類的反射技術等。本章的主要目的是向大家介紹這些元編程技術，並且給出實例來演示它們是怎樣定製化你的源代碼行爲的。

9.1 在函數上添加包裝器

問題

你想在函數上添加一個包裝器，增加額外的操作處理 (比如日誌、計時等)。

解決方案

如果你想使用額外的代碼包裝一個函數，可以定義一個裝飾器函數，例如：

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面是使用裝飾器的例子：

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
```

```
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

討論

一個裝飾器就是一個函數，它接受一個函數作爲參數並返回一個新的函數。當你像下面這樣寫：

```
@timethis
def countdown(n):
    pass
```

跟像下面這樣寫其實效果是一樣的：

```
def countdown(n):
    pass
countdown = timethis(countdown)
```

順便說一下，內置的裝飾器比如 `@staticmethod`, `@classmethod`,`@property` 原理也是一樣的。例如，下面這兩個代碼片段是等價的：

```
class A:
    @classmethod
    def method(cls):
        pass

class B:
    # Equivalent definition of a class method
    def method(cls):
        pass
    method = classmethod(method)
```

在上面的 `wrapper()` 函數中，裝飾器內部定義了一個使用 `*args` 和 `**kwargs` 來接受任意參數的函數。在這個函數裏面調用了原始函數並將其結果返回，不過你還可以添加其他額外的代碼 (比如計時)。然後這個新的函數包裝器被作爲結果返回來代替原始函數。

需要強調的是裝飾器並不會修改原始函數的參數簽名以及返回值。使用 `*args` 和 `**kwargs` 目的就是確保任何參數都能適用。而返回結果值基本都是調用原始函數 `func(*args, **kwargs)` 的返回結果，其中 func 就是原始函數。

剛開始學習裝飾器的時候，會使用一些簡單的例子來說明，比如上面演示的這個。不過實際場景使用時，還是有一些細節問題要注意的。比如上面使用 `@wraps(func)` 註解是很重要的，它能保留原始函數的元數據 (下一小節會講到)，新手經常會忽略這個細節。接下來的幾個小節我們會更加深入的講解裝飾器函數的細節問題，如果你想構造你自己的裝飾器函數，需要認真看一下。

9.2 創建裝飾器時保留函數元信息

問題

你寫了一個裝飾器作用在某個函數上，但是這個函數的重要的元信息比如名字、文檔字符串、註解和參數簽名都丟失了。

解決方案

任何時候你定義裝飾器的時候，都應該使用 `functools` 庫中的 `@wraps` 裝飾器來註解底層包裝函數。例如：

```
import time
from functools import wraps
def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

下面我們使用這個被包裝後的函數並檢查它的元信息：

```
>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

討論

在編寫裝飾器的時候複製元信息是一個非常重要的部分。如果你忘記了使用 @wraps，那麼你會發現被裝飾函數丟失了所有有用的信息。比如如果忽略 @wraps 後的效果是下面這樣的：

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

@wraps 有一個重要特徵是它能讓你通過屬性 __wrapped__ 直接訪問被包裝函數。例如:

```
>>> countdown.__wrapped__(100000)
>>>
```

__wrapped__ 屬性還能讓被裝飾函數正確暴露底層的參數簽名信息。例如：

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
>>>
```

一個很普遍的問題是怎樣讓裝飾器去直接複製原始函數的參數簽名信息，如果想自己手動實現的話需要做大量的工作，最好就簡單的使用 @wraps 裝飾器。通過底層的 __wrapped__ 屬性訪問到函數簽名信息。更多關於簽名的內容可以參考 9.16 小節。

9.3 解除一個裝飾器

問題

一個裝飾器已經作用在一個函數上，你想撤銷它，直接訪問原始的未包裝的那個函數。

解決方案

假設裝飾器是通過 @wraps (參考 9.2 小節) 來實現的，那麼你可以通過訪問 __wrapped__ 屬性來訪問原始函數：

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
```

```
7
>>>
```

討論

直接訪問未包裝的原始函數在調試、內省和其他函數操作時是很有用的。但是我們這裏的方案僅僅適用於在包裝器中正確使用了 `@wraps` 或者直接設置了 `__wrapped__` 屬性的情況。

如果有多個包裝器，那麼訪問 `__wrapped__` 屬性的行爲是不可預知的，應該避免這樣做。在 Python3.3 中，它會略過所有的包裝層，比如，假如你有如下的代碼：

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y
```

下面我們在 Python3.3 下測試：

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>
```

下面我們在 Python3.4 下測試：

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
```

```
>>> add.__wrapped__(2, 3)
Decorator 2
5
>>>
```

最後要說的是，並不是所有的裝飾器都使用了 `@wraps` ，因此這裏的方案並不全部適用。特別的，內置的裝飾器 `@staticmethod` 和 `@classmethod` 就沒有遵循這個約定(它們把原始函數存儲在屬性 `__func__` 中)。

9.4 定義一個帶參數的裝飾器

問題

你想定義一個可以接受參數的裝飾器

解決方案

我們用一個例子詳細闡述下接受參數的處理過程。假設你想寫一個裝飾器，給函數添加日誌功能，同時允許用戶指定日誌的級別和其他的選項。下面是這個裝飾器的定義和使用示例：

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y
```

```
@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

初看起來，這種實現看上去很複雜，但是核心思想很簡單。最外層的函數 logged() 接受參數並將它們作用在內部的裝飾器函數上面。內層的函數 decorate() 接受一個函數作爲參數，然後在函數上面放置一個包裝器。這裏的關鍵點是包裝器是可以使用傳遞給 logged() 的參數的。

討論

定義一個接受參數的包裝器看上去比較複雜主要是因爲底層的調用序列。特別的，如果你有下面這個代碼：

```
@decorator(x, y, z)
def func(a, b):
    pass
```

裝飾器處理過程跟下面的調用是等效的;

```
def func(a, b):
    pass
func = decorator(x, y, z)(func)
```

decorator(x, y, z) 的返回結果必須是一個可調用對象，它接受一個函數作爲參數并包裝它，可以參考 9.7 小節中另外一個可接受參數的包裝器例子。

9.5 可自定義屬性的裝飾器

問題

你想寫一個裝飾器來包裝一個函數，並且允許用戶提供參數在運行時控制裝飾器行爲。

解決方案

引入一個訪問函數，使用 nonlocal 來修改內部變量。然後這個訪問函數被作爲一個屬性賦值給包裝函數。

```
from functools import wraps, partial
import logging
# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func
```

```
def logged(level, name=None, message=None):
    '''
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    '''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper

    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

下面是交互環境下的使用例子：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
5
>>> # Change the log message
```

```
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5
>>> # Change the log level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>
```

討論

這一小節的關鍵點在於訪問函數 (如 `set_message()` 和 `set_level()`)，它們被作爲屬性賦給包裝器。每個訪問函數允許使用 `nonlocal` 來修改函數內部的變量。

還有一個令人吃驚的地方是訪問函數會在多層裝飾器間傳播 (如果你的裝飾器都使用了 `@functools.wraps` 註解)。例如，假設你引入另外一個裝飾器，比如 9.2 小節中的 `@timethis` ，像下面這樣：

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

你會發現訪問函數依舊有效：

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

你還會發現即使裝飾器像下面這樣以相反的方向排放，效果也是一樣的：

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

還能通過使用 lambda 表達式代碼來讓訪問函數的返回不同的設定值：

```
@attach_wrapper(wrapper)
def get_level():
    return level

# Alternative
wrapper.get_level = lambda: level
```

一個比較難理解的地方就是對於訪問函數的首次使用。例如，你可能會考慮另外一個方法直接訪問函數的屬性，如下：

```
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
```

這個方法也可能正常工作，但前提是它必須是最外層的裝飾器才行。如果它的上面還有另外的裝飾器 (比如上面提到的 `@timethis` 例子)，那麼它會隱藏底層屬性，使得修改它們沒有任何作用。而通過使用訪問函數就能避免這樣的侷限性。

最後提一點，這一小節的方案也可以作爲 9.9 小節中裝飾器類的另一種實現方法。

9.6 帶可選參數的裝飾器

問題

你想寫一個裝飾器，既可以不傳參數給它，比如 `@decorator` ，也可以傳遞可選參數給它，比如 `@decorator(x,y,z)` 。

解決方案

下面是 9.5 小節中日誌裝飾器的一個修改版本：

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__
```

```
    @wraps(func)
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)

    return wrapper

# Example use
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

可以看到，@logged 裝飾器可以同時不帶參數或帶參數。

討論

這裏提到的這個問題就是通常所說的編程一致性問題。當我們使用裝飾器的時候，大部分程序員習慣了要麼不給它們傳遞任何參數，要麼給它們傳遞確切參數。其實從技術上來講，我們可以定義一個所有參數都是可選的裝飾器，就像下面這樣：

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

但是，這種寫法並不符合我們的習慣，有時候程序員忘記加上後面的括號會導致錯誤。這裏我們向你展示瞭如何以一致的編程風格來同時滿足沒有括號和有括號兩種情況。

爲了理解代碼是如何工作的，你需要非常熟悉裝飾器是如何作用到函數上以及它們的調用規則。對於一個像下面這樣的簡單裝飾器：

```
# Example use
@logged
def add(x, y):
    return x + y
```

這個調用序列跟下面等價：

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

add = logged(add)
```

這時候，被裝飾函數會被當做第一個參數直接傳遞給 logged 裝飾器。因此，logged() 中的第一個參數就是被包裝函數本身。所有其他參數都必須有默認值。

而對於一個下面這樣有參數的裝飾器：

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

調用序列跟下面等價：

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

初始調用 `logged()` 函數時，被包裝函數並沒有傳遞進來。因此在裝飾器內，它必須是可選的。這個反過來會迫使其他參數必須使用關鍵字來指定。並且，但這些參數被傳遞進來後，裝飾器要返回一個接受一個函數參數幷包裝它的函數 (參考 9.5 小節)。爲了這樣做，我們使用了一個技巧，就是利用 `functools.partial` 。它會返回一個未完全初始化的自身，除了被包裝函數外其他參數都已經確定下來了。可以參考 7.8 小節獲取更多 `partial()` 方法的知識。

9.7 利用裝飾器強制函數上的類型檢查

問題

作爲某種編程規約，你想在對函數參數進行強制類型檢查。

解決方案

在演示實際代碼前，先說明我們的目標：能對函數參數類型進行斷言，類似下面這樣：

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

下面是使用裝飾器技術來實現 `@typeassert` ：

```
from inspect import signature
from functools import wraps
```

```
def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_
↪types[name])
                            )
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

可以看出這個裝飾器非常靈活，既可以指定所有參數類型，也可以只指定部分。並且可以通過位置或關鍵字來指定參數類型。下面是使用示例：

```
>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

討論

這節是高級裝飾器示例，引入了很多重要的概念。

首先，裝飾器只會在函數定義時被調用一次。有時候你去掉裝飾器的功能，那麼你只需要簡單的返回被裝飾函數即可。下面的代碼中，如果全局變量 `__debug__` 被設

置成了 False(當你使用-O 或-OO 參數的優化模式執行程序時)，那麼就直接返回未修改過的函數本身：

```
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
```

其次，這裏還對被包裝函數的參數簽名進行了檢查，我們使用了 inspect.signature() 函數。簡單來講，它運行你提取一個可調用對象的參數簽名信息。例如：

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>
↪)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

裝飾器的開始部分，我們使用了 bind_partial() 方法來執行從指定類型到名稱的部分綁定。下面是例子演示：

```
>>> bound_types = sig.bind_partial(int,z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

在這個部分綁定中，你可以注意到缺失的參數被忽略了 (比如並沒有對 y 進行綁定)。不過最重要的是創建了一個有序字典 bound_types.arguments 。這個字典會將參數名以函數簽名中相同順序映射到指定的類型值上面去。在我們的裝飾器例子中，這個映射包含了我們要強制指定的類型斷言。

在裝飾器創建的實際包裝函數中使用到了 sig.bind() 方法。bind() 跟 bind_partial() 類似，但是它不允許忽略任何參數。因此有了下面的結果：

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
```

```
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

使用這個映射我們可以很輕鬆的實現我們的強制類型檢查：

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
...
>>>
```

不過這個方案還有點小瑕疵，它對於有默認值的參數並不適用。比如下面的代碼可以正常工作，儘管 items 的類型是錯誤的：

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

最後一點是關於適用裝飾器參數和函數註解之間的爭論。例如，爲什麼不像下面這樣寫一個裝飾器來查找函數中的註解呢？

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

一個可能的原因是如果使用了函數參數註解，那麼就被限制了。如果註解被用來做類型檢查就不能做其他事情了。而且 @typeassert 不能再用於使用註解做其他事情的函數了。而使用上面的裝飾器參數靈活性大多了，也更加通用。

可以在 PEP 362 以及 inspect 模塊中找到更多關於函數參數對象的信息。在 9.16 小節還有另外一個例子。

9.8 將裝飾器定義爲類的一部分

問題

你想在類中定義裝飾器，並將其作用在其他函數或方法上。

解決方案

在類裏面定義裝飾器很簡單，但是你首先要確認它的使用方式。比如到底是作爲一個實例方法還是類方法。下面我們用例子來闡述它們的不同：

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

下面是一使用例子：

```
# As an instance method
a = A()
@a.decorator1
def spam():
    pass
# As a class method
@A.decorator2
def grok():
    pass
```

仔細觀察可以發現一個是實例調用，一個是類調用。

討論

在類中定義裝飾器初看上去好像很奇怪，但是在標準庫中有很多這樣的例子。特別的，`@property` 裝飾器實際上是一個類，它裏面定義了三個方法 `getter()`, `setter()`, `deleter()` , 每一個方法都是一個裝飾器。例如：

```
class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

它爲什麼要這麼定義的主要原因是各種不同的裝飾器方法會在關聯的 property 實例上操作它的狀態。因此，任何時候只要你碰到需要在裝飾器中記錄或綁定信息，那麼這不失爲一種可行方法。

在類中定義裝飾器有個難理解的地方就是對於額外參數 self 或 cls 的正確使用。儘管最外層的裝飾器函數比如 decorator1() 或 decorator2() 需要提供一個 self 或 cls 參數，但是在兩個裝飾器內部被創建的 wrapper() 函數並不需要包含這個 self 參數。你唯一需要這個參數是在你確實要訪問包裝器中這個實例的某些部分的時候。其他情況下都不用去管它。

對於類裏面定義的包裝器還有一點比較難理解，就是在涉及到繼承的時候。例如，假設你想讓在 A 中定義的裝飾器作用在子類 B 中。你需要像下面這樣寫：

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

也就是說，裝飾器要被定義成類方法並且你必須顯式的使用父類名去調用它。你不能使用 @B.decorator2 ，因爲在方法定義時，這個類 B 還沒有被創建。

9.9 將裝飾器定義爲類

問題

你想使用一個裝飾器去包裝函數，但是希望返回一個可調用的實例。你需要讓你的裝飾器可以同時工作在類定義的內部和外部。

解決方案

爲了將裝飾器定義成一個實例，你需要確保它實現了 __call__() 和 __get__() 方法。例如，下面的代碼定義了一個類，它在其他函數上放置一個簡單的記錄層：

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

你可以將它當做一個普通的裝飾器來使用，在類裏面或外面都可以：

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

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

在交互環境中的使用示例：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

討論

將裝飾器定義成類通常是很簡單的。但是這裏還是有一些細節需要解釋下，特別是當你想將它作用在實例方法上的時候。

首先，使用 functools.wraps() 函數的作用跟之前還是一樣，將被包裝函數的元信息複製到可調用實例中去。

其次，通常很容易會忽視上面的 __get__() 方法。如果你忽略它，保持其他代碼不變再次運行，你會發現當你去調用被裝飾實例方法時出現很奇怪的問題。例如：

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: bar() missing 1 required positional argument: 'x'
```

出錯原因是當方法函數在一個類中被查找時，它們的 __get__() 方法依據描述器協議被調用，在 8.9 小節已經講述過描述器協議了。在這裏，__get__() 的目的是創建一個綁定方法對象 (最終會給這個方法傳遞 self 參數)。下面是一個例子來演示底層原理：

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

__get__() 方法是爲了確保綁定方法對象能被正確的創建。type.MethodType() 手動創建一個綁定方法來使用。只有當實例被使用的時候綁定方法纔會被創建。如果這個方法是在類上面來訪問，那麼 __get__() 中的 instance 參數會被設置成 None 並直接返回 Profiled 實例本身。這樣的話我們就可以提取它的 ncalls 屬性了。

如果你想避免一些混亂，也可以考慮另外一個使用閉包和 nonlocal 變量實現的裝飾器，這個在 9.5 小節有講到。例如：

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
```

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

這個方式跟之前的效果幾乎一樣，除了對於 ncalls 的訪問現在是通過一個被綁定爲屬性的函數來實現，例如：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

9.10 爲類和靜態方法提供裝飾器

問題

你想給類或靜態方法提供裝飾器。

解決方案

給類或靜態方法提供裝飾器是很簡單的，不過要確保裝飾器在 @classmethod 或 @staticmethod 之前。例如：

```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Class illustrating application of the decorator to different kinds of␣
↪methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
```

```
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

装飾後的類和靜態方法可正常工作，只不過增加了額外的計時功能：

```
>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>
```

討論

如果你把裝飾器的順序寫錯了就會出錯。例如，假設你像下面這樣寫：

```
class Spam:
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

那麼你調用這個靜態方法時就會報錯：

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "timethis.py", line 6, in wrapper
start = time.time()
```

```
TypeError: 'staticmethod' object is not callable
>>>
```

問題在於 `@classmethod` 和 `@staticmethod` 實際上並不會創建可直接調用的對象，而是創建特殊的描述器對象 (參考 8.9 小節)。因此當你試着在其他裝飾器中將它們當做函數來使用時就會出錯。確保這種裝飾器出現在裝飾器鏈中的第一個位置可以修復這個問題。

當我們在抽象基類中定義類方法和靜態方法 (參考 8.12 小節) 時，這裏講到的知識就很有用了。例如，如果你想定義一個抽象類方法，可以使用類似下面的代碼：

```
from abc import ABCMeta, abstractmethod
class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

在這段代碼中，`@classmethod` 跟 `@abstractmethod` 兩者的順序是有講究的，如果你調換它們的順序就會出錯。

9.11 裝飾器爲被包裝函數增加參數

問題

你想在裝飾器中給被包裝函數增加額外的參數，但是不能影響這個函數現有的調用規則。

解決方案

可以使用關鍵字參數來給被包裝函數增加額外參數。考慮下面的裝飾器：

```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    return wrapper
```

```
>>> @optional_debug
... def spam(a,b,c):
...     print(a,b,c)
...
>>> spam(1,2,3)
```

```
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

討論

通過裝飾器來給被包裝函數增加參數的做法並不常見。儘管如此，有時候它可以避免一些重複代碼。例如，如果你有下面這樣的代碼：

```
def a(x, debug=False):
    if debug:
        print('Calling a')

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')

def c(x, y, debug=False):
    if debug:
        print('Calling c')
```

那麼你可以將其重構成這樣：

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper

@optional_debug
def a(x):
    pass

@optional_debug
def b(x, y, z):
    pass

@optional_debug
def c(x, y):
```

```
    pass
```

這種實現方案之所以行得通，在於強制關鍵字參數很容易被添加到接受 *args 和 **kwargs 參數的函數中。通過使用強制關鍵字參數，它被作爲一個特殊情況被挑選出來，並且接下來僅僅使用剩餘的位置和關鍵字參數去調用這個函數時，這個特殊參數會被排除在外。也就是說，它並不會被納入到 **kwargs 中去。

還有一個難點就是如何去處理被添加的參數與被包裝函數參數直接的名字衝突。例如，如果裝飾器 @optional_debug 作用在一個已經擁有一個 debug 參數的函數上時會有問題。這裏我們增加了一步名字檢查。

上面的方案還可以更完美一點，因爲精明的程序員應該發現了被包裝函數的函數簽名其實是錯誤的。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)
>>>
```

通過如下的修改，可以解決這個問題：

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                inspect.Parameter.KEYWORD_ONLY,
                default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper
```

通過這樣的修改，包裝後的函數簽名就能正確的顯示 debug 參數的存在了。例如：

```
>>> @optional_debug
... def add(x,y):
...     return x+y
```

```
...
>>> print(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

參考 9.16 小節獲取更多關於函數簽名的信息。

9.12 使用裝飾器擴充類的功能

問題

你想通過反省或者重寫類定義的某部分來修改它的行爲，但是你又不希望使用繼承或元類的方式。

解決方案

這種情況可能是類裝飾器最好的使用場景了。例如，下面是一個重寫了特殊方法 __getattribute__ 的類裝飾器，可以打印日誌：

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Make a new definition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattribute__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

下面是使用效果：

```
>>> a = A(42)
>>> a.x
getting: x
42
```

```
>>> a.spam()
getting: spam
>>>
```

討論

類裝飾器通常可以作爲其他高級技術比如混入或元類的一種非常簡潔的替代方案。比如，上面示例中的另外一種實現使用到繼承：

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

這種方案也行得通，但是爲了去理解它，你就必須知道方法調用順序、super() 以及其它 8.7 小節介紹的繼承知識。某種程度上來講，類裝飾器方案就顯得更加直觀，並且它不會引入新的繼承體系。它的運行速度也更快一些，因爲他並不依賴 super() 函數。

如果你係想在一個類上面使用多個類裝飾器，那麼就需要注意下順序問題。例如，一個裝飾器 A 會將其裝飾的方法完整替換成另一種實現，而另一個裝飾器 B 只是簡單的在其裝飾的方法中添加點額外邏輯。那麼這時候裝飾器 A 就需要放在裝飾器 B 的前面。

你還可以回顧一下 8.13 小節另外一個關於類裝飾器的有用的例子。

9.13 使用元類控制實例的創建

問題

你想通過改變實例創建方式來實現單例、緩存或其他類似的特性。

解決方案

Python 程序員都知道，如果你定義了一個類，就能像函數一樣的調用它來創建實例，例如：

```
class Spam:
    def __init__(self, name):
        self.name = name
```

```
a = Spam('Guido')
b = Spam('Diana')
```

如果你想自定義這個步驟，你可以定義一個元類並自己實現 `__call__()` 方法。

爲了演示，假設你不想任何人創建這個類的實例：

```
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')
```

這樣的話，用戶只能調用這個類的靜態方法，而不能使用通常的方法來創建它的實例。例如：

```
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example1.py", line 7, in __call__
        raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

現在，假如你想實現單例模式（只能創建唯一實例的類），實現起來也很簡單：

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

那麼 Spam 類就只能創建唯一的實例了，演示如下：

```
>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

最後，假設你想創建 8.25 小節中那樣的緩存實例。下面我們可以通過元類來實現：

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

# Example
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name
```

然後我也來測試一下：

```
>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido') # Cached
>>> a is b
False
>>> a is c # Cached value returned
True
>>>
```

討論

利用元類實現多種實例創建模式通常要比不使用元類的方式優雅得多。

假設你不使用元類，你可能需要將類隱藏在某些工廠函數後面。比如爲了實現一個單例，你你可能會像下面這樣寫：

```
class _Spam:
    def __init__(self):
        print('Creating Spam')

_spam_instance = None

def Spam():
    global _spam_instance

    if _spam_instance is not None:
        return _spam_instance
    else:
        _spam_instance = _Spam()
        return _spam_instance
```

儘管使用元類可能會涉及到比較高級點的技術，但是它的代碼看起來會更加簡潔舒服，而且也更加直觀。

更多關於創建緩存實例、弱引用等內容，請參考 8.25 小節。

9.14 捕獲類的屬性定義順序

問題

你想自動記錄一個類中屬性和方法定義的順序，然後可以利用它來做很多操作（比如序列化、映射到數據庫等等）。

解決方案

利用元類可以很容易的捕獲類的定義信息。下面是一個例子，使用了一個 OrderedDict 來記錄描述器的定義順序：

```
from collections import OrderedDict

# A set of descriptors for various types
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = 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

class Integer(Typed):
```

```
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()
```

在這個元類中，執行類主體時描述器的定義順序會被一個 OrderedDict``捕獲到，生成的有序名稱從字典中提取出來並放入類屬性 ``_order 中。這樣的話類中的方法可以通過多種方式來使用它。例如，下面是一個簡單的類，使用這個排序字典來實現將一個類實例的數據序列化爲一行 CSV 數據：

```
class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)

# Example use
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()

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

我們在交互式環境中測試一下這個 Stock 類：

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
```

```
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

討論

本節一個關鍵點就是 OrderedMeta 元類中定義的 “ ___prepare___()“ 方法。這個方法會在開始定義類和它的父類的時候被執行。它必須返回一個映射對象以便在類定義體中被使用到。我們這裏通過返回了一個 OrderedDict 而不是一個普通的字典，可以很容易的捕獲定義的順序。

如果你想構造自己的類字典對象，可以很容易的擴展這個功能。比如，下面的這個修改方案可以防止重複的定義：

```
from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self.
↪clsname))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)
```

下面我們測試重複的定義會出現什麼情況：

```
>>> class A(metaclass=OrderedMeta):
...     def spam(self):
...         pass
...     def spam(self):
...         pass
...
Traceback (most recent call last):
```

```
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in A
    File "dupmethod2.py", line 25, in __setitem__
        (name, self.clsname))
TypeError: spam already defined in A
>>>
```

最後還有一點很重要，就是在 `__new__()` 方法中對於元類中被修改字典的處理。儘管類使用了另外一個字典來定義，在構造最終的 `class` 對象的時候，我們仍然需要將這個字典轉換爲一個正確的 `dict` 實例。通過語句 `d = dict(clsdict)` 來完成這個效果。

對於很多應用程序而已，能夠捕獲類定義的順序是一個看似不起眼卻又非常重要的特性。例如，在對象關係映射中，我們通常會看到下面這種方式定義的類：

```
class Stock(Model):
    name = String()
    shares = Integer()
    price = Float()
```

在框架底層，我們必須捕獲定義的順序來將對象映射到元組或數據庫表中的行（就類似於上面例子中的 `as_csv()` 的功能）。這節演示的技術非常簡單，並且通常會比其他類似方法（通常都要在描述器類中維護一個隱藏的計數器）要簡單的多。

9.15 定義有可選參數的元類

問題

你想定義一個元類，允許類定義時提供可選參數，這樣可以控制或配置類型的創建過程。

解決方案

在定義類的時候，Python 允許我們使用“metaclass“關鍵字參數來指定特定的元類。例如使用抽象基類：

```
from abc import ABCMeta, abstractmethod
class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

然而，在自定義元類中我們還可以提供其他的關鍵字參數，如下所示：

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    pass
```

爲了使元類支持這些關鍵字參數，你必須確保在 __prepare__() ，__new__() 和 __init__() 方法中都使用強制關鍵字參數。就像下面這樣：

```
class MyMeta(type):
    # Optional
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__prepare__(name, bases)

    # Required
    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        return super().__new__(cls, name, bases, ns)

    # Required
    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        pass
        super().__init__(name, bases, ns)
```

討論

給一個元類添加可選關鍵字參數需要你完全弄懂類創建的所有步驟，因爲這些參數會被傳遞給每一個相關的方法。__prepare__() 方法在所有類定義開始執行前首先被調用，用來創建類命名空間。通常來講，這個方法只是簡單的返回一個字典或其他映射對象。__new__() 方法被用來實例化最終的類對象。它在類的主體被執行完後開始執行。__init__() 方法最後被調用，用來執行其他的一些初始化工作。

當我們構造元類的時候，通常只需要定義一個 __new__() 或 __init__() 方法，但不是兩個都定義。但是，如果需要接受其他的關鍵字參數的話，這兩個方法就要同時提供，並且都要提供對應的參數簽名。默認的 __prepare__() 方法接受任意的關鍵字參數，但是會忽略它們，所以只有當這些額外的參數可能會影響到類命名空間的創建時你才需要去定義 __prepare__() 方法。

通過使用強制關鍵字參數，在類的創建過程中我們必須通過關鍵字來指定這些參數。

使用關鍵字參數配置一個元類還可以視作對類變量的一種替代方式。例如：

```
class Spam(metaclass=MyMeta):
    debug = True
    synchronize = True
    pass
```

將這些屬性定義爲參數的好處在於它們不會污染類的名稱空間，這些屬性僅僅只從屬於類的創建階段，而不是類中的語句執行階段。另外，它們在 `__prepare__()` 方法中是可以被訪問的，因爲這個方法會在所有類主體執行前被執行。但是類變量只能在元類的 `__new__()` 和 `__init__()` 方法中可見。

9.16 *args 和 **kwargs 的強制參數簽名

問題

你有一個函數或方法，它使用 *args 和 **kwargs 作爲參數，這樣使得它比較通用，但有時候你想檢查傳遞進來的參數是不是某個你想要的類型。

解決方案

對任何涉及到操作函數調用簽名的問題，你都應該使用 `inspect` 模塊中的簽名特性。我們最主要關注兩個類：`Signature` 和 `Parameter`。下面是一個創建函數前面的交互例子：

```
>>> from inspect import Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...          Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),
...          Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print(sig)
(x, y=42, *, z=None)
>>>
```

一旦你有了一個簽名對象，你就可以使用它的 `bind()` 方法很容易的將它綁定到 *args 和 **kwargs 上去。下面是一個簡單的演示：

```
>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name,value)
...
>>> # Try various examples
>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
```

```
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
        raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
        raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
    File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
        '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>
```

可以看出來，通過將簽名和傳遞的參數綁定起來，可以強制函數調用遵循特定的規則，比如必填、默認、重複等等。

下面是一個強制函數簽名更具體的例子。在代碼中，我們在基類中先定義了一個非常通用的 `__init__()` 方法，然後我們強制所有的子類必須提供一個特定的參數簽名。

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example use
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

下面是使用這個 `Stock` 類的示例：

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>
```

討論

在我們需要構建通用函數庫、編寫裝飾器或實現代理的時候，對於 *args 和 **kwargs 的使用是很普遍的。但是，這樣的函數有一個缺點就是當你想要實現自己的參數檢驗時，代碼就會笨拙混亂。在 8.11 小節裏面有這樣一個例子。這時候我們可以通過一個簽名對象來簡化它。

在最後的一個方案實例中，我們還可以通過使用自定義元類來創建簽名對象。下面演示怎樣來實現：

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
             for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields',[]))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']
```

當我們自定義簽名的時候，將簽名存儲在特定的屬性 `__signature__` 中通常是很有用的。這樣的話，在使用 `inspect` 模塊執行內省的代碼就能發現簽名並將它作爲調用約定。

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>
```

9.17 在類上強制使用編程規約

問題

你的程序包含一個很大的類繼承體系，你希望強制執行某些編程規約（或者代碼診斷）來幫助程序員保持清醒。

解決方案

如果你想監控類的定義，通常可以通過定義一個元類。一個基本元類通常是繼承自 `type` 並重定義它的 `__new__()` 方法或者是 `__init__()` 方法。比如：

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)
```

另一種是，定義 `__init__()` 方法：

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
```

爲了使用這個元類，你通常要將它放到到一個頂級父類定義中，然後其他的類繼承這個頂級父類。例如：

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass
```

```
class B(Root):
    pass
```

元類的一個關鍵特點是它允許你在定義的時候檢查類的內容。在重新定義 __init__() 方法中，你可以很輕鬆的檢查類字典、父類等等。並且，一旦某個元類被指定給了某個類，那麼就會被繼承到所有子類中去。因此，一個框架的構建者就能在大型的繼承體系中通過給一個頂級父類指定一個元類去捕獲所有下面子類的定義。

作爲一個具體的應用例子，下面定義了一個元類，它會拒絕任何有混合大小寫名字作爲方法的類定義 (可能是想氣死 Java 程序員 ^_^):

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self): # Ok
        pass

class B(Root):
    def fooBar(self): # TypeError
        pass
```

作爲更高級和實用的例子，下面有一個元類，它用來檢測重載方法，確保它的調用參數跟父類中原始方法有着相同的參數簽名。

```
from inspect import signature
import logging

class MatchSignaturesMeta(type):

    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            # Get the previous definition (if any) and compare the signatures
            prev_dfn = getattr(sup,name,None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                    value.__qualname__, prev_sig, val_sig)
```

```
# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self,x,z):
        pass
```

如果你運行這段代碼，就會得到下面這樣的輸出結果：

```
WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)
```

這種警告信息對於捕獲一些微妙的程序 bug 是很有用的。例如，如果某個代碼依賴於傳遞給方法的關鍵字參數，那麼當子類改變參數名字的時候就會調用出錯。

討論

在大型面向對象的程序中，通常將類的定義放在元類中控制是很有用的。元類可以監控類的定義，警告編程人員某些沒有注意到的可能出現的問題。

有人可能會說，像這樣的錯誤可以通過程序分析工具或 IDE 去做會更好些。誠然，這些工具是很有用。但是，如果你在構建一個框架或函數庫供其他人使用，那麼你沒辦法去控制使用者要使用什麼工具。因此，對於這種類型的程序，如果可以在元類中做檢測或許可以帶來更好的用戶體驗。

在元類中選擇重新定義 `__new__()` 方法還是 `__init__()` 方法取決於你想怎樣使用結果類。`__new__()` 方法在類創建之前被調用，通常用於通過某種方式（比如通過改變類字典的內容）修改類的定義。而 `__init__()` 方法是在類被創建之後被調用，當你需要完整構建類對象的時候會很有用。在最後一個例子中，這是必要的，因爲它使用了 `super()` 函數來搜索之前的定義。它只能在類的實例被創建之後，並且相應的方法解析順序也已經被設置好了。

最後一個例子還演示了 Python 的函數簽名對象的使用。實際上，元類將每個可調用定義放在一個類中，搜索前一個定義（如果有的話），然後通過使用 `inspect.signature()` 來簡單的比較它們的調用簽名。

最後一點，代碼中有一行使用了 `super(self, self)` 並不是排版錯誤。當使用元

類的時候，我們要時刻記住一點就是 self 實際上是一個類對象。因此，這條語句其實就是用來尋找位於繼承體系中構建 self 父類的定義。

9.18 以編程方式定義類

問題

你在寫一段代碼，最終需要創建一個新的類對象。你考慮將類的定義源代碼以字符串的形式發佈出去。並且使用函數比如 exec() 來執行它，但是你想尋找一個更加優雅的解決方案。

解決方案

你可以使用函數 types.new_class() 來初始化新的類對象。你需要做的只是提供類的名字、父類元組、關鍵字參數，以及一個用成員變量填充類字典的回調函數。例如：

```
# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price
def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

這種方式會構建一個普通的類對象，並且按照你的期望工作：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>
```

這種方法中，一個比較難理解的地方是在調用完 types.new_class() 對 Stock.__module__ 的賦值。每次當一個類被定義後，它的 __module__ 屬性包含定義它的模塊

名。這個名字用於生成 __repr__() 方法的輸出。它同樣也被用於很多庫，比如 pickle 。因此，爲了讓你創建的類是“正確”的，你需要確保這個屬性也設置正確了。

如果你想創建的類需要一個不同的元類，可以通過 types.new_class() 第三個參數傳遞給它。例如：

```
>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                         lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>
```

第三個參數還可以包含其他的關鍵字參數。比如，一個類的定義如下：

```
class Spam(Base, debug=True, typecheck=False):
    pass
```

那麼可以將其翻譯成如下的 new_class() 調用形式：

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
                       lambda ns: ns.update(cls_dict))
```

new_class() 第四個參數最神祕，它是一個用來接受類命名空間的映射對象的函數。通常這是一個普通的字典，但是它實際上是 __prepare__() 方法返回的任意對象，這個在 9.14 小節已經介紹過了。這個函數需要使用上面演示的 update() 方法給命名空間增加內容。

討論

很多時候如果能構造新的類對象是很有用的。有個很熟悉的例子是調用 collections.namedtuple() 函數，例如：

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

namedtuple() 使用 exec() 而不是上面介紹的技術。但是，下面通過一個簡單的變化，我們直接創建一個類：

```
import operator
import types
import sys

```

```
def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

這段代碼的最後部分使用了一個所謂的”框架魔法”，通過調用 `sys._getframe()` 來獲取調用者的模塊名。另外一個框架魔法例子在 2.15 小節中有介紹過。

下面的例子演示了前面的代碼是如何工作的：

```
>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
>>>
```

這項技術一個很重要的方面是它對於元類的正確使用。你可能像通過直接實例化一個元類來直接創建一個類：

```
Stock = type('Stock', (), cls_dict)
```

這種方法的問題在於它忽略了一些關鍵步驟，比如對於元類中 `__prepare__()` 方法的調用。通過使用 `types.new_class()` ，你可以保證所有的必要初始化步驟都能得

到執行。比如，types.new_class() 第四個參數的回調函數接受 __prepare__() 方法返回的映射對象。

如果你僅僅只是想執行準備步驟，可以使用 types.prepare_class() 。例如：

```
import types
metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

它會查找合適的元類並調用它的 __prepare__() 方法。然後這個元類保存它的關鍵字參數，準備命名空間後被返回。

更多信息, 請參考 PEP 3115 , 以及 Python documentation .

9.19 在定義的時候初始化類的成員

問題

你想在類被定義的時候就初始化一部分類的成員，而不是要等到實例被創建後。

解決方案

在類定義時就執行初始化或設置操作是元類的一個典型應用場景。本質上講，一個元類會在定義時被觸發，這時候你可以執行一些額外的操作。

下面是一個例子，利用這個思路來創建類似於 collections 模塊中的命名元組的類：

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls,args)
```

這段代碼可以用來定義簡單的基於元組的數據結構，如下所示：

```
class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']
```

下面演示它如何工作：

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

討論

這一小節中，類 StructTupleMeta 獲取到類屬性 _fields 中的屬性名字列表，然後將它們轉換成相應的可訪問特定元組槽的方法。函數 operator.itemgetter() 創建一個訪問器函數，然後 property() 函數將其轉換成一個屬性。

本節最難懂的部分是知道不同的初始化步驟是什麼時候發生的。StructTupleMeta 中的 __init__() 方法只在每個類被定義時被調用一次。cls 參數就是那個被定義的類。實際上，上述代碼使用了 _fields 類變量來保存新的被定義的類，然後給它再添加一點新的東西。

StructTuple 類作爲一個普通的基類，供其他使用者來繼承。這個類中的 __new__() 方法用來構造新的實例。這裏使用 __new__() 並不是很常見，主要是因爲我們要修改元組的調用簽名，使得我們可以像普通的實例調用那樣創建實例。就像下面這樣：

```
s = Stock('ACME', 50, 91.1) # OK
s = Stock(('ACME', 50, 91.1)) # Error
```

跟 __init__() 不同的是，__new__() 方法在實例被創建之前被觸發。由於元組是不可修改的，所以一旦它們被創建了就不可能對它做任何改變。而 __init__() 會在實例創建的最後被觸發，這樣的話我們就可以做我們想做的了。這也是爲什麼 __new__() 方法已經被定義了。

儘管本節很短，還是需要你能仔細研讀，深入思考 Python 類是如何被定義的，實例是如何被創建的，還有就是元類和類的各個不同的方法究竟在什麼時候被調用。

PEP 422 提供了一個解決本節問題的另外一種方法。但是，截止到我寫這本書的時候，它還沒被採納和接受。儘管如此，如果你使用的是 Python 3.3 或更高的版本，那麼還是值得去看一下的。

9.20 利用函數註解實現方法重載

問題

你已經學過怎樣使用函數參數註解，那麼你可能會想利用它來實現基於類型的方法重載。但是你不確定應該怎樣去實現（或者到底行得通不）。

解決方案

本小節的技術是基於一個簡單的技術，那就是 Python 允許參數註解，代碼可以像下面這樣寫：

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3) # Prints Bar 1: 2 3
s.bar('hello') # Prints Bar 2: hello 0
```

下面是我們第一步的嘗試，使用到了一個元類和描述器：

```
# multiple.py
import inspect
import types

class MultiMethod:
    '''
    Represents a single multimethod.
    '''
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        '''
        Register a new method as a multimethod
        '''
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(name)
                )
```

```
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(name)
                )
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
            types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        '''
        Call a method based on type signature of the arguments
        '''
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            raise TypeError('No matching method for types {}'.format(types))

    def __get__(self, instance, cls):
        '''
        Descriptor method needed to make calls work in a class
        '''
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class MultiDict(dict):
    '''
    Special dictionary to build multimethods in a metaclass
    '''
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    '''
```

```
    Metaclass that allows multiple dispatch of methods
    '''
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()
```

為了使用這個類，你可以像下面這樣寫：

```
class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time

class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

下面是一個交互示例來驗證它能正確的工作：

```
>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "multiple.py", line 42, in __call__
        raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)
>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
```

```
2012
>>> e.month
12
>>> e.day
3
>>>
```

討論

坦白來講，相對於通常的代碼而已本節使用到了很多的魔法代碼。但是，它卻能讓我們深入理解元類和描述器的底層工作原理，並能加深對這些概念的印象。因此，就算你並不會立即去應用本節的技術，它的一些底層思想卻會影響到其它涉及到元類、描述器和函數註解的編程技術。

本節的實現中的主要思路其實是很簡單的。MutipleMeta 元類使用它的 __prepare__() 方法來提供一個作爲 MultiDict 實例的自定義字典。這個跟普通字典不一樣的是，MultiDict 會在元素被設置的時候檢查是否已經存在，如果存在的話，重複的元素會在 MultiMethod 實例中合併。

MultiMethod 實例通過構建從類型簽名到函數的映射來收集方法。在這個構建過程中，函數註解被用來收集這些簽名然後構建這個映射。這個過程在 MultiMethod.register() 方法中實現。這種映射的一個關鍵特點是對於多個方法，所有參數類型都必須要指定，否則就會報錯。

爲了讓 MultiMethod 實例模擬一個調用，它的 __call__() 方法被實現了。這個方法從所有排除 slef 的參數中構建一個類型元組，在內部 map 中查找這個方法，然後調用相應的方法。爲了能讓 MultiMethod 實例在類定義時正確操作，__get__() 是必須得實現的。它被用來構建正確的綁定方法。比如：

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

不過本節的實現還有一些限制，其中一個是它不能使用關鍵字參數。例如：

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
```

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

也許有其他的方法能添加這種支持，但是它需要一個完全不同的方法映射方式。問題在於關鍵字參數的出現是沒有順序的。當它跟位置參數混合使用時，那你的參數就會變得比較混亂了，這時候你不得不在 `__call__()` 方法中先去做個排序。

同樣對於繼承也是有限制的，例如，類似下面這種代碼就不能正常工作：

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)

    def foo(self, x:C):
        print('Foo 2:', x)
```

原因是因爲 `x:A` 註解不能成功匹配子類實例（比如 B 的實例），如下：

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "multiple.py", line 44, in __call__
        raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

作爲使用元類和註解的一種替代方案，可以通過描述器來實現類似的效果。例如：

```
import types

class multimethod:
    def __init__(self, func):
```

```
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self
```

爲了使用描述器版本，你需要像下面這樣寫：

```
class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
        print('Bar 2:', s, n)
```

描述器方案同樣也有前面提到的限制（不支持關鍵字參數和繼承）。

所有事物都是平等的，有好有壞，也許最好的辦法就是在普通代碼中避免使用方法重載。不過有些特殊情況下還是有意義的，比如基於模式匹配的方法重載程序中。舉個例子，8.21 小節中的訪問者模式可以修改爲一個使用方法重載的類。但是，除了這個以外，通常不應該使用方法重載（就簡單的使用不同名稱的方法就行了）。

在 Python 社區對於實現方法重載的討論已經由來已久。對於引發這個爭論的原因，可以參考下 Guido van Rossum 的這篇博客：Five-Minute Multimethods in Python

9.21 避免重複的屬性方法

問題

你在類中需要重複的定義一些執行相同邏輯的屬性方法，比如進行類型檢查，怎樣去簡化這些重複代碼呢？

解決方案

考慮下一個簡單的類，它的屬性由屬性方法包裝：

```
class Person:
    def __init__(self, name ,age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError('age must be an int')
        self._age = value
```

可以看到，爲了實現屬性值的類型檢查我們寫了很多的重複代碼。只要你以後看到類似這樣的代碼，你都應該想辦法去簡化它。一個可行的方法是創建一個函數用來定義屬性並返回它。例如：

```
def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
```

```
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)

    return prop

# Example use
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)

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

討論

本節我們演示內部函數或者閉包的一個重要特性，它們很像一個宏。例子中的函數 typed_property() 看上去有點難理解，其實它所做的僅僅就是爲你生成屬性並返回這個屬性對象。因此，當在一個類中使用它的時候，效果跟將它裏面的代碼放到類定義中去是一樣的。儘管屬性的 getter 和 setter 方法訪問了本地變量如 name , expected_type 以及 storate_name ，這個很正常，這些變量的值會保存在閉包當中。

我們還可以使用 functools.partial() 來稍稍改變下這個例子，很有趣。例如，你可以像下面這樣：

```
from functools import partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)

# Example:
class Person:
    name = String('name')
    age = Integer('age')

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

其實你可以發現，這裏的代碼跟 8.13 小節中的類型系統描述器代碼有些相似。

9.22 定義上下文管理器的簡單方法

問題

你想自己去實現一個新的上下文管理器，以便使用 with 語句。

解決方案

實現一個新的上下文管理器的最簡單的方法就是使用 contexlib 模塊中的 @contextmanager 裝飾器。下面是一個實現了代碼塊計時功能的上下文管理器例子：

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{}: {}'.format(label, end - start))

# Example use
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

在函數 timethis() 中，yield 之前的代碼會在上下文管理器中作爲 __enter__() 方法執行，所有在 yield 之後的代碼會作爲 __exit__() 方法執行。如果出現了異常，異常會在 yield 語句那裏拋出。

下面是一個更加高級一點的上下文管理器，實現了列表對象上的某種事務：

```
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working
```

這段代碼的作用是任何對列表的修改只有當所有代碼運行完成並且不出現異常的情況下才會生效。下面我們來演示一下：

```
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
```

```
    File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

討論

通常情況下，如果要寫一個上下文管理器，你需要定義一個類，裏面包含一個 `__enter__()` 和一個 `__exit__()` 方法，如下所示：

```
import time

class timethis:
    def __init__(self, label):
        self.label = label

    def __enter__(self):
        self.start = time.time()

    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

儘管這個也不難寫，但是相比較寫一個簡單的使用 `@contextmanager` 註解的函數而言還是稍顯乏味。

`@contextmanager` 應該僅僅用來寫自包含的上下文管理函數。如果你有一些對象 (比如一個文件、網絡連接或鎖)，需要支持 `with` 語句，那麼你就需要單獨實現 `__enter__()` 方法和 `__exit__()` 方法。

9.23 在局部變量域中執行代碼

問題

你想在使用範圍內執行某個代碼片段，並且希望在執行後所有的結果都不可見。

解決方案

爲了理解這個問題，先試試一個簡單場景。首先，在全局命名空間內執行一個代碼片段：

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

然後，再在一個函數中執行同樣的代碼：

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in test
NameError: global name 'b' is not defined
>>>
```

可以看出，最後拋出了一個 NameError 異常，就跟在 exec() 語句從沒執行過一樣。要是你想在後面的計算中使用到 exec() 執行結果的話就會有問題了。

爲了修正這樣的錯誤，你需要在調用 exec() 之前使用 locals() 函數來得到一個局部變量字典。之後你就能從局部字典中獲取修改過後的變量值了。例如：

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

討論

實際上對於 exec() 的正確使用是比較難的。大多數情況下當你要考慮使用 exec() 的時候，還有另外更好的解決方案 (比如裝飾器、閉包、元類等等)。

然而，如果你仍然要使用 exec() ，本節列出了一些如何正確使用它的方法。默認情況下，exec() 會在調用者局部和全局範圍內執行代碼。然而，在函數裏面，傳遞給 exec() 的局部範圍是拷貝實際局部變量組成的一個字典。因此，如果 exec() 如果執行了修改操作，這種修改後的結果對實際局部變量值是沒有影響的。下面是另外一個演示它的例子：

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
0
>>>
```

上面代碼裏，當你調用 locals() 獲取局部變量時，你獲得的是傳遞給 exec() 的局部變量的一個拷貝。通過在代碼執行後審查這個字典的值，那就能獲取修改後的值了。下面是一個演示例子：

```
>>> def test2():
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

仔細觀察最後一步的輸出，除非你將 loc 中被修改後的值手動賦值給 x，否則 x 變量值是不會變的。

在使用 locals() 的時候，你需要注意操作順序。每次它被調用的時候，locals() 會獲取局部變量值中的值並覆蓋字典中相應的變量。請注意觀察下下面這個試驗的輸出結果：

```
>>> def test3():
...     x = 0
...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

注意最後一次調用 locals() 的時候 x 的值是如何被覆蓋掉的。

作爲 locals() 的一個替代方案，你可以使用你自己的字典，並將它傳遞給 exec()。例如：

```
>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     glb = { }
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
```

```
...
>>> test4()
14
>>>
```

大部分情況下，這種方式是使用 exec() 的最佳實踐。你只需要保證全局和局部字典在後面代碼訪問時已經被初始化。

還有一點，在使用 exec() 之前，你可能需要問下自己是否有其他更好的替代方案。大多數情況下當你要考慮使用 exec() 的時候，還有另外更好的解決方案，比如裝飾器、閉包、元類，或其他一些元編程特性。

9.24 解析與分析 Python 源碼

問題

你想寫解析並分析 Python 源代碼的程序。

解決方案

大部分程序員知道 Python 能夠計算或執行字符串形式的源代碼。例如：

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>
```

儘管如此，ast 模塊能被用來將 Python 源碼編譯成一個可被分析的抽象語法樹（AST）。例如：

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
```

```
right=Name(id='x', ctx=Load()))))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None))], orelse=[])])"
>>>
```

分析源碼樹需要你自己更多的學習，它是由一系列 AST 節點組成的。分析這些節點最簡單的方法就是定義一個訪問者類，實現很多 `visit_NodeName()` 方法，`NodeName()` 匹配那些你感興趣的節點。下面是這樣一個類，記錄了哪些名字被加載、存儲和刪除的信息。

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
    for i in range(10):
        print(i)
    del i
    '''

    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
```

```
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

如果你運行這個程序，你會得到下面這樣的輸出：

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

最後，AST 可以通過 compile() 函數來編譯並執行。例如：

```
>>> exec(compile(top,'<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

討論

當你能夠分析源代碼並從中獲取信息的時候，你就能寫很多代碼分析、優化或驗證工具了。例如，相比盲目的傳遞一些代碼片段到類似 exec() 函數中，你可以先將它轉換成一個 AST，然後觀察它的細節看它到底是怎樣做的。你還可以寫一些工具來查看某個模塊的全部源碼，並且在此基礎上執行某些靜態分析。

需要注意的是，如果你知道自己在幹啥，你還能夠重寫 AST 來表示新的代碼。下面是一個裝飾器例子，可以通過重新解析函數體源碼、重寫 AST 並重新創建函數代碼對象來將全局訪問變量降爲函數體作用範圍，

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
```

```
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                          for name in self.lowered_names)
        code_ast = ast.parse(code, mode='exec')

        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ','\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
        cl.visit(top)

        # Execute the modified AST
        temp = {}
        exec(compile(top,'','exec'), temp, temp)

        # Pull out the modified code object
        func.__code__ = temp[func.__name__].__code__
        return func
    return lower
```

爲了使用這個代碼，你可以像下面這樣寫：

```
INCR = 1
@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR
```

裝飾器會將 countdown() 函數重寫爲類似下面這樣子：

```
def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
```

```
    while n > 0:
        n -= INCR
```

在性能測試中，它會讓函數運行快 20%

現在，你是不是想爲你所有的函數都加上這個裝飾器呢？或許不會。但是，這卻是對於一些高級技術比如 AST 操作、源碼操作等等的一個很好的演示說明

本節受另外一個在 ActiveState 中處理 Python 字節碼的章節的啓示。使用 AST 是一個更加高級點的技術，並且也更簡單些。參考下面一節獲得字節碼的更多信息。

9.25 拆解 Python 字節碼

問題

你想通過將你的代碼反編譯成低級的字節碼來查看它底層的工作機制。

解決方案

dis 模塊可以被用來輸出任何 Python 函數的反編譯結果。例如：

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
...
>>>
```

討論

當你想要知道你的程序底層的運行機制的時候，dis 模塊是很有用的。比如如果你想試着理解性能特徵。被 dis() 函數解析的原始字節碼如下所示：

```
>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00Wt\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

如果你想自己解釋這段代碼，你需要使用一些在 opcode 模塊中定義的常量。例如：

```
>>> c = countdown.__code__.co_code
>>> import opcode
```

```
>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>
```

奇怪的是，在 dis 模塊中並沒有函數讓你以編程方式很容易的來處理字節碼。不過，下面的生成器函數可以將原始字節碼序列轉換成 opcodes 和參數。

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)
```

使用方法如下：

```
>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
```

這種方式很少有人知道，你可以利用它替換任何你想要替換的函數的原始字節碼。下面我們用一個示例來演示整個過程：

```
>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
```

```
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

你可以像這樣耍大招讓解釋器奔潰。但是，對於編寫更高級優化和元編程工具的程序員來講，他們可能真的需要重寫字節碼。本節最後的部分演示了這個是怎樣做到的。你還可以參考另外一個類似的例子：this code on ActiveState

第十章：模塊與包

模塊與包是任何大型程序的核心，就連 Python 安裝程序本身也是一個包。本章重點涉及有關模塊和包的常用編程技術，例如如何組織包、把大型模塊分割成多個文件、創建命名空間包。同時，也給出了讓你自定義導入語句的祕籍。

10.1 構建一個模塊的層級包

問題

你想將你的代碼組織成由很多分層模塊構成的包。

解決方案

封裝成包是很簡單的。在文件系統上組織你的代碼，並確保每個目錄都定義了一個 ___init___.py 文件。例如：

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

一旦你做到了這一點，你應該能夠執行各種 import 語句，如下：

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

討論

定義模塊的層次結構就像在文件系統上建立目錄結構一樣容易。文件 ___init___.py 的目的是要包含不同運行級別的包的可選的初始化代碼。舉個例子，如果你執行了語句 import graphics，文件 graphics/___init___.py 將被導入，建立 graphics 命名空間的內容。像 import graphics.format.jpg 這樣導入，文件 graphics/___init___.py 和文件 graphics/formats/___init___.py 將在文件 graphics/formats/jpg.py 導入之前導入。

絕大部分時候讓 ___init___.py 空着就好。但是有些情況下可能包含代碼。舉個例子，___init___.py 能夠用來自動加載子模塊：

```
# graphics/formats/__init__.py
from . import jpg
from . import png
```

像這樣一個文件, 用戶可以僅僅通過 import grahpics.formats 來代替 import graphics.formats.jpg 以及 import graphics.formats.png。

___init___.py 的其他常用用法包括將多個文件合併到一個邏輯命名空間，這將在 10.4 小節討論。

敏銳的程序員會發現，即使沒有 ___init___.py 文件存在，python 仍然會導入包。如果你沒有定義 ___init___.py 時，實際上創建了一個所謂的“命名空間包”，這將在 10.5 小節討論。萬物平等，如果你着手創建一個新的包的話，包含一個 ___init___.py 文件吧。

10.2 控制模塊被全部導入的內容

問題

當使用’from module import *’語句時，希望對從模塊或包導出的符號進行精確控制。

解決方案

在你的模塊中定義一個變量 ___all___ 來明確地列出需要導出的內容。

舉個例子:

```
# somemodule.py
def spam():
    pass

def grok():
    pass

blah = 42
# Only export 'spam' and 'grok'
__all__ = ['spam', 'grok']
```

討論

儘管強烈反對使用‘from module import *’, 但是在定義了大量變量名的模塊中頻繁使用。如果你不做任何事, 這樣的導入將會導入所有不以下劃線開頭的。另一方面, 如果定義了 ___all___ , 那麼只有被列舉出的東西會被導出。

如果你將 ___all___ 定義成一個空列表, 沒有東西將被導入。如果 ___all___ 包含未定義的名字, 在導入時引起 AttributeError。

10.3 使用相對路徑名導入包中子模塊

問題

將代碼組織成包, 想用 import 語句從另一個包名沒有硬編碼過的包的中導入子模塊。

解決方案

使用包的相對導入，使一個模塊導入同一個包的另一個模塊舉個例子，假設在你的文件系統上有 mypackage 包，組織如下：

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

如果模塊 mypackage.A.spam 要導入同目錄下的模塊 grok，它應該包括的 import 語句如下：

```
# mypackage/A/spam.py
from . import grok
```

如果模塊 mypackage.A.spam 要導入不同目錄下的模塊 B.bar，它應該使用的 import 語句如下：

```
# mypackage/A/spam.py
from ..B import bar
```

兩個 import 語句都沒包含頂層包名，而是使用了 spam.py 的相對路徑。

討論

在包內，既可以使用相對路徑也可以使用絕對路徑來導入。舉個例子：

```
# mypackage/A/spam.py
from mypackage.A import grok # OK
from . import grok # OK
import grok # Error (not found)
```

像 mypackage.A 這樣使用絕對路徑名的不利之處是這將頂層包名硬編碼到你的源碼中。如果你想重新組織它，你的代碼將更脆，很難工作。舉個例子，如果你改變了包名，你就必須檢查所有文件來修正源碼。同樣，硬編碼的名稱會使移動代碼變得困難。

舉個例子，也許有人想安裝兩個不同版本的軟件包，只通過名稱區分它們。如果使用相對導入，那一切都 ok，然而使用絕對路徑名很可能會出問題。

import 語句的 . 和 .. 看起來很滑稽, 但它指定目錄名. 爲當前目錄，..B 爲目錄../B。這種語法只適用於 import。舉個例子：

```
from . import grok # OK
import .grok # ERROR
```

儘管使用相對導入看起來像是瀏覽文件系統，但是不能到定義包的目錄之外。也就是說，使用點的這種模式從不是包的目錄中導入將會引發錯誤。

最後，相對導入只適用於在合適的包中的模塊。尤其是在頂層的腳本的簡單模塊中，它們將不起作用。如果包的部分被作爲腳本直接執行，那它們將不起作用例如：

```
% python3 mypackage/A/spam.py # Relative imports fail
```

另一方面，如果你使用 Python 的-m 選項來執行先前的腳本，相對導入將會正確運行。例如：

```
% python3 -m mypackage.A.spam # Relative imports work
```

更多的包的相對導入的背景知識, 請看 PEP 328 .

10.4 將模塊分割成多個文件

問題

你想將一個模塊分割成多個文件。但是你不想將分離的文件統一成一個邏輯模塊時使已有的代碼遭到破壞。

解決方案

程序模塊可以通過變成包來分割成多個獨立的文件。考慮下下面簡單的模塊：

```
# mymodule.py
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

假設你想 mymodule.py 分爲兩個文件，每個定義的一個類。要做到這一點，首先用 mymodule 目錄來替換文件 mymodule.py。這這個目錄下，創建以下文件：

```
mymodule/
    __init__.py
```

```
    a.py
    b.py
```

在 a.py 文件中插入以下代碼：

```
# a.py
class A:
    def spam(self):
        print('A.spam')
```

在 b.py 文件中插入以下代碼：

```
# b.py
from .a import A
class B(A):
    def bar(self):
        print('B.bar')
```

最後，在 __init__.py 中，將 2 個文件粘合在一起：

```
# __init__.py
from .a import A
from .b import B
```

如果按照這些步驟，所產生的包 MyModule 將作爲一個單一的邏輯模塊：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

討論

在這個章節中的主要問題是一個設計問題，不管你是否希望用戶使用很多小模塊或只是一個模塊。舉個例子，在一個大型的代碼庫中，你可以將這一切都分割成獨立的文件，讓用戶使用大量的 import 語句，就像這樣：

```
from mymodule.a import A
from mymodule.b import B
...
```

這樣能工作，但這讓用戶承受更多的負擔，用戶要知道不同的部分位於何處。通常情況下，將這些統一起來，使用一條 import 將更加容易，就像這樣：

```
from mymodule import A, B
```

對後者而言，讓 mymodule 成爲一個大的源文件是最常見的。但是，這一章節展示瞭如何合併多個文件合併成一個單一的邏輯命名空間。這樣做的關鍵是創建一個包目錄，使用 ___init___.py 文件來將每部分粘合在一起。

當一個模塊被分割，你需要特別注意交叉引用的文件名。舉個例子，在這一章節中，B 類需要訪問 A 類作爲基類。用包的相對導入 from .a import A 來獲取。

整個章節都使用包的相對導入來避免將頂層模塊名硬編碼到源代碼中。這使得重命名模塊或者將它移動到別的位置更容易。(見 10.3 小節)

作爲這一章節的延伸，將介紹延遲導入。如圖所示，___init___.py 文件一次導入所有必需的組件的。但是對於一個很大的模塊，可能你只想組件在需要時被加載。要做到這一點，___init___.py 有細微的變化：

```
# __init__.py
def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

在這個版本中，類 A 和類 B 被替換爲在第一次訪問時加載所需的類的函數。對於用戶，這看起來不會有太大的不同。例如：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

延遲加載的主要缺點是繼承和類型檢查可能會中斷。你可能會稍微改變你的代碼，例如:

```
if isinstance(x, mymodule.A): # Error
...

if isinstance(x, mymodule.a.A): # Ok
...
```

延遲加載的真實例子, 見標準庫 multiprocessing/___init___.py 的源碼.

10.5 利用命名空間導入目錄分散的代碼

問題

你可能有大量的代碼，由不同的人來分散地維護。每個部分被組織爲文件目錄，如一個包。然而，你希望能用共同的包前綴將所有組件連接起來，不是將每一個部分作爲獨立的包來安裝。

解決方案

從本質上講，你要定義一個頂級 Python 包，作爲一個大集合分開維護子包的命名空間。這個問題經常出現在大的應用框架中，框架開發者希望鼓勵用戶發佈插件或附加包。

在統一不同的目錄裏統一相同的命名空間，但是要刪去用來將組件聯合起來的 ___init___.py 文件。假設你有 Python 代碼的兩個不同的目錄如下：

```
foo-package/
    spam/
        blah.py

bar-package/
    spam/
        grok.py
```

在這 2 個目錄裏，都有着共同的命名空間 spam。在任何一個目錄裏都沒有 ___init___.py 文件。

讓我們看看，如果將 foo-package 和 bar-package 都加到 python 模塊路徑並嘗試導入會發生什麼

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah
>>> import spam.grok
>>>
```

兩個不同的包目錄被合併到一起，你可以導入 spam.blah 和 spam.grok，並且它們能夠工作。

討論

在這裏工作的機制被稱爲“包命名空間”的一個特徵。從本質上講，包命名空間是一種特殊的封裝設計，爲合併不同的目錄的代碼到一個共同的命名空間。對於大的框架，這可能是有用的，因爲它允許一個框架的部分被單獨地安裝下載。它也使人們能夠輕鬆地爲這樣的框架編寫第三方附加組件和其他擴展。

包命名空間的關鍵是確保頂級目錄中沒有 ___init___.py 文件來作爲共同的命名空間。缺失 ___init___.py 文件使得在導入包的時候會發生有趣的事情：這並沒有產生錯誤，解釋器創建了一個由所有包含匹配包名的目錄組成的列表。特殊的包命名空間模塊被創建，只讀的目錄列表副本被存儲在其 ___path___ 變量中。舉個例子：

```
>>> import spam
>>> spam.__path__
_NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

在定位包的子組件時，目錄 __path__ 將被用到 (例如, 當導入 spam.grok 或者 spam.blah 的時候).

包命名空間的一個重要特點是任何人都可以用自己的代碼來擴展命名空間。舉個例子，假設你自己的代碼目錄像這樣：

```
my-package/
    spam/
        custom.py
```

如果你將你的代碼目錄和其他包一起添加到 sys.path，這將無縫地合併到別的 spam 包目錄中：

```
>>> import spam.custom
>>> import spam.grok
>>> import spam.blah
>>>
```

一個包是否被作爲一個包命名空間的主要方法是檢查其 __file__ 屬性。如果沒有，那包是個命名空間。這也可以由其字符表現形式中的“namespace”這個詞體現出來。

```
>>> spam.__file__
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>> spam
<module 'spam' (namespace)>
>>>
```

更多的包命名空間信息可以查看 PEP 420.

10.6 重新加載模塊

問題

你想重新加載已經加載的模塊，因爲你對其源碼進行了修改。

解決方案

使用 imp.reload() 來重新加載先前加載的模塊。舉個例子：

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

討論

重新加載模塊在開發和調試過程中常常很有用。但在生產環境中的代碼使用會不安全，因爲它並不總是像您期望的那樣工作。

reload() 擦除了模塊底層字典的內容，並通過重新執行模塊的源代碼來刷新它。模塊對象本身的身份保持不變。因此，該操作在程序中所有已經被導入了的地方更新了模塊。

儘管如此，reload() 沒有更新像”from module import name”這樣使用 import 語句導入的定義。舉個例子：

```
# spam.py
def bar():
    print('bar')

def grok():
    print('grok')
```

現在啓動交互式會話：

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

不退出 Python 修改 spam.py 的源碼，將 grok() 函數改成這樣：

```
def grok():
    print('New grok')
```

現在回到交互式會話，重新加載模塊，嘗試下這個實驗：

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok() # Notice old output
grok
>>> spam.grok() # Notice new output
```

```
New grok
>>>
```

在這個例子中，你看到有 2 個版本的 grok() 函數被加載。通常來說，這不是你想要的，而是令人頭疼的事。

因此，在生產環境中可能需要避免重新加載模塊。在交互環境下調試，解釋程序並試圖弄懂它。

10.7 運行目錄或壓縮文件

問題

您有一個已成長爲包含多個文件的應用，它已遠不再是一個簡單的腳本，你想向用戶提供一些簡單的方法運行這個程序。

解決方案

如果你的應用程序已經有多個文件，你可以把你的應用程序放進它自己的目錄並添加一個 __main__.py 文件。舉個例子，你可以像這樣創建目錄：

```
myapplication/
    spam.py
    bar.py
    grok.py
    __main__.py
```

如果 __main__.py 存在，你可以簡單地在頂級目錄運行 Python 解釋器：

```
bash % python3 myapplication
```

解釋器將執行 __main__.py 文件作爲主程序。

如果你將你的代碼打包成 zip 文件，這種技術同樣也適用，舉個例子：

```
bash % ls
spam.py bar.py grok.py __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

討論

創建一個目錄或 zip 文件並添加 __main__.py 文件來將一個更大的 Python 應用打包是可行的。這和作爲標準庫被安裝到 Python 庫的代碼包是有一點區別的。相反，這只是讓別人執行的代碼包。

由於目錄和 zip 文件與正常文件有一點不同，你可能還需要增加一個 shell 腳本，使執行更加容易。例如，如果代碼文件名爲 myapp.zip，你可以創建這樣一個頂級腳本：

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

10.8 讀取位於包中的數據文件

問題

你的包中包含代碼需要去讀取的數據文件。你需要儘可能地用最便捷的方式來做這件事。

解決方案

假設你的包中的文件組織成如下：

```
mypackage/
    __init__.py
    somedata.dat
    spam.py
```

現在假設 spam.py 文件需要讀取 somedata.dat 文件中的內容。你可以用以下代碼來完成：

```
# spam.py
import pkgutil
data = pkgutil.get_data(__package__, 'somedata.dat')
```

由此產生的變量是包含該文件的原始內容的字節字符串。

討論

要讀取數據文件，你可能會傾向於編寫使用內置的 I/ O 功能的代碼，如 open()。但是這種方法也有一些問題。

首先，一個包對解釋器的當前工作目錄幾乎沒有控制權。因此，編程時任何 I/O 操作都必須使用絕對文件名。由於每個模塊包含有完整路徑的 __file__ 變量，這弄清楚它的路徑不是不可能，但它很凌亂。

第二，包通常安裝作爲.zip 或.egg 文件，這些文件並不像在文件系統上的一個普通目錄裏那樣被保存。因此，你試圖用 open() 對一個包含數據文件的歸檔文件進行操作，它根本不會工作。

pkgutil.get_data() 函數是一個讀取數據文件的高級工具，不用管包是如何安裝以及安裝在哪。它只是工作並將文件內容以字節字符串返回給你

get_data() 的第一個參數是包含包名的字符串。你可以直接使用包名，也可以使用特殊的變量，比如 __package__。第二個參數是包內文件的相對名稱。如果有必要，可以使用標準的 Unix 命名規範到不同的目錄，只有最後的目錄仍然位於包中。

10.9 將文件夾加入到 sys.path

問題

你無法導入你的 Python 代碼因爲它所在的目錄不在 sys.path 裏。你想將添加新目錄到 Python 路徑，但是不想硬鏈接到你的代碼。

解決方案

有兩種常用的方式將新目錄添加到 sys.path。第一種，你可以使用 PYTHONPATH 環境變量來添加。例如：

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct 4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

在自定義應用程序中，這樣的環境變量可在程序啓動時設置或通過 shell 腳本。

第二種方法是創建一個.pth 文件，將目錄列舉出來，像這樣：

```
# myapplication.pth
/some/dir
/other/dir
```

這個.pth 文件需要放在某個 Python 的 site-packages 目錄，通常位於/usr/local/lib/python3.3/site-packages 或者 ~/.local/lib/python3.3/sitepackages。當解釋器啓動時，.pth 文件裏列舉出來的存在於文件系統的目錄將被添加到 sys.path。安裝一個.pth 文件可能需要管理員權限，如果它被添加到系統級的 Python 解釋器。

討論

比起費力地找文件，你可能會傾向於寫一個代碼手動調節 sys.path 的值。例如:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

雖然這能“工作”，它是在實踐中極爲脆弱，應儘量避免使用。這種方法的問題是，它將目錄名硬編碼到了你的源代碼。如果你的代碼被移到一個新的位置，這會導致維護問題。更好的做法是在不修改源代碼的情況下，將 path 配置到其他地方。如果您使用模塊級的變量來精心構造一個適當的絕對路徑，有時你可以解決硬編碼目錄的問題，比如 ___file___。舉個例子：

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, join(abspath(dirname(__file__)), 'src'))
```

這將 src 目錄添加到 path 裏，和執行插入步驟的代碼在同一個目錄裏。

site-packages 目錄是第三方包和模塊安裝的目錄。如果你手動安裝你的代碼，它將被安裝到 site-packages 目錄。雖然用於配置 path 的.pth 文件必須放置在 site-packages 裏，但它配置的路徑可以是系統上任何你希望的目錄。因此，你可以把你的代碼放在一系列不同的目錄，只要那些目錄包含在.pth 文件裏。

10.10 通過字符串名導入模塊

問題

你想導入一個模塊，但是模塊的名字在字符串裏。你想對字符串調用導入命令。

解決方案

使用 importlib.import_module() 函數來手動導入名字爲字符串給出的一個模塊或者包的一部分。舉個例子：

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

import_module 只是簡單地執行和 import 相同的步驟，但是返回生成的模塊對象。你只需要將其存儲在一個變量，然後像正常的模塊一樣使用。

如果你正在使用的包，import_module() 也可用於相對導入。但是，你需要給它一個額外的參數。例如：

```
import importlib
# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

討論

使用 import_module() 手動導入模塊的問題通常出現在以某種方式編寫修改或覆蓋模塊的代碼時候。例如，也許你正在執行某種自定義導入機制，需要通過名稱來加載一個模塊，通過補丁加載代碼。

在舊的代碼，有時你會看到用於導入的內建函數 ___import___()。儘管它能工作，但是 importlib.import_module() 通常更容易使用。

自定義導入過程的高級實例見 10.11 小節

10.11 通過鉤子遠程加載模塊

問題

你想自定義 Python 的 import 語句，使得它能從遠程機器上面透明的加載模塊。

解決方案

首先要提出來的是安全問題。本節討論的思想如果沒有一些額外的安全和認知機制的話會很糟糕。也就是說，我們的主要目的是深入分析 Python 的 import 語句機制。如果你理解了本節內部原理，你就能夠爲其他任何目的而自定義 import。有了這些，讓我們繼續向前走。

本節核心是設計導入語句的擴展功能。有很多種方法可以做這個，不過爲了演示的方便，我們開始先構造下面這個 Python 代碼結構：

```
testcode/
    spam.py
    fib.py
    grok/
        __init__.py
        blah.py
```

這些文件的內容並不重要，不過我們在每個文件中放入了少量的簡單語句和函數，這樣你可以測試它們並查看當它們被導入時的輸出。例如：

```
# spam.py
print("I'm spam")

def hello(name):
    print('Hello %s' % name)

# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# grok/__init__.py
print("I'm grok.__init__")
```

```
# grok/blah.py
print("I'm grok.blah")
```

這裏的目的是允許這些文件作爲模塊被遠程訪問。也許最簡單的方式就是將它們發佈到一個 web 服務器上面。在 testcode 目錄中像下面這樣運行 Python：

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

服務器運行起來後再啓動一個單獨的 Python 解釋器。確保你可以使用 urllib 訪問到遠程文件。例如：

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

從這個服務器加載源代碼是接下來本節的基礎。爲了替代手動的通過 urlopen() 來收集源文件，我們通過自定義 import 語句來在後臺自動幫我們做到。

加載遠程模塊的第一種方法是創建一個顯示的加載函數來完成它。例如：

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

這個函數會下載源代碼，並使用 compile() 將其編譯到一個代碼對象中，然後在一個新創建的模塊對象的字典中來執行它。下面是使用這個函數的方式：

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
```

```
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py
↪'>
>>>
```

正如你所見，對於簡單的模塊這個是行得通的。不過它並沒有嵌入到通常的 import 語句中，如果要支持更高級的結構比如包就需要更多的工作了。

一個更酷的做法是創建一個自定義導入器。第一種方法是創建一個元路徑導入器。如下：

```
# urlimport.py
import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Debugging
import logging
log = logging.getLogger(__name__)

# Get links from a given URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))
    links = set()
    try:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
    except Exception as e:
        log.debug('Could not get links. %s', e)
    log.debug('links: %r', links)
    return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
```

```
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]
        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Check link cache
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl)
                log.debug('find_module: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return loader
        # A normal module
        filename = basename + '.py'
        if filename in self._links[baseurl]:
            log.debug('find_module: module %r found', fullname)
            return self._loaders[baseurl]
        else:
            log.debug('find_module: module %r not found', fullname)
            return None

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links.clear()

# Module Loader for a URL
class UrlModuleLoader(importlib.abc.SourceLoader):
```

```
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

    # Required method
    def load_module(self, fullname):
        code = self.get_code(fullname)
        mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
        mod.__file__ = self.get_filename(fullname)
        mod.__loader__ = self
        mod.__package__ = fullname.rpartition('.')[0]
        exec(code, mod.__dict__)
        return mod

    # Optional extensions
    def get_code(self, fullname):
        src = self.get_source(fullname)
        return compile(src, self.get_filename(fullname), 'exec')

    def get_data(self, path):
        pass

    def get_filename(self, fullname):
        return self._baseurl + '/' + fullname.split('.')[-1] + '.py'

    def get_source(self, fullname):
        filename = self.get_filename(fullname)
        log.debug('loader: reading %r', filename)
        if filename in self._source_cache:
            log.debug('loader: cached %r', filename)
            return self._source_cache[filename]
        try:
            u = urlopen(filename)
            source = u.read().decode('utf-8')
            log.debug('loader: %r loaded', filename)
            self._source_cache[filename] = source
            return source
        except (HTTPError, URLError) as e:
            log.debug('loader: %r failed. %s', filename, e)
            raise ImportError("Can't load %s" % filename)

    def is_package(self, fullname):
        return False

# Package loader for a URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
```

```
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Utility functions for installing/uninstalling the loader
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)
```

下面是一個交互會話，演示瞭如何使用前面的代碼：

```
>>> # importing currently fails
>>> import fib
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> # Load the importer and retry (it works)
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

這個特殊的方案會安裝一個特別的查找器 `UrlMetaFinder` 實例，作爲 `sys.meta_path` 中最後的實體。當模塊被導入時，會依據 `sys.meta_path` 中的查找器定位模塊。在這個例子中，`UrlMetaFinder` 實例是最後一個查找器方案，當模塊在任何一個普通地方都找不到的時候就觸發它。

作爲常見的實現方案，UrlMetaFinder 類包裝在一個用戶指定的 URL 上。在內部，查找器通過抓取指定 URL 的內容構建合法的鏈接集合。導入的時候，模塊名會跟已有的鏈接作對比。如果找到了一個匹配的，一個單獨的 UrlModuleLoader 類被用來從遠程機器上加載源代碼並創建最終的模塊對象。這裏緩存鏈接的一個原因是避免不必要的 HTTP 請求重複導入。

自定義導入的第二種方法是編寫一個鉤子直接嵌入到 sys.path 變量中去，識別某些目錄命名模式。在 urlimport.py 中添加如下的類和支持函數：

```
# urlimport.py
# ... include previous code above ...
# Path finder class for a URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')
        basename = parts[-1]
        # Check link cache
        if self._links is None:
            self._links = [] # See discussion
            self._links = _get_links(self._baseurl)

        # Check if it's a package
        if basename in self._links:
            log.debug('find_loader: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                log.debug('find_loader: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_loader: %r is a namespace package', fullname)
                loader = None
            return (loader, [fullurl])

        # A normal module
        filename = basename + '.py'
        if filename in self._links:
            log.debug('find_loader: module %r found', fullname)
            return (self._loader, [])
        else:
            log.debug('find_loader: module %r not found', fullname)
            return (None, [])

    def invalidate_caches(self):
```

```
        log.debug('invalidating link cache')
        self._links = None

# Check path to see if it looks like a URL
_url_path_cache = {}
def handle_url(path):
    if path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if path in _url_path_cache:
            finder = _url_path_cache[path]
        else:
            finder = UrlPathFinder(path)
            _url_path_cache[path] = finder
        return finder
    else:
        log.debug('Handle path? %s. [No]', path)

def install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def remove_path_hook():
    sys.path_hooks.remove(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')
```

要使用這個路徑查找器，你只需要在 sys.path 中加入 URL 鏈接。例如：

```
>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)
>>> import fib
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
```

```
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

關鍵點就是 handle_url() 函數，它被添加到了 sys.path_hooks 變量中。當 sys.path 的實體被處理時，會調用 sys.path_hooks 中的函數。如果任何一個函數返回了一個查找器對象，那麼這個對象就被用來爲 sys.path 實體加載模塊。

遠程模塊加載跟其他的加載使用方法幾乎是一樣的。例如：

```
>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
>>> print(inspect.getsource(fib))
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

討論

在詳細討論之前，有點要強調的是，Python 的模塊、包和導入機制是整個語言中最複雜的部分，即使經驗豐富的 Python 程序員也很少能精通它們。我在這裏推薦一些值的去讀的文檔和書籍，包括 importlib module 和 PEP 302. 文檔內容在這裏不會被重複提到，不過我在這裏會討論一些最重要的部分。

首先，如果你想創建一個新的模塊對象，使用 imp.new_module() 函數：

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

模塊對象通常有一些期望屬性，包括 __file__ （運行模塊加載語句的文件名）和 __package__ （包名）。

其次，模塊會被解釋器緩存起來。模塊緩存可以在字典 sys.modules 中被找到。因爲有了這個緩存機制，通常可以將緩存和模塊的創建通過一個步驟完成：

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

如果給定模塊已經存在那麼就會直接獲得已經被創建過的模塊，例如：

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

由於創建模塊很簡單，很容易編寫簡單函數比如第一部分的 load_module() 函數。這個方案的一個缺點是很難處理複雜情況比如包的導入。爲了處理一個包，你要重新實現普通 import 語句的底層邏輯 (比如檢查目錄，查找 __init__.py 文件，執行那些文件，設置路徑等)。這個複雜性就是爲什麼最好直接擴展 import 語句而不是自定義函數的一個原因。

擴展 import 語句很簡單，但是會有很多移動操作。最高層上，導入操作被一個位於 sys.meta_path 列表中的“元路徑”查找器處理。如果你輸出它的值，會看到下面這樣：

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
<class '_frozen_importlib.FrozenImporter'>,
<class '_frozen_importlib.PathFinder'>]
>>>
```

當執行一個語句比如 import fib 時，解釋器會遍歷 sys.mata_path 中的查找器對象，調用它們的 find_module() 方法定位正確的模塊加載器。可以通過實驗來看看：

```
>>> class Finder:
...     def find_module(self, fullname, path):
...         print('Looking for', fullname, path)
...         return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder()) # Insert as first entry
>>> import math
Looking for math None
```

```
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

注意看 `find_module()` 方法是怎樣在每一個導入就被觸發的。這個方法中的 path 參數的作用是處理包。多個包被導入，就是一個可在包的 `__path__` 屬性中找到的路徑列表。要找到包的子組件就要檢查這些路徑。比如注意對於 `xml.etree` 和 `xml.etree.ElementTree` 的路徑配置：

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>
```

在 `sys.meta_path` 上查找器的位置很重要，將它從隊頭移到隊尾，然後再試試導入看：

```
>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime
```

現在你看不到任何輸出了，因爲導入被 sys.meta_path 中的其他實體處理。這時候，你只有在導入不存在模塊的時候才能看到它被觸發：

```
>>> import fib
Looking for fib None
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
ImportError: No module named 'xml.superfast'
>>>
```

你之前安裝過一個捕獲未知模塊的查找器，這個是 UrlMetaFinder 類的關鍵。一個 UrlMetaFinder 實例被添加到 sys.meta_path 的末尾，作爲最後一個查找器方案。如果被請求的模塊名不能定位，就會被這個查找器處理掉。處理包的時候需要注意，在 path 參數中指定的值需要被檢查，看它是否以查找器中註冊的 URL 開頭。如果不是，該子模塊必須歸屬於其他查找器並被忽略掉。

對於包的其他處理可在 UrlPackageLoader 類中被找到。這個類不會導入包名，而是去加載對應的 __init__.py 文件。它也會設置模塊的 __path__ 屬性，這一步很重要，因爲在加載包的子模塊時這個值會被傳給後面的 find_module() 調用。基於路徑的導入鉤子是這些思想的一個擴展，但是採用了另外的方法。我們都知道，sys.path 是一個 Python 查找模塊的目錄列表，例如：

```
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
'/usr/local/lib/python33.zip',
'/usr/local/lib/python3.3',
'/usr/local/lib/python3.3/plat-darwin',
'/usr/local/lib/python3.3/lib-dynload',
'/usr/local/lib/...3.3/site-packages']
>>>
```

在 sys.path 中的每一個實體都會被額外的綁定到一個查找器對象上。你可以通過查看 sys.path_importer_cache 去看下這些查找器：

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
'/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
'/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
'/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collections
↪'),
'/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodings'),
'/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dynload
↪'),
'/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-darwin
↪'),
'/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site-
↪packages'),
'/usr/local/lib/python33.zip': None}
>>>
```

sys.path_importer_cache 比 sys.path 會更大點，因爲它會爲所有被加載代碼的目錄記錄它們的查找器。這包括包的子目錄，這些通常在 sys.path 中是不存在的。

要執行 import fib ，會順序檢查 sys.path 中的目錄。對於每個目錄，名稱“fib”會被傳給相應的 sys.path_importer_cache 中的查找器。這個可以讓你創建自己的查找器並在緩存中放入一個實體。試試這個：

```
>>> class Finder:
... def find_loader(self, name):
...     print('Looking for', name)
...     return (None, [])
...
>>> import sys
>>> # Add a "debug" entry to the importer cache
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>
```

在這裏，你可以爲名字“debug”創建一個新的緩存實體並將它設置成 sys.path 上的第一個。在所有接下來的導入中，你會看到你的查找器被觸發了。不過，由於它返回 (None, [])，那麼處理進程會繼續處理下一個實體。

sys.path_importer_cache 的使用被一個存儲在 sys.path_hooks 中的函數列表控制。試試下面的例子，它會清除緩存並給 sys.path_hooks 添加一個新的路徑檢查函數

```
>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> import fib
Checked debug
Checking .
Checking /usr/local/lib/python33.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

正如你所見，check_path() 函數被每個 sys.path 中的實體調用。不顧，由於拋出了 ImportError 異常，啥都不會發生了（僅僅將檢查轉移到 sys.path_hooks 的下一個函數）。

知道了怎樣 sys.path 是怎樣被處理的，你就能構建一個自定義路徑檢查函數來查找文件名，不然 URL。例如：

```
>>> def check_url(path):
...     if path.startswith('http://'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib # Finder output!
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

這就是本節最後部分的關鍵點。事實上，一個用來在 sys.path 中查找 URL 的自定義路徑檢查函數已經構建完畢。當它們被碰到的時候，一個新的 UrlPathFinder 實例被創建並被放入 sys.path_importer_cache. 之後，所有需要檢查 sys.path 的導入語句都會使用你的自定義查找器。

基於路徑導入的包處理稍微有點複雜，並且跟 find_loader() 方法返回值有關。對於簡單模塊，find_loader() 返回一個元組 (loader, None)，其中的 loader 是一個用於導入模塊的加載器實例。

對於一個普通的包，find_loader() 返回一個元組 (loader, path)，其中的 loader 是一個用於導入包（並執行 __init__.py）的加載器實例，path 是一個會初始化包的 __path__ 屬性的目錄列表。例如，如果基礎 URL 是 http://localhost:15000 並且一個用戶執行 import grok，那麼 find_loader() 返回的 path 就會是 ['http://localhost:15000/grok']

find_loader() 還要能處理一個命名空間包。一個命名空間包中有一個合法的包目錄名，但是不存在 __init__.py 文件。這樣的話，find_loader() 必須返回一個元組 (None, path)，path 是一個目錄列表，由它來構建包的定義有 __init__.py 文件的 __path__ 屬性。對於這種情況，導入機制會繼續前行去檢查 sys.path 中的目錄。如果找到了命名空間包，所有的結果路徑被加到一起來構建最終的命名空間包。關於命名空間包的更多信息請參考 10.5 小節。

所有的包都包含了一個內部路徑設置，可以在 __path__ 屬性中看到，例如：

```
>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

之前提到，__path__ 的設置是通過 find_loader() 方法返回值控制的。不過，__path__ 接下來也被 sys.path_hooks 中的函數處理。因此，但包的子組件被加載後，位於 __path__ 中的實體會被 handle_url() 函數檢查。這會導致新的 UrlPathFinder 實例被創建並且被加入到 sys.path_importer_cache 中。

還有個難點就是 handle_url() 函數以及它跟內部使用的 _get_links() 函數之間的交互。如果你的查找器實現需要使用到其他模塊（比如 urllib.request），有可能這些模塊會在查找器操作期間進行更多的導入。它可以導致 handle_url() 和其他查找器部分陷入一種遞歸循環狀態。爲了解釋這種可能性，實現中有一個被創建的查找器緩存（每一個 URL 一個）。它可以避免創建重複查找器的問題。另外，下面的代碼片段可以確保查找器不會在初始化鏈接集合的時候響應任何導入請求：

```
# Check link cache
if self._links is None:
    self._links = [] # See discussion
    self._links = _get_links(self._baseurl)
```

最後，查找器的 invalidate_caches() 方法是一個工具方法，用來清理內部緩存。這個方法再用戶調用 importlib.invalidate_caches() 的時候被觸發。如果你想讓 URL 導入者重新讀取鏈接列表的話可以使用它。

對比下兩種方案（修改 sys.meta_path 或使用一個路徑鉤子）。使用 sys.meta_path 的導入者可以按照自己的需要自由處理模塊。例如，它們可以從數據庫中導入或以不同於一般模塊/包處理方式導入。這種自由同樣意味着導入者需要自己進行內部的一些管理。另外，基於路徑的鉤子只是適用於對 sys.path 的處理。通過這種擴展加載的模塊跟普通方式加載的特性是一樣的。

如果到現在爲止你還是不是很明白，那麼可以通過增加一些日誌打印來測試下本節。像下面這樣：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
```

```
>>>
```

最後，建議你花點時間看看 PEP 302 以及 importlib 的文檔。

10.12 導入模塊的同時修改模塊

問題

你想給某個已存在模塊中的函數添加裝飾器。不過，前提是這個模塊已經被導入並且被使用過。

解決方案

這裏問題的本質就是你想在模塊被加載時執行某個動作。可能是你想在一個模塊被加載時觸發某個回調函數來通知你。

這個問題可以使用 10.11 小節中同樣的導入鉤子機制來實現。下面是一個可能的方案：

```
# postimport.py
import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module
```

```
def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())
```

這樣，你就可以使用 when_imported() 裝飾器了，例如：

```
>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads? Are you crazy?')
...
>>>
>>> import threading
Threads? Are you crazy?
>>>
```

作爲一個更實際的例子，你可能想在已存在的定義上面添加裝飾器，如下所示：

```
from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Example
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

討論

本節技術依賴於 10.11 小節中講述過的導入鉤子，並稍作修改。

@when_imported 裝飾器的作用是註冊在導入時被激活的處理器函數。該裝飾器檢查 sys.modules 來查看模塊是否真的已經被加載了。如果是的話，該處理器被立即調用。不然，處理器被添加到 _post_import_hooks 字典中的一個列表中去。_post_import_hooks 的作用就是收集所有的爲每個模塊註冊的處理器對象。一個模塊可以註冊多個處理器。

要讓模塊導入後觸發添加的動作，PostImportFinder 類被設置爲 sys.meta_path 第一個元素。它會捕獲所有模塊導入操作。

本節中的 PostImportFinder 的作用並不是加載模塊，而是自帶導入完成後觸發相應的動作。實際的導入被委派給位於 sys.meta_path 中的其他查找器。PostImportLoader 類中的 imp.import_module() 函數被遞歸的調用。爲了避免陷入無線循環，PostImportFinder 保持了一個所有被加載過的模塊集合。如果一個模塊名存在就會直接被忽略掉。

當一個模塊被 imp.import_module() 加載後，所有在 _post_import_hooks 被註冊的處理器被調用，使用新加載模塊作爲一個參數。

有一點需要注意的是本機不適用於那些通過 imp.reload() 被顯式加載的模塊。也就是說，如果你加載一個之前已被加載過的模塊，那麼導入處理器將不會再被觸發。另外，要是你從 sys.modules 中刪除模塊然後再重新導入，處理器又會再一次觸發。

更多關於導入後鉤子信息請參考 PEP 369.

10.13 安裝私有的包

問題

你想要安裝一個第三方包，但是沒有權限將它安裝到系統 Python 庫中去。或者，你可能想要安裝一個供自己使用的包，而不是系統上面所有用戶。

解決方案

Python 有一個用戶安裝目錄，通常類似”~/.local/lib/python3.3/site-packages”。要強制在這個目錄中安裝包，可使用安裝選項“–user”。例如：

```
python3 setup.py install --user
```

或者

```
pip install --user packagename
```

在 sys.path 中用戶的“site-packages”目錄位於系統的“site-packages”目錄之前。因此，你安裝在裏面的包就比系統已安裝的包優先級高（儘管並不總是這樣，要取決於第三方包管理器，比如 distribute 或 pip）。

討論

通常包會被安裝到系統的 site-packages 目錄中去，路徑類似“/usr/local/lib/python3.3/site-packages”。不過，這樣做需要有管理員權限並且使用 sudo 命令。就算你有這樣的權限去執行命令，使用 sudo 去安裝一個新的，可能沒有被驗證過的包有時候也不安全。

安裝包到用戶目錄中通常是一個有效的方案，它允許你創建一個自定義安裝。

另外，你還可以創建一個虛擬環境，這個我們在下一節會講到。

10.14 創建新的 Python 環境

問題

你想創建一個新的 Python 環境，用來安裝模塊和包。不過，你不想安裝一個新的 Python 克隆，也不想對系統 Python 環境產生影響。

解決方案

你可以使用 pyvenv 命令創建一個新的“虛擬”環境。這個命令被安裝在 Python 解釋器同一目錄，或 Windows 上面的 Scripts 目錄中。下面是一個例子：

```
bash % pyvenv Spam
bash %
```

傳給 pyvenv 命令的名字是將要被創建的目錄名。當被創建後，Span 目錄像下面這樣：

```
bash % cd Spam
bash % ls
bin include lib pyvenv.cfg
bash %
```

在 bin 目錄中，你會找到一個可以使用的 Python 解釋器：

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct 6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
'/usr/local/lib/python33.zip',
'/usr/local/lib/python3.3',
'/usr/local/lib/python3.3/plat-darwin',
'/usr/local/lib/python3.3/lib-dynload',
'/Users/beazley/Spam/lib/python3.3/site-packages']
>>>
```

這個解釋器的特點就是他的 site-packages 目錄被設置爲新創建的環境。如果你要安裝第三方包，它們會被安裝在那裏，而不是通常系統的 site-packages 目錄。

討論

創建虛擬環境通常是爲了安裝和管理第三方包。正如你在例子中看到的那樣，sys.path 變量包含來自於系統 Python 的目錄，而 site-packages 目錄已經被重定位到一個新的目錄。

有了一個新的虛擬環境，下一步就是安裝一個包管理器，比如 distribute 或 pip。但安裝這樣的工具和包的時候，你需要確保你使用的是虛擬環境的解釋器。它會將包安裝到新創建的 site-packages 目錄中去。

儘管一個虛擬環境看上去是 Python 安裝的一個複製，不過它實際上只包含了少量幾個文件和一些符號鏈接。所有標準庫函文件和可執行解釋器都來自原來的 Python 安裝。因此，創建這樣的環境是很容易的，並且幾乎不會消耗機器資源。

默認情況下，虛擬環境是空的，不包含任何額外的第三方庫。如果你想將一個已經安裝的包作爲虛擬環境的一部分，可以使用“–system-site-packages”選項來創建虛擬環境，例如：

```
bash % pyvenv --system-site-packages Spam
bash %
```

跟多關於 pyvenv 和虛擬環境的信息可以參考 PEP 405.

10.15 分發包

問題

你已經編寫了一個有用的庫，想將它分享給其他人。

解決方案

如果你想分發你的代碼，第一件事就是給它一個唯一的名字，並且清理它的目錄結構。例如，一個典型的函數庫包會類似下面這樣：

```
projectname/
    README.txt
    Doc/
        documentation.txt
    projectname/
        __init__.py
        foo.py
        bar.py
        utils/
            __init__.py
            spam.py
            grok.py
    examples/
        helloworld.py
        ...
```

要讓你的包可以發佈出去，首先你要編寫一個 setup.py ，類似下面這樣：

```
# setup.py
from distutils.core import setup

setup(name='projectname',
    version='1.0',
    author='Your Name',
    author_email='you@youraddress.com',
    url='http://www.you.com/projectname',
    packages=['projectname', 'projectname.utils'],
)
```

下一步，就是創建一個 MANIFEST.in 文件，列出所有在你的包中需要包含進來的非源碼文件：

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

確保 setup.py 和 MANIFEST.in 文件放在你的包的最頂級目錄中。一旦你已經做了這些，你就可以像下面這樣執行命令來創建一個源碼分發包了：

```
% bash python3 setup.py sdist
```

它會創建一個文件比如”projectname-1.0.zip”或“projectname-1.0.tar.gz”，具體依賴於你的系統平臺。如果一切正常，這個文件就可以發送給別人使用或者上傳至 Python Package Index.

討論

對於純 Python 代碼，編寫一個普通的 setup.py 文件通常很簡單。一個可能的問題是你必須手動列出所有構成包源碼的子目錄。一個常見錯誤就是僅僅只列出一個包的最頂級目錄，忘記了包含包的子組件。這也是爲什麼在 setup.py 中對於包的說明包含了列表 packages=['projectname', 'projectname.utils']

大部分 Python 程序員都知道，有很多第三方包管理器供選擇，包括 setuptools、distribute 等等。有些是爲了替代標準庫中的 distutils。注意如果你依賴這些包，用戶可能不能安裝你的軟件，除非他們已經事先安裝過所需要的包管理器。正因如此，你更應該時刻記住越簡單越好的道理。最好讓你的代碼使用標準的 Python 3 安裝。如果其他包也需要的話，可以通過一個可選項來支持。

對於涉及到 C 擴展的代碼打包與分發就更復雜點了。第 15 章對關於 C 擴展的這方面知識有一些詳細講解，特別是在 15.2 小節中。

第十一章：網絡與 Web 編程

本章是關於在網絡應用和分佈式應用中使用的各種主題。主題劃分爲使用 Python 編寫客戶端程序來訪問已有的服務，以及使用 Python 實現網絡服務端程序。也給出了一些常見的技術，用於編寫涉及協同或通信的的代碼。

11.1 作爲客戶端與 HTTP 服務交互

問題

你需要通過 HTTP 協議以客戶端的方式訪問多種服務。例如，下載數據或者與基於 REST 的 API 進行交互。

解決方案

對於簡單的事情來說，通常使用 urllib.request 模塊就夠了。例如，發送一個簡單的 HTTP GET 請求到遠程的服務上，可以這樣做：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/get'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a GET request and read the response
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

如果你需要使用 POST 方法在請求主體中發送查詢參數，可以將參數編碼後作爲可選參數提供給 urlopen() 函數，就像這樣：

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
```

```
    'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a POST request and read the response
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

如果你需要在發出的請求中提供一些自定義的 HTTP 頭，例如修改 user-agent 字段，可以創建一個包含字段值的字典，並創建一個 Request 實例然後將其傳給 urlopen()，如下：

```
from urllib import request, parse
...

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Make a request and read the response
u = request.urlopen(req)
resp = u.read()
```

如果需要交互的服務比上面的例子都要複雜，也許應該去看看 requests 庫（https://pypi.python.org/pypi/requests）。例如，下面這個示例採用 requests 庫重新實現了上面的操作：

```
import requests

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}
```

```
resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request
text = resp.text
```

關於 requests 庫，一個值得一提的特性就是它能以多種方式從請求中返回響應結果的內容。從上面的代碼來看，resp.text 帶給我們的是以 Unicode 解碼的響應文本。但是，如果去訪問 resp.content ，就會得到原始的二進制數據。另一方面，如果訪問 resp.json ，那麼就會得到 JSON 格式的響應內容。

下面這個示例利用 requests 庫發起一個 HEAD 請求，並從響應中提取出一些 HTTP 頭數據的字段：

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

下面是一個利用 requests 通過基本認證登錄 Pypi 的例子：

```
import requests

resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

下面是一個利用 requests 將 HTTP cookies 從一個請求傳遞到另一個的例子：

```
import requests

# First request
resp1 = requests.get(url)
...

# Second requests with cookies received on first requests
resp2 = requests.get(url, cookies=resp1.cookies)
```

最後但並非最不重要的一個例子是用 requests 上傳內容：

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

討論

對於真的很簡單 HTTP 客戶端代碼，用內置的 urllib 模塊通常就足夠了。但是，如果你要做的不僅僅只是簡單的 GET 或 POST 請求，那就真的不能再依賴它的功能了。這時候就是第三方模塊比如 requests 大顯身手的時候了。

例如，如果你決定堅持使用標準的程序庫而不考慮像 requests 這樣的第三方庫，那麼也許就不得不使用底層的 http.client 模塊來實現自己的代碼。比方說，下面的代碼展示瞭如何執行一個 HEAD 請求：

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

同樣地，如果必須編寫涉及代理、認證、cookies 以及其他一些細節方面的代碼，那麼使用 urllib 就顯得特別彆扭和囉嗦。比方說，下面這個示例實現在 Python 包索引上的認證：

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=login')
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener
...
```

坦白說，所有的這些操作在 requests 庫中都變得簡單的多。

在開發過程中測試 HTTP 客戶端代碼常常是很令人沮喪的，因爲所有棘手的細節問題都需要考慮 (例如 cookies、認證、HTTP 頭、編碼方式等)。要完成這些任務，考慮使用 httpbin 服務 (http://httpbin.org)。這個站點會接收發出的請求，然後以 JSON 的形式將相應信息回傳回來。下面是一個交互式的例子：

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
```

```
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

在要同一個真正的站點進行交互前，先在 httpbin.org 這樣的網站上做實驗常常是可取的辦法。尤其是當我們面對 3 次登錄失敗就會關閉賬戶這樣的風險時尤爲有用（不要嘗試自己編寫 HTTP 認證客戶端來登錄你的銀行賬戶）。

儘管本節沒有涉及，request 庫還對許多高級的 HTTP 客戶端協議提供了支持，比如 OAuth。requests 模塊的文檔（http://docs.python-requests.org）質量很高（坦白說比在這短短的一節的篇幅中所提供的任何信息都好），可以參考文檔以獲得更多地信息。

11.2 創建 TCP 服務器

問題

你想實現一個服務器，通過 TCP 協議和客戶端通信。

解決方案

創建一個 TCP 服務器的一個簡單方法是使用 socketserver 庫。例如，下面是一個簡單的應答服務器：

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:

            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在這段代碼中，你定義了一個特殊的處理類，實現了一個 handle() 方法，用來爲客戶端連接服務。request 屬性是客戶端 socket，client_address 有客戶端地址。爲了測試這個服務器，運行它並打開另外一個 Python 進程連接這個服務器：

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
```

```
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

很多時候，可以很容易的定義一個不同的處理器。下面是一個使用 StreamRequestHandler 基類將一個類文件接口放置在底層 socket 上的例子：

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading
        for line in self.rfile:
            # self.wfile is a file-like object for writing
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

討論

socketserver 可以讓我們很容易的創建簡單的 TCP 服務器。但是，你需要注意的是，默認情況下這種服務器是單線程的，一次只能爲一個客戶端連接服務。如果你想處理多個客戶端，可以初始化一個 ForkingTCPServer 或者是 ThreadingTCPServer 對象。例如：

```
from socketserver import ThreadingTCPServer


if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

使用 fork 或線程服務器有個潛在問題就是它們會爲每個客戶端連接創建一個新的進程或線程。由於客戶端連接數是沒有限制的，因此一個惡意的黑客可以同時發送大量的連接讓你的服務器奔潰。

如果你擔心這個問題，你可以創建一個預先分配大小的工作線程池或進程池。你先創建一個普通的非線程服務器，然後在一個線程池中使用 serve_forever() 方法來啓動它們。

```
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
```

```
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

一般來講，一個 TCPServer 在實例化的時候會綁定並激活相應的 socket 。不過，有時候你想通過設置某些選項去調整底下的 *socket*‘ ，可以設置參數 bind_and_activate=False 。如下：

```
if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Set up various socket options
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Bind and activate
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

上面的 socket 選項是一個非常普遍的配置項，它允許服務器重新綁定一個之前使用過的端口號。由於要被經常使用到，它被放置到類變量中，可以直接在 TCPServer 上面設置。在實例化服務器的時候去設置它的值，如下所示：

```
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

在上面示例中，我們演示了兩種不同的處理器基類（BaseRequestHandler 和 StreamRequestHandler ）。StreamRequestHandler 更加靈活點，能通過設置其他的類變量來支持一些新的特性。比如：

```
import socket

class EchoHandler(StreamRequestHandler):
    # Optional settings (defaults shown)
    timeout = 5                     # Timeout on all socket operations
    rbufsize = -1                   # Read buffer size
    wbufsize = 0                    # Write buffer size
    disable_nagle_algorithm = False  # Sets TCP_NODELAY socket option
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

最後，還需要注意的是巨大部分 Python 的高層網絡模塊（比如 HTTP、XML-RPC 等）都是建立在 socketserver 功能之上。也就是說，直接使用 socket 庫來實現服務器也並不是很難。下面是一個使用 socket 直接編程實現的一個服務器簡單例子：

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server(('', 20000))
```

11.3 創建 UDP 服務器

問題

你想實現一個基於 UDP 協議的服務器來與客戶端通信。

解決方案

跟 TCP 一樣，UDP 服務器也可以通過使用 socketserver 庫很容易的被創建。例如，下面是一個簡單的時間服務器：

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)
```

```
if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

跟之前一樣，你先定義一個實現 handle() 特殊方法的類，爲客戶端連接服務。這個類的 request 屬性是一個包含了數據報和底層 socket 對象的元組。client_address 包含了客戶端地址。

我們來測試下這個服務器，首先運行它，然後打開另外一個 Python 進程向服務器發送消息：

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

討論

一個典型的 UDP 服務器接收到達的數據報 (消息) 和客戶端地址。如果服務器需要做應答，它要給客戶端回發一個數據報。對於數據報的傳送，你應該使用 socket 的 sendto() 和 recvfrom() 方法。儘管傳統的 send() 和 recv() 也可以達到同樣的效果，但是前面的兩個方法對於 UDP 連接而言更普遍。

由於沒有底層的連接，UPD 服務器相對於 TCP 服務器來講實現起來更加簡單。不過，UDP 天生是不可靠的 (因爲通信沒有建立連接，消息可能丟失)。因此需要由你自己來決定該怎樣處理丟失消息的情況。這個已經不在本書討論範圍內了，不過通常來說，如果可靠性對於你程序很重要，你需要藉助於序列號、重試、超時以及一些其他方法來保證。UDP 通常被用在那些對於可靠傳輸要求不是很高的場合。例如，在實時應用如多媒體流以及遊戲領域，無需返回恢復丟失的數據包 (程序只需簡單的忽略它並繼續向前運行)。

UDPServer 類是單線程的，也就是說一次只能爲一個客戶端連接服務。實際使用中，這個無論是對於 UDP 還是 TCP 都不是什麼大問題。如果你想要併發操作，可以實例化一個 ForkingUDPServer 或 ThreadingUDPServer 對象：

```
from socketserver import ThreadingUDPServer

   if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

直接使用 socket 來實現一個 UDP 服務器也不難，下面是一個例子：

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
```

```
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4 通過 CIDR 地址生成對應的 IP 地址集

問題

你有一個 CIDR 網絡地址比如“123.45.67.89/27”，你想將其轉換成它所代表的所有 IP （比如，“123.45.67.64”, “123.45.67.65”, ···, “123.45.67.95”））

解決方案

可以使用 ipaddress 模塊很容易的實現這樣的計算。例如：

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...     print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
```

```
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

Network 也允許像數組一樣的索引取值，例如：

```
>>> net.num_addresses
32
>>> net[0]

IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

另外，你還可以執行網絡成員檢查之類的操作：

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
>>>
```

一個 IP 地址和網絡地址能通過一個 IP 接口來指定，例如：

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

討論

ipaddress 模塊有很多類可以表示 IP 地址、網絡和接口。當你需要操作網絡地址（比如解析、打印、驗證等）的時候會很有用。

要注意的是，ipaddress 模塊跟其他一些和網絡相關的模塊比如 socket 庫交集很少。所以，你不能使用 IPv4Address 的實例來代替一個地址字符串，你首先得顯式的使用 str() 轉換它。例如：

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

更多相關內容，請參考 An Introduction to the ipaddress Module

11.5 創建一個簡單的 REST 接口

問題

你想使用一個簡單的 REST 接口通過網絡遠程控制或訪問你的應用程序，但是你又不想自己去安裝一個完整的 web 框架。

解決方案

構建一個 REST 風格的接口最簡單的方法是創建一個基於 WSGI 標準 (PEP 3333) 的很小的庫，下面是一個例子：

```
# resty.py

import cgi

def notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method,path), notfound_404)
        return handler(environ, start_response)

    def register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return function
```

爲了使用這個調度器，你只需要編寫不同的處理器，就像下面這樣：

```
import time

_hello_resp = '''\
<html>
  <head>
     <title>Hello {name}</title>
   </head>
   <body>
     <h1>Hello {name}!</h1>
   </body>
</html>'''

def hello_world(environ, start_response):
    start_response('200 OK', [ ('Content-type','text/html')])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
  <year>{t.tm_year}</year>
  <month>{t.tm_mon}</month>
  <day>{t.tm_mday}</day>
  <hour>{t.tm_hour}</hour>
  <minute>{t.tm_min}</minute>
  <second>{t.tm_sec}</second>
</time>'''

def localtime(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

要測試下這個服務器，你可以使用一個瀏覽器或 `urllib` 和它交互。例如：

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
     <title>Hello Guido</title>
   </head>
   <body>
     <h1>Hello Guido!</h1>
   </body>
</html>

>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
  <second>17</second>
</time>
>>>
```

討論

在編寫 REST 接口時，通常都是服務於普通的 HTTP 請求。但是跟那些功能完整的網站相比，你通常只需要處理數據。這些數據以各種標準格式編碼，比如 XML、JSON 或 CSV。儘管程序看上去很簡單，但是以這種方式提供的 API 對於很多應用程序來講是非常有用的。

例如，長期運行的程序可能會使用一個 REST API 來實現監控或診斷。大數據應用程序可以使用 REST 來構建一個數據查詢或提取系統。REST 還能用來控制硬件設備比如機器人、傳感器、工廠或燈泡。更重要的是，REST API 已經被大量客戶端編程環境所支持，比如 Javascript, Android, iOS 等。因此，利用這種接口可以讓你開發出更加複雜的應用程序。

爲了實現一個簡單的 REST 接口，你只需讓你的程序代碼滿足 Python 的 WSGI 標準即可。WSGI 被標準庫支持，同時也被絕大部分第三方 web 框架支持。因此，如果你的代碼遵循這個標準，在後面的使用過程中就會更加的靈活！

在 WSGI 中，你可以像下面這樣約定的方式以一個可調用對象形式來實現你的程序。

```
import cgi

def wsgi_app(environ, start_response):
    pass
```

environ 屬性是一個字典，包含了從 web 服務器如 Apache[參考 Internet RFC

3875] 提供的 CGI 接口中獲取的值。要將這些不同的值提取出來，你可以像這麼這樣寫：

```
def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Parse the query parameters
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
```

我們展示了一些常見的值。environ['REQUEST_METHOD'] 代表請求類型如 GET、POST、HEAD 等。environ['PATH_INFO'] 表示被請求資源的路徑。調用 cgi.FieldStorage() 可以從請求中提取查詢參數並將它們放入一個類字典對象中以便後面使用。

start_response 參數是一個爲了初始化一個請求對象而必須被調用的函數。第一個參數是返回的 HTTP 狀態值，第二個參數是一個 (名, 值) 元組列表，用來構建返回的 HTTP 頭。例如：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
```

爲了返回數據，一個 WSGI 程序必須返回一個字節字符串序列。可以像下面這樣使用一個列表來完成：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

或者，你還可以使用 yield：

```
def wsgi_app(environ, start_response):
    pass
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

這裏要強調的一點是最後返回的必須是字節字符串。如果返回結果包含文本字符串，必須先將其編碼成字節。當然，並沒有要求你返回的一定是文本，你可以很輕鬆的編寫一個生成圖片的程序。

儘管 WSGI 程序通常被定義成一個函數，不過你也可以使用類實例來實現，只要它實現了合適的 __call__() 方法。例如：

```
class WSGIApplication:
    def __init__(self):
        ...
```

```
    def __call__(self, environ, start_response)
        ...
```

我們已經在上面使用這種技術創建 PathDispatcher 類。這個分發器僅僅只是管理一個字典，將 (方法, 路徑) 對映射到處理器函數上面。當一個請求到來時，它的方法和路徑被提取出來，然後被分發到對應的處理器上面去。另外，任何查詢變量會被解析後放到一個字典中，以 environ['params'] 形式存儲。後面這個步驟太常見，所以建議你在分發器裏面完成，這樣可以省掉很多重複代碼。使用分發器的時候，你只需簡單的創建一個實例，然後通過它註冊各種 WSGI 形式的函數。編寫這些函數應該超級簡單了，只要你遵循 start_response() 函數的編寫規則，並且最後返回字節字符串即可。

當編寫這種函數的時候還需注意的一點就是對於字符串模板的使用。沒人願意寫那種到處混合着 print() 函數、XML 和大量格式化操作的代碼。我們上面使用了三引號包含的預先定義好的字符串模板。這種方式的可以讓我們很容易的在以後修改輸出格式 (只需要修改模板本身，而不用動任何使用它的地方)。

最後，使用 WSGI 還有一個很重要的部分就是沒有什麼地方是針對特定 web 服務器的。因爲標準對於服務器和框架是中立的，你可以將你的程序放入任何類型服務器中。我們使用下面的代碼測試測試本節代碼：

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    pass

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

上面代碼創建了一個簡單的服務器，然後你就可以來測試下你的實現是否能正常工作。最後，當你準備進一步擴展你的程序的時候，你可以修改這個代碼，讓它可以爲特定服務器工作。

WSGI 本身是一個很小的標準。因此它並沒有提供一些高級的特性比如認證、cookies、重定向等。這些你自己實現起來也不難。不過如果你想要更多的支持，可以考慮第三方庫，比如 WebOb 或者 Paste

11.6 通過 XML-RPC 實現簡單的遠程調用

問題

你想找到一個簡單的方式去執行運行在遠程機器上面的 Python 程序中的函數或方法。

解決方案

實現一個遠程方法調用的最簡單方式是使用 XML-RPC。下面我們演示一下一個實現了鍵-值存儲功能的簡單服務器：

```
from xmlrpc.server import SimpleXMLRPCServer

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

# Example
if __name__ == '__main__':
    kvserv = KeyValueServer(('', 15000))
    kvserv.serve_forever()
```

下面我們從一個客戶端機器上面來訪問服務器：

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
```

```
>>> s.exists('spam')
False
>>>
```

討論

XML-RPC 可以讓我們很容易的構造一個簡單的遠程調用服務。你所需要做的僅僅是創建一個服務器實例，通過它的方法 `register_function()` 來註冊函數，然後使用方法 `serve_forever()` 啓動它。在上面我們將這些步驟放在一起寫到一個類中，不夠這並不是必須的。比如你還可以像下面這樣創建一個服務器：

```
from xmlrpc.server import SimpleXMLRPCServer
def add(x,y):
    return x+y

serv = SimpleXMLRPCServer(('', 15000))
serv.register_function(add)
serv.serve_forever()
```

XML-RPC 暴露出來的函數只能適用於部分數據類型，比如字符串、整形、列表和字典。對於其他類型就得需要做些額外的功課了。例如，如果你想通過 XML-RPC 傳遞一個對象實例，實際上只有他的實例字典被處理：

```
>>> class Point:
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...
>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>
```

類似的，對於二進制數據的處理也跟你想象的不太一樣：

```
>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>

>>> _.data
b'Hello World'
>>>
```

一般來講，你不應該將 XML-RPC 服務以公共 API 的方式暴露出來。對於這種情況，通常分佈式應用程序會是一個更好的選擇。

XML-RPC 的一個缺點是它的性能。`SimpleXMLRPCServer` 的實現是單線程的，所以它不適合於大型程序，儘管我們在 11.2 小節中演示過它是可以通過多線程來執行的。

另外，由於 XML-RPC 將所有數據都序列化爲 XML 格式，所以它會比其他的方式運行的慢一些。但是它也有優點，這種方式的編碼可以被絕大部分其他編程語言支持。通過使用這種方式，其他語言的客戶端程序都能訪問你的服務。

雖然 XML-RPC 有很多缺點，但是如果你需要快速構建一個簡單遠程過程調用系統的話，它仍然值得去學習的。有時候，簡單的方案就已經足夠了。

11.7 在不同的 Python 解釋器之間交互

問題

你在不同的機器上面運行着多個 Python 解釋器實例，並希望能夠在這些解釋器之間通過消息來交換數據。

解決方案

通過使用 multiprocessing.connection 模塊可以很容易的實現解釋器之間的通信。下面是一個簡單的應答服務器例子：

```
from multiprocessing.connection import Listener
import traceback

def echo_client(conn):
    try:
        while True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while True:
        try:
            client = serv.accept()

            echo_client(client)
        except Exception:
            traceback.print_exc()

echo_server(('', 25000), authkey=b'peekaboo')
```

然後客戶端連接服務器併發送消息的簡單示例：

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
```

```
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

跟底層 socket 不同的是，每個消息會完整保存（每一個通過 send() 發送的對象能通過 recv() 來完整接受）。另外，所有對象會通過 pickle 序列化。因此，任何兼容 pickle 的對象都能在此連接上面被髮送和接受。

討論

目前有很多用來實現各種消息傳輸的包和函數庫，比如 ZeroMQ、Celery 等。你還有另外一種選擇就是自己在底層 socket 基礎之上來實現一個消息傳輸層。但是你想要簡單一點的方案，那麼這時候 multiprocessing.connection 就派上用場了。僅僅使用一些簡單的語句即可實現多個解釋器之間的消息通信。

如果你的解釋器運行在同一臺機器上面，那麼你可以使用另外的通信機制，比如 Unix 域套接字或者是 Windows 命名管道。要想使用 UNIX 域套接字來創建一個連接，只需簡單的將地址改寫一個文件名即可：

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

要想使用 Windows 命名管道來創建連接，只需像下面這樣使用一個文件名：

```
s = Listener(r'\\.\pipe\myconn', authkey=b'peekaboo')
```

一個通用準則是，你不要使用 multiprocessing 來實現一個對外的公共服務。Client() 和 Listener() 中的 authkey 參數用來認證發起連接的終端用戶。如果密鑰不對會產生一個異常。此外，該模塊最適合用來建立長連接（而不是大量的短連接），例如，兩個解釋器之間啓動後就開始建立連接並在處理某個問題過程中會一直保持連接狀態。

如果你需要對底層連接做更多的控制，比如需要支持超時、非阻塞 I/O 或其他類似的特性，你最好使用另外的庫或者是在高層 socket 上來實現這些特性。

11.8 實現遠程方法調用

問題

你想在一個消息傳輸層如 sockets、multiprocessing connections 或 ZeroMQ 的基礎之上實現一個簡單的遠程過程調用（RPC）。

解決方案

將函數請求、參數和返回值使用 pickle 編碼後，在不同的解釋器直接傳送 pickle 字節字符串，可以很容易的實現 RPC。下面是一個簡單的 PRC 處理器，可以被整合到一個服務器中去：

```
# rpcserver.py

import pickle
class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = pickle.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(pickle.dumps(r))
                except Exception as e:
                    connection.send(pickle.dumps(e))
        except EOFError:
             pass
```

要使用這個處理器，你需要將它加入到一個消息服務器中。你有很多種選擇，但是使用 multiprocessing 庫是最簡單的。下面是一個 RPC 服務器例子：

```
from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Some remote functions
def add(x, y):
    return x + y

def sub(x, y):
    return x - y
```

```
# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

爲了從一個遠程客戶端訪問服務器，你需要創建一個對應的用來傳送請求的 RPC 代理類。例如

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

要使用這個代理類，你需要將其包裝到一個服務器的連接上面，例如：

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)

5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

要注意的是很多消息層（比如 `multiprocessing` ）已經使用 pickle 序列化了數據。如果是這樣的話，對 `pickle.dumps()` 和 `pickle.loads()` 的調用要去掉。

討論

`RPCHandler` 和 `RPCProxy` 的基本思路是很比較簡單的。如果一個客戶端想要調用一個遠程函數，比如 `foo(1, 2, z=3)` ，代理類創建一個包含了函數名和參數的元組

`('foo', (1, 2), {'z': 3})`。這個元組被 pickle 序列化後通過網絡連接發生出去。這一步在 `RPCProxy` 的 `__getattr__()` 方法返回的 `do_rpc()` 閉包中完成。服務器接收後通過 pickle 反序列化消息，查找函數名看看是否已經註冊過，然後執行相應的函數。執行結果 (或異常) 被 pickle 序列化後返回發送給客戶端。我們的實例需要依賴 `multiprocessing` 進行通信。不過，這種方式可以適用於其他任何消息系統。例如，如果你想在 ZeroMQ 之上實習 RPC，僅僅只需要將連接對象換成合適的 ZeroMQ 的 socket 對象即可。

由於底層需要依賴 pickle，那麼安全問題就需要考慮了 (因爲一個聰明的黑客可以創建特定的消息，能夠讓任意函數通過 pickle 反序列化後被執行)。因此你永遠不要允許來自不信任或未認證的客戶端的 RPC。特別是你絕對不要允許來自 Internet 的任意機器的訪問，這種只能在內部被使用，位於防火牆後面並且不要對外暴露。

作爲 pickle 的替代，你也許可以考慮使用 JSON、XML 或一些其他的編碼格式來序列化消息。例如，本機實例可以很容易的改寫成 JSON 編碼方案。還需要將 `pickle.loads()` 和 `pickle.dumps()` 替換成 `json.loads()` 和 `json.dumps()` 即可：

```
# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = json.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(json.dumps(r))
                except Exception as e:
                    connection.send(json.dumps(str(e)))
        except EOFError:
             pass

# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
```

```
            return result
        return do_rpc
```

實現 RPC 的一個比較複雜的問題是如何去處理異常。至少，當方法產生異常時服務器不應該奔潰。因此，返回給客戶端的異常所代表的含義就要好好設計了。如果你使用 pickle，異常對象實例在客戶端能被反序列化並拋出。如果你使用其他的協議，那得想想另外的方法了。不過至少，你應該在響應中返回異常字符串。我們在 JSON 的例子中就是使用的這種方式。

對於其他的 RPC 實現例子，我推薦你看看在 XML-RPC 中使用的 SimpleXMLRPCServer 和 ServerProxy 的實現，也就是 11.6 小節中的內容。

11.9 簡單的客戶端認證

問題

你想在分佈式系統中實現一個簡單的客戶端連接認證功能，又不想像 SSL 那樣的複雜。

解決方案

可以利用 hmac 模塊實現一個連接握手，從而實現一個簡單而高效的認證過程。下面是代碼示例：

```
import hmac
import os

def client_authenticate(connection, secret_key):
    '''
    Authenticate client to a remote service.
    connection represents a network connection.
    secret_key is a key known only to both client/server.
    '''
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def server_authenticate(connection, secret_key):
    '''
    Request client authentication.
    '''
    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest,response)
```

基本原理是當連接建立後，服務器給客戶端發送一個隨機的字節消息（這裏例子中使用了 os.urandom() 返回值）。客戶端和服務器同時利用 hmac 和一個只有雙方知道的密鑰來計算出一個加密哈希值。然後客戶端將它計算出的摘要發送給服務器，服務器通過比較這個值和自己計算的是否一致來決定接受或拒絕連接。摘要的比較需要使用 hmac.compare_digest() 函數。使用這個函數可以避免遭到時間分析攻擊，不要用簡單的比較操作符（==）。爲了使用這些函數，你需要將它集成到已有的網絡或消息代碼中。例如，對於 sockets，服務器代碼應該類似下面：

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
def echo_handler(client_sock):
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
        return
    while True:

        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        c,a = s.accept()
        echo_handler(c)

echo_server(('', 18000))

Within a client, you would do this:

from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'

s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 18000))
client_authenticate(s, secret_key)
s.send(b'Hello World')
resp = s.recv(1024)
```

討論

hmac 認證的一個常見使用場景是內部消息通信系統和進程間通信。例如，如果你編寫的系統涉及到一個集羣中多個處理器之間的通信，你可以使用本節方案來確保只有被允許的進程之間才能彼此通信。事實上，基於 hmac 的認證被 multiprocessing 模

塊使用來實現子進程直接的通信。

還有一點需要強調的是連接認證和加密是兩碼事。認證成功之後的通信消息是以明文形式發送的，任何人只要想監聽這個連接線路都能看到消息（儘管雙方的密鑰不會被傳輸）。

hmac 認證算法基於哈希函數如 MD5 和 SHA-1，關於這個在 IETF RFC 2104 中有詳細介紹。

11.10 在網絡服務中加入 SSL

問題

你想實現一個基於 sockets 的網絡服務，客戶端和服務器通過 SSL 協議認證並加密傳輸的數據。

解決方案

ssl 模塊能爲底層 socket 連接添加 SSL 的支持。ssl.wrap_socket() 函數接受一個已存在的 socket 作爲參數並使用 SSL 層來包裝它。例如，下面是一個簡單的應答服務器，能在服務器端爲所有客戶端連接做認證。

```
from socket import socket, AF_INET, SOCK_STREAM
import ssl

KEYFILE = 'server_key.pem'   # Private key of the server
CERTFILE = 'server_cert.pem' # Server certificate (given to client)

def echo_client(s):
    while True:
        data = s.recv(8192)
        if data == b'':
            break
        s.send(data)
    s.close()
    print('Connection closed')

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(1)

    # Wrap with an SSL layer requiring client certs
    s_ssl = ssl.wrap_socket(s,
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            server_side=True
                            )
    # Wait for connections
```

```
    while True:
        try:
            c,a = s_ssl.accept()
            print('Got connection', c, a)
            echo_client(c)
        except Exception as e:
            print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))
```

下面我們演示一個客戶端連接服務器的交互例子。客戶端會請求服務器來認證並確認連接：

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
                cert_reqs=ssl.CERT_REQUIRED,
                ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>
```

這種直接處理底層 socket 方式有個問題就是它不能很好的跟標準庫中已存在的網絡服務兼容。例如，絕大部分服務器代碼（HTTP、XML-RPC 等）實際上是基於 socketserver 庫的。客戶端代碼在一個較高層上實現。我們需要另外一種稍微不同的方式來將 SSL 添加到已存在的服務中：

首先，對於服務器而言，可以通過像下面這樣使用一個 mixin 類來添加 SSL：

```
import ssl

class SSLMixin:
'''
Mixin class that adds support for SSL to existing servers based
on the socketserver module.
'''
def __init__(self, *args,
             keyfile=None, certfile=None, ca_certs=None,
             cert_reqs=ssl.CERT_NONE,
             **kwargs):
    self._keyfile = keyfile
    self._certfile = certfile
    self._ca_certs = ca_certs
    self._cert_reqs = cert_reqs
    super().__init__(*args, **kwargs)

def get_request(self):
```

```
    client, addr = super().get_request()
    client_ssl = ssl.wrap_socket(client,
                                 keyfile = self._keyfile,
                                 certfile = self._certfile,
                                 ca_certs = self._ca_certs,
                                 cert_reqs = self._cert_reqs,
                                 server_side = True)
    return client_ssl, addr
```

爲了使用這個 mixin 類，你可以將它跟其他服務器類混合。例如，下面是定義一個基於 SSL 的 XML-RPC 服務器例子：

```
# XML-RPC server with SSL

from xmlrpc.server import SimpleXMLRPCServer

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

Here's the XML-RPC server from Recipe 11.6 modified only slightly to use SSL:

import ssl
from xmlrpc.server import SimpleXMLRPCServer
from sslmixin import SSLMixin

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True, **kwargs)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)
```

```
    def serve_forever(self):
        self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem'     # Private key of the server
    CERTFILE='server_cert.pem'   # Server certificate
    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE)
    kvserv.serve_forever()
```

使用這個服務器時，你可以使用普通的 xmlrpc.client 模塊來連接它。只需要在 URL 中指定 https: 即可，例如：

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo','bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

對於 SSL 客戶端來講一個比較複雜的問題是如何確認服務器證書或爲服務器提供客戶端認證（比如客戶端證書）。不幸的是，暫時還沒有一個標準方法來解決這個問題，需要自己去研究。不過，下面給出一個例子，用來建立一個安全的 XML-RPC 連接來確認服務器證書：

```
from xmlrpc.client import SafeTransport, ServerProxy
import ssl

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if certfile:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def make_connection(self, host):
        # Items in the passed dictionary are passed as keyword
        # arguments to the http.client.HTTPSConnection() constructor.
```

```
        # The context argument allows an ssl.SSLContext instance to
        # be passed with information about the SSL configuration
        s = super().make_connection((host, {'context': self._ssl_context}))

        return s

# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem'),
                allow_none=True)
```

服務器將證書發送給客戶端，客戶端來確認它的合法性。這種確認可以是相互的。如果服務器想要確認客戶端，可以將服務器啓動代碼修改如下：

```
if __name__ == '__main__':
    KEYFILE='server_key.pem'    # Private key of the server
    CERTFILE='server_cert.pem' # Server certificate
    CA_CERTS='client_cert.pem' # Certificates of accepted clients

    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            ca_certs=CA_CERTS,
                            cert_reqs=ssl.CERT_REQUIRED,
                            )
    kvserv.serve_forever()
```

爲了讓 XML-RPC 客戶端發送證書，修改 ServerProxy 的初始化代碼如下：

```
# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem',
                                                  'client_cert.pem',
                                                  'client_key.pem'),
                allow_none=True)
```

討論

試着去運行本節的代碼能測試你的系統配置能力和理解 SSL。可能最大的挑戰是如何一步步的獲取初始配置 key、證書和其他所需依賴。

我解釋下到底需要啥，每一個 SSL 連接終端一般都會有一個私鑰和一個簽名證書文件。這個證書包含了公鑰並在每一次連接的時候都會發送給對方。對於公共服務器，它們的證書通常是被權威證書機構比如 Verisign、Equifax 或其他類似機構（需要付費的）簽名過的。爲了確認服務器簽名，客戶端回保存一份包含了信任授權機構的證書列表文件。例如，web 瀏覽器保存了主要的認證機構的證書，並使用它來爲每一個 HTTPS 連接確認證書的合法性。對本小節示例而言，只是爲了測試，我們可以創建自簽名的證書，下面是主要步驟：

bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem -keyout server_key.pem

Generating a 1024 bit RSA private key ……++++++ …++++++

writing new private key to ‘server_key.pem’

You are about to be asked to enter information that will be incorporated into your certificate request. What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter ‘.’, the field will be left blank.

Country Name (2 letter code) [AU]:US State or Province Name (full name) [Some-State]:Illinois Locality Name (eg, city) []:Chicago Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabeaz, LLC Organizational Unit Name (eg, section) []: Common Name (eg, YOUR name) []:localhost Email Address []: bash %

在創建證書的時候，各個值的設定可以是任意的，但是”Common Name**“的值通常要包含服務器的** DNS **主機名。如果你只是在本機測試，那麼就使用”**localhost**“，否則使用服務器的域名。**

—–BEGIN RSA PRIVATE KEY —– MIICXQIBAAKBgQCZrCN-LoEyAKF+f9UNcFaz5Osa6jf7qkbUl8si5xQrY3ZYC7juu nL1dZLn/ VbE-FIITaUOgvBtPv1qUWTJGwga62VSG1oFE0ODIx3g2Nh4sRf+rySsx2 L4442nx0z4O5vJQ7k6eRNHAZUUnCL50+YvjyLyt7ryLSjSuKhCcJsbZgPwIDAQAB AoGAB5evrr7eyL4160tM5rHTeATlaLY3UBOe5Z8XN8Z6gLiB/ ucSX9AysviVD/6F 3oD6z2aL8jbeJc1vHqjt0dC2dwwm32vVl8mRdyoAsQpWmiqXrkvP4Bsl04Vp Qt8xNSW9SFhceL3LEvw9M8i9MV39viih1ILyH8OuHdvJyFECQQDLEjl2d2ppxND9 PoLqVFAirDfX2JnLTdWbc+M11a9Jdn3hKF8TcxfEnFVs5Gav1MusicY5KB0ylYPb YbTvqKc7AkEAwbnRBO2VYEZsJZp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMcigL7v WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBAKDdkv+xoW6gJx42Auc2WzTcUHCA eXR/+BLpPrhKykzbvOQ8YvS5W764SUO1u1LWs3G+wnRMvrRvlMCZKgggBjkCQQCG Jewto2+a+WkOKQXrNNScCDE5aPTmZQc5waCYq4UmCZQcOjkUOiN3ST1U5iuxRqfb V/ yX6fw0qh+fLWtkOs/ JAkA+okMSxZwqRtfgOFGBfwQ8/ iKrnizeanTQ3L6scFXI CHZXdJ3XQ6qUmNxNn7iJ7S/ LDawo1QfWkCfD9FYoxBlg —–END RSA PRIVATE KEY—–

服務器證書文件 server_cert.pem **內容類似下面這樣：**

—–BEGIN CERTIFICATE—– MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQ BAYTAlVTMREwDwYDVQQIEwhJbGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIG A1UEChMLRGFiZWF6LCBMTEMxEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0xMzAxMTEx ODQyMjdaFw0xNDAxMTExODQyMjdaMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwh bGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMx EjAQBgNVBAMTCWxvY2FsaG9zdDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA mawjS6BMgChfn/VDXBWs+TrGuo3+6pG1JfLIucUK2N2WAu47rpy9XWS5/1WxBSCE

2lDoLwbT79alFkyRsIGutlUhtaBRNDgyMd4NjYeLEX/
q8krMdi+OONp8dM+DubyU

O5OnkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAaOBwTCBvjAdBgNV
HQ4EFgQUrtoLHHgXiDZTr26NMmgKJLJLFtIwgY4GA1UdIwSBhjCBg4AUrtoLHHgX
iDZTr26NMmgKJLJLFtKhYKReMFwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhJbGxp
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWF6LCBMTEMxEjAQ
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAf8wDQYJKoZI
hvcNAQEFBQADgYEAFci+dqvMG4xF8UTnbGVvZJPIzJDRee6Nbt6AHQo9pOdAIMAu
WsGCplSOaDNdKKzl+b2UT2Zp3AIW4Qd51bouSNnR4M/
gnr9ZD1ZctFd3jS+C5XRp D3vvcW5lAnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/
06NALGDflrrOwxF3Y= —–END CERTIFICATE—–

在服務器端代碼中，私鑰和證書文件會被傳給 SSL 相關的包裝函數。證書來自於客戶端，私鑰應該在保存在服務器中，並加以安全保護。

在客戶端代碼中，需要保存一個合法證書授權文件來確認服務器證書。如果你沒有這個文件，你可以在客戶端複製一份服務器的證書並使用它來確認。連接建立後，服務器會提供它的證書，然後你就能使用已經保存的證書來確認它是否正確。

服務器也能選擇是否要確認客戶端的身份。如果要這樣做的話，客戶端需要有自己的私鑰和認證文件。服務器也需要保存一個被信任證書授權文件來確認客戶端證書。

如果你要在真實環境中爲你的網絡服務加上 SSL 的支持，這小節只是一個入門介紹而已。你還應該參考其他的文檔，做好花費不少時間來測試它正常工作的準備。反正，就是得慢慢折騰吧 ~ ^_^

11.11 進程間傳遞 Socket 文件描述符

問題

你有多個 Python 解釋器進程在同時運行，你想將某個打開的文件描述符從一個解釋器傳遞給另外一個。比如，假設有個服務器進程相應連接請求，但是實際的相應邏輯是在另一個解釋器中執行的。

解決方案

爲了在多個進程中傳遞文件描述符，你首先需要將它們連接到一起。在 Unix 機器上，你可能需要使用 Unix 域套接字，而在 windows 上面你需要使用命名管道。不過你無需真的需要去操作這些底層，通常使用 `multiprocessing` 模塊來創建這樣的連接會更容易一些。

一旦一個連接被創建，你可以使用 `multiprocessing.reduction` 中的 `send_handle()` 和 `recv_handle()` 函數在不同的處理器直接傳遞文件描述符。下面的例子演示了最基本的用法：

```
import multiprocessing
from multiprocessing.reduction import recv_handle, send_handle
import socket
```

```
def worker(in_p, out_p):
    out_p.close()
    while True:
        fd = recv_handle(in_p)
        print('CHILD: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as␣
↪s:
            while True:
                msg = s.recv(1024)
                if not msg:
                    break
                print('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1,c2))
    worker_p.start()

    server_p = multiprocessing.Process(target=server,
                  args=(('', 15000), c1, c2, worker_p.pid))
    server_p.start()

    c1.close()
    c2.close()
```

在這個例子中，兩個進程被創建並通過一個 multiprocessing 管道連接起來。服務器進程打開一個 socket 並等待客戶端連接請求。工作進程僅僅使用 recv_handle() 在管道上面等待接收一個文件描述符。當服務器接收到一個連接，它將產生的 socket 文件描述符通過 send_handle() 傳遞給工作進程。工作進程接收到 socket 後向客戶端迴應數據，然後此次連接關閉。

如果你使用 Telnet 或類似工具連接到服務器，下面是一個演示例子：

bash % python3 passfd.py SERVER: Got connection from (‘127.0.0.1’ , 55543) CHILD: GOT FD 7 CHILD: RECV b’ Hellorn’ CHILD: RECV b’ Worldrn’

此例最重要的部分是服務器接收到的客戶端 socket 實際上被另外一個不同的進程

處理。服務器僅僅只是將其轉手並關閉此連接，然後等待下一個連接。

討論

對於大部分程序員來講在不同進程之間傳遞文件描述符好像沒什麼必要。但是，有時候它是構建一個可擴展系統的很有用的工具。例如，在一個多核機器上面，你可以有多個 Python 解釋器實例，將文件描述符傳遞給其它解釋器來實現負載均衡。

`send_handle()` 和 `recv_handle()` 函數只能夠用於 `multiprocessing` 連接。使用它們來代替管道的使用（參考 11.7 節），只要你使用的是 Unix 域套接字或 Windows 管道。例如，你可以讓服務器和工作者各自以單獨的程序來啓動。下面是服務器的實現例子：

```
# servermp.py
from multiprocessing.connection import Listener
from multiprocessing.reduction import send_handle
import socket

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)

        send_handle(worker, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

運行這個服務器，只需要執行 *python3 servermp.py /tmp/servconn 15000*，下面是相應的工作者代碼：

```
# workermp.py

from multiprocessing.connection import Client
```

```
from multiprocessing.reduction import recv_handle
import os
from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
    while True:
        fd = recv_handle(serv)
        print('WORKER: GOT FD', fd)
        with socket(AF_INET, SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])
```

要運行工作者，執行執行命令 *python3 workermp.py /tmp/servconn* . 效果跟使用 Pipe() 例子是完全一樣的。文件描述符的傳遞會涉及到 UNIX 域套接字的創建和套接字的 sendmsg() 方法。不過這種技術並不常見，下面是使用套接字來傳遞描述符的另外一種實現：

```
# server.py
import socket

import struct

def send_fd(sock, fd):
    '''
    Send a single file descriptor.
    '''
    sock.sendmsg([b'x'],
                 [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i',␣
↪fd))])
    ack = sock.recv(2)
    assert ack == b'OK'

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    work_serv.bind(work_address)
```

```
    work_serv.listen(1)
    worker, addr = work_serv.accept()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('',port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_fd(worker, client.fileno())
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

下面是使用套接字的工作者實現：

```
# worker.py
import socket
import struct

def recv_fd(sock):
    '''
    Receive a single file descriptor
    '''
    msg, ancdata, flags, addr = sock.recvmsg(1,
                                     socket.CMSG_LEN(struct.calcsize('i')))

    cmsg_level, cmsg_type, cmsg_data = ancdata[0]
    assert cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS
    sock.sendall(b'OK')

    return struct.unpack('i', cmsg_data)[0]

def worker(server_address):
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address)
    while True:
        fd = recv_fd(serv)
        print('WORKER: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as␣
↪client:
            while True:
```

```
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])
```

如果你想在你的程序中傳遞文件描述符，建議你參閱其他一些更加高級的文檔，比如 Unix Network Programming by W. Richard Stevens (Prentice Hall, 1990). 在 Windows 上傳遞文件描述符跟 Unix 是不一樣的，建議你研究下 multiprocessing.reduction 中的源代碼看看其工作原理。

11.12 理解事件驅動的 IO

問題

你應該已經聽過基於事件驅動或異步 I/O 的包，但是你還不能完全理解它的底層到底是怎樣工作的，或者是如果使用它的話會對你的程序產生什麼影響。

解決方案

事件驅動 I/O 本質上來講就是將基本 I/O 操作（比如讀和寫）轉化爲你程序需要處理的事件。例如，當數據在某個 socket 上被接受後，它會轉換成一個 receive 事件，然後被你定義的回調方法或函數來處理。作爲一個可能的起始點，一個事件驅動的框架可能會以一個實現了一系列基本事件處理器方法的基類開始：

```
class EventHandler:
    def fileno(self):
        'Return the associated file descriptor'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed'
        return False

    def handle_receive(self):
        'Perform the receive operation'
        pass

    def wants_to_send(self):
```

```
        'Return True if sending is requested'
        return False

    def handle_send(self):
        'Send outgoing data'
        pass
```

這個類的實例作爲插件被放入類似下面這樣的事件循環中：

```
import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()
```

事件循環的關鍵部分是 `select()` 調用，它會不斷輪詢文件描述符從而激活它。在調用 `select()` 之前，時間循環會詢問所有的處理器來決定哪一個想接受或發生。然後它將結果列表提供給 `select()` 。然後 `select()` 返回準備接受或發送的對象組成的列表。然後相應的 `handle_receive()` 或 `handle_send()` 方法被觸發。

編寫應用程序的時候，`EventHandler` 的實例會被創建。例如，下面是兩個簡單的基於 UDP 網絡服務的處理器例子：

```
import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
```

```
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [ UDPTimeServer(('',14000)), UDPEchoServer(('',15000))  ]
    event_loop(handlers)
```

測試這段代碼，試着從另外一個 Python 解釋器連接它：

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'',('localhost',14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello',('localhost',15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

實現一個 TCP 服務器會更加複雜一點，因爲每一個客戶端都要初始化一個新的處理器對象。下面是一個 TCP 應答客戶端例子：

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

    def handle_receive(self):
        client, addr = self.sock.accept()
        # Add the client to the event loop's handler list
        self.handler_list.append(self.client_handler(client, self.handler_
↪list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def fileno(self):
```

```
        return self.sock.fileno()

    def close(self):
        self.sock.close()
        # Remove myself from the event loop's handler list
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False

    def handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192)
        if not data:
            self.close()
        else:
            self.outgoing.extend(data)

if __name__ == '__main__':
   handlers = []
   handlers.append(TCPServer(('',16000), TCPEchoClient, handlers))
   event_loop(handlers)
```

TCP 例子的關鍵點是從處理器中列表增加和刪除客戶端的操作。對每一個連接，一個新的處理器被創建並加到列表中。當連接被關閉後，每個客戶端負責將其從列表中刪除。如果你運行程序並試着用 Telnet 或類似工具連接，它會將你發送的消息回顯給你。並且它能很輕鬆的處理多客戶端連接。

討論

實際上所有的事件驅動框架原理跟上面的例子相差無幾。實際的實現細節和軟件架構可能不一樣，但是在最核心的部分，都會有一個輪詢的循環來檢查活動 socket，並執行響應操作。

事件驅動 I/O 的一個可能好處是它能處理非常大的併發連接，而不需要使用多線程或多進程。也就是說，select() 調用（或其他等效的）能監聽大量的 socket 並響應它們中任何一個產生事件的。在循環中一次處理一個事件，並不需要其他的併發機制。

事件驅動 I/O 的缺點是沒有真正的同步機制。如果任何事件處理器方法阻塞或執行一個耗時計算，它會阻塞所有的處理進程。調用那些並不是事件驅動風格的庫函數也會有問題，同樣要是某些庫函數調用會阻塞，那麼也會導致整個事件循環停止。

對於阻塞或耗時計算的問題可以通過將事件發送個其他單獨的現場或進程來處理。

不過，在事件循環中引入多線程和多進程是比較棘手的，下面的例子演示瞭如何使用 concurrent.futures 模塊來實現：

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Callback that executes when the thread is done
    def _complete(self, callback, r):

        self.pending.append((callback, r.result()))
        self.signal_done_sock.send(b'x')

    # Run a function in a thread pool
    def run(self, func, args=(), kwargs={},*,callback):
        r = self.pool.submit(func, *args, **kwargs)
        r.add_done_callback(lambda r: self._complete(callback, r))

    def wants_to_receive(self):
        return True

    # Run callback functions of completed work
    def handle_receive(self):
        # Invoke all pending callback functions
        for callback, result in self.pending:
            callback(result)
            self.done_sock.recv(1)
        self.pending = []
```

在代碼中，run() 方法被用來將工作提交給回調函數池，處理完成後被激發。實際工作被提交給 ThreadPoolExecutor 實例。不過一個難點是協調計算結果和事件循環，爲了解決它，我們創建了一對 socket 並將其作爲某種信號量機制來使用。當線程池完

成工作後，它會執行類中的 `_complete()` 方法。這個方法再某個 socket 上寫入字節之前會講掛起的回調函數和結果放入隊列中。`fileno()` 方法返回另外的那個 socket。因此，這個字節被寫入時，它會通知事件循環，然後 `handle_receive()` 方法被激活併爲所有之前提交的工作執行回調函數。坦白講，說了這麼多連我自己都暈了。下面是一個簡單的服務器，演示瞭如何使用線程池來實現耗時的計算：

```
# A really bad Fibonacci implementation
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

    def respond(self, result, addr):
        self.sock.sendto(str(result).encode('ascii'), addr)

if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer(('',16000))]
    event_loop(handlers)
```

運行這個服務器，然後試着用其它 Python 程序來測試它：

```
from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print(resp[0])
```

你應該能在不同窗口中重複的執行這個程序，並且不會影響到其他程序，儘管當數字便越來越大時候它會變得越來越慢。

已經閱讀完了這一小節，那麼你應該使用這裏的代碼嗎？也許不會。你應該選擇一個可以完成同樣任務的高級框架。不過，如果你理解了基本原理，你就能理解這些框架所使用的核心技術。作爲對回調函數編程的替代，事件驅動編碼有時候會使用到協程，參考 12.12 小節的一個例子。

11.13 發送與接收大型數組

問題

你要通過網絡連接發送和接受連續數據的大型數組，並儘量減少數據的複製操作。

解決方案

下面的函數利用 memoryviews 來發送和接受大數組：

```
# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]
```

爲了測試程序，首先創建一個通過 socket 連接的服務器和客戶端程序：

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

在客戶端（另外一個解釋器中）：

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

本節的目標是你能通過連接傳輸一個超大數組。這種情況的話，可以通過 array 模塊或 numpy 模塊來創建數組：

```
# Server
>>> import numpy
>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Client
>>> import numpy
>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

```
>>>
```

討論

在數據密集型分佈式計算和平行計算程序中，自己寫程序來實現發送/接受大量數據並不常見。不過，要是你確實想這樣做，你可能需要將你的數據轉換成原始字節，以便給低層的網絡函數使用。你可能還需要將數據切割成多個塊，因爲大部分和網絡相關的函數並不能一次性發送或接受超大數據塊。

一種方法是使用某種機制序列化數據——可能將其轉換成一個字節字符串。不過，這樣最終會創建數據的一個複製。就算你只是零碎的做這些，你的代碼最終還是會有大量的小型複製操作。

本節通過使用內存視圖展示了一些魔法操作。本質上，一個內存視圖就是一個已存在數組的覆蓋層。不僅僅是那樣，內存視圖還能以不同的方式轉換成不同類型來表現數據。這個就是下面這個語句的目的：

```
view = memoryview(arr).cast('B')
```

它接受一個數組 arr 並將其轉換爲一個無符號字節的內存視圖。這個視圖能被傳遞給 socket 相關函數，比如 `socket.send()` 或 `send.recv_into()` 。在內部，這些方法能夠直接操作這個內存區域。例如，`sock.send()` 直接從內存中發生數據而不需要複製。`send.recv_into()` 使用這個內存區域作爲接受操作的輸入緩衝區。

剩下的一個難點就是 socket 函數可能只操作部分數據。通常來講，我們得使用很多不同的 `send()` 和 `recv_into()` 來傳輸整個數組。不用擔心，每次操作後，視圖會通過發送或接受字節數量被切割成新的視圖。新的視圖同樣也是內存覆蓋層。因此，還是沒有任何的複製操作。

這裏有個問題就是接受者必須事先知道有多少數據要被髮送，以便它能預分配一個數組或者確保它能將接受的數據放入一個已經存在的數組中。如果沒辦法知道的話，發送者就得先將數據大小發送過來，然後再發送實際的數組數據。

第十二章：併發編程

對於併發編程, Python 有多種長期支持的方法, 包括多線程, 調用子進程, 以及各種各樣的關於生成器函數的技巧. 這一章將會給出併發編程各種方面的技巧, 包括通用的多線程技術以及並行計算的實現方法.

像經驗豐富的程序員所知道的那樣, 大家擔心併發的程序有潛在的危險. 因此, 本章的主要目標之一是給出更加可信賴和易調試的代碼.

12.1 啓動與停止線程

問題

你要爲需要併發執行的代碼創建/銷燬線程

解決方案

threading 庫可以在單獨的線程中執行任何的在 Python 中可以調用的對象。你可以創建一個 Thread 對象並將你要執行的對象以 target 參數的形式提供給該對象。下面是一個簡單的例子：

```
# Code to execute in an independent thread
import time
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create and launch a thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

當你創建好一個線程對象後，該對象並不會立即執行，除非你調用它的 start() 方法（當你調用 start() 方法時，它會調用你傳遞進來的函數，並把你傳遞進來的參數傳遞給該函數）。Python 中的線程會在一個單獨的系統級線程中執行（比如說一個 POSIX 線程或者一個 Windows 線程），這些線程將由操作系統來全權管理。線程一旦啓動，將獨立執行直到目標函數返回。你可以查詢一個線程對象的狀態，看它是否還在執行：

```
if t.is_alive():
    print('Still running')
else:
    print('Completed')
```

你也可以將一個線程加入到當前線程，並等待它終止：

```
t.join()
```

Python 解釋器直到所有線程都終止前仍保持運行。對於需要長時間運行的線程或者需要一直運行的後臺任務，你應當考慮使用後臺線程。例如：

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

後臺線程無法等待，不過，這些線程會在主線程終止時自動銷燬。除了如上所示的兩個操作，並沒有太多可以對線程做的事情。你無法結束一個線程，無法給它發送信號，無法調整它的調度，也無法執行其他高級操作。如果需要這些特性，你需要自己添加。比如說，如果你需要終止線程，那麼這個線程必須通過編程在某個特定點輪詢來退出。你可以像下邊這樣把線程放入一個類中：

```
class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
c.terminate() # Signal termination
t.join()      # Wait for actual termination (if needed)
```

如果線程執行一些像 I/O 這樣的阻塞操作，那麼通過輪詢來終止線程將使得線程之間的協調變得非常棘手。比如，如果一個線程一直阻塞在一個 I/O 操作上，它就永遠無法返回，也就無法檢查自己是否已經被結束了。要正確處理這些問題，你需要利用超時循環來小心操作線程。例子如下：

```
class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)        # Set timeout period
        while self._running:
            # Perform a blocking I/O operation w/ timeout
            try:
                data = sock.recv(8192)
                break
```

```
            except socket.timeout:
                continue
            # Continued processing
            ...
        # Terminated
        return
```

討論

由於全局解釋鎖（GIL）的原因，Python 的線程被限制到同一時刻只允許一個線程執行這樣一個執行模型。所以，Python 的線程更適用於處理 I/O 和其他需要併發執行的阻塞操作（比如等待 I/O、等待從數據庫獲取數據等等），而不是需要多處理器並行的計算密集型任務。

有時你會看到下邊這種通過繼承 Thread 類來實現的線程：

```
from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = 0
    def run(self):
        while self.n > 0:

            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

c = CountdownThread(5)
c.start()
```

儘管這樣也可以工作，但這使得你的代碼依賴於 threading 庫，所以你的這些代碼只能在線程上下文中使用。上文所寫的那些代碼、函數都是與 threading 庫無關的，這樣就使得這些代碼可以被用在其他的上下文中，可能與線程有關，也可能與線程無關。比如，你可以通過 multiprocessing 模塊在一個單獨的進程中執行你的代碼：

```
import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
```

再次重申，這段代碼僅適用於 CountdownTask 類是以獨立於實際的併發手段（多線程、多進程等等）實現的情況。

12.2 判斷線程是否已經啓動

問題

你已經啓動了一個線程，但是你想知道它是不是真的已經開始運行了。

解決方案

線程的一個關鍵特性是每個線程都是獨立運行且狀態不可預測。如果程序中的其他線程需要通過判斷某個線程的狀態來確定自己下一步的操作，這時線程同步問題就會變得非常棘手。爲了解決這些問題，我們需要使用 threading 庫中的 Event 對象。Event 對象包含一個可由線程設置的信號標誌，它允許線程等待某些事件的發生。在初始情況下，event 對象中的信號標誌被設置爲假。如果有線程等待一個 event 對象，而這個 event 對象的標誌爲假，那麼這個線程將會被一直阻塞直至該標誌爲真。一個線程如果將一個 event 對象的信號標誌設置爲真，它將喚醒所有等待這個 event 對象的線程。如果一個線程等待一個已經被設置爲真的 event 對象，那麼它將忽略這個事件，繼續執行。下邊的代碼展示瞭如何使用 Event 來協調線程的啓動：

```
from threading import Thread, Event
import time

# Code to execute in an independent thread
def countdown(n, started_evt):
    print('countdown starting')
    started_evt.set()
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create the event object that will be used to signal startup
started_evt = Event()

# Launch the thread and pass the startup event
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Wait for the thread to start
started_evt.wait()
print('countdown is running')
```

當你執行這段代碼，“countdown is running”總是顯示在“countdown starting”之後顯示。這是由於使用 event 來協調線程，使得主線程要等到 countdown() 函數輸出啓動信息後，才能繼續執行。

討論

event 對象最好單次使用，就是說，你創建一個 event 對象，讓某個線程等待這個對象，一旦這個對象被設置爲真，你就應該丟棄它。儘管可以通過 clear() 方法來重

置 event 對象，但是很難確保安全地清理 event 對象並對它重新賦值。很可能會發生錯過事件、死鎖或者其他問題（特別是，你無法保證重置 event 對象的代碼會在線程再次等待這個 event 對象之前執行）。如果一個線程需要不停地重複使用 event 對象，你最好使用 Condition 對象來代替。下面的代碼使用 Condition 對象實現了一個週期定時器，每當定時器超時的時候，其他線程都可以監測到：

```
import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
        t.daemon = True

        t.start()

    def run(self):
        '''
        Run the timer and notify waiting threads after each interval
        '''
        while True:
            time.sleep(self._interval)
            with self._cv:
                 self._flag ^= 1
                 self._cv.notify_all()

    def wait_for_tick(self):
        '''
        Wait for the next tick of the timer
        '''
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()

# Two threads that synchronize on the timer
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1
```

```
def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()
```

event 對象的一個重要特點是當它被設置爲真時會喚醒所有等待它的線程。如果你只想喚醒單個線程，最好是使用信號量或者 Condition 對象來替代。考慮一下這段使用信號量實現的代碼：

```
# Worker thread
def worker(n, sema):
    # Wait to be signaled
    sema.acquire()

    # Do some work
    print('Working', n)

# Create some threads
sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.start()
```

運行上邊的代碼將會啓動一個線程池，但是並沒有什麼事情發生。這是因爲所有的線程都在等待獲取信號量。每次信號量被釋放，只有一個線程會被喚醒並執行，示例如下：

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

編寫涉及到大量的線程間同步問題的代碼會讓你痛不欲生。比較合適的方式是使用隊列來進行線程間通信或者每個把線程當作一個 Actor，利用 Actor 模型來控制併發。下一節將會介紹到隊列，而 Actor 模型將在 12.10 節介紹。

12.3 線程間通信

問題

你的程序中有多個線程，你需要在這些線程之間安全地交換信息或數據

解決方案

從一個線程向另一個線程發送數據最安全的方式可能就是使用 queue 庫中的隊列了。創建一個被多個線程共享的 Queue 對象，這些線程通過使用 put() 和 get() 操作來向隊列中添加或者刪除元素。例如：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
# Get some data
        data = in_q.get()
        # Process the data
        ...

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Queue 對象已經包含了必要的鎖，所以你可以通過它在多個線程間多安全地共享數據。當使用隊列時，協調生產者和消費者的關閉問題可能會有一些麻煩。一個通用的解決方法是在隊列中放置一個特殊的值，當消費者讀到這個值的時候，終止執行。例如：

```
from queue import Queue
from threading import Thread

# Object that signals shutdown
_sentinel = object()

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

    # Put the sentinel on the queue to indicate completion
    out_q.put(_sentinel)
```

```
# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Check for termination
        if data is _sentinel:
            in_q.put(_sentinel)
            break

        # Process the data
        ...
```

本例中有一個特殊的地方：消費者在讀到這個特殊值之後立即又把它放回到隊列中，將之傳遞下去。這樣，所有監聽這個隊列的消費者線程就可以全部關閉了。儘管隊列是最常見的線程間通信機制，但是仍然可以自己通過創建自己的數據結構並添加所需的鎖和同步機制來實現線程間通信。最常見的方法是使用 Condition 變量來包裝你的數據結構。下邊這個例子演示瞭如何創建一個線程安全的優先級隊列，如同 1.5 節中介紹的那樣。

```
import heapq
import threading

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

使用隊列來進行線程間通信是一個單向、不確定的過程。通常情況下，你沒有辦法知道接收數據的線程是什麼時候接收到的數據並開始工作的。不過隊列對象提供一些基本完成的特性，比如下邊這個例子中的 task_done() 和 join()：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
```

```
    while running:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Process the data
        ...
        # Indicate completion
        in_q.task_done()

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

# Wait for all produced items to be consumed
q.join()
```

如果一個線程需要在一個“消費者”線程處理完特定的數據項時立即得到通知，你可以把要發送的數據和一個 Event 放到一起使用，這樣“生產者”就可以通過這個 Event 對象來監測處理的過程了。示例如下：

```
from queue import Queue
from threading import Thread, Event

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        # Make an (data, event) pair and hand it to the consumer
        evt = Event()
        out_q.put((data, evt))
        ...
        # Wait for the consumer to process the item
        evt.wait()

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data, evt = in_q.get()
```

```
        # Process the data
        ...
        # Indicate completion
        evt.set()
```

討論

基於簡單隊列編寫多線程程序在多數情況下是一個比較明智的選擇。從線程安全隊列的底層實現來看，你無需在你的代碼中使用鎖和其他底層的同步機制，這些只會把你的程序弄得亂七八糟。此外，使用隊列這種基於消息的通信機制可以被擴展到更大的應用範疇，比如，你可以把你的程序放入多個進程甚至是分佈式系統而無需改變底層的隊列結構。使用線程隊列有一個要注意的問題是，向隊列中添加數據項時並不會複製此數據項，線程間通信實際上是在線程間傳遞對象引用。如果你擔心對象的共享狀態，那你最好只傳遞不可修改的數據結構（如：整型、字符串或者元組）或者一個對象的深拷貝。例如：

```
from queue import Queue
from threading import Thread
import copy

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(copy.deepcopy(data))

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...
```

Queue 對象提供一些在當前上下文很有用的附加特性。比如在創建 Queue 對象時提供可選的 size 參數來限制可以添加到隊列中的元素數量。對於“生產者”與“消費者”速度有差異的情況，爲隊列中的元素數量添加上限是有意義的。比如，一個“生產者”產生項目的速度比“消費者”“消費”的速度快，那麼使用固定大小的隊列就可以在隊列已滿的時候阻塞隊列，以免未預期的連鎖效應擴散整個程序造成死鎖或者程序運行失常。在通信的線程之間進行“流量控制”是一個看起來容易實現起來困難的問題。如果你發現自己曾經試圖通過擺弄隊列大小來解決一個問題，這也許就標誌着你的程序可能存在脆弱設計或者固有的可伸縮問題。get() 和 put() 方法都支持非阻塞方式和設定超時，例如：

```
import queue
q = queue.Queue()
```

```
try:
    data = q.get(block=False)
except queue.Empty:
    ...

try:
    q.put(item, block=False)
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...
```

這些操作都可以用來避免當執行某些特定隊列操作時發生無限阻塞的情況，比如，一個非阻塞的 put() 方法和一個固定大小的隊列一起使用，這樣當隊列已滿時就可以執行不同的代碼。比如輸出一條日誌信息並丟棄。

```
def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)
```

如果你試圖讓消費者線程在執行像 q.get() 這樣的操作時，超時自動終止以便檢查終止標誌，你應該使用 q.get() 的可選參數 timeout ，如下：

```
_running = True

def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Process item
            ...
        except queue.Empty:
            pass
```

最後，有 q.qsize() ，q.full() ，q.empty() 等實用方法可以獲取一個隊列的當前大小和狀態。但要注意，這些方法都不是線程安全的。可能你對一個隊列使用 empty() 判斷出這個隊列爲空，但同時另外一個線程可能已經向這個隊列中插入一個數據項。所以，你最好不要在你的代碼中使用這些方法。

12.4 給關鍵部分加鎖

問題

你需要對多線程程序中的臨界區加鎖以避免競爭條件。

解決方案

要在多線程程序中安全使用可變對象，你需要使用 threading 庫中的 Lock 對象，就像下邊這個例子這樣：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with self._value_lock:
             self._value += delta

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        with self._value_lock:
             self._value -= delta
```

Lock 對象和 with 語句塊一起使用可以保證互斥執行，就是每次只有一個線程可以執行 with 語句包含的代碼塊。with 語句會在這個代碼塊執行前自動獲取鎖，在執行結束後自動釋放鎖。

討論

線程調度本質上是不確定的，因此，在多線程程序中錯誤地使用鎖機制可能會導致隨機數據損壞或者其他的異常行爲，我們稱之爲競爭條件。爲了避免競爭條件，最好只在臨界區（對臨界資源進行操作的那部分代碼）使用鎖。在一些“老的”Python 代碼中，顯式獲取和釋放鎖是很常見的。下邊是一個上一個例子的變種：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
```

```
    '''
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self,delta=1):
        '''
        Decrement the counter with locking
        '''
        self._value_lock.acquire()
        self._value -= delta
        self._value_lock.release()
```

相比於這種顯式調用的方法，with 語句更加優雅，也更不容易出錯，特別是程序員可能會忘記調用 release() 方法或者程序在獲得鎖之後產生異常這兩種情況（使用 with 語句可以保證在這兩種情況下仍能正確釋放鎖）。爲了避免出現死鎖的情況，使用鎖機制的程序應該設定爲每個線程一次只允許獲取一個鎖。如果不能這樣做的話，你就需要更高級的死鎖避免機制，我們將在 12.5 節介紹。在 threading 庫中還提供了其他的同步原語，比如 RLock 和 Semaphore 對象。但是根據以往經驗，這些原語是用於一些特殊的情況，如果你只是需要簡單地對可變對象進行鎖定，那就不應該使用它們。一個 RLock（可重入鎖）可以被同一個線程多次獲取，主要用來實現基於監測對象模式的鎖定和同步。在使用這種鎖的情況下，當鎖被持有時，只有一個線程可以使用完整的函數或者類中的方法。比如，你可以實現一個這樣的 SharedCounter 類：

```
import threading

class SharedCounter:
    '''
    A counter object that can be shared by multiple threads.
    '''
    _lock = threading.RLock()
    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self,delta=1):
        '''
        Increment the counter with locking
        '''
        with SharedCounter._lock:
            self._value += delta

    def decr(self,delta=1):
```

```
        '''
        Decrement the counter with locking
        '''
        with SharedCounter._lock:
            self.incr(-delta)
```

在上邊這個例子中，沒有對每一個實例中的可變對象加鎖，取而代之的是一個被所有實例共享的類級鎖。這個鎖用來同步類方法，具體來說就是，這個鎖可以保證一次只有一個線程可以調用這個類方法。不過，與一個標準的鎖不同的是，已經持有這個鎖的方法在調用同樣使用這個鎖的方法時，無需再次獲取鎖。比如 decr 方法。這種實現方式的一個特點是，無論這個類有多少個實例都只用一個鎖。因此在需要大量使用計數器的情況下內存效率更高。不過這樣做也有缺點，就是在程序中使用大量線程並頻繁更新計數器時會有爭用鎖的問題。信號量對象是一個建立在共享計數器基礎上的同步原語。如果計數器不爲 0，with 語句將計數器減 1，線程被允許執行。with 語句執行結束後，計數器加 1。如果計數器爲 0，線程將被阻塞，直到其他線程結束將計數器加 1。儘管你可以在程序中像標準鎖一樣使用信號量來做線程同步，但是這種方式並不被推薦，因爲使用信號量爲程序增加的複雜性會影響程序性能。相對於簡單地作爲鎖使用，信號量更適用於那些需要在線程之間引入信號或者限制的程序。比如，你需要限制一段代碼的併發訪問量，你就可以像下面這樣使用信號量完成：

```
from threading import Semaphore
import urllib.request

# At most, five threads allowed to run at once
_fetch_url_sema = Semaphore(5)

def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)
```

如果你對線程同步原語的底層理論和實現感興趣，可以參考操作系統相關書籍，絕大多數都有提及。

12.5 防止死鎖的加鎖機制

問題

你正在寫一個多線程程序，其中線程需要一次獲取多個鎖，此時如何避免死鎖問題。

解決方案

在多線程程序中，死鎖問題很大一部分是由於線程同時獲取多個鎖造成的。舉個例子：一個線程獲取了第一個鎖，然後在獲取第二個鎖的時候發生阻塞，那麼這個線程就可能阻塞其他線程的執行，從而導致整個程序假死。解決死鎖問題的一種方案是爲程序中的每一個鎖分配一個唯一的 id，然後只允許按照升序規則來使用多個鎖，這個規則使用上下文管理器是非常容易實現的，示例如下：

```
import threading
from contextlib import contextmanager

# Thread-local state to stored information on locks already acquired
_local = threading.local()

@contextmanager
def acquire(*locks):
    # Sort locks by object identifier
    locks = sorted(locks, key=lambda x: id(x))

    # Make sure lock order of previously acquired locks is not violated
    acquired = getattr(_local,'acquired',[])
    if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
        raise RuntimeError('Lock Order Violation')

    # Acquire all of the locks
    acquired.extend(locks)
    _local.acquired = acquired

    try:
        for lock in locks:
            lock.acquire()
        yield
    finally:
        # Release locks in reverse order of acquisition
        for lock in reversed(locks):
            lock.release()
        del acquired[-len(locks):]
```

如何使用這個上下文管理器呢？你可以按照正常途徑創建一個鎖對象，但不論是單個鎖還是多個鎖中都使用 `acquire()` 函數來申請鎖，示例如下：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()
```

```
t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果你執行這段代碼，你會發現它即使在不同的函數中以不同的順序獲取鎖也沒有發生死鎖。其關鍵在於，在第一段代碼中，我們對這些鎖進行了排序。通過排序，使得不管用戶以什麼樣的順序來請求鎖，這些鎖都會按照固定的順序被獲取。如果有多個 acquire() 操作被嵌套調用，可以通過線程本地存儲（TLS）來檢測潛在的死鎖問題。假設你的代碼是這樣寫的：

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():

    while True:
        with acquire(x_lock):
            with acquire(y_lock):
                print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

如果你運行這個版本的代碼，必定會有一個線程發生崩潰，異常信息可能像這樣：

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_inner
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
  File "deadlock.py", line 15, in acquire
    raise RuntimeError("Lock Order Violation")
```

```
RuntimeError: Lock Order Violation
>>>
```

發生崩潰的原因在於，每個線程都記錄着自己已經獲取到的鎖。acquire() 函數會檢查之前已經獲取的鎖列表，由於鎖是按照升序排列獲取的，所以函數會認爲之前已獲取的鎖的 id 必定小於新申請到的鎖，這時就會觸發異常。

討論

死鎖是每一個多線程程序都會面臨的一個問題（就像它是每一本操作系統課本的共同話題一樣）。根據經驗來講，儘可能保證每一個線程只能同時保持一個鎖，這樣程序就不會被死鎖問題所困擾。一旦有線程同時申請多個鎖，一切就不可預料了。

死鎖的檢測與恢復是一個幾乎沒有優雅的解決方案的擴展話題。一個比較常用的死鎖檢測與恢復的方案是引入看門狗計數器。當線程正常運行的時候會每隔一段時間重置計數器，在沒有發生死鎖的情況下，一切都正常進行。一旦發生死鎖，由於無法重置計數器導致定時器超時，這時程序會通過重啓自身恢復到正常狀態。

避免死鎖是另外一種解決死鎖問題的方式，在進程獲取鎖的時候會嚴格按照對象 id 升序排列獲取，經過數學證明，這樣保證程序不會進入死鎖狀態。證明就留給讀者作爲練習了。避免死鎖的主要思想是，單純地按照對象 id 遞增的順序加鎖不會產生循環依賴，而循環依賴是死鎖的一個必要條件，從而避免程序進入死鎖狀態。

下面以一個關於線程死鎖的經典問題：“哲學家就餐問題”，作爲本節最後一個例子。題目是這樣的：五位哲學家圍坐在一張桌子前，每個人面前有一碗飯和一隻筷子。在這裏每個哲學家可以看做是一個獨立的線程，而每隻筷子可以看做是一個鎖。每個哲學家可以處在靜坐、思考、吃飯三種狀態中的一個。需要注意的是，每個哲學家吃飯是需要兩隻筷子的，這樣問題就來了：如果每個哲學家都拿起自己左邊的筷子，那麼他們五個都只能拿着一隻筷子坐在那兒，直到餓死。此時他們就進入了死鎖狀態。下面是一個簡單的使用死鎖避免機制解決“哲學家就餐問題”的實現：

```
import threading

# The philosopher thread
def philosopher(left, right):
    while True:
        with acquire(left,right):
             print(threading.currentThread(), 'eating')

# The chopsticks (represented by locks)
NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

# Create all of the philosophers
for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                         args=(chopsticks[n],chopsticks[(n+1) % NSTICKS]))
    t.start()
```

最後，要特別注意到，爲了避免死鎖，所有的加鎖操作必須使用 acquire() 函數。如果代碼中的某部分繞過 acquire 函數直接申請鎖，那麼整個死鎖避免機制就不起作用了。

12.6 保存線程的狀態信息

問題

你需要保存正在運行線程的狀態，這個狀態對於其他的線程是不可見的。

解決方案

有時在多線程編程中，你需要只保存當前運行線程的狀態。要這麼做，可使用 thread.local() 創建一個本地線程存儲對象。對這個對象的屬性的保存和讀取操作都只會對執行線程可見，而其他線程並不可見。

作爲使用本地存儲的一個有趣的實際例子，考慮在 8.3 小節定義過的 LazyConnection 上下文管理器類。下面我們對它進行一些小的修改使得它可以適用於多線程：

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        if hasattr(self.local, 'sock'):
            raise RuntimeError('Already connected')
        self.local.sock = socket(self.family, self.type)
        self.local.sock.connect(self.address)
        return self.local.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.sock.close()
        del self.local.sock
```

代碼中，自己觀察對於 self.local 屬性的使用。它被初始化尾一個 threading.local() 實例。其他方法操作被存儲爲 self.local.sock 的套接字對象。有了這些就可以在多線程中安全的使用 LazyConnection 實例了。例如：

```
from functools import partial
def test(conn):
    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
```

```
        s.send(b'Host: www.python.org\r\n')

        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got {} bytes'.format(len(resp)))

if __name__ == '__main__':
    conn = LazyConnection(('www.python.org', 80))

    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

它之所以行得通的原因是每個線程會創建一個自己專屬的套接字連接（存儲爲 self.local.sock ）。因此，當不同的線程執行套接字操作時，由於操作的是不同的套接字，因此它們不會相互影響。

討論

在大部分程序中創建和操作線程特定狀態並不會有什麼問題。不過，當出了問題的時候，通常是因爲某個對象被多個線程使用到，用來操作一些專用的系統資源，比如一個套接字或文件。你不能讓所有線程貢獻一個單獨對象，因爲多個線程同時讀和寫的時候會產生混亂。本地線程存儲通過讓這些資源只能在被使用的線程中可見來解決這個問題。

本節中，使用 `thread.local()` 可以讓 `LazyConnection` 類支持一個線程一個連接，而不是對於所有的進程都只有一個連接。

其原理是，每個 `threading.local()` 實例爲每個線程維護着一個單獨的實例字典。所有普通實例操作比如獲取、修改和刪除值僅僅操作這個字典。每個線程使用一個獨立的字典就可以保證數據的隔離了。

12.7 創建一個線程池

問題

你創建一個工作者線程池，用來相應客戶端請求或執行其他的工作。

解決方案

`concurrent.futures` 函數庫有一個 `ThreadPoolExecutor` 類可以被用來完成這個任務。下面是一個簡單的 TCP 服務器，使用了一個線程池來響應客戶端：

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr):
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

如果你想手動創建你自己的線程池，通常可以使用一個 Queue 來輕鬆實現。下面是一個稍微不同但是手動實現的例子：

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    '''
    Handle a client connection
    '''
    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')

    sock.close()

def echo_server(addr, nworkers):
```

```
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

使用 ThreadPoolExecutor 相對於手動實現的一個好處在於它使得任務提交者更方便的從被調用函數中獲取返回值。例如，你可能會像下面這樣寫：

```
from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)
# Submit work to the pool
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Get the results back
x = a.result()
y = b.result()
```

例子中返回的 handle 對象會幫你處理所有的阻塞與協作，然後從工作線程中返回數據給你。特別的，a.result() 操作會阻塞進程直到對應的函數執行完成並返回一個結果。

討論

通常來講，你應該避免編寫線程數量可以無限制增長的程序。例如，看看下面這個服務器：

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM
```

```
def echo_client(sock, client_addr):
    '''
    Handle a client connection
    '''
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        t = Thread(target=echo_client, args=(client_sock, client_addr))
        t.daemon = True
        t.start()

echo_server(('',15000))
```

儘管這個也可以工作，但是它不能抵禦有人試圖通過創建大量線程讓你服務器資源枯竭而崩潰的攻擊行爲。通過使用預先初始化的線程池，你可以設置同時運行線程的上限數量。

你可能會關心創建大量線程會有什麼後果。現代操作系統可以很輕鬆的創建幾千個線程的線程池。甚至，同時幾千個線程等待工作並不會對其他代碼產生性能影響。當然了，如果所有線程同時被喚醒並立即在 CPU 上執行，那就不同了——特別是有了全局解釋器鎖 GIL。通常，你應該只在 I/O 處理相關代碼中使用線程池。

創建大的線程池的一個可能需要關注的問題是內存的使用。例如，如果你在 OS X 系統上面創建 2000 個線程，系統顯示 Python 進程使用了超過 9GB 的虛擬內存。不過，這個計算通常是有誤差的。當創建一個線程時，操作系統會預留一個虛擬內存區域來放置線程的執行棧（通常是 8MB 大小）。但是這個內存只有一小片段被實際映射到真實內存中。因此，Python 進程使用到的真實內存其實很小（比如，對於 2000 個線程來講，只使用到了 70MB 的真實內存，而不是 9GB）。如果你擔心虛擬內存大小，可以使用 `threading.stack_size()` 函數來降低它。例如：

```
import threading
threading.stack_size(65536)
```

如果你加上這條語句並再次運行前面的創建 2000 個線程試驗，你會發現 Python 進程只使用到了大概 210MB 的虛擬內存，而真實內存使用量沒有變。注意線程棧大小必須至少爲 32768 字節，通常是系統內存頁大小（4096、8192 等）的整數倍。

12.8 簡單的並行編程

問題

你有個程序要執行 CPU 密集型工作，你想讓他利用多核 CPU 的優勢來運行的快一點。

解決方案

concurrent.futures 庫提供了一個 ProcessPoolExecutor 類，可被用來在一個單獨的 Python 解釋器中執行計算密集型函數。不過，要使用它，你首先要有一些計算密集型的任務。我們通過一個簡單而實際的例子來演示它。假定你有個 Apache web 服務器日誌目錄的 gzip 壓縮包：

```
logs/
   20120701.log.gz
   20120702.log.gz
   20120703.log.gz
   20120704.log.gz
   20120705.log.gz
   20120706.log.gz
   ...
```

進一步假設每個日誌文件內容類似下面這樣：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

下面是一個腳本，在這些日誌文件中查找出所有訪問過 robots.txt 文件的主機：

```
# findrobots.py

import gzip
import io
import glob

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots
```

```
def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

前面的程序使用了通常的 map-reduce 風格來編寫。函數 `find_robots()` 在一個文件名集合上做 map 操作，並將結果彙總爲一個單獨的結果，也就是 `find_all_robots()` 函數中的 `all_robots` 集合。現在，假設你想要修改這個程序讓它使用多核 CPU。很簡單——只需要將 map() 操作替換爲一個 `concurrent.futures` 庫中生成的類似操作即可。下面是一個簡單修改版本：

```
# findrobots.py

import gzip
import io
import glob
from concurrent import futures

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file

    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    with futures.ProcessPoolExecutor() as pool:
        for robots in pool.map(find_robots, files):
```

```
            all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

通過這個修改後，運行這個腳本產生同樣的結果，但是在四核機器上面比之前快了 3.5 倍。實際的性能優化效果根據你的機器 CPU 數量的不同而不同。

討論

ProcessPoolExecutor 的典型用法如下：

```
from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as pool:
    ...
    do work in parallel using pool
    ...
```

其原理是，一個 ProcessPoolExecutor 創建 N 個獨立的 Python 解釋器，N 是系統上面可用 CPU 的個數。你可以通過提供可選參數給 ProcessPoolExecutor(N) 來修改處理器數量。這個處理池會一直運行到 with 塊中最後一個語句執行完成，然後處理池被關閉。不過，程序會一直等待直到所有提交的工作被處理完成。

被提交到池中的工作必須被定義爲一個函數。有兩種方法去提交。如果你想讓一個列表推導或一個 map() 操作並行執行的話，可使用 pool.map() :

```
# A function that performs a lot of work
def work(x):
    ...
    return result

# Nonparallel code
results = map(work, data)

# Parallel implementation
with ProcessPoolExecutor() as pool:
    results = pool.map(work, data)
```

另外，你可以使用 pool.submit() 來手動的提交單個任務：

```
# Some function
def work(x):
    ...
    return result

with ProcessPoolExecutor() as pool:
```

```
    ...
    # Example of submitting work to the pool
    future_result = pool.submit(work, arg)

    # Obtaining the result (blocks until done)
    r = future_result.result()
    ...
```

如果你手動提交一個任務，結果是一個 Future 實例。要獲取最終結果，你需要調用它的 result() 方法。它會阻塞進程直到結果被返回來。

如果不想阻塞，你還可以使用一個回調函數，例如：

```
def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
     future_result = pool.submit(work, arg)
     future_result.add_done_callback(when_done)
```

回調函數接受一個 Future 實例，被用來獲取最終的結果（比如通過調用它的 result() 方法）。儘管處理池很容易使用，在設計大程序的時候還是有很多需要注意的地方，如下幾點：

- 這種並行處理技術只適用於那些可以被分解爲互相獨立部分的問題。
- 被提交的任務必須是簡單函數形式。對於方法、閉包和其他類型的並行執行還不支持。
- 函數參數和返回值必須兼容 pickle，因爲要使用到進程間的通信，所有解釋器之間的交換數據必須被序列化
- 被提交的任務函數不應保留狀態或有副作用。除了打印日誌之類簡單的事情，

一旦啓動你不能控制子進程的任何行爲，因此最好保持簡單和純潔——函數不要去修改環境。

- 在 Unix 上進程池通過調用 fork() 系統調用被創建，

它會克隆 Python 解釋器，包括 fork 時的所有程序狀態。而在 Windows 上，克隆解釋器時不會克隆狀態。實際的 fork 操作會在第一次調用 pool.map() 或 pool.submit() 後發生。

- 當你混合使用進程池和多線程的時候要特別小心。

你應該在創建任何線程之前先創建並激活進程池（比如在程序啓動的 main 線程中創建進程池）。

12.9 Python 的全局鎖問題

問題

你已經聽說過全局解釋器鎖 GIL，擔心它會影響到多線程程序的執行性能。

解決方案

儘管 Python 完全支持多線程編程，但是解釋器的 C 語言實現部分在完全並行執行時並不是線程安全的。實際上，解釋器被一個全局解釋器鎖保護着，它確保任何時候都只有一個 Python 線程執行。GIL 最大的問題就是 Python 的多線程程序並不能利用多核 CPU 的優勢 (比如一個使用了多個線程的計算密集型程序只會在一個單 CPU 上面運行)。

在討論普通的 GIL 之前，有一點要強調的是 GIL 只會影響到那些嚴重依賴 CPU 的程序 (比如計算型的)。如果你的程序大部分只會涉及到 I/O，比如網絡交互，那麼使用多線程就很合適，因爲它們大部分時間都在等待。實際上，你完全可以放心的創建幾千個 Python 線程，現代操作系統運行這麼多線程沒有任何壓力，沒啥可擔心的。

而對於依賴 CPU 的程序，你需要弄清楚執行的計算的特點。例如，優化底層算法要比使用多線程運行快得多。類似的，由於 Python 是解釋執行的，如果你將那些性能瓶頸代碼移到一個 C 語言擴展模塊中，速度也會提升的很快。如果你要操作數組，那麼使用 NumPy 這樣的擴展會非常的高效。最後，你還可以考慮下其他可選實現方案，比如 PyPy，它通過一個 JIT 編譯器來優化執行效率 (不過在寫這本書的時候它還不能支持 Python 3)。

還有一點要注意的是，線程不是專門用來優化性能的。一個 CPU 依賴型程序可能會使用線程來管理一個圖形用戶界面、一個網絡連接或其他服務。這時候，GIL 會產生一些問題，因爲如果一個線程長期持有 GIL 的話會導致其他非 CPU 型線程一直等待。事實上，一個寫的不好的 C 語言擴展會導致這個問題更加嚴重，儘管代碼的計算部分會比之前運行的更快些。

說了這麼多，現在想說的是我們有兩種策略來解決 GIL 的缺點。首先，如果你完全工作於 Python 環境中，你可以使用 `multiprocessing` 模塊來創建一個進程池，並像協同處理器一樣的使用它。例如，假如你有如下的線程代碼：

```
# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = some_work(args)
    ...
```

修改代碼，使用進程池：

```
# Processing pool (see below for initiazation)
pool = None

# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result
```

```
# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Initiaze the pool
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
```

這個通過使用一個技巧利用進程池解決了 GIL 的問題。當一個線程想要執行 CPU 密集型工作時，會將任務發給進程池。然後進程池會在另外一個進程中啓動一個單獨的 Python 解釋器來工作。當線程等待結果的時候會釋放 GIL。並且，由於計算任務在單獨解釋器中執行，那麼就不會受限於 GIL 了。在一個多核系統上面，你會發現這個技術可以讓你很好的利用多 CPU 的優勢。

另外一個解決 GIL 的策略是使用 C 擴展編程技術。主要思想是將計算密集型任務轉移給 C，跟 Python 獨立，在工作的時候在 C 代碼中釋放 GIL。這可以通過在 C 代碼中插入下面這樣的特殊宏來完成：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code
   ...
   Py_END_ALLOW_THREADS
   ...
}
```

如果你使用其他工具訪問 C 語言，比如對於 Cython 的 ctypes 庫，你不需要做任何事。例如，ctypes 在調用 C 時會自動釋放 GIL。

討論

許多程序員在面對線程性能問題的時候，馬上就會怪罪 GIL，什麼都是它的問題。其實這樣子太不厚道也太天真了點。作爲一個真實的例子，在多線程的網絡編程中神祕的 `stalls` 可能是因爲其他原因比如一個 DNS 查找延時，而跟 GIL 毫無關係。最後你真的需要先去搞懂你的代碼是否真的被 GIL 影響到。同時還要明白 GIL 大部分都應該只關注 CPU 的處理而不是 I/O.

如果你準備使用一個處理器池，注意的是這樣做涉及到數據序列化和在不同 Python 解釋器通信。被執行的操作需要放在一個通過 def 語句定義的 Python 函數中，不能是 lambda、閉包可調用實例等，並且函數參數和返回值必須要兼容 pickle。同樣，要執行的任務量必須足夠大以彌補額外的通信開銷。

另外一個難點是當混合使用線程和進程池的時候會讓你很頭疼。如果你要同時使用兩者，最好在程序啓動時，創建任何線程之前先創建一個單例的進程池。然後線程使用同樣的進程池來進行它們的計算密集型工作。

C 擴展最重要的特徵是它們和 Python 解釋器是保持獨立的。也就是說，如果你準備將 Python 中的任務分配到 C 中去執行，你需要確保 C 代碼的操作跟 Python 保持獨立，這就意味着不要使用 Python 數據結構以及不要調用 Python 的 C API。另外一個就是你要確保 C 擴展所做的工作是足夠的，值得你這樣做。也就是說 C 擴展擔負起了大量的計算任務，而不是少數幾個計算。

這些解決 GIL 的方案並不能適用於所有問題。例如，某些類型的應用程序如果被分解爲多個進程處理的話並不能很好的工作，也不能將它的部分代碼改成 C 語言執行。對於這些應用程序，你就要自己需求解決方案了（比如多進程訪問共享內存區，多解析器運行於同一個進程等）。或者，你還可以考慮下其他的解釋器實現，比如 PyPy。

瞭解更多關於在 C 擴展中釋放 GIL，請參考 15.7 和 15.10 小節。

12.10 定義一個 Actor 任務

問題

你想定義跟 actor 模式中類似“actors”角色的任務

解決方案

actor 模式是一種最古老的也是最簡單的並行和分佈式計算解決方案。事實上，它天生的簡單性是它如此受歡迎的重要原因之一。簡單來講，一個 actor 就是一個併發執行的任務，只是簡單的執行發送給它的消息任務。響應這些消息時，它可能還會給其他 actor 發送更進一步的消息。actor 之間的通信是單向和異步的。因此，消息發送者不知道消息是什麼時候被髮送，也不會接收到一個消息已被處理的迴應或通知。

結合使用一個線程和一個隊列可以很容易的定義 actor，例如：

```
from queue import Queue
from threading import Thread, Event

# Sentinel used for shutdown
class ActorExit(Exception):
    pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        '''
        Send a message to the actor
        '''
        self._mailbox.put(msg)
```

```
    def recv(self):
        '''
        Receive an incoming message
        '''
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        '''
        Close the actor, thus shutting it down
        '''
        self.send(ActorExit)

    def start(self):
        '''
        Start concurrent execution
        '''
        self._terminated = Event()
        t = Thread(target=self._bootstrap)

        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
        self._terminated.wait()

    def run(self):
        '''
        Run method to be implemented by the user
        '''
        while True:
            msg = self.recv()

# Sample ActorTask
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print('Got:', msg)
```

```
# Sample use
p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()
```

這個例子中，你使用 actor 實例的 send() 方法發送消息給它們。其機制是，這個方法會將消息放入一個隊裏中，然後將其轉交給處理被接受消息的一個內部線程。close() 方法通過在隊列中放入一個特殊的哨兵值（ ActorExit ）來關閉這個 actor。用戶可以通過繼承 Actor 並定義實現自己處理邏輯 run() 方法來定義新的 actor。ActorExit 異常的使用就是用戶自定義代碼可以在需要的時候來捕獲終止請求（異常被 get() 方法拋出並傳播出去 ）。

如果你放寬對於同步和異步消息發送的要求，類 actor 對象還可以通過生成器來簡化定義。例如：

```
def print_actor():
    while True:

        try:
            msg = yield      # Get a message
            print('Got:', msg)
        except GeneratorExit:
            print('Actor terminating')

# Sample use
p = print_actor()
next(p)      # Advance to the yield (ready to receive)
p.send('Hello')
p.send('World')
p.close()
```

討論

actor 模式的魅力就在於它的簡單性。實際上，這裏僅僅只有一個核心操作 send() . 甚至，對於在基於 actor 系統中的“消息”的泛化概念可以已多種方式被擴展。例如，你可以以元組形式傳遞標籤消息，讓 actor 執行不同的操作，如下：

```
class TaggedActor(Actor):
    def run(self):
        while True:
             tag, *payload = self.recv()
             getattr(self,'do_'+tag)(*payload)

    # Methods correponding to different message tags
    def do_A(self, x):
```

```
        print('Running A', x)

    def do_B(self, x, y):
        print('Running B', x, y)

# Example
a = TaggedActor()
a.start()
a.send(('A', 1))      # Invokes do_A(1)
a.send(('B', 2, 3))   # Invokes do_B(2,3)
```

作爲另外一個例子，下面的 actor 允許在一個工作者中運行任意的函數，並且通過一個特殊的 Result 對象返回結果：

```
from threading import Event
class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value

        self._evt.set()

    def result(self):
        self._evt.wait()
        return self._result

class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Example use
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
print(r.result())
```

最後，“發送”一個任務消息的概念可以被擴展到多進程甚至是大型分佈式系統中去。例如，一個類 actor 對象的 send() 方法可以被編程讓它能在一個套接字連接上傳輸數據或通過某些消息中間件（比如 AMQP、ZMQ 等）來發送。

12.11 實現消息發佈/訂閱模型

問題

你有一個基於線程通信的程序，想讓它們實現發佈/訂閱模式的消息通信。

解決方案

要實現發佈/訂閱的消息通信模式，你通常要引入一個單獨的“交換機”或“網關”對象作爲所有消息的中介。也就是說，不直接將消息從一個任務發送到另一個，而是將其發送給交換機，然後由交換機將它發送給一個或多個被關聯任務。下面是一個非常簡單的交換機實現例子：

```
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]
```

一個交換機就是一個普通對象，負責維護一個活躍的訂閱者集合，併爲綁定、解綁和發送消息提供相應的方法。每個交換機通過一個名稱定位，`get_exchange()` 通過給定一個名稱返回相應的 `Exchange` 實例。

下面是一個簡單例子，演示瞭如何使用一個交換機：

```
# Example of a task.  Any object with a send() method

class Task:
    ...
    def send(self, msg):
        ...

task_a = Task()
```

```
task_b = Task()

# Example of getting an exchange
exc = get_exchange('name')

# Examples of subscribing tasks to it
exc.attach(task_a)
exc.attach(task_b)

# Example of sending messages
exc.send('msg1')
exc.send('msg2')

# Example of unsubscribing
exc.detach(task_a)
exc.detach(task_b)
```

儘管對於這個問題有很多的變種，不過萬變不離其宗。消息會被髮送給一個交換機，然後交換機會將它們發送給被綁定的訂閱者。

討論

通過隊列發送消息的任務或線程的模式很容易被實現並且也非常普遍。不過，使用發佈/訂閱模式的好處更加明顯。

首先，使用一個交換機可以簡化大部分涉及到線程通信的工作。無需去寫通過多進程模塊來操作多個線程，你只需要使用這個交換機來連接它們。某種程度上，這個就跟日誌模塊的工作原理類似。實際上，它可以輕鬆的解耦程序中多個任務。

其次，交換機廣播消息給多個訂閱者的能力帶來了一個全新的通信模式。例如，你可以使用多任務系統、廣播或扇出。你還可以通過以普通訂閱者身份綁定來構建調試和診斷工具。例如，下面是一個簡單的診斷類，可以顯示被髮送的消息：

```
class DisplayMessages:
    def __init__(self):
        self.count = 0
    def send(self, msg):
        self.count += 1
        print('msg[{}]: {!r}'.format(self.count, msg))

exc = get_exchange('name')
d = DisplayMessages()
exc.attach(d)
```

最後，該實現的一個重要特點是它能兼容多個“task-like”對象。例如，消息接受者可以是 actor（12.10 小節介紹）、協程、網絡連接或任何實現了正確的 `send()` 方法的東西。

關於交換機的一個可能問題是對於訂閱者的正確綁定和解綁。爲了正確的管理資源，每一個綁定的訂閱者必須最終要解綁。在代碼中通常會是像下面這樣的模式：

```
exc = get_exchange('name')
exc.attach(some_task)
try:
    ...
finally:
    exc.detach(some_task)
```

某種意義上，這個和使用文件、鎖和類似對象很像。通常很容易會忘記最後的 detach() 步驟。爲了簡化這個，你可以考慮使用上下文管理器協議。例如，在交換機對象上增加一個 subscribe() 方法，如下：

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
        for task in tasks:
            self.attach(task)
        try:
            yield
        finally:
            for task in tasks:
                self.detach(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]

# Example of using the subscribe() method
exc = get_exchange('name')
with exc.subscribe(task_a, task_b):
     ...
     exc.send('msg1')
```

```
    exc.send('msg2')
    ...

# task_a and task_b detached here
```

最後還應該注意的是關於交換機的思想有很多種的擴展實現。例如，交換機可以實現一整個消息通道集合或提供交換機名稱的模式匹配規則。交換機還可以被擴展到分佈式計算程序中（比如，將消息路由到不同機器上面的任務中去）。

12.12 使用生成器代替線程

問題

你想使用生成器（協程）替代系統線程來實現併發。這個有時又被稱爲用戶級線程或綠色線程。

解決方案

要使用生成器實現自己的併發，你首先要對生成器函數和 yield 語句有深刻理解。yield 語句會讓一個生成器掛起它的執行，這樣就可以編寫一個調度器，將生成器當做某種“任務”並使用任務協作切換來替換它們的執行。要演示這種思想，考慮下面兩個使用簡單的 yield 語句的生成器函數：

```
# Two simple generator functions
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
    print('Blastoff!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1
```

這些函數在內部使用 yield 語句，下面是一個實現了簡單任務調度器的代碼：

```
from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        '''
```

```
        Admit a newly started task to the scheduler

        '''
        self._task_queue.append(task)

    def run(self):
        '''
        Run until there are no more tasks
        '''
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                # Run until the next yield statement
                next(task)
                self._task_queue.append(task)
            except StopIteration:
                # Generator is no longer executing
                pass

# Example use
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

TaskScheduler 類在一個循環中運行生成器集合——每個都運行到碰到 yield 語句爲止。運行這個例子，輸出如下：

```
T-minus 10
T-minus 5
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...
```

到此爲止，我們實際上已經實現了一個“操作系統”的最小核心部分。生成器函數就是認爲，而 yield 語句是任務掛起的信號。調度器循環檢查任務列表直到沒有任務要執行爲止。

實際上，你可能想要使用生成器來實現簡單的併發。那麼，在實現 actor 或網絡服務器的時候你可以使用生成器來替代線程的使用。

下面的代碼演示了使用生成器來實現一個不依賴線程的 actor：

```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = { }          # Mapping of names to actors
        self._msg_queue = deque()   # Message queue

    def new_actor(self, name, actor):
        '''
        Admit a newly started actor to the scheduler and give it a name
        '''
        self._msg_queue.append((actor,None))
        self._actors[name] = actor

    def send(self, name, msg):
        '''
        Send a message to a named actor
        '''
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor,msg))

    def run(self):
        '''
        Run as long as there are pending messages.
        '''
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                 actor.send(msg)
            except StopIteration:
                 pass

# Example use
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            # Receive the current count
            n = yield
            if n == 0:
                break
            # Send to the printer task
            sched.send('printer', n)
            # Send the next count to the counter task (recursive)
```

```
            sched.send('counter', n-1)

    sched = ActorScheduler()
    # Create the initial actors
    sched.new_actor('printer', printer())
    sched.new_actor('counter', counter(sched))

    # Send an initial message to the counter to initiate
    sched.send('counter', 10000)
    sched.run()
```

完全弄懂這段代碼需要更深入的學習，但是關鍵點在於收集消息的隊列。本質上，調度器在有需要發送的消息時會一直運行着。計數生成器會給自己發送消息並在一個遞歸循環中結束。

下面是一個更加高級的例子，演示了使用生成器來實現一個併發網絡應用程序：

```
from collections import deque
from select import select

# This class represents a generic yield event in the scheduler
class YieldEvent:
    def handle_yield(self, sched, task):
        pass
    def handle_resume(self, sched, task):
        pass

# Task Scheduler
class Scheduler:
    def __init__(self):
        self._numtasks = 0        # Total num of tasks
        self._ready = deque()     # Tasks ready to run
        self._read_waiting = {}   # Tasks waiting to read
        self._write_waiting = {} # Tasks waiting to write

    # Poll for I/O events and restart waiting tasks
    def _iopoll(self):
        rset,wset,eset = select(self._read_waiting,
                                self._write_waiting,[])
        for r in rset:
            evt, task = self._read_waiting.pop(r)
            evt.handle_resume(self, task)
        for w in wset:
            evt, task = self._write_waiting.pop(w)
            evt.handle_resume(self, task)

    def new(self,task):
        '''
        Add a newly started task to the scheduler
        '''
```

```
        self._ready.append((task, None))
        self._numtasks += 1

    def add_ready(self, task, msg=None):
        '''
        Append an already started task to the ready queue.
        msg is what to send into the task when it resumes.
        '''
        self._ready.append((task, msg))

    # Add a task to the reading set
    def _read_wait(self, fileno, evt, task):
        self._read_waiting[fileno] = (evt, task)

    # Add a task to the write set
    def _write_wait(self, fileno, evt, task):
        self._write_waiting[fileno] = (evt, task)

    def run(self):
        '''
        Run the task scheduler until there are no tasks
        '''
        while self._numtasks:
            if not self._ready:
                self._iopoll()
            task, msg = self._ready.popleft()
            try:
                # Run the coroutine to the next yield
                r = task.send(msg)
                if isinstance(r, YieldEvent):
                    r.handle_yield(self, task)
                else:
                    raise RuntimeError('unrecognized yield event')
            except StopIteration:
                self._numtasks -= 1

# Example implementation of coroutine-based socket I/O
class ReadSocket(YieldEvent):
    def __init__(self, sock, nbytes):
        self.sock = sock
        self.nbytes = nbytes
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        data = self.sock.recv(self.nbytes)
        sched.add_ready(task, data)

class WriteSocket(YieldEvent):
    def __init__(self, sock, data):
        self.sock = sock
```

```
        self.data = data
    def handle_yield(self, sched, task):

        sched._write_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket(YieldEvent):
    def __init__(self, sock):
        self.sock = sock
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Wrapper around a socket object for use with yield
class Socket(object):
    def __init__(self, sock):
        self._sock = sock
    def recv(self, maxbytes):
        return ReadSocket(self._sock, maxbytes)
    def send(self, data):
        return WriteSocket(self._sock, data)
    def accept(self):
        return AcceptSocket(self._sock)
    def __getattr__(self, name):
        return getattr(self._sock, name)

if __name__ == '__main__':
    from socket import socket, AF_INET, SOCK_STREAM
    import time

    # Example of a function involving generators.  This should
    # be called using line = yield from readline(sock)
    def readline(sock):
        chars = []
        while True:
            c = yield sock.recv(1)
            if not c:
                break
            chars.append(c)
            if c == b'\n':
                break
        return b''.join(chars)

    # Echo server using generators
    class EchoServer:
        def __init__(self,addr,sched):
```

```
            self.sched = sched
            sched.new(self.server_loop(addr))

        def server_loop(self,addr):
            s = Socket(socket(AF_INET,SOCK_STREAM))

            s.bind(addr)
            s.listen(5)
            while True:
                c,a = yield s.accept()
                print('Got connection from ', a)
                self.sched.new(self.client_handler(Socket(c)))

        def client_handler(self,client):
            while True:
                line = yield from readline(client)
                if not line:
                    break
                line = b'GOT:' + line
                while line:
                    nsent = yield client.send(line)
                    line = line[nsent:]
            client.close()
            print('Client closed')

    sched = Scheduler()
    EchoServer(('',16000),sched)
    sched.run()
```

這段代碼有點複雜。不過，它實現了一個小型的操作系統。有一個就緒的任務隊列，並且還有因 I/O 休眠的任務等待區域。還有很多調度器負責在就緒隊列和 I/O 等待區域之間移動任務。

討論

在構建基於生成器的併發框架時，通常會使用更常見的 yield 形式：

```
def some_generator():
    ...
    result = yield data
    ...
```

使用這種形式的 yield 語句的函數通常被稱爲“協程”。通過調度器，yield 語句在一個循環中被處理，如下：

```
f = some_generator()

# Initial result. Is None to start since nothing has been computed
result = None
```

```
while True:
    try:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration:
        break
```

這裏的邏輯稍微有點複雜。不過，被傳給 send() 的值定義了在 yield 語句醒來時的返回值。因此，如果一個 yield 準備在對之前 yield 數據的迴應中返回結果時，會在下一次 send() 操作返回。如果一個生成器函數剛開始運行，發送一個 None 值會讓它排在第一個 yield 語句前面。

除了發送值外，還可以在一個生成器上面執行一個 close() 方法。它會導致在執行 yield 語句時拋出一個 GeneratorExit 異常，從而終止執行。如果進一步設計，一個生成器可以捕獲這個異常並執行清理操作。同樣還可以使用生成器的 throw() 方法在 yield 語句執行時生成一個任意的執行指令。一個任務調度器可利用它來在運行的生成器中處理錯誤。

最後一個例子中使用的 yield from 語句被用來實現協程，可以被其它生成器作爲子程序或過程來調用。本質上就是將控制權透明的傳輸給新的函數。不像普通的生成器，一個使用 yield from 被調用的函數可以返回一個作爲 yield from 語句結果的值。關於 yield from 的更多信息可以在 PEP 380 中找到。

最後，如果使用生成器編程，要提醒你的是它還是有很多缺點的。特別是，你得不到任何線程可以提供的好處。例如，如果你執行 CPU 依賴或 I/O 阻塞程序，它會將整個任務掛起知道操作完成。爲了解決這個問題，你只能選擇將操作委派給另外一個可以獨立運行的線程或進程。另外一個限制是大部分 Python 庫並不能很好的兼容基於生成器的線程。如果你選擇這個方案，你會發現你需要自己改寫很多標準庫函數。作爲本節提到的協程和相關技術的一個基礎背景，可以查看 PEP 342 和 “協程和併發的一門有趣課程”

PEP 3156 同樣有一個關於使用協程的異步 I/O 模型。特別的，你不可能自己去實現一個底層的協程調度器。不過，關於協程的思想是很多流行庫的基礎，包括 gevent, greenlet, Stackless Python 以及其他類似工程。

12.13 多個線程隊列輪詢

問題

你有一個線程隊列集合，想爲到來的元素輪詢它們，就跟你爲一個客戶端請求去輪詢一個網絡連接集合的方式一樣。

解決方案

對於輪詢問題的一個常見解決方案中有個很少有人知道的技巧，包含了一個隱藏的迴路網絡連接。本質上講其思想就是：對於每個你想要輪詢的隊列，你創建一對連接的套接字。然後你在其中一個套接字上面編寫代碼來標識存在的數據，另外一個套接字被傳給 select() 或類似的一個輪詢數據到達的函數。下面的例子演示了這個思想：

```
import queue
import socket
import os

class PollableQueue(queue.Queue):
    def __init__(self):
        super().__init__()
        # Create a pair of connected sockets
        if os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair()
        else:
            # Compatibility on non-POSIX systems
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self._putsocket = socket.socket(socket.AF_INET, socket.SOCK_
↪STREAM)
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()

    def fileno(self):
        return self._getsocket.fileno()

    def put(self, item):
        super().put(item)
        self._putsocket.send(b'x')

    def get(self):
        self._getsocket.recv(1)
        return super().get()
```

在這個代碼中，一個新的 Queue 實例類型被定義，底層是一個被連接套接字對。在 Unix 機器上的 socketpair() 函數能輕鬆的創建這樣的套接字。在 Windows 上面，你必須使用類似代碼來模擬它。然後定義普通的 get() 和 put() 方法在這些套接字上面來執行 I/O 操作。put() 方法再將數據放入隊列後會寫一個單字節到某個套接字中去。而 get() 方法在從隊列中移除一個元素時會從另外一個套接字中讀取到這個單字節數據。

fileno() 方法使用一個函數比如 select() 來讓這個隊列可以被輪詢。它僅僅只是暴露了底層被 get() 函數使用到的 socket 的文件描述符而已。

下面是一個例子，定義了一個爲到來的元素監控多個隊列的消費者：

```
import select
import threading

def consumer(queues):
    '''
    Consumer that reads data on multiple queues simultaneously
    '''
```

```
    while True:
        can_read, _, _ = select.select(queues,[],[])
        for r in can_read:
            item = r.get()
            print('Got:', item)

q1 = PollableQueue()
q2 = PollableQueue()
q3 = PollableQueue()
t = threading.Thread(target=consumer, args=([q1,q2,q3],))
t.daemon = True
t.start()

# Feed data to the queues
q1.put(1)
q2.put(10)
q3.put('hello')
q2.put(15)
...
```

如果你試着運行它，你會發現這個消費者會接受到所有的被放入的元素，不管元素被放進了哪個隊列中。

討論

對於輪詢非類文件對象，比如隊列通常都是比較棘手的問題。例如，如果你不使用上面的套接字技術，你唯一的選擇就是編寫代碼來循環遍歷這些隊列並使用一個定時器。像下面這樣：

```
import time
def consumer(queues):
    while True:
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)

        # Sleep briefly to avoid 100% CPU
        time.sleep(0.01)
```

這樣做其實不合理，還會引入其他的性能問題。例如，如果新的數據被加入到一個隊列中，至少要花 10 毫秒才能被發現。如果你之前的輪詢還要去輪詢其他對象，比如網絡套接字那還會有更多問題。例如，如果你想同時輪詢套接字和隊列，你可能要像下面這樣使用：

```
import select

def event_loop(sockets, queues):
    while True:
```

```
        # polling with a timeout
        can_read, _, _ = select.select(sockets, [], [], 0.01)
        for r in can_read:
            handle_read(r)
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)
```

這個方案通過將隊列和套接字等同對待來解決了大部分的問題。一個單獨的 `select()` 調用可被同時用來輪詢。使用超時或其他基於時間的機制來執行週期性檢查並沒有必要。甚至，如果數據被加入到一個隊列，消費者幾乎可以實時的被通知。儘管會有一點點底層的 I/O 損耗，使用它通常會獲得更好的響應時間並簡化編程。

12.14 在 Unix 系統上面啓動守護進程

問題

你想編寫一個作爲一個在 Unix 或類 Unix 系統上面運行的守護進程運行的程序。

解決方案

創建一個正確的守護進程需要一個精確的系統調用序列以及對於細節的控制。下面的代碼展示了怎樣定義一個守護進程，可以啓動後很容易的停止它。

```
#!/usr/bin/env python3
# daemon.py

import os
import sys

import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
                          stdout='/dev/null',
                          stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # First fork (detaches from parent)
    try:
        if os.fork() > 0:
            raise SystemExit(0)   # Parent exit
    except OSError as e:
        raise RuntimeError('fork #1 failed.')
```

```
    os.chdir('/')
    os.umask(0)
    os.setsid()
    # Second fork (relinquish session leadership)
    try:
        if os.fork() > 0:
            raise SystemExit(0)
    except OSError as e:
        raise RuntimeError('fork #2 failed.')

    # Flush I/O buffers
    sys.stdout.flush()
    sys.stderr.flush()

    # Replace file descriptors for stdin, stdout, and stderr
    with open(stdin, 'rb', 0) as f:
        os.dup2(f.fileno(), sys.stdin.fileno())
    with open(stdout, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stdout.fileno())
    with open(stderr, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stderr.fileno())

    # Write the PID file
    with open(pidfile,'w') as f:
        print(os.getpid(),file=f)

    # Arrange to have the PID file removed on exit/signal
    atexit.register(lambda: os.remove(pidfile))

    # Signal handler for termination (required)
    def sigterm_handler(signo, frame):
        raise SystemExit(1)

    signal.signal(signal.SIGTERM, sigterm_handler)

def main():
    import time
    sys.stdout.write('Daemon started with pid {}\n'.format(os.getpid()))
    while True:
        sys.stdout.write('Daemon Alive! {}\n'.format(time.ctime()))
        time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit(1)

    if sys.argv[1] == 'start':
```

```
        try:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
                      stderr='/tmp/dameon.log')
        except RuntimeError as e:
            print(e, file=sys.stderr)
            raise SystemExit(1)

        main()

    elif sys.argv[1] == 'stop':
        if os.path.exists(PIDFILE):
            with open(PIDFILE) as f:
                os.kill(int(f.read()), signal.SIGTERM)
        else:
            print('Not running', file=sys.stderr)
            raise SystemExit(1)

    else:
        print('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr)
        raise SystemExit(1)
```

要啓動這個守護進程，用戶需要使用如下的命令：

```
bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...
```

守護進程可以完全在後臺運行，因此這個命令會立即返回。不過，你可以像上面那樣查看與它相關的 pid 文件和日誌。要停止這個守護進程，使用：

```
bash % daemon.py stop
bash %
```

討論

本節定義了一個函數 daemonize() ，在程序啓動時被調用使得程序以一個守護進程來運行。daemonize() 函數只接受關鍵字參數，這樣的話可選參數在被使用時就更清晰了。它會強制用戶像下面這樣使用它：

```
daemonize('daemon.pid',
          stdin='/dev/null,
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

而不是像下面這樣含糊不清的調用：

```
# Illegal. Must use keyword arguments
daemonize('daemon.pid',
          '/dev/null', '/tmp/daemon.log','/tmp/daemon.log')
```

創建一個守護進程的步驟看上去不是很易懂，但是大體思想是這樣的，首先，一個守護進程必須要從父進程中脫離。這是由 os.fork() 操作來完成的，並立即被父進程終止。

在子進程變成孤兒後，調用 os.setsid() 創建了一個全新的進程會話，並設置子進程爲首領。它會設置這個子進程爲新的進程組的首領，並確保不會再有控制終端。如果這些聽上去太魔幻，因爲它需要將守護進程同終端分離開並確保信號機制對它不起作用。調用 os.chdir() 和 os.umask(0) 改變了當前工作目錄並重置文件權限掩碼。修改目錄通常是個好主意，因爲這樣可以使得它不再工作在被啓動時的目錄。

另外一個調用 os.fork() 在這裏更加神祕點。這一步使得守護進程失去了獲取新的控制終端的能力並且讓它更加獨立 (本質上，該 daemon 放棄了它的會話首領低位，因此再也沒有權限去打開控制終端了)。儘管你可以忽略這一步，但是最好不要這麼做。

一旦守護進程被正確的分離，它會重新初始化標準 I/O 流指向用戶指定的文件。這一部分有點難懂。跟標準 I/O 流相關的文件對象的引用在解釋器中多個地方被找到 (sys.stdout, sys.__stdout__ 等)。僅僅簡單的關閉 sys.stdout 並重新指定它是行不通的，因爲沒辦法知道它是否全部都是用的是 sys.stdout 。這裏，我們打開了一個單獨的文件對象，並調用 os.dup2() ，用它來代替被 sys.stdout 使用的文件描述符。這樣，sys.stdout 使用的原始文件會被關閉並由新的來替換。還要強調的是任何用於文件編碼或文本處理的標準 I/O 流還會保留原狀。

守護進程的一個通常實踐是在一個文件中寫入進程 ID，可以被其他程序後面使用到。daemonize() 函數的最後部分寫了這個文件，但是在程序終止時刪除了它。atexit.register() 函數註冊了一個函數在 Python 解釋器終止時執行。一個對於 SIGTERM 的信號處理器的定義同樣需要被優雅的關閉。信號處理器簡單的拋出了 SystemExit() 異常。或許這一步看上去沒必要，但是沒有它，終止信號會使得不執行 atexit.register() 註冊的清理操作的時候就殺掉了解釋器。一個殺掉進程的例子代碼可以在程序最後的 stop 命令的操作中看到。

更多關於編寫守護進程的信息可以查看《UNIX 環境高級編程》, 第二版 by W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005)。儘管它是關注與 C 語言編程，但是所有的內容都適用於 Python，因爲所有需要的 POSIX 函數都可以在標準庫中找到。

第十三章：腳本編程與系統管理

許多人使用 Python 作爲一個 shell 腳本的替代，用來實現常用系統任務的自動化，如文件的操作，系統的配置等。本章的主要目標是描述關於編寫腳本時候經常遇到的一些功能。例如，解析命令行選項、獲取有用的系統配置數據等等。第 5 章也包含了與文件和目錄相關的一般信息。

13.1 通過重定向/管道/文件接受輸入

問題

你希望你的腳本接受任何用戶認爲最簡單的輸入方式。包括將命令行的輸出通過管道傳遞給該腳本、重定向文件到該腳本，或在命令行中傳遞一個文件名或文件名列表給該腳本。

解決方案

Python 內置的 `fileinput` 模塊讓這個變得簡單。如果你有一個下面這樣的腳本：

```
#!/usr/bin/env python3
import fileinput

with fileinput.input() as f_input:
    for line in f_input:
        print(line, end='')
```

那麼你就能以前面提到的所有方式來爲此腳本提供輸入。假設你將此腳本保存爲 `filein.py` 並將其變爲可執行文件，那麼你可以像下面這樣調用它，得到期望的輸出：

```
$ ls | ./filein.py          # Prints a directory listing to stdout.
$ ./filein.py /etc/passwd   # Reads /etc/passwd to stdout.
$ ./filein.py < /etc/passwd # Reads /etc/passwd to stdout.
```

討論

`fileinput.input()` 創建並返回一個 `FileInput` 類的實例。該實例除了擁有一些有用的幫助方法外，它還可被當做一個上下文管理器使用。因此，整合起來，如果我們要寫一個打印多個文件輸出的腳本，那麼我們需要在輸出中包含文件名和行號，如下所示：

```
>>> import fileinput
>>> with fileinput.input('/etc/passwd') as f:
>>>     for line in f:
...         print(f.filename(), f.lineno(), line, end='')
...
/etc/passwd 1 ##
```

```
/etc/passwd 2 # User Database
/etc/passwd 3 #

<other output omitted>
```

通過將它作爲一個上下文管理器使用，可以確保它不再使用時文件能自動關閉，而且我們在之後還演示了 FileInput 的一些有用的幫助方法來獲取輸出中的一些其他信息。

13.2 終止程序並給出錯誤信息

問題

你想向標準錯誤打印一條消息並返回某個非零狀態碼來終止程序運行

解決方案

你有一個程序像下面這樣終止，拋出一個 SystemExit 異常，使用錯誤消息作爲參數。例如：

```
raise SystemExit('It failed!')
```

它會將消息在 sys.stderr 中打印，然後程序以狀態碼 1 退出。

討論

本節雖然很短小，但是它能解決在寫腳本時的一個常見問題。也就是說，當你想要終止某個程序時，你可能會像下面這樣寫：

```
import sys
sys.stderr.write('It failed!\n')
raise SystemExit(1)
```

如果你直接將消息作爲參數傳給 SystemExit() ，那麼你可以省略其他步驟，比如 import 語句或將錯誤消息寫入 sys.stderr

13.3 解析命令行選項

問題

你的程序如何能夠解析命令行選項（位於 sys.argv 中）

解決方案

argparse 模塊可被用來解析命令行選項。下面一個簡單例子演示了最基本的用法：

```
# search.py
'''
Hypothetical command-line tool for searching a collection of
files for one or more text patterns.
'''
import argparse
parser = argparse.ArgumentParser(description='Search some files')

parser.add_argument(dest='filenames',metavar='filename', nargs='*')

parser.add_argument('-p', '--pat',metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')

parser.add_argument('--speed', dest='speed', action='store',
                    choices={'slow','fast'}, default='slow',
                    help='search speed')

args = parser.parse_args()

# Output the collected arguments
print(args.filenames)
print(args.patterns)
print(args.verbose)
print(args.outfile)
print(args.speed)
```

該程序定義了一個如下使用的命令行解析器：

```
bash % python3 search.py -h
usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE] [--speed {slow,fast}]
                 [filename [filename ...]]

Search some files

positional arguments:
  filename

optional arguments:
  -h, --help            show this help message and exit
  -p pattern, --pat pattern
```

```
                        text pattern to search for
  -v                    verbose mode
  -o OUTFILE            output file
  --speed {slow,fast}   search speed
```

下面的部分演示了程序中的數據部分。仔細觀察 print() 語句的打印輸出。

```
bash % python3 search.py foo.txt bar.txt
usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [--speed {fast,slow}]
                 [filename [filename ...]]
search.py: error: the following arguments are required: -p/--pat

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = None
speed     = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results \
             --speed=fast
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = fast
```

對於選項值的進一步處理由程序來決定，用你自己的邏輯來替代 print() 函數。

討論

argparse 模塊是標準庫中最大的模塊之一，擁有大量的配置選項。本節只是演示了其中最基礎的一些特性，幫助你入門。

爲了解析命令行選項，你首先要創建一個 ArgumentParser 實例，並使用 add_argument() 方法聲明你想要支持的選項。在每個 add_argument() 調用中，dest 參數指定解析結果被指派給屬性的名字。metavar 參數被用來生成幫助信息。action 參數指定跟屬性對應的處理邏輯，通常的值爲 store，被用來存儲某個值或講多個參數值收集到一個列表中。下面的參數收集所有剩餘的命令行參數到一個列表中。在本例中它被用來構造一個文件名列表：

```
parser.add_argument(dest='filenames',metavar='filename', nargs='*')
```

下面的參數根據參數是否存在來設置一個 Boolean 標誌：

```
parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')
```

下面的參數接受一個單獨值並將其存儲爲一個字符串：

```
parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')
```

下面的參數說明允許某個參數重複出現多次，並將它們追加到一個列表中去。required 標誌表示該參數至少要有一個。-p 和 --pat 表示兩個參數名形式都可使用。

```
parser.add_argument('-p', '--pat',metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')
```

最後，下面的參數說明接受一個值，但是會將其和可能的選擇值做比較，以檢測其合法性：

```
parser.add_argument('--speed', dest='speed', action='store',
                    choices={'slow','fast'}, default='slow',
                    help='search speed')
```

一旦參數選項被指定，你就可以執行 parser.parse() 方法了。它會處理 sys.argv 的值並返回一個結果實例。每個參數值會被設置成該實例中 add_argument() 方法的 dest 參數指定的屬性值。

還很多種其他方法解析命令行選項。例如，你可能會手動的處理 sys.argv 或者使用 getopt 模塊。但是，如果你採用本節的方式，將會減少很多冗餘代碼，底層細節 argparse 模塊已經幫你處理了。你可能還會碰到使用 optparse 庫解析選項的代碼。儘管 optparse 和 argparse 很像，但是後者更先進，因此在新的程序中你應該使用它。

13.4 運行時彈出密碼輸入提示

問題

你寫了個腳本，運行時需要一個密碼。此腳本是交互式的，因此不能將密碼在腳本中硬編碼，而是需要彈出一個密碼輸入提示，讓用戶自己輸入。

解決方案

這時候 Python 的 getpass 模塊正是你所需要的。你可以讓你很輕鬆的彈出密碼輸入提示，並且不會在用戶終端回顯密碼。下面是具體代碼：

```
import getpass

user = getpass.getuser()
passwd = getpass.getpass()
```

```
if svc_login(user, passwd):    # You must write svc_login()
   print('Yay!')
else:
   print('Boo!')
```

在此代碼中，svc_login() 是你要實現的處理密碼的函數，具體的處理過程你自己決定。

討論

注意在前面代碼中 getpass.getuser() 不會彈出用戶名的輸入提示。它會根據該用戶的 shell 環境或者會依據本地系統的密碼庫（支持 *pwd* 模塊的平臺）來使用當前用戶的登錄名，

如果你想顯示的彈出用戶名輸入提示，使用內置的 input 函數：

```
user = input('Enter your username: ')
```

還有一點很重要，有些系統可能不支持 getpass() 方法隱藏輸入密碼。這種情況下，Python 會提前警告你這些問題（例如它會警告你說密碼會以明文形式顯示）

13.5 獲取終端的大小

問題

你需要知道當前終端的大小以便正確的格式化輸出。

解決方案

使用 os.get_terminal_size() 函數來做到這一點。

代碼示例：

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz
os.terminal_size(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

討論

有太多方式來得知終端大小了，從讀取環境變量到執行底層的 `ioctl()` 函數等等。不過，爲什麼要去研究這些複雜的辦法而不是僅僅調用一個簡單的函數呢？

13.6 執行外部命令並獲取它的輸出

問題

你想執行一個外部命令並以 Python 字符串的形式獲取執行結果。

解決方案

使用 `subprocess.check_output()` 函數。例如：

```
import subprocess
out_bytes = subprocess.check_output(['netstat','-a'])
```

這段代碼執行一個指定的命令並將執行結果以一個字節字符串的形式返回。如果你需要文本形式返回，加一個解碼步驟即可。例如：

```
out_text = out_bytes.decode('utf-8')
```

如果被執行的命令以非零碼返回，就會拋出異常。下面的例子捕獲到錯誤並獲取返回碼：

```
try:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'])
except subprocess.CalledProcessError as e:
    out_bytes = e.output       # Output generated before error
    code      = e.returncode   # Return code
```

默認情況下，`check_output()` 僅僅返回輸入到標準輸出的值。如果你需要同時收集標準輸出和錯誤輸出，使用 `stderr` 參數：

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'],
                                    stderr=subprocess.STDOUT)
```

如果你需要用一個超時機制來執行命令，使用 `timeout` 參數：

```
try:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'], timeout=5)
except subprocess.TimeoutExpired as e:
    ...
```

通常來講，命令的執行不需要使用到底層 shell 環境（比如 sh、bash）。一個字符串列表會被傳遞給一個低級系統命令，比如 `os.execve()` 。如果你想讓命令被一個 shell 執行，傳遞一個字符串參數，並設置參數 `shell=True` . 有時候你想要 Python 去執行

一個複雜的 shell 命令的時候這個就很有用了，比如管道流、I/O 重定向和其他特性。例如：

```
out_bytes = subprocess.check_output('grep python | wc > out', shell=True)
```

需要注意的是在 shell 中執行命令會存在一定的安全風險，特別是當參數來自於用戶輸入時。這時候可以使用 shlex.quote() 函數來講參數正確的用雙引用引起來。

討論

使用 check_output() 函數是執行外部命令並獲取其返回值的最簡單方式。但是，如果你需要對子進程做更復雜的交互，比如給它發送輸入，你得采用另外一種方法。這時候可直接使用 subprocess.Popen 類。例如：

```
import subprocess

# Some text to send
text = b'''
hello world
this is a test
goodbye
'''

# Launch a command with pipes
p = subprocess.Popen(['wc'],
          stdout = subprocess.PIPE,
          stdin = subprocess.PIPE)

# Send the data and get the output
stdout, stderr = p.communicate(text)

# To interpret as text, decode
out = stdout.decode('utf-8')
err = stderr.decode('utf-8')
```

subprocess 模塊對於依賴 TTY 的外部命令不合適用。例如，你不能使用它來自動化一個用戶輸入密碼的任務（比如一個 ssh 會話）。這時候，你需要使用到第三方模塊了，比如基於著名的 expect 家族的工具（pexpect 或類似的）

13.7 複製或者移動文件和目錄

問題

你想要複製或移動文件和目錄，但是又不想調用 shell 命令。

解決方案

shutil 模塊有很多便捷的函數可以複製文件和目錄。使用起來非常簡單，比如：

```
import shutil

# Copy src to dst. (cp src dst)
shutil.copy(src, dst)

# Copy files, but preserve metadata (cp -p src dst)
shutil.copy2(src, dst)

# Copy directory tree (cp -R src dst)
shutil.copytree(src, dst)

# Move src to dst (mv src dst)
shutil.move(src, dst)
```

這些函數的參數都是字符串形式的文件或目錄名。底層語義模擬了類似的 Unix 命令，如上面的註釋部分。

默認情況下，對於符號鏈接而已這些命令處理的是它指向的東西。例如，如果源文件是一個符號鏈接，那麼目標文件將會是符號鏈接指向的文件。如果你只想複製符號鏈接本身，那麼需要指定關鍵字參數 `follow_symlinks` , 如下：

如果你想保留被複制目錄中的符號鏈接，像這樣做：

```
shutil.copytree(src, dst, symlinks=True)
```

`copytree()` 可以讓你在複製過程中選擇性的忽略某些文件或目錄。你可以提供一個忽略函數，接受一個目錄名和文件名列表作爲輸入，返回一個忽略的名稱列表。例如：

```
def ignore_pyc_files(dirname, filenames):
    return [name in filenames if name.endswith('.pyc')]

shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

由於忽略某種模式的文件名是很常見的，因此一個便捷的函數 `ignore_patterns()` 已經包含在裏面了。例如：

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('*~', '*.pyc'))
```

討論

使用 `shutil` 複製文件和目錄也忒簡單了點吧。不過，對於文件元數據信息，`copy2()` 這樣的函數只能儘自己最大能力來保留它。訪問時間、創建時間和權限這些基本信息會被保留，但是對於所有者、ACLs、資源 fork 和其他更深層次的文件元信息就說不準了，這個還得依賴於底層操作系統類型和用戶所擁有的訪問權限。你通常不會去使用 `shutil.copytree()` 函數來執行系統備份。當處理文件名的時候，最好使用 `os.path` 中的函數來確保最大的可移植性（特別是同時要適用於 Unix 和 Windows）。例如：

```
>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path
```

```
>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
>>> os.path.expanduser('~/guido/programs/spam.py')
'/Users/guido/programs/spam.py'
>>>
```

使用 copytree() 複製文件夾的一個棘手的問題是對於錯誤的處理。例如，在複製過程中，函數可能會碰到損壞的符號鏈接，因爲權限無法訪問文件的問題等等。爲了解決這個問題，所有碰到的問題會被收集到一個列表中並打包爲一個單獨的異常，到了最後再拋出。下面是一個例子：

```
try:
    shutil.copytree(src, dst)
except shutil.Error as e:
    for src, dst, msg in e.args[0]:
         # src is source name
         # dst is destination name
         # msg is error message from exception
         print(dst, src, msg)
```

如果你提供關鍵字參數 ignore_dangling_symlinks=True ，這時候 copytree() 會忽略掉無效符號鏈接。

本節演示的這些函數都是最常見的。不過，shutil 還有更多的和複製數據相關的操作。它的文檔很值得一看，參考 Python documentation

13.8 創建和解壓歸檔文件

問題

你需要創建或解壓常見格式的歸檔文件（比如.tar, .tgz 或.zip）

解決方案

shutil 模塊擁有兩個函數——make_archive() 和 unpack_archive() 可派上用場。例如：

```
>>> import shutil
>>> shutil.unpack_archive('Python-3.3.0.tgz')

>>> shutil.make_archive('py33','zip','Python-3.3.0')
```

```
'/Users/beazley/Downloads/py33.zip'
>>>
```

make_archive() 的第二個參數是期望的輸出格式。可以使用 get_archive_formats() 獲取所有支持的歸檔格式列表。例如：

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>>
```

討論

Python 還有其他的模塊可用來處理多種歸檔格式（比如 tarfile, zipfile, gzip, bz2）的底層細節。不過，如果你僅僅只是要創建或提取某個歸檔，就沒有必要使用底層庫了。可以直接使用 shutil 中的這些高層函數。

這些函數還有很多其他選項，用於日誌打印、預檢、文件權限等等。參考 shutil 文檔

13.9 通過文件名查找文件

問題

你需要寫一個涉及到文件查找操作的腳本，比如對日誌歸檔文件的重命名工具，你不想在 Python 腳本中調用 shell，或者你要實現一些 shell 不能做的功能。

解決方案

查找文件，可使用 os.walk() 函數，傳一個頂級目錄名給它。下面是一個例子，查找特定的文件名並答應所有符合條件的文件全路徑：

```
#!/usr/bin/env python3.3
import os

def findfile(start, name):
    for relpath, dirs, files in os.walk(start):
        if name in files:
            full_path = os.path.join(start, relpath, name)
            print(os.path.normpath(os.path.abspath(full_path)))

if __name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```

保存腳本爲文件 findfile.py，然後在命令行中執行它。指定初始查找目錄以及名字作爲位置參數，如下：

討論

os.walk() 方法爲我們遍歷目錄樹，每次進入一個目錄，它會返回一個三元組，包含相對於查找目錄的相對路徑，一個該目錄下的目錄名列表，以及那個目錄下面的文件名列表。

對於每個元組，只需檢測一下目標文件名是否在文件列表中。如果是就使用 os.path.join() 合併路徑。爲了避免奇怪的路徑名比如 ././foo//bar ，使用了另外兩個函數來修正結果。第一個是 os.path.abspath() , 它接受一個路徑，可能是相對路徑，最後返回絕對路徑。第二個是 os.path.normpath() ，用來返回正常路徑，可以解決雙斜杆、對目錄的多重引用的問題等。

儘管這個腳本相對於 UNIX 平臺上面的很多查找來講要簡單很多，它還有跨平臺的優勢。並且，還能很輕鬆的加入其他的功能。我們再演示一個例子，下面的函數打印所有最近被修改過的文件：

```
#!/usr/bin/env python3.3

import os
import time

def modified_within(top, seconds):
    now = time.time()
    for path, dirs, files in os.walk(top):
        for name in files:
            fullpath = os.path.join(path, name)
            if os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if mtime > (now - seconds):
                    print(fullpath)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit(1)

    modified_within(sys.argv[1], float(sys.argv[2]))
```

在此函數的基礎之上，使用 os,os.path,glob 等類似模塊，你就能實現更加複雜的操作了。可參考 5.11 小節和 5.13 小節等相關章節。

13.10 讀取配置文件

問題

怎樣讀取普通.ini 格式的配置文件？

解決方案

configparser 模塊能被用來讀取配置文件。例如，假設你有如下的配置文件：

```
; config.ini
; Sample configuration file

[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Setting related to debug configuration
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
    =================================
    Brought to you by the Python Cookbook
    =================================
```

下面是一個讀取和提取其中值的例子：

```
>>> from configparser import ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation','library')
'/usr/local/lib'
>>> cfg.getboolean('debug','log_errors')

True
>>> cfg.getint('server','port')
8080
>>> cfg.getint('server','nworkers')
32
>>> print(cfg.get('server','signature'))

\=================================
Brought to you by the Python Cookbook
\=================================
>>>
```

如果有需要，你還能修改配置並使用 `cfg.write()` 方法將其寫回到文件中。例如：

```
>>> cfg.set('server','port','9000')
>>> cfg.set('debug','log_errors','False')
>>> import sys
>>> cfg.write(sys.stdout)
```

```
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
          ==================================
          Brought to you by the Python Cookbook
          ==================================
>>>
```

討論

配置文件作爲一種可讀性很好的格式，非常適用於存儲程序中的配置數據。在每個配置文件中，配置數據會被分組（比如例子中的“installation”、“debug”和“server”）。每個分組在其中指定對應的各個變量值。

對於可實現同樣功能的配置文件和 Python 源文件是有很大的不同的。首先，配置文件的語法要更自由些，下面的賦值語句是等效的：

```
prefix=/usr/local
prefix: /usr/local
```

配置文件中的名字是不區分大小寫的。例如：

```
>>> cfg.get('installation','PREFIX')
'/usr/local'
>>> cfg.get('installation','prefix')
'/usr/local'
>>>
```

在解析值的時候，`getboolean()` 方法查找任何可行的值。例如下面都是等價的：

```
log_errors = true
log_errors = TRUE
log_errors = Yes
log_errors = 1
```

或許配置文件和 Python 代碼最大的不同在於，它並不是從上而下的順序執行。文件是安裝一個整體被讀取的。如果碰到了變量替換，它實際上已經被替換完成了。例如，在下面這個配置中，prefix 變量在使用它的變量之前或之後定義都是可以的：

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

ConfigParser 有個容易被忽視的特性是它能一次讀取多個配置文件然後合併成一個配置。例如，假設一個用戶像下面這樣構造了他們的配置文件：

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test

[debug]
log_errors=False
```

讀取這個文件，它就能跟之前的配置合併起來。如：

```
>>> # Previously read configuration
>>> cfg.get('installation', 'prefix')
'/usr/local'

>>> # Merge in user-specific configuration
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']

>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

仔細觀察下 prefix 變量是怎樣覆蓋其他相關變量的，比如 library 的設定值。產生這種結果的原因是變量的改寫採取的是後發制人策略，以最後一個爲準。你可以像下面這樣做試驗：

```
>>> cfg.get('installation','library')
'/Users/beazley/test/lib'
```

```
>>> cfg.set('installation','prefix','/tmp/dir')
>>> cfg.get('installation','library')
'/tmp/dir/lib'
>>>
```

最後還有很重要一點要注意的是 Python 並不能支持.ini 文件在其他程序（比如 windows 應用程序）中的所有特性。確保你已經參閱了 configparser 文檔中的語法詳情以及支持特性。

13.11 給簡單腳本增加日誌功能

問題

你希望在腳本和程序中將診斷信息寫入日誌文件。

解決方案

打印日誌最簡單方式是使用 `logging` 模塊。例如：

```
import logging

def main():
    # Configure the logging system
    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Variables (to make the calls that follow work)
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'

    # Example logging calls (insert into your program)
    logging.critical('Host %s unknown', hostname)
    logging.error("Couldn't find %r", item)
    logging.warning('Feature is deprecated')
    logging.info('Opening file %r, mode=%r', filename, mode)
    logging.debug('Got here')

if __name__ == '__main__':
    main()
```

上面五個日誌調用（critical(), error(), warning(), info(), debug()）以降序方式表示不同的嚴重級別。`basicConfig()` 的 `level` 參數是一個過濾器。所有級別低於此級別的日誌消息都會被忽略掉。每個 logging 操作的參數是一個消息字符串，後面再跟一個或多個參數。構造最終的日誌消息的時候我們使用了% 操作符來格式化消息字符串。

運行這個程序後，在文件 app.log 中的內容應該是下面這樣：

```
CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'
```

如果你想改變輸出等級，你可以修改 basicConfig() 調用中的參數。例如：

```
logging.basicConfig(
     filename='app.log',
     level=logging.WARNING,
     format='%(levelname)s:%(asctime)s:%(message)s')
```

最後輸出變成如下：

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

上面的日誌配置都是硬編碼到程序中的。如果你想使用配置文件，可以像下面這樣修改 basicConfig() 調用：

```
import logging
import logging.config

def main():
    # Configure the logging system
    logging.config.fileConfig('logconfig.ini')
    ...
```

創建一個下面這樣的文件，名字叫 logconfig.ini ：

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
format=%(levelname)s:%(name)s:%(message)s
```

如果你想修改配置，可以直接編輯文件 logconfig.ini 即可。

討論

儘管對於 logging 模塊而已有很多更高級的配置選項，不過這裏的方案對於簡單的程序和腳本已經足夠了。只想在調用日誌操作前先執行下 basicConfig() 函數方法，你的程序就能產生日誌輸出了。

如果你想要你的日誌消息寫到標準錯誤中，而不是日誌文件中，調用 basicConfig() 時不傳文件名參數即可。例如：

```
logging.basicConfig(level=logging.INFO)
```

basicConfig() 在程序中只能被執行一次。如果你稍後想改變日誌配置，就需要先獲取 root logger ，然後直接修改它。例如：

```
logging.getLogger().level = logging.DEBUG
```

需要強調的是本節只是演示了 logging 模塊的一些基本用法。它可以做更多更高級的定製。關於日誌定製化一個很好的資源是 Logging Cookbook

13.12 給函數庫增加日誌功能

問題

你想給某個函數庫增加日誌功能，但是又不能影響到那些不使用日誌功能的程序。

解決方案

對於想要執行日誌操作的函數庫而已，你應該創建一個專屬的 logger 對象，並且像下面這樣初始化配置：

```
# somelib.py

import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

# Example function (for testing)
def func():
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

使用這個配置，默認情況下不會打印日誌。例如：

```
>>> import somelib
>>> somelib.func()
>>>
```

不過，如果配置過日誌系統，那麼日誌消息打印就開始生效，例如：

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

討論

通常來講，你不應該在函數庫代碼中自己配置日誌系統，或者是已經假定有個已經存在的日誌配置了。

調用 `getLogger(__name__)` 創建一個和調用模塊同名的 logger 模塊。由於模塊都是唯一的，因此創建的 logger 也將是唯一的。

`log.addHandler(logging.NullHandler())` 操作將一個空處理器綁定到剛剛已經創建好的 logger 對象上。一個空處理器默認會忽略調用所有的日誌消息。因此，如果使用該函數庫的時候還沒有配置日誌，那麼將不會有消息或警告出現。

還有一點就是對於各個函數庫的日誌配置可以是相互獨立的，不影響其他庫的日誌配置。例如，對於如下的代碼：

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)

>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!

>>> # Change the logging level for 'somelib' only
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

在這裏，根日誌被配置成僅僅輸出 ERROR 或更高級別的消息。不過，somelib 的日誌級別被單獨配置成可以輸出 debug 級別的消息，它的優先級比全局配置高。像這樣更改單獨模塊的日誌配置對於調試來講是很方便的，因爲你無需去更改任何的全局日誌配置——只需要修改你想要更多輸出的模塊的日誌等級。

Logging HOWTO 詳細介紹瞭如何配置日誌模塊和其他有用技巧，可以參閱下。

13.13 實現一個計時器

問題

你想記錄程序執行多個任務所花費的時間

解決方案

time 模塊包含很多函數來執行跟時間有關的函數。儘管如此，通常我們會在此基礎之上構造一個更高級的接口來模擬一個計時器。例如：

```
import time

class Timer:
    def __init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def start(self):
        if self._start is not None:
            raise RuntimeError('Already started')
        self._start = self._func()

    def stop(self):
        if self._start is None:
            raise RuntimeError('Not started')
        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.elapsed = 0.0

    @property
    def running(self):
        return self._start is not None

    def __enter__(self):
        self.start()
        return self

    def __exit__(self, *args):
        self.stop()
```

這個類定義了一個可以被用戶根據需要啓動、停止和重置的計時器。它會在 elapsed 屬性中記錄整個消耗時間。下面是一個例子來演示怎樣使用它：

```
def countdown(n):
    while n > 0:
```

```
        n -= 1

# Use 1: Explicit start/stop
t = Timer()
t.start()
countdown(1000000)
t.stop()
print(t.elapsed)

# Use 2: As a context manager
with t:
    countdown(1000000)

print(t.elapsed)

with Timer() as t2:
    countdown(1000000)
print(t2.elapsed)
```

討論

本節提供了一個簡單而實用的類來實現時間記錄以及耗時計算。同時也是對使用 with 語句以及上下文管理器協議的一個很好的演示。

在計時中要考慮一個底層的時間函數問題。一般來說，使用 `time.time()` 或 `time.clock()` 計算的時間精度因操作系統的不同會有所不同。而使用 `time.perf_counter()` 函數可以確保使用系統上面最精確的計時器。

上述代碼中由 `Timer` 類記錄的時間是鐘錶時間，幷包含了所有休眠時間。如果你只想計算該進程所花費的 CPU 時間，應該使用 `time.process_time()` 來代替：

```
t = Timer(time.process_time)
with t:
    countdown(1000000)
print(t.elapsed)
```

`time.perf_counter()` 和 `time.process_time()` 都會返回小數形式的秒數時間。實際的時間值沒有任何意義，爲了得到有意義的結果，你得執行兩次函數然後計算它們的差值。

更多關於計時和性能分析的例子請參考 14.13 小節。

13.14 限制內存和 CPU 的使用量

問題

你想對在 Unix 系統上面運行的程序設置內存或 CPU 的使用限制。

解決方案

resource 模塊能同時執行這兩個任務。例如，要限制 CPU 時間，可以像下面這樣做：

```
import signal
import resource
import os

def time_exceeded(signo, frame):
    print("Time's up!")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # Install the signal handler and set a resource limit
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

if __name__ == '__main__':
    set_max_runtime(15)
    while True:
        pass
```

程序運行時，SIGXCPU 信號在時間過期時被生成，然後執行清理並退出。

要限制內存使用，設置可使用的總內存值即可，如下：

```
import resource

def limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))
```

像這樣設置了內存限制後，程序運行到沒有多餘內存時會拋出 MemoryError 異常。

討論

在本節例子中，setrlimit() 函數被用來設置特定資源上面的軟限制和硬限制。軟限制是一個值，當超過這個值的時候操作系統通常會發送一個信號來限制或通知該進程。硬限制是用來指定軟限制能設定的最大值。通常來講，這個由系統管理員通過設置系統級參數來決定。儘管硬限制可以改小一點，但是最好不要使用用戶進程去修改。

setrlimit() 函數還能被用來設置子進程數量、打開文件數以及類似系統資源的限制。更多詳情請參考 resource 模塊的文檔。

需要注意的是本節內容只能適用於 Unix 系統，並且不保證所有系統都能如期工作。比如我們在測試的時候，它能在 Linux 上面正常運行，但是在 OS X 上卻不能。

13.15 啓動一個 WEB 瀏覽器

問題

你想通過腳本啓動瀏覽器並打開指定的 URL 網頁

解決方案

webbrowser 模塊能被用來啓動一個瀏覽器，並且與平臺無關。例如：

```
>>> import webbrowser
>>> webbrowser.open('http://www.python.org')
True
>>>
```

它會使用默認瀏覽器打開指定網頁。如果你還想對網頁打開方式做更多控制，還可以使用下面這些函數：

```
>>> # Open the page in a new browser window
>>> webbrowser.open_new('http://www.python.org')
True
>>>

>>> # Open the page in a new browser tab
>>> webbrowser.open_new_tab('http://www.python.org')
True
>>>
```

這樣就可以打開一個新的瀏覽器窗口或者標籤，只要瀏覽器支持就行。

如果你想指定瀏覽器類型，可以使用 webbrowser.get() 函數來指定某個特定瀏覽器。例如：

```
>>> c = webbrowser.get('firefox')
>>> c.open('http://www.python.org')
True
>>> c.open_new_tab('http://docs.python.org')
True
>>>
```

對於支持的瀏覽器名稱列表可查閱 ‘Python 文檔 <http://docs.python.org/3/library/webbrowser.html>‘_

討論

在腳本中打開瀏覽器有時候會很有用。例如，某個腳本執行某個服務器發佈任務，你想快速打開一個瀏覽器來確保它已經正常運行了。或者是某個程序以 HTML 網頁格式輸出數據，你想打開瀏覽器查看結果。不管是上面哪種情況，使用 webbrowser 模塊都是一個簡單實用的解決方案。

第十四章：測試、調試和異常

試驗還是很棒的，但是調試？就沒那麼有趣了。事實是，在 Python 測試代碼之前沒有編譯器來分析你的代碼，因此使的測試成爲開發的一個重要部分。本章的目標是討論一些關於測試、調試和異常處理的常見問題。但是並不是爲測試驅動開發或者單元測試模塊做一個簡要的介紹。因此，筆者假定讀者熟悉測試概念。

14.1 測試 stdout 輸出

問題

你的程序中有個方法會輸出到標準輸出中（sys.stdout）。也就是說它會將文本打印到屏幕上面。你想寫個測試來證明它，給定一個輸入，相應的輸出能正常顯示出來。

解決方案

使用 unittest.mock 模塊中的 patch() 函數，使用起來非常簡單，可以爲單個測試模擬 sys.stdout 然後回滾，並且不產生大量的臨時變量或在測試用例直接暴露狀態變量。

作爲一個例子，我們在 mymodule 模塊中定義如下一個函數：

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

默認情況下內置的 print 函數會將輸出發送到 sys.stdout 。爲了測試輸出真的在那裏，你可以使用一個替身對象來模擬它，然後使用斷言來確認結果。使用 unittest.mock 模塊的 patch() 方法可以很方便的在測試運行的上下文中替換對象，並且當測試完成時候自動返回它們的原有狀態。下面是對 mymodule 模塊的測試代碼：

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, domain)

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

討論

urlprint() 函數接受三個參數，測試方法開始會先設置每一個參數的值。expected_url 變量被設置成包含期望的輸出的字符串。

unittest.mock.patch() 函數被用作一個上下文管理器，使用 StringIO 對象來代替 sys.stdout . fake_out 變量是在該進程中被創建的模擬對象。在 with 語句中使用它可以執行各種檢查。當 with 語句結束時，patch 會將所有東西恢復到測試開始前的狀態。有一點需要注意的是某些對 Python 的 C 擴展可能會忽略掉 sys.stdout 的配置二直接寫入到標準輸出中。限於篇幅，本節不會涉及到這方面的講解，它適用於純 Python 代碼。如果你真的需要在 C 擴展中捕獲 I/O，你可以先打開一個臨時文件，然後將標準輸出重定向到該文件中。更多關於捕獲以字符串形式捕獲 I/O 和 StringIO 對象請參閱 5.6 小節。

14.2 在單元測試中給對象打補丁

問題

你寫的單元測試中需要給指定的對象打補丁，用來斷言它們在測試中的期望行爲 (比如，斷言被調用時的參數個數，訪問指定的屬性等)。

解決方案

unittest.mock.patch() 函數可被用來解決這個問題。patch() 還可被用作一個裝飾器、上下文管理器或單獨使用，儘管並不常見。例如，下面是一個將它當做裝飾器使用的例子：

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)       # Uses patched example.func
    mock_func.assert_called_with(x)
```

它還可以被當做一個上下文管理器：

```
with patch('example.func') as mock_func:
    example.func(x)      # Uses patched example.func
    mock_func.assert_called_with(x)
```

最後，你還可以手動的使用它打補丁：

```
p = patch('example.func')
mock_func = p.start()
```

```
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

如果可能的話，你能夠疊加裝飾器和上下文管理器來給多個對象打補丁。例如：

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
         patch('example.patch2') as mock2, \
         patch('example.patch3') as mock3:
    ...
```

討論

patch() 接受一個已存在對象的全路徑名，將其替換爲一個新的值。原來的值會在裝飾器函數或上下文管理器完成後自動恢復回來。默認情況下，所有值會被 MagicMock 實例替代。例如：

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

不過，你可以通過給 patch() 提供第二個參數來將值替換成任何你想要的：

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

被用來作爲替換值的 MagicMock 實例能夠模擬可調用對象和實例。他們記錄對象的使用信息並允許你執行斷言檢查，例如：

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called

>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

一般來講，這些操作會在一個單元測試中完成。例如，假設你已經有了像下面這樣的函數：

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=sl1')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

正常來講，這個函數會使用 urlopen() 從 Web 上面獲取數據並解析它。在單元測試中，你可以給它一個預先定義好的數據集。下面是使用補丁操作的例子：

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                         {'IBM': 91.1,
                          'AA': 13.25,
                          'MSFT' : 27.72})

if __name__ == '__main__':
    unittest.main()
```

本例中，位於 example 模塊中的 urlopen() 函數被一個模擬對象替代，該對象會返回一個包含測試數據的 ByteIO().

還有一點，在打補丁時我們使用了 example.urlopen 來代替 urllib.request.urlopen 。當你創建補丁的時候，你必須使用它們在測試代碼中的名稱。由於測試代碼使用了 from urllib.request import urlopen , 那麼 dowprices() 函數中使用的 urlopen() 函數實際上就位於 example 模塊了。

本節實際上只是對 unittest.mock 模塊的一次淺嘗輒止。更多更高級的特性，請參考 官方文檔

14.3 在單元測試中測試異常情況

問題

你想寫個測試用例來準確的判斷某個異常是否被拋出。

解決方案

對於異常的測試可使用 assertRaises() 方法。例如，如果你想測試某個函數拋出了 ValueError 異常，像下面這樣寫：

```
import unittest

# A simple function to illustrate
def parse_int(s):
    return int(s)

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')
```

如果你想测試異常的具體值，需要用到另外一種方法：

```
import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')
        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)

        else:
            self.fail('IOError not raised')
```

討論

assertRaises() 方法爲測試異常存在性提供了一個簡便方法。一個常見的陷阱是手動去進行異常檢測。比如：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
```

這種方法的問題在於它很容易遺漏其他情況，比如沒有任何異常抛出的時候。那麼你還得需要增加另外的檢測過程，如下面這樣：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
        else:
            self.fail('ValueError not raised')
```

assertRaises() 方法會處理所有細節，因此你應該使用它。

assertRaises() 的一個缺點是它測不了異常具體的值是多少。爲了測試異常值，可以使用 assertRaisesRegex() 方法，它可同時測試異常的存在以及通過正則式匹配異常的字符串表示。例如：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                               parse_int, 'N/A')
```

assertRaises() 和 assertRaisesRegex() 還有一個容易忽略的地方就是它們還能被當做上下文管理器使用：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')
```

但你的測試涉及到多個執行步驟的時候這種方法就很有用了。

14.4 將測試輸出用日誌記錄到文件中

問題

你希望將單元測試的輸出寫到到某個文件中去，而不是打印到標準輸出。

解決方案

運行單元測試一個常見技術就是在測試文件底部加入下面這段代碼片段：

```
import unittest

class MyTest(unittest.TestCase):
    pass

if __name__ == '__main__':
    unittest.main()
```

這樣的話測試文件就是可執行的，並且會將運行測試的結果打印到標準輸出上。如果你想重定向輸出，就需要像下面這樣修改 main() 函數：

```
import sys

def main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out,verbosity=verbosity).run(suite)

if __name__ == '__main__':
```

```
    with open('testing.out', 'w') as f:
        main(f)
```

討論

本節感興趣的部分並不是將測試結果重定向到一個文件中，而是通過這樣做向你展示了 unittest 模塊中一些值得關注的內部工作原理。

unittest 模塊首先會組裝一個測試套件。這個測試套件包含了你定義的各種方法。一旦套件組裝完成，它所包含的測試就可以被執行了。

這兩步是分開的，unittest.TestLoader 實例被用來組裝測試套件。loadTestsFromModule() 是它定義的方法之一，用來收集測試用例。它會爲 TestCase 類掃描某個模塊並將其中的測試方法提取出來。如果你想進行細粒度的控制，可以使用 loadTestsFromTestCase() 方法來從某個繼承 TestCase 的類中提取測試方法。TextTestRunner 類是一個測試運行類的例子，這個類的主要用途是執行某個測試套件中包含的測試方法。這個類跟執行 unittest.main() 函數所使用的測試運行器是一樣的。不過，我們在這裏對它進行了一些列底層配置，包括輸出文件和提升級別。儘管本節例子代碼很少，但是能指導你如何對 unittest 框架進行更進一步的自定義。要想自定義測試套件的裝配方式，你可以對 TestLoader 類執行更多的操作。爲了自定義測試運行，你可以構造一個自己的測試運行類來模擬 TextTestRunner 的功能。而這些已經超出了本節的範圍。unittest 模塊的文檔對底層實現原理有更深入的講解，可以去看看。

14.5 忽略或期望測試失敗

問題

你想在單元測試中忽略或標記某些測試會按照預期運行失敗。

解決方案

unittest 模塊有裝飾器可用來控制對指定測試方法的處理，例如：

```
import unittest
import os
import platform

class Tests(unittest.TestCase):
    def test_0(self):
        self.assertTrue(True)

    @unittest.skip('skipped test')
    def test_1(self):
        self.fail('should have failed!')
```

```
    @unittest.skipIf(os.name=='posix', 'Not supported on Unix')
    def test_2(self):
        import winreg

    @unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific test')
    def test_3(self):
        self.assertTrue(True)

    @unittest.expectedFailure
    def test_4(self):
        self.assertEqual(2+2, 5)

if __name__ == '__main__':
    unittest.main()
```

如果你在 Mac 上運行這段代碼，你會得到如下輸出：

```
bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'skipped test'
test_2 (__main__.Tests) ... skipped 'Not supported on Unix'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure

----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)
```

討論

skip() 裝飾器能被用來忽略某個你不想運行的測試。skipIf() 和 skipUnless() 對於你只想在某個特定平臺或 Python 版本或其他依賴成立時才運行測試的時候非常有用。使用 @expected 的失敗裝飾器來標記那些確定會失敗的測試，並且對這些測試你不想讓測試框架打印更多信息。

忽略方法的裝飾器還可以被用來裝飾整個測試類，比如：

```
@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific tests')
class DarwinTests(unittest.TestCase):
    pass
```

14.6 處理多個異常

問題

你有一個代碼片段可能會拋出多個不同的異常，怎樣才能不創建大量重複代碼就能處理所有的可能異常呢？

解決方案

如果你可以用單個代碼塊處理不同的異常，可以將它們放入一個元組中，如下所示：

```
try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)
```

在這個例子中，元祖中任何一個異常發生時都會執行 `remove_url()` 方法。如果你想對其中某個異常進行不同的處理，可以將其放入另外一個 `except` 語句中：

```
try:
    client_obj.get_url(url)
except (URLError, ValueError):
    client_obj.remove_url(url)
except SocketTimeout:
    client_obj.handle_url_timeout(url)
```

很多的異常會有層級關係，對於這種情況，你可能使用它們的一個基類來捕獲所有的異常。例如，下面的代碼：

```
try:
    f = open(filename)
except (FileNotFoundError, PermissionError):
    pass
```

可以被重寫爲：

```
try:
    f = open(filename)
except OSError:
    pass
```

`OSError` 是 `FileNotFoundError` 和 `PermissionError` 異常的基類。

討論

儘管處理多個異常本身並沒什麼特殊的，不過你可以使用 `as` 關鍵字來獲得被拋出異常的引用：

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
```

```
    else:
        logger.error('Unexpected error: %d', e.errno)
```

這個例子中，e 變量指向一個被拋出的 OSError 異常實例。這個在你想更進一步分析這個異常的時候會很有用，比如基於某個狀態碼來處理它。

同時還要注意的時候 except 語句是順序檢查的，第一個匹配的會執行。你可以很容易的構造多個 except 同時匹配的情形，比如：

```
>>> f = open('missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing'
>>> try:
...     f = open('missing')
... except OSError:
...     print('It failed')
... except FileNotFoundError:
...     print('File not found')
...
It failed
>>>
```

這裏的 FileNotFoundError 語句並沒有執行的原因是 OSError 更一般，它可匹配 FileNotFoundError 異常，於是就是第一個匹配的。在調試的時候，如果你對某個特定異常的類成層級關係不是很確定，你可以通過查看該異常的 __mro__ 屬性來快速瀏覽。比如：

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

上面列表中任何一個直到 BaseException 的類都能被用於 except 語句。

14.7 捕獲所有異常

問題

怎樣捕獲代碼中的所有異常？

解決方案

想要捕獲所有的異常，可以直接捕獲 Exception 即可：

```
try:
    ...
except Exception as e:
```

```
    ...
    log('Reason:', e)       # Important!
```

這個將會捕獲除了 SystemExit 、KeyboardInterrupt 和 GeneratorExit 之外的所有異常。如果你還想捕獲這三個異常，將 Exception 改成 BaseException 即可。

討論

捕獲所有異常通常是由於程序員在某些複雜操作中並不能記住所有可能的異常。如果你不是很細心的人，這也是編寫不易調試代碼的一個簡單方法。

正因如此，如果你選擇捕獲所有異常，那麼在某個地方（比如日誌文件、打印異常到屏幕）打印確切原因就比較重要了。如果你沒有這樣做，有時候你看到異常打印時可能摸不着頭腦，就像下面這樣：

```
def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Couldn't parse")
```

試着運行這個函數，結果如下：

```
>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>
```

這時候你就會撓頭想：“這咋回事啊？”假如你像下面這樣重寫這個函數：

```
def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Couldn't parse")
        print('Reason:', e)
```

這時候你能獲取如下輸出，指明瞭有個編程錯誤：

```
>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>
```

很明顯，你應該儘可能將異常處理器定義的精準一些。不過，要是你必須捕獲所有異常，確保打印正確的診斷信息或將異常傳播出去，這樣不會丟失掉異常。

14.8 創建自定義異常

問題

在你構建的應用程序中，你想將底層異常包裝成自定義的異常。

解決方案

創建新的異常很簡單——定義新的類，讓它繼承自 Exception （或者是任何一個已存在的異常類型）。例如，如果你編寫網絡相關的程序，你可能會定義一些類似如下的異常：

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

然後用戶就可以像通常那樣使用這些異常了，例如：

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

討論

自定義異常類應該總是繼承自內置的 Exception 類，或者是繼承自那些本身就是從 Exception 繼承而來的類。儘管所有類同時也繼承自 BaseException ，但你不應該使用這個基類來定義新的異常。BaseException 是爲系統退出異常而保留的，比如 KeyboardInterrupt 或 SystemExit 以及其他那些會給應用發送信號而退出的異常。因此，捕獲這些異常本身沒什麼意義。這樣的話，假如你繼承 BaseException 可能會導致你的自定義異常不會被捕獲而直接發送信號退出程序運行。

在程序中引入自定義異常可以使得你的代碼更具可讀性，能清晰顯示誰應該閱讀這個代碼。還有一種設計是將自定義異常通過繼承組合起來。在複雜應用程序中，使用基類來分組各種異常類也是很有用的。它可以讓用戶捕獲一個範圍很窄的特定異常，比如下面這樣的：

```
try:
    s.send(msg)
except ProtocolError:
    ...
```

你還能捕獲更大範圍的異常，就像下面這樣：

```
try:
    s.send(msg)
except NetworkError:
    ...
```

如果你想定義的新異常重寫了 `__init__()` 方法，確保你使用所有參數調用 `Exception.__init__()`，例如：

```
class CustomError(Exception):
    def __init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

看上去有點奇怪，不過 Exception 的默認行爲是接受所有傳遞的參數並將它們以元組形式存儲在 `.args` 屬性中. 很多其他函數庫和部分 Python 庫默認所有異常都必須有 `.args` 屬性，因此如果你忽略了這一步，你會發現有些時候你定義的新異常不會按照期望運行。爲了演示 `.args` 的使用，考慮下下面這個使用內置的 *RuntimeError‘* 異常的交互會話，注意看 raise 語句中使用的參數個數是怎樣的：

```
>>> try:
...     raise RuntimeError('It failed')
... except RuntimeError as e:
...     print(e.args)
...
('It failed',)
>>> try:
...     raise RuntimeError('It failed', 42, 'spam')
... except RuntimeError as e:

...     print(e.args)
...
('It failed', 42, 'spam')
>>>
```

關於創建自定義異常的更多信息，請參考 ‘Python 官方文檔 <https://docs.python.org/3/tutorial/errors.html>‘_

14.9 捕獲異常後拋出另外的異常

問題

你想捕獲一個異常後拋出另外一個不同的異常，同時還得在異常回溯中保留兩個異常的信息。

解決方案

爲了鏈接異常，使用 raise from 語句來代替簡單的 raise 語句。它會讓你同時保留兩個異常的信息。例如：

```
>>> def example():
...     try:
...             int('N/A')
...     except ValueError as e:
...             raise RuntimeError('A parsing error occurred') from e
...
>>> example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
```

上面的異常是下面的異常產生的直接原因：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example
RuntimeError: A parsing error occurred
>>>
```

在回溯中可以看到，兩個異常都被捕獲。要想捕獲這樣的異常，你可以使用一個簡單的 except 語句。不過，你還可以通過查看異常對象的 __cause__ 屬性來跟蹤異常鏈。例如：

```
try:
    example()
except RuntimeError as e:
    print("It didn't work:", e)

    if e.__cause__:
        print('Cause:', e.__cause__)
```

當在 except 塊中又有另外的異常被拋出時會導致一個隱藏的異常鏈的出現。例如：

```
>>> def example2():
...     try:
...             int('N/A')
...     except ValueError as e:
...             print("Couldn't parse:", err)
...
>>>
```

```
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'N/A'
```

在處理上述異常的時候，另外一個異常發生了：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
NameError: global name 'err' is not defined
>>>
```

這個例子中，你同時獲得了兩個異常的信息，但是對異常的解釋不同。這時候，NameError 異常被作爲程序最終異常被拋出，而不是位於解析異常的直接回應中。

如果，你想忽略掉異常鏈，可使用 raise from None：

```
>>> def example3():
...     try:
...             int('N/A')
...     except ValueError:
...             raise RuntimeError('A parsing error occurred') from None
...
>>>
example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>
```

討論

在設計代碼時，在另外一個 except 代碼塊中使用 raise 語句的時候你要特別小心了。大多數情況下，這種 raise 語句都應該被改成 raise from 語句。也就是說你應該使用下面這種形式：

```
try:
   ...
except SomeException as e:
   raise DifferentException() from e
```

這樣做的原因是你應該顯示的將原因鏈接起來。也就是說，DifferentException 是直接從 SomeException 衍生而來。這種關係可以從回溯結果中看出來。

如果你像下面這樣寫代碼，你仍然會得到一個鏈接異常，不過這個並沒有很清晰的說明這個異常鏈到底是內部異常還是某個未知的編程錯誤。

```
try:
    ...
except SomeException:
    raise DifferentException()
```

當你使用 raise from 語句的話，就很清楚的表明拋出的是第二個異常。

最後一個例子中隱藏異常鏈信息。儘管隱藏異常鏈信息不利於回溯，同時它也丟失了很多有用的調試信息。不過萬事皆平等，有時候只保留適當的信息也是很有用的。

14.10 重新拋出被捕獲的異常

問題

你在一個 except 塊中捕獲了一個異常，現在想重新拋出它。

解決方案

簡單的使用一個單獨的 rasie 語句即可，例如：

```
>>> def example():
...     try:
...             int('N/A')
...     except ValueError:
...             print("Didn't work")
...             raise
...

>>> example()
Didn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

討論

這個問題通常是當你需要在捕獲異常後執行某個操作（比如記錄日誌、清理等），但是之後想將異常傳播下去。一個很常見的用法是在捕獲所有異常的處理器中：

```
try:
    ...
except Exception as e:
    # Process exception information in some way
    ...
```

```
    # Propagate the exception
    raise
```

14.11 輸出警告信息

問題

你希望自己的程序能生成警告信息（比如廢棄特性或使用問題）。

解決方案

要輸出一個警告消息，可使用 warning.warn() 函數。例如：

```
import warnings

def func(x, y, logfile=None, debug=False):
    if logfile is not None:
         warnings.warn('logfile argument deprecated', DeprecationWarning)
    ...
```

warn() 的參數是一個警告消息和一個警告類，警告類有如下幾種：UserWarning, DeprecationWarning, SyntaxWarning, RuntimeWarning, ResourceWarning, 或 FutureWarning.

對警告的處理取決於你如何運行解釋器以及一些其他配置。例如，如果你使用 -W all 選項去運行 Python，你會得到如下的輸出：

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: logfile argument is deprecated
  warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

通常來講，警告會輸出到標準錯誤上。如果你想講警告轉換爲異常，可以使用 -W error 選項：

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('logfile argument is deprecated', DeprecationWarning)
DeprecationWarning: logfile argument is deprecated
bash %
```

討論

在你維護軟件，提示用戶某些信息，但是又不需要將其上升爲異常級別，那麼輸出警告信息就會很有用了。例如，假設你準備修改某個函數庫或框架的功能，你可以先爲

你要更改的部分輸出警告信息，同時向後兼容一段時間。你還可以警告用戶一些對代碼有問題的使用方式。

作爲另外一個內置函數庫的警告使用例子，下面演示了一個沒有關閉文件就銷燬它時產生的警告消息：

```
>>> import warnings
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper name='/etc/
↪passwd'
 mode='r' encoding='UTF-8'>
>>>
```

默認情況下，並不是所有警告消息都會出現。`-W` 選項能控制警告消息的輸出。`-W all` 會輸出所有警告消息，`-W ignore` 忽略掉所有警告，`-W error` 將警告轉換成異常。另外一種選擇，你還可以使用 `warnings.simplefilter()` 函數控制輸出。`always` 參數會讓所有警告消息出現，`ignore 忽略調所有的警告，error 將警告轉換成異常。

對於簡單的生成警告消息的情況這些已經足夠了。`warnings` 模塊對過濾和警告消息處理提供了大量的更高級的配置選項。更多信息請參考 Python 文檔

14.12 調試基本的程序崩潰錯誤

問題

你的程序崩潰後該怎樣去調試它？

解決方案

如果你的程序因爲某個異常而崩潰，運行 `python3 -i someprogram.py` 可執行簡單的調試。`-i` 選項可讓程序結束後打開一個交互式 shell。然後你就能查看環境，例如，假設你有下面的代碼：

```
# sample.py

def func(n):
    return n + 10

func('Hello')
```

運行 `python3 -i sample.py` 會有類似如下的輸出：

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Hello')
  File "sample.py", line 4, in func
```

```
    return n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

如果你看不到上面這樣的，可以在程序崩潰後打開 Python 的調試器。例如：

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
  sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```

如果你的代碼所在的環境很難獲取交互 shell（比如在某個服務器上面），通常可以捕獲異常後自己打印跟蹤信息。例如：

```
import traceback
import sys

try:
    func(arg)
except:
    print('**** AN ERROR OCCURRED ****')
    traceback.print_exc(file=sys.stderr)
```

要是你的程序沒有崩潰，而只是產生了一些你看不懂的結果，你在感興趣的地方插入一下 `print()` 語句也是個不錯的選擇。不過，要是你打算這樣做，有一些小技巧可以幫助你。首先，`traceback.print_stack()` 函數會你程序運行到那個點的時候創建一個跟蹤棧。例如：

```
>>> def sample(n):
...     if n > 0:
...             sample(n-1)
...     else:
...             traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
```

```
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 5, in sample
>>>
```

另外，你還可以像下面這樣使用 pdb.set_trace() 在任何地方手動的啓動調試器：

```
import pdb

def func(arg):
    ...
    pdb.set_trace()
    ...
```

當程序比較大而你想調試控制流程以及函數參數的時候這個就比較有用了。例如，一旦調試器開始運行，你就能夠使用 print 來觀測變量值或敲擊某個命令比如 w 來獲取追蹤信息。

討論

不要將調試弄的過於複雜化。一些簡單的錯誤只需要觀察程序堆棧信息就能知道了，實際的錯誤一般是堆棧的最後一行。你在開發的時候，也可以在你需要調試的地方插入一下 print() 函數來診斷信息（只需要最後發佈的時候刪除這些打印語句即可）。

調試器的一個常見用法是觀測某個已經崩潰的函數中的變量。知道怎樣在函數崩潰後進入調試器是一個很有用的技能。

當你想解剖一個非常複雜的程序，底層的控制邏輯你不是很清楚的時候，插入 pdb.set_trace() 這樣的語句就很有用了。

實際上，程序會一直運行到碰到 set_trace() 語句位置，然後立馬進入調試器。然後你就可以做更多的事了。

如果你使用 IDE 來做 Python 開發，通常 IDE 都會提供自己的調試器來替代 pdb。更多這方面的信息可以參考你使用的 IDE 手冊。

14.13 給你的程序做性能測試

問題

你想測試你的程序運行所花費的時間並做性能測試。

解決方案

如果你只是簡單的想測試下你的程序整體花費的時間，通常使用 Unix 時間函數就行了，比如：

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys  0m0.098s
bash %
```

如果你還需要一個程序各個細節的詳細報告，可以使用 cProfile 模塊：

```
bash % python3 -m cProfile someprogram.py
         859647 function calls in 16.016 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   263169    0.080    0.000    0.080    0.000 someprogram.py:16(frange)
      513    0.001    0.000    0.002    0.000 someprogram.py:30(generate_
↪mandel)
   262656    0.194    0.000   15.295    0.000 someprogram.py:32(<genexpr>)
        1    0.036    0.036   16.077   16.077 someprogram.py:4(<module>)
   262144   15.021    0.000   15.021    0.000 someprogram.py:4(in_mandelbrot)
        1    0.000    0.000    0.000    0.000 os.py:746(urandom)
        1    0.000    0.000    0.000    0.000 png.py:1056(_readable)
        1    0.000    0.000    0.000    0.000 png.py:1073(Reader)
        1    0.227    0.227    0.438    0.438 png.py:163(<module>)
      512    0.010    0.000    0.010    0.000 png.py:200(group)
    ...
bash %
```

不過通常情況是介於這兩個極端之間。比如你已經知道代碼運行時在少數幾個函數中花費了絕大部分時間。對於這些函數的性能測試，可以使用一個簡單的裝飾器：

```
# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end -␣
↪start))
        return r
    return wrapper
```

要使用這個裝飾器，只需要將其放置在你要進行性能測試的函數定義前即可，比如：

```
>>> @timethis
... def countdown(n):
...     while n > 0:
...             n -= 1
...
>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>
```

要測試某個代碼塊運行時間，你可以定義一個上下文管理器，例如：

```
from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))
```

下面是使用這個上下文管理器的例子：

```
>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...             n -= 1
...
counting : 1.5551159381866455
>>>
```

對於測試很小的代碼片段運行性能，使用 `timeit` 模塊會很方便，例如：

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

`timeit` 會執行第一個參數中語句 100 萬次並計算運行時間。第二個參數是運行測試之前配置環境。如果你想改變循環執行次數，可以像下面這樣設置 `number` 參數的值：

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

討論

當執行性能測試的時候，需要注意的是你獲取的結果都是近似值。time.perf_counter() 函數會在給定平臺上獲取最高精度的計時值。不過，它仍然還是基於時鐘時間，很多因素會影響到它的精確度，比如機器負載。如果你對於執行時間更感興趣，使用 time.process_time() 來代替它。例如：

```
from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end -␣
↪start))
        return r
    return wrapper
```

最後，如果你想進行更深入的性能分析，那麼你需要詳細閱讀 time 、timeit 和其他相關模塊的文檔。這樣你可以理解和平臺相關的差異以及一些其他陷阱。還可以參考 13.13 小節中相關的一個創建計時器類的例子。

14.14 加速程序運行

問題

你的程序運行太慢，你想在不使用複雜技術比如 C 擴展或 JIT 編譯器的情況下加快程序運行速度。

解決方案

關於程序優化的第一個準則是“不要優化”，第二個準則是“不要優化那些無關緊要的部分”。如果你的程序運行緩慢，首先你得使用 14.13 小節的技術先對它進行性能測試找到問題所在。

通常來講你會發現你得程序在少數幾個熱點地方花費了大量時間，比如內存的數據處理循環。一旦你定位到這些點，你就可以使用下面這些實用技術來加速程序運行。

使用函數

很多程序員剛開始會使用 Python 語言寫一些簡單腳本。當編寫腳本的時候，通常習慣了寫毫無結構的代碼，比如：

```
# somescript.py

import sys
import csv
```

```
with open(sys.argv[1]) as f:
     for row in csv.reader(f):

          # Some kind of processing
          pass
```

很少有人知道，像這樣定義在全局範圍的代碼運行起來要比定義在函數中運行慢的多。這種速度差異是由於局部變量和全局變量的實現方式（使用局部變量要更快些）。因此，如果你想讓程序運行更快些，只需要將腳本語句放入函數中即可：

```
# somescript.py
import sys
import csv

def main(filename):
    with open(filename) as f:
         for row in csv.reader(f):
             # Some kind of processing
             pass

main(sys.argv[1])
```

速度的差異取決於實際運行的程序，不過根據經驗，使用函數帶來 15-30% 的性能提升是很常見的。

儘可能去掉屬性訪問

每一次使用點 (.) 操作符來訪問屬性的時候會帶來額外的開銷。它會觸發特定的方法，比如 `__getattribute__()` 和 `__getattr__()`，這些方法會進行字典操作操作。

通常你可以使用 `from module import name` 這樣的導入形式，以及使用綁定的方法。假設你有如下的代碼片段：

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Test
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

在我們機器上面測試的時候，這個程序花費了大概 40 秒。現在我們修改 `compute_roots()` 函數如下：

```
from math import sqrt

```

```
def compute_roots(nums):

    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

修改後的版本運行時間大概是 29 秒。唯一不同之處就是消除了屬性訪問。用 sqrt() 代替了 math.sqrt() 。The result.append() 方法被賦給一個局部變量 result_append ，然後在內部循環中使用它。

不過，這些改變只有在大量重複代碼中才有意義，比如循環。因此，這些優化也只是在某些特定地方纔應該被使用。

理解局部變量

之前提過，局部變量會比全局變量運行速度快。對於頻繁訪問的名稱，通過將這些名稱變成局部變量可以加速程序運行。例如，看下之前對於 compute_roots() 函數進行修改後的版本：

```
import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

在這個版本中，sqrt 從 match 模塊被拿出並放入了一個局部變量中。如果你運行這個代碼，大概花費 25 秒 (對於之前 29 秒又是一個改進)。這個額外的加速原因是因爲對於局部變量 sqrt 的查找要快於全局變量 sqrt

對於類中的屬性訪問也同樣適用於這個原理。通常來講，查找某個值比如 self.name 會比訪問一個局部變量要慢一些。在內部循環中，可以將某個需要頻繁訪問的屬性放入到一個局部變量中。例如：

```
# Slower
class SomeClass:
    ...
    def method(self):
         for x in s:
             op(self.value)

# Faster
class SomeClass:

    ...
    def method(self):
         value = self.value
```

```
        for x in s:
            op(value)
```

避免不必要的抽象

任何時候當你使用額外的處理層（比如裝飾器、屬性訪問、描述器）去包裝你的代碼時，都會讓程序運行變慢。比如看下如下的這個類：

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

現在進行一個簡單測試：

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

可以看到，訪問屬性 y 相比屬性 x 而言慢的不止一點點，大概慢了 4.5 倍。如果你在意性能的話，那麼就需要重新審視下對於 y 的屬性訪問器的定義是否真的有必要了。如果沒有必要，就使用簡單屬性吧。如果僅僅是因爲其他編程語言需要使用 getter/setter 函數就去修改代碼風格，這個真的沒有必要。

使用內置的容器

內置的數據類型比如字符串、元組、列表、集合和字典都是使用 C 來實現的，運行起來非常快。如果你想自己實現新的數據結構（比如鏈接列表、平衡樹等），那麼要想在性能上達到內置的速度幾乎不可能，因此，還是乖乖的使用內置的吧。

避免創建不必要的數據結構或複製

有時候程序員想顯擺下，構造一些並沒有必要的數據結構。例如，有人可能會像下面這樣寫：

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

也許這裏的想法是首先將一些值收集到一個列表中，然後使用列表推導來執行操作。不過，第一個列表完全沒有必要，可以簡單的像下面這樣寫：

```
squares = [x*x for x in sequence]
```

與此相關，還要注意下那些對 Python 的共享數據機制過於偏執的程序所寫的代碼。有些人並沒有很好的理解或信任 Python 的內存模型，濫用 `copy.deepcopy()` 之類的函數。通常在這些代碼中是可以去掉複製操作的。

討論

在優化之前，有必要先研究下使用的算法。選擇一個複雜度爲 O(n log n) 的算法要比你去調整一個複雜度爲 O(n**2) 的算法所帶來的性能提升要大得多。

如果你覺得你還是得進行優化，那麼請從整體考慮。作爲一般準則，不要對程序的每一個部分都去優化, 因爲這些修改會導致代碼難以閱讀和理解。你應該專注於優化產生性能瓶頸的地方，比如內部循環。

你還要注意微小優化的結果。例如考慮下面創建一個字典的兩種方式：

```
a = {
    'name' : 'AAPL',
    'shares' : 100,
    'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

後面一種寫法更簡潔一些（你不需要在關鍵字上輸入引號）。不過，如果你將這兩個代碼片段進行性能測試對比時，會發現使用 `dict()` 的方式會慢了 3 倍。看到這個，你是不是有衝動把所有使用 `dict()` 的代碼都替換成第一種。不夠，聰明的程序員只會關注他應該關注的地方，比如內部循環。在其他地方，這點性能損失沒有什麼影響。

如果你的優化要求比較高，本節的這些簡單技術滿足不了，那麼你可以研究下基於即時編譯（JIT）技術的一些工具。例如，PyPy 工程是 Python 解釋器的另外一種實現，它會分析你的程序運行並對那些頻繁執行的部分生成本機機器碼。它有時候能極大的提升性能，通常可以接近 C 代碼的速度。不過可惜的是，到寫這本書位置，PyPy 還不能完全支持 Python3. 因此，這個是你將來需要去研究的。你還可以考慮下 Numba 工程，Numba 是一個在你使用裝飾器來選擇 Python 函數進行優化時的動態編譯器。這些函數會使用 LLVM 被編譯成本地機器碼。它同樣可以極大的提升性能。但是，跟 PyPy 一樣，它對於 Python 3 的支持現在還停留在實驗階段。

最後我引用 John Ousterhout 說過的話作爲結尾：“最好的性能優化是從不工作到工作狀態的遷移”。直到你真的需要優化的時候再去考慮它。確保你程序正確的運行通常比讓它運行更快要更重要一些（至少開始是這樣的）.

第十五章：C 語言擴展

本章着眼於從 Python 訪問 C 代碼的問題。許多 Python 內置庫是用 C 寫的，訪問 C 是讓 Python 的對現有庫進行交互一個重要的組成部分。這也是一個當你面臨從 Python 2 到 Python 3 擴展代碼的問題。雖然 Python 提供了一個廣泛的編程 API，實際上有很多方法來處理 C 的代碼。相比試圖給出對於每一個可能的工具或技術的詳細參考，我麼採用的是是集中在一個小片段的 C++ 代碼，以及一些有代表性的例子來展示如何與代碼交互。這個目標是提供一系列的編程模板，有經驗的程序員可以擴展自己的使用。

這裏是我們將在大部分祕籍中工作的代碼：

```
/* sample.c */_method
#include <math.h>

/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

/* Test if (x0,y0) is in the Mandelbrot set or not */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = xtemp;
        n -= 1;
        if (x*x + y*y > 4) return 0;
    }
    return 1;
}

/* Divide two numbers */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}

/* Average values in an array */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
```

```
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}

/* A C data structure */
typedef struct Point {
    double x,y;
} Point;

/* Function involving a C data structure */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}
```

這段代碼包含了多種不同的 C 語言編程特性。首先，這裏有很多函數比如 `gcd()` 和 `is_mandel()` 。`divide()` 函數是一個返回多個值的 C 函數例子，其中有一個是通過指針參數的方式。`avg()` 函數通過一個 C 數組執行數據聚集操作。`Point` 和 `distance()` 函數涉及到了 C 結構體。

對於接下來的所有小節，先假定上面的代碼已經被寫入了一個名叫“sample.c”的文件中，然後它們的定義被寫入一個名叫“sample.h”的頭文件中，並且被編譯爲一個庫叫“libsample”，能被鏈接到其他 C 語言代碼中。編譯和鏈接的細節依據系統的不同而不同，但是這個不是我們關注的。如果你要處理 C 代碼，我們假定這些基礎的東西你都掌握了。

15.1 使用 ctypes 訪問 C 代碼

問題

你有一些 C 函數已經被編譯到共享庫或 DLL 中。你希望可以使用純 Python 代碼調用這些函數，而不用編寫額外的 C 代碼或使用第三方擴展工具。

解決方案

對於需要調用 C 代碼的一些小的問題，通常使用 Python 標準庫中的 `ctypes` 模塊就足夠了。要使用 `ctypes` ，你首先要確保你要訪問的 C 代碼已經被編譯到和 Python 解釋器兼容（同樣的架構、字大小、編譯器等）的某個共享庫中了。爲了進行本節的演示，假設你有一個共享庫名字叫 `libsample.so` ，裏面的內容就是 15 章介紹部分那樣。另外還假設這個 `libsample.so` 文件被放置到位於 `sample.py` 文件相同的目錄中了。

要訪問這個函數庫，你要先構建一個包裝它的 Python 模塊，如下這樣：

```
# sample.py
import ctypes
import os
```

```
# Try to locate the .so file in the same directory as this file
_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)

    return quot,rem.value

# void avg(double *, int n)
# Define a special type for the 'double *' argument
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

    # Cast from array.array objects
    def from_array(self, param):
        if param.typecode != 'd':
            raise TypeError('must be an array of doubles')
        ptr, _ = param.buffer_info()
        return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))

    # Cast from lists/tuples
    def from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return val
```

```
    from_tuple = from_list

    # Cast from a numpy array
    def from_ndarray(self, param):
        return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double
```

如果一切正常，你就可以加載並使用裏面定義的 C 函數了。例如：

```
>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

討論

本小節有很多值得我們詳細討論的地方。首先是對於 C 和 Python 代碼一起打包的問題，如果你在使用 ctypes 來訪問編譯後的 C 代碼，那麼需要確保這個共享庫放在 sample.py 模塊同一個地方。一種可能是將生成的 .so 文件放置在要使用它的 Python 代碼同一個目錄下。我們在 recipe-sample.py 中使用 __file__ 變量來查看它被安裝

的位置，然後構造一個指向同一個目錄中的 libsample.so 文件的路徑。

如果 C 函數庫被安裝到其他地方，那麼你就要修改相應的路徑。如果 C 函數庫在你機器上被安裝爲一個標準庫了，那麼可以使用 ctypes.util.find_library() 函數來查找：

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

一旦你知道了 C 函數庫的位置，那麼就可以像下面這樣使用 ctypes.cdll.LoadLibrary() 來加載它，其中 _path 是標準庫的全路徑：

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

函數庫被加載後，你需要編寫幾個語句來提取特定的符號並指定它們的類型。就像下面這個代碼片段一樣：

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

在這段代碼中，.argtypes 屬性是一個元組，包含了某個函數的輸入按時，而 .restype 就是相應的返回類型。ctypes 定義了大量的類型對象 (比如 c_double, c_int, c_short, c_float 等)，代表了對應的 C 數據類型。如果你想讓 Python 能夠傳遞正確的參數類型並且正確的轉換數據的話，那麼這些類型簽名的綁定是很重要的一步。如果你沒有這麼做，不但代碼不能正常運行，還可能會導致整個解釋器進程掛掉。使用 ctypes 有一個麻煩點的地方是原生的 C 代碼使用的術語可能跟 Python 不能明確的對應上來。divide() 函數是一個很好的例子，它通過一個參數除以另一個參數返回一個結果值。儘管這是一個很常見的 C 技術，但是在 Python 中卻不知道怎樣清晰的表達出來。例如，你不能像下面這樣簡單的做：

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_
↪int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```

就算這個能正確的工作，它會違反 Python 對於整數的不可更改原則，並且可能會導致整個解釋器陷入一個黑洞中。對於涉及到指針的參數，你通常需要先構建一個相應

的 ctypes 對象並像下面這樣傳進去：

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

在這裏，一個 ctypes.c_int 實例被創建並作爲一個指針被傳進去。跟普通 Python 整形不同的是，一個 c_int 對象是可以被修改的。.value 屬性可被用來獲取或更改這個值。

對於那些不像 Python 的 C 調用，通常可以寫一個小的包裝函數。這裏，我們讓 divide() 函數通過元組來返回兩個結果：

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

avg() 函數又是一個新的挑戰。C 代碼期望接受到一個指針和一個數組的長度值。但是，在 Python 中，我們必須考慮這個問題：數組是啥？它是一個列表？一個元組？還是 array 模塊中的一個數組？還是一個 numpy 數組？還是說所有都是？實際上，一個 Python“數組”有多種形式，你可能想要支持多種可能性。

DoubleArrayType 演示了怎樣處理這種情況。在這個類中定義了一個單個方法 from_param() 。這個方法的角色是接受一個單個參數然後將其向下轉換爲一個合適的 ctypes 對象（本例中是一個 ctypes.c_double 的指針）。在 from_param() 中，你可以做任何你想做的事。參數的類型名被提取出來並被用於分發到一個更具體的方法中去。例如，如果一個列表被傳遞過來，那麼 typename 就是 list ，然後 from_list 方法被調用。

對於列表和元組，from_list 方法將其轉換爲一個 ctypes 的數組對象。這個看上去有點奇怪，下面我們使用一個交互式例子來將一個列表轉換爲一個 ctypes 數組：

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

對於數組對象，from_array() 提取底層的內存指針並將其轉換爲一個 ctypes 指針對象。例如：

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>
```

from_ndarray() 演示了對於 numpy 數組的轉換操作。通過定義 DoubleArrayType 類並在 avg() 類型簽名中使用它，那麼這個函數就能接受多個不同的類數組輸入了：

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>
```

本節最後一部分向你演示了怎樣處理一個簡單的 C 結構。對於結構體，你只需要像下面這樣簡單的定義一個類，包含相應的字段和類型即可：

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

一旦類被定義後，你就可以在類型簽名中或者是需要實例化結構體的代碼中使用它。例如：

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

最後一些小的提示：如果你想在 Python 中訪問一些小的 C 函數，那麼 ctypes 是一個很有用的函數庫。儘管如此，如果你想要去訪問一個很大的庫，那麼可能就需要其他的方法了，比如 Swig (15.9 節會講到) 或 Cython (15.10 節)。

對於大型庫的訪問有個主要問題，由於 ctypes 並不是完全自動化，那麼你就必須花費大量時間來編寫所有的類型簽名，就像例子中那樣。如果函數庫夠複雜，你還得去編寫很多小的包裝函數和支持類。另外，除非你已經完全精通了所有底層的 C 接口細節，包括內存分配和錯誤處理機制，通常一個很小的代碼缺陷、訪問越界或其他類似錯誤就能讓 Python 程序奔潰。

作爲 ctypes 的一個替代，你還可以考慮下 CFFI。CFFI 提供了很多類似的功能，但是使用 C 語法並支持更多高級的 C 代碼類型。到寫這本書爲止，CFFI 還是一個相對較新的工程，但是它的流行度正在快速上升。甚至還有在討論在 Python 將來的版本中將它包含進去。因此，這個真的值得一看。

15.2 簡單的 C 擴展模塊

問題

你想不依靠其他工具，直接使用 Python 的擴展 API 來編寫一些簡單的 C 擴展模塊。

解決方案

對於簡單的 C 代碼，構建一個自定義擴展模塊是很容易的。作爲第一步，你需要確保你的 C 代碼有一個正確的頭文件。例如：

```
/* sample.h */

#include <math.h>

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

通常來講，這個頭文件要對應一個已經被單獨編譯過的庫。有了這些，下面我們演示下編寫擴展函數的一個簡單例子：

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */
```

```
static PyObject *py_gcd(PyObject *self, PyObject *args) {
  int x, y, result;

  if (!PyArg_ParseTuple(args,"ii", &x, &y)) {
    return NULL;
  }
  result = gcd(x,y);
  return Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */
static PyObject *py_in_mandel(PyObject *self, PyObject *args) {
  double x0, y0;
  int n;
  int result;

  if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
    return NULL;
  }
  result = in_mandel(x0,y0,n);
  return Py_BuildValue("i", result);
}

/* int divide(int, int, int *) */
static PyObject *py_divide(PyObject *self, PyObject *args) {
  int a, b, quotient, remainder;
  if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
    return NULL;
  }
  quotient = divide(a,b, &remainder);
  return Py_BuildValue("(ii)", quotient, remainder);
}

/* Module method table */
static PyMethodDef SampleMethods[] = {
  {"gcd",  py_gcd, METH_VARARGS, "Greatest common divisor"},
  {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},
  {"divide", py_divide, METH_VARARGS, "Integer division"},
  { NULL, NULL, 0, NULL}
};

/* Module structure */
static struct PyModuleDef samplemodule = {
  PyModuleDef_HEAD_INIT,

  "sample",           /* name of module */
  "A sample module",  /* Doc string (may be NULL) */
  -1,                 /* Size of per-interpreter state or -1 */
  SampleMethods       /* Method table */
};
```

```
/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  return PyModule_Create(&samplemodule);
}
```

要綁定這個擴展模塊，像下面這樣創建一個 setup.py 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      ext_modules=[
        Extension('sample',
                  ['pysample.c'],
                  include_dirs = ['/some/dir'],
                  define_macros = [('FOO','1')],
                  undef_macros = ['BAR'],
                  library_dirs = ['/usr/local/lib'],
                  libraries = ['sample']
                  )
        ]
)
```

爲了構建最終的函數庫，只需簡單的使用 python3 buildlib.py build_ext --inplace 命令即可：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
 -I/usr/local/include/python3.3m -c pysample.c
 -o build/temp.macosx-10.6-x86_64-3.3/pysample.o
gcc -bundle -undefined dynamic_lookup
build/temp.macosx-10.6-x86_64-3.3/pysample.o \
 -L/usr/local/lib -lsample -o sample.so
bash %
```

如上所示，它會創建一個名字叫 sample.so 的共享庫。當被編譯後，你就能將它作爲一個模塊導入進來了：

```
>>> import sample
>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)

0
>>> sample.divide(42, 8)
```

```
(5, 2)
>>>
```

如果你是在 Windows 機器上面嘗試這些步驟，可能會遇到各種環境和編譯問題，你需要花更多點時間去配置。Python 的二進制分發通常使用了 Microsoft Visual Studio 來構建。爲了讓這些擴展能正常工作，你需要使用同樣或兼容的工具來編譯它。參考相應的 Python 文檔

討論

在嘗試任何手寫擴展之前，最好能先參考下 Python 文檔中的 擴展和嵌入 Python 解釋器 . Python 的 C 擴展 API 很大，在這裏整個去講述它沒什麼實際意義。不過對於最核心的部分還是可以討論下的。

首先，在擴展模塊中，你寫的函數都是像下面這樣的一個普通原型：

```
static PyObject *py_func(PyObject *self, PyObject *args) {
  ...
}
```

PyObject 是一個能表示任何 Python 對象的 C 數據類型。在一個高級層面，一個擴展函數就是一個接受一個 Python 對象 (在 PyObject *args 中) 元組並返回一個新 Python 對象的 C 函數。函數的 self 參數對於簡單的擴展函數沒有被使用到，不過如果你想定義新的類或者是 C 中的對象類型的話就能派上用場了。比如如果擴展函數是一個類的一個方法，那麼 self 就能引用那個實例了。

PyArg_ParseTuple() 函數被用來將 Python 中的值轉換成 C 中對應表示。它接受一個指定輸入格式的格式化字符串作爲輸入，比如“i”代表整數，“d”代表雙精度浮點數，同樣還有存放轉換後結果的 C 變量的地址。如果輸入的值不匹配這個格式化字符串，就會拋出一個異常並返回一個 NULL 值。通過檢查並返回 NULL，一個合適的異常會在調用代碼中被拋出。

Py_BuildValue() 函數被用來根據 C 數據類型創建 Python 對象。它同樣接受一個格式化字符串來指定期望類型。在擴展函數中，它被用來返回結果給 Python。Py_BuildValue() 的一個特性是它能構建更加複雜的對象類型，比如元組和字典。在 py_divide() 代碼中，一個例子演示了怎樣返回一個元組。不過，下面還有一些實例：

```
return Py_BuildValue("i", 34);      // Return an integer
return Py_BuildValue("d", 3.4);     // Return a double
return Py_BuildValue("s", "Hello"); // Null-terminated UTF-8 string
return Py_BuildValue("(ii)", 3, 4); // Tuple (3, 4)
```

在擴展模塊底部，你會發現一個函數表，比如本節中的 SampleMethods 表。這個表可以列出 C 函數、Python 中使用的名字、文檔字符串。所有模塊都需要指定這個表，因爲它在模塊初始化時要被使用到。

最後的函數 PyInit_sample() 是模塊初始化函數，但該模塊第一次被導入時執行。這個函數的主要工作是在解釋器中註冊模塊對象。

最後一個要點需要提出來，使用 C 函數來擴展 Python 要考慮的事情還有很多，本

節只是一小部分。（實際上，C API 包含了超過 500 個函數）。你應該將本節當做是一個入門篇。更多高級內容，可以看看 `PyArg_ParseTuple()` 和 `Py_BuildValue()` 函數的文檔，然後進一步擴展開。

15.3 編寫擴展函數操作數組

問題

你想編寫一個 C 擴展函數來操作數組，可能是被 array 模塊或類似 Numpy 庫所創建。不過，你想讓你的函數更加通用，而不是針對某個特定的庫所生成的數組。

解決方案

爲了能讓接受和處理數組具有可移植性，你需要使用到 *Buffer Protocol*。下面是一個手寫的 C 擴展函數例子，用來接受數組數據並調用本章開篇部分的 `avg(double *buf, int len)` 函數：

```
/* Call double avg(double *, int) */
static PyObject *py_avg(PyObject *self, PyObject *args) {
  PyObject *bufobj;
  Py_buffer view;
  double result;
  /* Get the passed Python object */
  if (!PyArg_ParseTuple(args, "O", &bufobj)) {
    return NULL;
  }

  /* Attempt to extract buffer information from it */

  if (PyObject_GetBuffer(bufobj, &view,
      PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return NULL;
  }

  if (view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Check the type of items in the array */
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Pass the raw buffer and size to the C function */
```

```
  result = avg(view.buf, view.shape[0]);

  /* Indicate we're done working with the buffer */
  PyBuffer_Release(&view);
  return Py_BuildValue("d", result);
}
```

下面我們演示下這個擴展函數是如何工作的：

```
>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])

2.0
>>>
```

討論

將一個數組對象傳給 C 函數可能是一個擴展函數做的最常見的事。很多 Python 應用程序，從圖像處理到科學計算，都是基於高性能的數組處理。通過編寫能接受並操作數組的代碼，你可以編寫很好的兼容這些應用程序的自定義代碼，而不是隻能兼容你自己的代碼。

代碼的關鍵點在於 `PyBuffer_GetBuffer()` 函數。給定一個任意的 Python 對象，它會試着去獲取底層內存信息，它簡單的拋出一個異常並返回-1. 傳給 `PyBuffer_GetBuffer()` 的特殊標誌給出了所需的內存緩衝類型。例如，`PyBUF_ANY_CONTIGUOUS` 表示是一個聯繫的內存區域。

對於數組、字節字符串和其他類似對象而言，一個 `Py_buffer` 結構體包含了所有

底層內存的信息。它包含一個指向內存地址、大小、元素大小、格式和其他細節的指針。下面是這個結構體的定義：

```
typedef struct bufferinfo {
    void *buf;              /* Pointer to buffer memory */
    PyObject *obj;          /* Python object that is the owner */
    Py_ssize_t len;         /* Total size in bytes */
    Py_ssize_t itemsize;    /* Size in bytes of a single item */
    int readonly;           /* Read-only access flag */
    int ndim;               /* Number of dimensions */
    char *format;           /* struct code of a single item */
    Py_ssize_t *shape;      /* Array containing dimensions */
    Py_ssize_t *strides;    /* Array containing strides */
    Py_ssize_t *suboffsets; /* Array containing suboffsets */
} Py_buffer;
```

本節中，我們只關注接受一個雙精度浮點數數組作爲參數。要檢查元素是否是一個雙精度浮點數，只需驗證 format 屬性是不是字符串”d”. 這個也是 struct 模塊用來編碼二進制數據的。通常來講，format 可以是任何兼容 struct 模塊的格式化字符串，並且如果數組包含了 C 結構的話它可以包含多個值。一旦我們已經確定了底層的緩存區信息，那隻需要簡單的將它傳給 C 函數，然後會被當做是一個普通的 C 數組了。實際上，我們不必擔心是怎樣的數組類型或者它是被什麼庫創建出來的。這也是爲什麼這個函數能兼容 array 模塊也能兼容 numpy 模塊中的數組了。

在返回最終結果之前，底層的緩衝區視圖必須使用 PyBuffer_Release() 釋放掉。之所以要這一步是爲了能正確的管理對象的引用計數。

同樣，本節也僅僅只是演示了接受數組的一個小的代碼片段。如果你真的要處理數組，你可能會碰到多維數據、大數據、不同的數據類型等等問題，那麼就得去學更高級的東西了。你需要參考官方文檔來獲取更多詳細的細節。

如果你需要編寫涉及到數組處理的多個擴展，那麼通過 Cython 來實現會更容易下。參考 15.11 節。

15.4 在 C 擴展模塊中操作隱形指針

問題

你有一個擴展模塊需要處理 C 結構體中的指針，但是你又不想暴露結構體中任何內部細節給 Python。

解決方案

隱形結構體可以很容易的通過將它們包裝在膠囊對象中來處理。考慮我們例子代碼中的下列 C 代碼片段：

```
typedef struct Point {
    double x,y;
} Point;
```

```
extern double distance(Point *p1, Point *p2);
```

下面是一個使用膠囊包裝 Point 結構體和 distance() 函數的擴展代碼實例：

```
/* Destructor function for points */
static void del_Point(PyObject *obj) {
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
  return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

/* Create a new Point object */
static PyObject *py_Point(PyObject *self, PyObject *args) {

  Point *p;
  double x,y;
  if (!PyArg_ParseTuple(args,"dd",&x,&y)) {
    return NULL;
  }
  p = (Point *) malloc(sizeof(Point));
  p->x = x;
  p->y = y;
  return PyPoint_FromPoint(p, 1);
}

static PyObject *py_distance(PyObject *self, PyObject *args) {
  Point *p1, *p2;
  PyObject *py_p1, *py_p2;
  double result;

  if (!PyArg_ParseTuple(args,"OO",&py_p1, &py_p2)) {
    return NULL;
  }
  if (!(p1 = PyPoint_AsPoint(py_p1))) {
    return NULL;
  }
  if (!(p2 = PyPoint_AsPoint(py_p2))) {
    return NULL;
  }
  result = distance(p1,p2);
  return Py_BuildValue("d", result);
}
```

在 Python 中可以像下面這樣來使用這些函數：

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1004ea330>
>>> p2
<capsule object "Point" at 0x1005d1db0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

討論

膠囊和 C 指針類似。在內部，它們獲取一個通用指針和一個名稱，可以使用 PyCapsule_New() 函數很容易的被創建。另外，一個可選的析構函數能被綁定到膠囊上，用來在膠囊對象被垃圾回收時釋放底層的內存。

要提取膠囊中的指針，可使用 PyCapsule_GetPointer() 函數並指定名稱。如果提供的名稱和膠囊不匹配或其他錯誤出現，那麼就會拋出異常並返回 NULL。

本節中，一對工具函數——PyPoint_FromPoint() 和 PyPoint_AsPoint() 被用來創建和從膠囊對象中提取 Point 實例。在任何擴展函數中，我們會使用這些函數而不是直接使用膠囊對象。這種設計使得我們可以很容易的應對將來對 Point 底下的包裝的更改。例如，如果你決定使用另外一個膠囊了，那麼只需要更改這兩個函數即可。

對於膠囊對象一個難點在於垃圾回收和內存管理。PyPoint_FromPoint() 函數接受一個 must_free 參數，用來指定當膠囊被銷燬時底層 Point * 結構體是否應該被回收。在某些 C 代碼中，歸屬問題通常很難被處理（比如一個 Point 結構體被嵌入到一個被單獨管理的大結構體中）。程序員可以使用 extra 參數來控制，而不是單方面的決定垃圾回收。要注意的是和現有膠囊有關的析構器能使用 PyCapsule_SetDestructor() 函數來更改。

對於涉及到結構體的 C 代碼而言，使用膠囊是一個比較合理的解決方案。例如，有時候你並不關心暴露結構體的內部信息或者將其轉換成一個完整的擴展類型。通過使用膠囊，你可以在它上面放一個輕量級的包裝器，然後將它傳給其他的擴展函數。

15.5 從擴展模塊中定義和導出 C 的 API

問題

你有一個 C 擴展模塊，在內部定義了很多有用的函數，你想將它們導出爲一個公共的 C API 供其他地方使用。你想在其他擴展模塊中使用這些函數，但是不知道怎樣將它們鏈接起來，並且通過 C 編譯器/鏈接器來做看上去特別複雜（或者不可能做到）。

解決方案

本節主要問題是如何處理 15.4 小節中提到的 Point 對象。仔細回一下，在 C 代碼中包含了如下這些工具函數：

```
/* Destructor function for points */
static void del_Point(PyObject *obj) {

  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
  return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}
```

現在的問題是怎樣將 PyPoint_AsPoint() 和 Point_FromPoint() 函數作爲 API 導出，這樣其他擴展模塊能使用並鏈接它們，比如如果你有其他擴展也想使用包裝的 Point 對象。

要解決這個問題，首先要爲 sample 擴展寫個新的頭文件名叫 pysample.h ，如下：

```
/* pysample.h */
#include "Python.h"
#include "sample.h"
#ifdef __cplusplus
extern "C" {
#endif

/* Public API Table */
typedef struct {
  Point *(*aspoint)(PyObject *);
  PyObject *(*frompoint)(Point *, int);
} _PointAPIMethods;

#ifndef PYSAMPLE_MODULE
/* Method table in external module */
static _PointAPIMethods *_point_api = 0;

/* Import the API table from sample */
static int import_sample(void) {
  _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._point_api",0);
  return (_point_api != NULL) ? 1 : 0;
}

/* Macros to implement the programming interface */
#define PyPoint_AsPoint(obj) (_point_api->aspoint)(obj)
```

```
#define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus
}
#endif
```

這裏最重要的部分是函數指針表 _PointAPIMethods . 它會在導出模塊時被初始化，然後導入模塊時被查找到。修改原始的擴展模塊來填充表格並將它像下面這樣導出：

```
/* pysample.c */

#include "Python.h"
#define PYSAMPLE_MODULE
#include "pysample.h"

...
/* Destructor function for points */
static void del_Point(PyObject *obj) {
  printf("Deleting point\n");
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int free) {
  return PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

static _PointAPIMethods _point_api = {
  PyPoint_AsPoint,
  PyPoint_FromPoint
};
...

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  PyObject *m;
  PyObject *py_point_api;

  m = PyModule_Create(&samplemodule);
  if (m == NULL)
    return NULL;

  /* Add the Point C API functions */
```

```
  py_point_api = PyCapsule_New((void *) &_point_api, "sample._point_api",␣
↪NULL);
  if (py_point_api) {
    PyModule_AddObject(m, "_point_api", py_point_api);
  }
  return m;
}
```

最後，下面是一個新的擴展模塊例子，用來加載並使用這些 API 函數：

```
/* ptexample.c */

/* Include the header associated with the other module */
#include "pysample.h"

/* An extension function that uses the exported API */
static PyObject *print_point(PyObject *self, PyObject *args) {
  PyObject *obj;
  Point *p;
  if (!PyArg_ParseTuple(args,"O", &obj)) {
    return NULL;
  }

  /* Note: This is defined in a different module */
  p = PyPoint_AsPoint(obj);
  if (!p) {
    return NULL;
  }
  printf("%f %f\n", p->x, p->y);
  return Py_BuildValue("");
}

static PyMethodDef PtExampleMethods[] = {
  {"print_point", print_point, METH_VARARGS, "output a point"},
  { NULL, NULL, 0, NULL}
};

static struct PyModuleDef ptexamplemodule = {
  PyModuleDef_HEAD_INIT,
  "ptexample",           /* name of module */
  "A module that imports an API",  /* Doc string (may be NULL) */
  -1,                 /* Size of per-interpreter state or -1 */
  PtExampleMethods       /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_ptexample(void) {
  PyObject *m;
```

```
  m = PyModule_Create(&ptexamplemodule);
  if (m == NULL)
    return NULL;

  /* Import sample, loading its API functions */
  if (!import_sample()) {
    return NULL;
  }

  return m;
}
```

編譯這個新模塊時，你甚至不需要去考慮怎樣將函數庫或代碼跟其他模塊鏈接起來。例如，你可以像下面這樣創建一個簡單的 setup.py 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='ptexample',
      ext_modules=[
        Extension('ptexample',
                  ['ptexample.c'],
                  include_dirs = [],  # May need pysample.h directory
                  )
        ]
)
```

如果一切正常，你會發現你的新擴展函數能和定義在其他模塊中的 C API 函數一起運行的很好。

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample
>>> ptexample.print_point(p1)
2.000000 3.000000
>>>
```

討論

本節基於一個前提就是，膠囊對象能獲取任何你想要的對象的指針。這樣的話，定義模塊會填充一個函數指針的結構體，創建一個指向它的膠囊，並在一個模塊級屬性中保存這個膠囊，例如 sample._point_api .

其他模塊能夠在導入時獲取到這個屬性並提取底層的指針。事實上，Python 提供了 PyCapsule_Import() 工具函數，爲了完成所有的步驟。你只需提供屬性的名字即可 (比如 sample._point_api)，然後他就會一次性找到膠囊對象並提取出指針來。

在將被導出函數變爲其他模塊中普通函數時，有一些 C 編程陷阱需要指出來。在

pysample.h 文件中，一個 _point_api 指針被用來指向在導出模塊中被初始化的方法表。一個相關的函數 import_sample() 被用來指向膠囊導入並初始化這個指針。這個函數必須在任何函數被使用之前被調用。通常來講，它會在模塊初始化時被調用到。最後，C 的預處理宏被定義，被用來通過方法表去分發這些 API 函數。用戶只需要使用這些原始函數名稱即可，不需要通過宏去了解其他信息。

最後，還有一個重要的原因讓你去使用這個技術來鏈接模塊——它非常簡單並且可以使得各個模塊很清晰的解耦。如果你不想使用本機的技術，那你就必須使用共享庫的高級特性和動態加載器來鏈接模塊。例如，將一個普通的 API 函數放入一個共享庫並確保所有擴展模塊鏈接到那個共享庫。這種方法確實可行，但是它相對繁瑣，特別是在大型系統中。本節演示瞭如何通過 Python 的普通導入機制和僅僅幾個膠囊調用來將多個模塊鏈接起來的魔法。對於模塊的編譯，你只需要定義頭文件，而不需要考慮函數庫的內部細節。

更多關於利用 C API 來構造擴展模塊的信息可以參考 Python 的文檔

15.6 從 C 語言中調用 Python 代碼

問題

你想在 C 中安全的執行某個 Python 調用並返回結果給 C。例如，你想在 C 語言中使用某個 Python 函數作爲一個回調。

解決方案

在 C 語言中調用 Python 非常簡單，不過設計到一些小竅門。下面的 C 代碼告訴你怎樣安全的調用：

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter.  The
   arguments and return result of the function must
   be Python floats */

double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;
  PyObject *result = 0;
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();

  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  /* Build arguments */
```

```
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;

  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);

  /* Check for Python exceptions (if any) */
  if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
  }

  /* Verify the result is a float object */
  if (!PyFloat_Check(result)) {
    fprintf(stderr,"call_func: callable didn't return a float\n");
    goto fail;
  }

  /* Create the return value */
  retval = PyFloat_AsDouble(result);
  Py_DECREF(result);

  /* Restore previous GIL state and return */
  PyGILState_Release(state);
  return retval;

fail:
  Py_XDECREF(result);
  PyGILState_Release(state);
  abort();   // Change to something more appropriate
}
```

要使用這個函數，你需要獲取傳遞過來的某個已存在 Python 調用的引用。有很多種方法可以讓你這樣做，比如將一個可調用對象傳給一個擴展模塊或直接寫 C 代碼從已存在模塊中提取出來。

下面是一個簡單例子用來掩飾從一個嵌入的 Python 解釋器中調用一個函數：

```
#include <Python.h>

/* Definition of call_func() same as above */
...

/* Load a symbol from a module */
PyObject *import_name(const char *modname, const char *symbol) {
  PyObject *u_name, *module;
  u_name = PyUnicode_FromString(modname);
  module = PyImport_Import(u_name);
  Py_DECREF(u_name);
```

```
  return PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */
int main() {
  PyObject *pow_func;
  double x;

  Py_Initialize();
  /* Get a reference to the math.pow function */
  pow_func = import_name("math","pow");

  /* Call it using our call_func() code */
  for (x = 0.0; x < 10.0; x += 0.1) {
    printf("%0.2f %0.2f\n", x, call_func(pow_func,x,2.0));
  }
  /* Done */
  Py_DECREF(pow_func);
  Py_Finalize();
  return 0;
}
```

要構建例子代碼，你需要編譯 C 並將它鏈接到 Python 解釋器。下面的 Makefile 可以教你怎樣做（不過在你機器上面需要一些配置）。

```
all::
        cc -g embed.c -I/usr/local/include/python3.3m \
          -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m
```

編譯並運行會產生類似下面的輸出：

```
0.00 0.00
0.10 0.01
0.20 0.04
0.30 0.09
0.40 0.16
...
```

下面是一個稍微不同的例子，展示了一個擴展函數，它接受一個可調用對象和其他參數，並將它們傳遞給 call_func() 來做測試：

```
/* Extension function for testing the C-Python callback */
PyObject *py_call_func(PyObject *self, PyObject *args) {
  PyObject *func;

  double x, y, result;
  if (!PyArg_ParseTuple(args,"Odd", &func,&x,&y)) {
    return NULL;
  }
  result = call_func(func, x, y);
  return Py_BuildValue("d", result);
```

```
}
```

使用這個擴展函數，你要像下面這樣測試它：

```
>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>
```

討論

如果你在 C 語言中調用 Python，要記住最重要的是 C 語言會是主體。也就是說，C 語言負責構造參數、調用 Python 函數、檢查異常、檢查類型、提取返回值等。

作爲第一步，你必須先有一個表示你將要調用的 Python 可調用對象。這可以是一個函數、類、方法、內置方法或其他任意實現了 `__call__()` 操作的東西。爲了確保是可調用的，可以像下面的代碼這樣利用 `PyCallable_Check()` 做檢查：

```
double call_func(PyObject *func, double x, double y) {
  ...
  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  ...
```

在 C 代碼裏處理錯誤你需要格外的小心。一般來講，你不能僅僅拋出一個 Python 異常。錯誤應該使用 C 代碼方式來被處理。在這裏，我們打算將對錯誤的控制傳給一個叫 `abort()` 的錯誤處理器。它會結束掉整個程序，在真實環境下面你應該要處理的更加優雅些（返回一個狀態碼）。你要記住的是在這裏 C 是主角，因此並沒有跟拋出異常相對應的操作。錯誤處理是你在編程時必須要考慮的事情。

調用一個函數相對來講很簡單——只需要使用 `PyObject_Call()`，傳一個可調用對象給它、一個參數元組和一個可選的關鍵字字典。要構建參數元組或字典，你可以使用 `Py_BuildValue()`，如下：

```
double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;

  ...
  /* Build arguments */
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;
```

```
  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);
  ...
```

如果沒有關鍵字參數，你可以傳遞 NULL。當你要調用函數時，需要確保使用了 Py_DECREF() 或者 Py_XDECREF() 清理參數。第二個函數相對安全點，因爲它允許傳遞 NULL 指針（直接忽略它），這也是爲什麼我們使用它來清理可選的關鍵字參數。

調用萬 Python 函數之後，你必須檢查是否有異常發生。PyErr_Occurred() 函數可被用來做這件事。對對於異常的處理就有點麻煩了，由於是用 C 語言寫的，你沒有像 Python 那麼的異常機制。因此，你必須要設置一個異常狀態碼，打印異常信息或其他相應處理。在這裏，我們選擇了簡單的 abort() 來處理。另外，傳統 C 程序員可能會直接讓程序奔潰。

```
...
/* Check for Python exceptions (if any) */
if (PyErr_Occurred()) {
  PyErr_Print();
  goto fail;
}
...
fail:
  PyGILState_Release(state);
  abort();
```

從調用 Python 函數的返回值中提取信息通常要進行類型檢查和提取值。要這樣做的話，你必須使用 Python 對象層中的函數。在這裏我們使用了 PyFloat_Check() 和 PyFloat_AsDouble() 來檢查和提取 Python 浮點數。

最後一個問題是對於 Python 全局鎖的管理。在 C 語言中訪問 Python 的時候，你需要確保 GIL 被正確的獲取和釋放了。不然的話，可能會導致解釋器返回錯誤數據或者直接奔潰。調用 PyGILState_Ensure() 和 PyGILState_Release() 可以確保一切都能正常。

```
double call_func(PyObject *func, double x, double y) {
  ...
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();
  ...
  /* Code that uses Python C API functions */
  ...
  /* Restore previous GIL state and return */
  PyGILState_Release(state);
  return retval;

fail:
  PyGILState_Release(state);
```

```
  abort();
}
```

一旦返回，PyGILState_Ensure() 可以確保調用線程獨佔 Python 解釋器。就算 C 代碼運行於另外一個解釋器不知道的線程也沒事。這時候，C 代碼可以自由的使用任何它想要的 Python C-API 函數。調用成功後，PyGILState_Release() 被用來講解釋器恢復到原始狀態。

要注意的是每一個 PyGILState_Ensure() 調用必須跟着一個匹配的 PyGILState_Release() 調用——即便有錯誤發生。在這裏，我們使用一個 goto 語句看上去是個可怕的設計，但是實際上我們使用它來講控制權轉移給一個普通的 exit 塊來執行相應的操作。在 fail: 標籤後面的代碼和 Python 的 fianl: 塊的用途是一樣的。

如果你使用所有這些約定來編寫 C 代碼，包括對 GIL 的管理、異常檢查和錯誤檢查，你會發現從 C 語言中調用 Python 解釋器是可靠的——就算再複雜的程序，用到了高級編程技巧比如多線程都沒問題。

15.7 從 C 擴展中釋放全局鎖

問題

你想讓 C 擴展代碼和 Python 解釋器中的其他進程一起正確的執行，那麼你就需要去釋放並重新獲取全局解釋器鎖（GIL）。

解決方案

在 C 擴展代碼中，GIL 可以通過在代碼中插入下面這樣的宏來釋放和重新獲取：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code.  Must not use Python API functions
   ...
   Py_END_ALLOW_THREADS
   ...
   return result;
}
```

討論

只有當你確保沒有 Python C API 函數在 C 中執行的時候你才能安全的釋放 GIL。GIL 需要被釋放的常見的場景是在計算密集型代碼中需要在 C 數組上執行計算（比如

在 numpy 中）或者是要執行阻塞的 I/O 操作時（比如在一個文件描述符上讀取或寫入時）。

當 GIL 被釋放後，其他 Python 線程才被允許在解釋器中執行。Py_END_ALLOW_THREADS 宏會阻塞執行直到調用線程重新獲取了 GIL。

15.8 C 和 Python 中的線程混用

問題

你有一個程序需要混合使用 C、Python 和線程，有些線程是在 C 中創建的，超出了 Python 解釋器的控制範圍。並且一些線程還使用了 Python C API 中的函數。

解決方案

如果你想將 C、Python 和線程混合在一起，你需要確保正確的初始化和管理 Python 的全局解釋器鎖（GIL）。要想這樣做，可以將下列代碼放到你的 C 代碼中並確保它在任何線程被創建之前被調用。

```
#include <Python.h>
  ...
  if (!PyEval_ThreadsInitialized()) {
    PyEval_InitThreads();
  }
  ...
```

對於任何調用 Python 對象或 Python C API 的 C 代碼，確保你首先已經正確地獲取和釋放了 GIL。這可以用 PyGILState_Ensure() 和 PyGILState_Release() 來做到，如下所示：

```
...
/* Make sure we own the GIL */
PyGILState_STATE state = PyGILState_Ensure();

/* Use functions in the interpreter */
...
/* Restore previous GIL state and return */
PyGILState_Release(state);
...
```

每次調用 PyGILState_Ensure() 都要相應的調用 PyGILState_Release() .

討論

在涉及到 C 和 Python 的高級程序中，很多事情一起做是很常見的——可能是對 C、Python、C 線程、Python 線程的混合使用。只要你確保解釋器被正確的初始化，並且涉及到解釋器的 C 代碼執行了正確的 GIL 管理，應該沒什麼問題。

要注意的是調用 PyGILState_Ensure() 並不會立刻搶佔或中斷解釋器。如果有其他代碼正在執行，這個函數被中斷知道那個執行代碼釋放掉 GIL。在內部，解釋器會執行週期性的線程切換，因此如果其他線程在執行，調用者最終還是可以運行的（儘管可能要先等一會）。

15.9 用 WSIG 包裝 C 代碼

問題

你想讓你寫的 C 代碼作爲一個 C 擴展模塊來訪問，想通過使用 Swig 包裝生成器來完成。

解決方案

Swig 通過解析 C 頭文件並自動創建擴展代碼來操作。要使用它，你先要有一個 C 頭文件。例如，我們示例的頭文件如下：

```
/* sample.h */

#include <math.h>
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

一旦你有了這個頭文件，下一步就是編寫一個 Swig”接口”文件。按照約定，這些文件以”.i”後綴並且類似下面這樣：

```
// sample.i - Swig interface
%module sample
%{
#include "sample.h"
%}

/* Customizations */
%extend Point {
    /* Constructor for Point objects */
    Point(double x, double y) {
        Point *p = (Point *) malloc(sizeof(Point));
        p->x = x;
        p->y = y;
        return p;
```

```
   };
};

/* Map int *remainder as an output argument */
%include typemaps.i
%apply int *OUTPUT { int * remainder };

/* Map the argument pattern (double *a, int n) to arrays */
%typemap(in) (double *a, int n)(Py_buffer view) {
  view.obj = NULL;
  if (PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT)␣
↪== -1) {
    SWIG_fail;
  }
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    SWIG_fail;
  }
  $1 = (double *) view.buf;
  $2 = view.len / sizeof(double);
}

%typemap(freearg) (double *a, int n) {
  if (view$argnum.obj) {
    PyBuffer_Release(&view$argnum);
  }
}

/* C declarations to be included in the extension module */

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

一旦你寫好了接口文件，就可以在命令行工具中調用 Swig 了：

```
bash % swig -python -py3 sample.i
bash %
```

swig 的輸出就是兩個文件，sample_wrap.c 和 sample.py。後面的文件就是用戶需要導入的。而 sample_wrap.c 文件是需要被編譯到名叫 _sample 的支持模塊的 C 代碼。這個可以通過跟普通擴展模塊一樣的技術來完成。例如，你創建了一個如下所示的 setup.py 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      py_modules=['sample.py'],
      ext_modules=[
        Extension('_sample',
                  ['sample_wrap.c'],
                  include_dirs = [],
                  define_macros = [],

                  undef_macros = [],
                  library_dirs = [],
                  libraries = ['sample']
                  )
        ]
)
```

要編譯和測試，在 setup.py 上執行 python3，如下：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building '_sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c sample_wrap.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
sample_wrap.c: In function 'SWIG_InitializeModule':
sample_wrap.c:3589: warning: statement with no effect
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/
↪sample.o
 build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample
bash %
```

如果一切正常的話，你會發現你就可以很方便的使用生成的 C 擴展模塊了。例如：

```
>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
```

```
2.0
>>>
```

討論

Swig 是 Python 歷史中構建擴展模塊的最古老的工具之一。Swig 能自動化很多包裝生成器的處理。

所有 Swig 接口都以類似下面這樣的爲開頭：

```
%module sample
%{
#include "sample.h"
%}
```

這個僅僅只是聲明瞭擴展模塊的名稱並指定了 C 頭文件，爲了能讓編譯通過必須要包含這些頭文件（位於%{ 和%} 的代碼），將它們之間複製粘貼到輸出代碼中，這也是你要放置所有包含文件和其他編譯需要的定義的地方。

Swig 接口的底下部分是一個 C 聲明列表，你需要在擴展中包含它。這通常從頭文件中被複制。在我們的例子中，我們僅僅像下面這樣直接粘貼在頭文件中：

```
%module sample
%{
#include "sample.h"
%}
...
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

有一點需要強調的是這些聲明會告訴 Swig 你想要在 Python 模塊中包含哪些東西。通常你需要編輯這個聲明列表或相應的修改下它。例如，如果你不想某些聲明被包含進來，你要將它從聲明列表中移除掉。

使用 Swig 最複雜的地方是它能給 C 代碼提供大量的自定義操作。這個主題太大，這裏無法展開，但是我們在本節還剩展示了一些自定義的東西。

第一個自定義是 %extend 指令允許方法被附加到已存在的結構體和類定義上。我例子中，這個被用來添加一個 Point 結構體的構造器方法。它可以讓你像下面這樣使用這個結構體：

```
>>> p1 = sample.Point(2,3)
>>>
```

如果略過的話，Point 對象就必須以更加複雜的方式來被創建：

```
>>> # Usage if %extend Point is omitted
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

第二個自定義涉及到對 `typemaps.i` 庫的引入和 `%apply` 指令，它會指示 Swig 參數簽名 `int *remainder` 要被當做是輸出值。這個實際上是一個模式匹配規則。在接下來的所有聲明中，任何時候只要碰上 `int *remainder`，他就會被作爲輸出。這個自定義方法可以讓 `divide()` 函數返回兩個值。

```
>>> sample.divide(42,8)
[5, 2]
>>>
```

最後一個涉及到 `%typemap` 指令的自定義可能是這裏展示的最高級的特性了。一個 typemap 就是一個在輸入中特定參數模式的規則。在本節中，一個 typemap 被定義爲匹配參數模式 `(double *a, int n)`. 在 typemap 內部是一個 C 代碼片段，它告訴 Swig 怎樣將一個 Python 對象轉換爲相應的 C 參數。本節代碼使用了 Python 的緩存協議去匹配任何看上去類似雙精度數組的輸入參數（比如 NumPy 數組、array 模塊創建的數組等），更多請參考 15.3 小節。

在 typemap 代碼內部，$1 和 $2 這樣的變量替換會獲取 typemap 模式的 C 參數值（比如 $1 映射爲 `double *a`）。$input 指向一個作爲輸入的 `PyObject *` 參數，而 `$argnum` 就代表參數的個數。

編寫和理解 typemaps 是使用 Swig 最基本的前提。不僅是說代碼更神祕，而且你需要理解 Python C API 和 Swig 和它交互的方式。Swig 文檔有更多這方面的細節，可以參考下。

不過，如果你有大量的 C 代碼需要被暴露爲擴展模塊。Swig 是一個非常強大的工具。關鍵點在於 Swig 是一個處理 C 聲明的編譯器，通過強大的模式匹配和自定義組件，可以讓你更改聲明指定和類型處理方式。更多信息請去查閱 Swig 網站，還有 特定於 Python 的相關文檔

15.10 用 Cython 包裝 C 代碼

問題

你想使用 Cython 來創建一個 Python 擴展模塊，用來包裝某個已存在的 C 函數庫。

解決方案

使用 Cython 構建一個擴展模塊看上去很手寫擴展有些類似，因爲你需要創建很多包裝函數。不過，跟前面不同的是，你不需要在 C 語言中做這些——代碼看上去更像是 Python。

作爲準備，假設本章介紹部分的示例代碼已經被編譯到某個叫 `libsample` 的 C 函數庫中了。首先創建一個名叫 `csample.pxd` 的文件，如下所示：

```
# csample.pxd
#
# Declarations of "external" C functions and structures

cdef extern from "sample.h":
    int gcd(int, int)
    bint in_mandel(double, double, int)
    int divide(int, int, int *)
    double avg(double *, int) nogil

    ctypedef struct Point:
         double x
         double y

    double distance(Point *, Point *)
```

這個文件在 Cython 中的作用就跟 C 的頭文件一樣。初始聲明 `cdef extern from "sample.h"` 指定了所學的 C 頭文件。接下來的聲明都是來自於那個頭文件。文件名是 `csample.pxd` ，而不是 `sample.pxd` ——這點很重要。

下一步，創建一個名爲 `sample.pyx` 的問題。該文件會定義包裝器，用來橋接 Python 解釋器到 `csample.pxd` 中聲明的 C 代碼。

```
# sample.pyx

# Import the low-level C declarations
cimport csample

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *

from libc.stdlib cimport malloc, free

# Wrappers
def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x, y)

def in_mandel(x, y, unsigned int n):
    return csample.in_mandel(x, y, n)

def divide(x, y):
    cdef int rem
```

```
    quot = csample.divide(x, y, &rem)
    return quot, rem

def avg(double[:] a):
    cdef:
        int sz
        double result

    sz = a.size
    with nogil:
        result = csample.avg(<double *> &a[0], sz)
    return result

# Destructor for cleaning up Point objects
cdef del_Point(object obj):
    pt = <csample.Point *> PyCapsule_GetPointer(obj,"Point")
    free(<void *> pt)

# Create a Point object and return as a capsule
def Point(double x,double y):
    cdef csample.Point *p
    p = <csample.Point *> malloc(sizeof(csample.Point))
    if p == NULL:
        raise MemoryError("No memory to make a Point")
    p.x = x
    p.y = y
    return PyCapsule_New(<void *>p,"Point",<PyCapsule_Destructor>del_Point)

def distance(p1, p2):
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1,"Point")
    pt2 = <csample.Point *> PyCapsule_GetPointer(p2,"Point")
    return csample.distance(pt1,pt2)
```

該文件更多的細節部分會在討論部分詳細展開。最後，爲了構建擴展模塊，像下面這樣創建一個 `setup.py` 文件：

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',

              ['sample.pyx'],
              libraries=['sample'],
              library_dirs=['.'])]
setup(
  name = 'Sample extension module',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
```

```
)
```

要構建我們測試的目標模塊，像下面這樣做：

```
bash % python3 setup.py build_ext --inplace
running build_ext
cythoning sample.pyx to sample.c
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
 -I/usr/local/include/python3.3m -c sample.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample.o
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/
↪sample.o
  -L. -lsample -o sample.so
bash %
```

如果一切順利的話，你應該有了一個擴展模塊 `sample.so` ，可在下面例子中使用：

```
>>> import sample
>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

討論

本節包含了很多前面所講的高級特性，包括數組操作、包裝隱形指針和釋放 GIL。每一部分都會逐個被講述到，但是我們最好能複習一下前面幾小節。在頂層，使用 Cython 是基於 C 之上。.pxd 文件僅僅只包含 C 定義（類似.h 文件），.pyx 文件包含了實現（類似.c 文件）。`cimport` 語句被 Cython 用來導入.pxd 文件中的定義。它跟使用普通的加載 Python 模塊的導入語句是不同的。

儘管 *.pxd* 文件包含了定義，但它們並不是用來自動創建擴展代碼的。因此，你還是要寫包裝函數。例如，就算 `csample.pxd` 文件聲明瞭 `int gcd(int, int)` 函數，你仍然需要在 `sample.pyx` 中爲它寫一個包裝函數。例如：

```
cimport csample

def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x,y)
```

對於簡單的函數，你並不需要去做太多的時。Cython 會生成包裝代碼來正確的轉換參數和返回值。綁定到屬性上的 C 數據類型是可選的。不過，如果你包含了它們，你可以另外做一些錯誤檢查。例如，如果有人使用負數來調用這個函數，會拋出一個異常：

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

如果你想對包裝函數做另外的檢查，只需要使用另外的包裝代碼。例如：

```
def gcd(unsigned int x, unsigned int y):
    if x <= 0:
        raise ValueError("x must be > 0")
    if y <= 0:
        raise ValueError("y must be > 0")
    return csample.gcd(x,y)
```

在 csample.pxd 文件中的 “in_mandel()“ 聲明有個很有趣但是比較難理解的定義。在這個文件中，函數被聲明爲然後一個 bint 而不是一個 int。它會讓函數創建一個正確的 Boolean 值而不是簡單的整數。因此，返回值 0 表示 False 而 1 表示 True。

在 Cython 包裝器中，你可以選擇聲明 C 數據類型，也可以使用所有的常見 Python 對象。對於 `divide()` 的包裝器展示了這樣一個例子，同時還有如何去處理一個指針參數。

```
def divide(x,y):
    cdef int rem
    quot = csample.divide(x,y,&rem)
    return quot, rem
```

在這裏，`rem` 變量被顯示的聲明爲一個 C 整型變量。當它被傳入 `divide()` 函數的時候，`&rem` 創建一個跟 C 一樣的指向它的指針。`avg()` 函數的代碼演示了 Cython 更高級的特性。首先 `def avg(double[:] a)` 聲明瞭 `avg()` 接受一個一維的雙精度內存視圖。最驚奇的部分是返回的結果函數可以接受任何兼容的數組對象，包括被 numpy 創建的。例如：

```
>>> import array
>>> a = array.array('d',[1,2,3])
```

```
>>> import numpy
>>> b = numpy.array([1., 2., 3.])
>>> import sample
>>> sample.avg(a)
2.0
>>> sample.avg(b)
2.0
>>>
```

在此包裝器中，a.size0 和 &a[0] 分別引用數組元素個數和底層指針。語法 <double *> &a[0] 教你怎樣將指針轉換爲不同的類型。前提是 C 中的 avg() 接受一個正確類型的指針。參考下一節關於 Cython 內存視圖的更高級講述。

除了處理通常的數組外，avg() 的這個例子還展示瞭如何處理全局解釋器鎖。語句 with nogil: 聲明瞭一個不需要 GIL 就能執行的代碼塊。在這個塊中，不能有任何的普通 Python 對象——只能使用被聲明爲 cdef 的對象和函數。另外，外部函數必須現實的聲明它們能不依賴 GIL 就能執行。因此，在 csample.pxd 文件中，avg() 被聲明爲 double avg(double *, int) nogil .

對 Point 結構體的處理是一個挑戰。本節使用膠囊對象將 Point 對象當做隱形指針來處理，這個在 15.4 小節介紹過。要這樣做的話，底層 Cython 代碼稍微有點複雜。首先，下面的導入被用來引入 C 函數庫和 Python C API 中定義的函數：

```
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free
```

函數 del_Point() 和 Point() 使用這個功能來創建一個膠囊對象，它會包裝一個 Point * 指針。cdef del_Point() 將 del_Point() 聲明爲一個函數，只能通過 Cython 訪問，而不能從 Python 中訪問。因此，這個函數對外部是不可見的——它被用來當做一個回調函數來清理膠囊分配的內存。函數調用比如 PyCapsule_New() 、PyCapsule_GetPointer() 直接來自 Python C API 並且以同樣的方式被使用。

distance 函數從 Point() 創建的膠囊對象中提取指針。這裏要注意的是你不需要擔心異常處理。如果一個錯誤的對象被傳進來，PyCapsule_GetPointer() 會拋出一個異常，但是 Cython 已經知道怎麼查找到它，並將它從 distance() 傳遞出去。

處理 Point 結構體一個缺點是它的實現是不可見的。你不能訪問任何屬性來查看它的內部。這裏有另外一種方法去包裝它，就是定義一個擴展類型，如下所示：

```
# sample.pyx

cimport csample
from libc.stdlib cimport malloc, free
...

cdef class Point:
    cdef csample.Point *_c_point
    def __cinit__(self, double x, double y):
        self._c_point = <csample.Point *> malloc(sizeof(csample.Point))
        self._c_point.x = x
        self._c_point.y = y
```

```
    def __dealloc__(self):
        free(self._c_point)

    property x:
        def __get__(self):
            return self._c_point.x
        def __set__(self, value):
            self._c_point.x = value

    property y:
        def __get__(self):
            return self._c_point.y
        def __set__(self, value):
            self._c_point.y = value

def distance(Point p1, Point p2):
    return csample.distance(p1._c_point, p2._c_point)
```

在這裏，cdif 類 Point 將 Point 聲明爲一個擴展類型。類屬性 cdef csample.Point *_c_point 聲明瞭一個實例變量，擁有一個指向底層 Point 結構體的指針。__cinit__() 和 __dealloc__() 方法通過 malloc() 和 free() 創建並銷燬底層 C 結構體。x 和 y 屬性的聲明讓你獲取和設置底層結構體的屬性值。distance() 的包裝器還可以被修改，使得它能接受 Point 擴展類型實例作爲參數，而傳遞底層指針給 C 函數。

做了這個改變後，你會發現操作 Point 對象就顯得更加自然了：

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

本節已經演示了很多 Cython 的核心特性，你可以以此爲基準來構建更多更高級的包裝。不過，你最好先去閱讀下官方文檔來了解更多信息。

接下來幾節還會繼續演示一些 Cython 的其他特性。

15.11 用 Cython 寫高性能的數組操作

問題

你要寫高性能的操作來自 NumPy 之類的數組計算函數。你已經知道了 Cython 這樣的工具會讓它變得簡單，但是並不確定該怎樣去做。

解決方案

作爲一個例子，下面的代碼演示了一個 Cython 函數，用來修整一個簡單的一維雙精度浮點數數組中元素的值。

```
# sample.pyx (Cython)

cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    '''
    Clip the values in a to be between min and max. Result in out
    '''
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        if a[i] < min:
            out[i] = min
        elif a[i] > max:
            out[i] = max
        else:
            out[i] = a[i]
```

要編譯和構建這個擴展，你需要一個像下面這樣的 `setup.py` 文件（使用 `python3 setup.py build_ext --inplace` 來構建它）：

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',
              ['sample.pyx'])
]

setup(
  name = 'Sample app',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

你會發現結果函數確實對數組進行的修正，並且可以適用於多種類型的數組對象。例如：

```
>>> # array module example
>>> import sample
>>> import array
>>> a = array.array('d',[1,-3,4,7,2,0])
>>> a

array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0])
>>> sample.clip(a,1,4,a)
>>> a
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])

>>> # numpy example
>>> import numpy
>>> b = numpy.random.uniform(-10,10,size=1000000)
>>> b
array([-9.55546017,  7.45599334,  0.69248932, ...,  0.69583148,
       -3.86290931,  2.37266888])
>>> c = numpy.zeros_like(b)
>>> c
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
>>> sample.clip(b,-5,5,c)
>>> c
array([-5.        ,  5.        ,  0.69248932, ...,  0.69583148,
       -3.86290931,  2.37266888])
>>> min(c)
-5.0
>>> max(c)
5.0
>>>
```

你還會發現運行生成結果非常的快。下面我們將本例和 numpy 中的已存在的 clip() 函數做一個性能對比：

```
>>> timeit('numpy.clip(b,-5,5,c)','from __main__ import b,c,numpy',
↪number=1000)
8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)','from __main__ import b,c,sample',
...         number=1000)
3.760528204000366
>>>
```

正如你看到的，它要快很多——這是一個很有趣的結果，因爲 NumPy 版本的核心代碼還是用 C 語言寫的。

討論

本節利用了 Cython 類型的內存視圖，極大的簡化了數組的操作。cpdef clip() 聲明瞭 clip() 同時爲 C 級別函數以及 Python 級別函數。在 Cython 中，這個是很重要的，因爲它表示此函數調用要比其他 Cython 函數更加高效（比如你想在另外一個不同的 Cython 函數中調用 clip()）。

類型參數 double[:] a 和 double[:] out 聲明這些參數爲一維的雙精度數組。作爲輸入，它們會訪問任何實現了內存視圖接口的數組對象，這個在 PEP 3118 有詳細定義。包括了 NumPy 中的數組和內置的 array 庫。

當你編寫生成結果爲數組的代碼時，你應該遵循上面示例那樣設置一個輸出參數。它會將創建輸出數組的責任給調用者，不需要知道你操作的數組的具體細節（它僅僅假設數組已經準備好了，只需要做一些小的檢查比如確保數組大小是正確的）。在像 NumPy 之類的庫中，使用 numpy.zeros() 或 numpy.zeros_like() 創建輸出數組相對而言比較容易。另外，要創建未初始化數組，你可以使用 numpy.empty() 或 numpy.empty_like() . 如果你想覆蓋數組內容作爲結果的話選擇這兩個會比較快點。

你你的函數實現中，你只需要簡單的通過下標運算和數組查找（比如 a[i],out[i] 等）來編寫代碼操作數組。Cython 會負責爲你生成高效的代碼。

clip() 定義之前的兩個裝飾器可以優化下性能。@cython.boundscheck(False) 省去了所有的數組越界檢查，當你知道下標訪問不會越界的時候可以使用它。@cython.wraparound(False) 消除了相對數組尾部的負數下標的處理（類似 Python 列表）。引入這兩個裝飾器可以極大的提升性能（測試這個例子的時候大概快了 2.5 倍）。

任何時候處理數組時，研究並改善底層算法同樣可以極大的提示性能。例如，考慮對 clip() 函數的如下修正，使用條件表達式：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

實際測試結果是，這個版本的代碼運行速度要快 50% 以上（2.44 秒對比之前使用 timeit() 測試的 3.76 秒）。

到這裏爲止，你可能想知道這種代碼怎麼能跟手寫 C 語言 PK 呢？例如，你可能寫了如下的 C 函數並使用前面幾節的技術來手寫擴展：

```
void clip(double *a, int n, double min, double max, double *out) {
  double x;
  for (; n >= 0; n--, a++, out++) {
    x = *a;

    *out = x > max ? max : (x < min ? min : x);
  }
```

```
}
```

我們沒有展示這個的擴展代碼，但是試驗之後，我們發現一個手寫 C 擴展要比使用 Cython 版本的慢了大概 10%。最底下的一行比你想象的運行的快很多。

你可以對實例代碼構建多個擴展。對於某些數組操作，最好要釋放 GIL，這樣多個線程能並行運行。要這樣做的話，需要修改代碼，使用 `with nogil:` 語句：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    with nogil:
        for i in range(a.shape[0]):
            out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

如果你想寫一個操作二維數組的版本，下面是可以參考下：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip2d(double[:,:] a, double min, double max, double[:,:] out):
    if min > max:
        raise ValueError("min must be <= max")
    for n in range(a.ndim):
        if a.shape[n] != out.shape[n]:
            raise TypeError("a and out have different shapes")
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if a[i,j] < min:
                out[i,j] = min
            elif a[i,j] > max:
                out[i,j] = max
            else:
                out[i,j] = a[i,j]
```

希望讀者不要忘了本節所有代碼都不會綁定到某個特定數組庫（比如 NumPy）上面。這樣代碼就更有靈活性。不過，要注意的是如果處理數組要涉及到多維數組、切片、偏移和其他因素的時候情況會變得複雜起來。這些內容已經超出本節範圍，更多信息請參考 PEP 3118，同時 Cython 文檔中關於“類型內存視圖”篇也值得一讀。

15.12 將函數指針轉換爲可調用對象

問題

你已經獲得了一個被編譯函數的內存地址，想將它轉換成一個 Python 可調用對象，這樣的話你就可以將它作爲一個擴展函數使用了。

解決方案

ctypes 模塊可被用來創建包裝任意內存地址的 Python 可調用對象。下面的例子演示了怎樣獲取 C 函數的原始、底層地址，以及如何將其轉換爲一個可調用對象：

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Get the address of sin() from the C math library
>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760

>>> # Turn the address into a callable function
>>> functype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
>>> func = functype(addr)
>>> func
<CFunctionType object at 0x1006816d0>

>>> # Call the resulting function
>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

討論

要構建一個可調用對象，你首先需要創建一個 CFUNCTYPE 實例。CFUNCTYPE() 的第一個參數是返回類型。接下來的參數是參數類型。一旦你定義了函數類型，你就能將它包裝在一個整型內存地址上來創建一個可調用對象了。生成的對象被當做普通的可通過 ctypes 訪問的函數來使用。

本節看上去可能有點神祕，偏底層一點。但是，但是它被廣泛使用於各種高級代碼生成技術比如即時編譯，在 LLVM 函數庫中可以看到。

例如，下面是一個使用 llvmpy 擴展的簡單例子，用來構建一個小的聚集函數，獲取它的函數指針，並將其轉換爲一個 Python 可調用對象。

```
>>> from llvm.core import Module, Function, Type, Builder
>>> mod = Module.new('example')
>>> f = Function.new(mod,Type.function(Type.double(), \
                    [Type.double(), Type.double()], False), 'foo')
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0],f.args[0])
>>> y2 = builder.fmul(f.args[1],f.args[1])
>>> r = builder.fadd(x2,y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import ExecutionEngine
```

```
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, ctypes.c_
↪double)(ptr)

>>> # Call the resulting function
>>> foo(2,3)
13.0
>>> foo(4,5)
41.0
>>> foo(1,2)
5.0
>>>
```

並不是說在這個層面犯了任何錯誤就會導致 Python 解釋器掛掉。要記得的是你是在直接跟機器級別的內存地址和本地機器碼打交道，而不是 Python 函數。

15.13 傳遞 NULL 結尾的字符串給 C 函數庫

問題

你要寫一個擴展模塊，需要傳遞一個 NULL 結尾的字符串給 C 函數庫。不過，你不是很確定怎樣使用 Python 的 Unicode 字符串去實現它。

解決方案

許多 C 函數庫包含一些操作 NULL 結尾的字符串，被聲明類型爲 `char *` . 考慮如下的 C 函數，我們用來做演示和測試用的：

```
void print_chars(char *s) {
    while (*s) {
        printf("%2x ", (unsigned char) *s);

        s++;
    }
    printf("\n");
}
```

此函數會打印被傳進來字符串的每個字符的十六進制表示，這樣的話可以很容易的進行調試了。例如：

```
print_chars("Hello");   // Outputs: 48 65 6c 6c 6f
```

對於在 Python 中調用這樣的 C 函數，你有幾種選擇。首先，你可以通過調用 `PyArg_ParseTuple()` 並指定”y“轉換碼來限制它只能操作字節，如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;

  if (!PyArg_ParseTuple(args, "y", &s)) {
    return NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

結果函數的使用方法如下。仔細觀察嵌入了 NULL 字節的字符串以及 Unicode 支持是怎樣被拒絕的：

```
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>>
```

如果你想傳遞 Unicode 字符串，在 PyArg_ParseTuple() 中使用”s“格式碼，如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;

  if (!PyArg_ParseTuple(args, "s", &s)) {
    return NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

當被使用的時候，它會自動將所有字符串轉換爲以 NULL 結尾的 UTF-8 編碼。例如：

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1o')  # Note: UTF-8 encoding
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

如果因爲某些原因，你要直接使用 PyObject * 而不能使用 PyArg_ParseTuple()，下面的例子向你展示了怎樣從字節和字符串對象中檢查和提取一個合適的 char * 引用：

```
/* Some Python Object (obtained somehow) */
PyObject *obj;

/* Conversion from bytes */
{
   char *s;
   s = PyBytes_AsString(o);
   if (!s) {
      return NULL;   /* TypeError already raised */
   }
   print_chars(s);
}

/* Conversion to UTF-8 bytes from a string */
{
   PyObject *bytes;
   char *s;
   if (!PyUnicode_Check(obj)) {
       PyErr_SetString(PyExc_TypeError, "Expected string");
       return NULL;
   }
   bytes = PyUnicode_AsUTF8String(obj);
   s = PyBytes_AsString(bytes);
   print_chars(s);
   Py_DECREF(bytes);
}
```

前面兩種轉換都可以確保是 NULL 結尾的數據，但是它們並不檢查字符串中間是否嵌入了 NULL 字節。因此，如果這個很重要的話，那你需要自己去做檢查了。

討論

如果可能的話，你應該避免去寫一些依賴於 NULL 結尾的字符串，因爲 Python 並沒有這個需要。最好結合使用一個指針和長度值來處理字符串。不過，有時候你必須去處理 C 語言遺留代碼時就沒得選擇了。

儘管很容易使用，但是很容易忽視的一個問題是在 PyArg_ParseTuple() 中使用“s”格式化碼會有內存損耗。但你需要使用這種轉換的時候，一個 UTF-8 字符串被創建並永久附加在原始字符串對象上面。如果原始字符串包含非 ASCII 字符的話，就會導致字符串的尺寸增到一直到被垃圾回收。例如：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)      # Passing string
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)    # Notice increased size
103
>>>
```

如果你在乎這個內存的損耗，你最好重寫你的 C 擴展代碼，讓它使用 PyUnicode_AsUTF8String() 函數。如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *o, *bytes;
  char *s;

  if (!PyArg_ParseTuple(args, "U", &o)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(o);
  s = PyBytes_AsString(bytes);
  print_chars(s);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

通過這個修改，一個 UTF-8 編碼的字符串根據需要被創建，然後在使用過後被丟棄。下面是修訂後的效果：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>
```

如果你試着傳遞 NULL 結尾字符串給 ctypes 包裝過的函數，要注意的是 ctypes 只能允許傳遞字節，並且它不會檢查中間嵌入的 NULL 字節。例如：

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsample.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
```

```
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>>
```

如果你想傳遞字符串而不是字節，你需要先執行手動的 UTF-8 編碼。例如：

```
>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>
```

對於其他擴展工具（比如 Swig、Cython），在你使用它們傳遞字符串給 C 代碼時要先好好學習相應的東西了。

15.14 傳遞 Unicode 字符串給 C 函數庫

問題

你要寫一個擴展模塊，需要將一個 Python 字符串傳遞給 C 的某個庫函數，但是這個函數不知道該怎麼處理 Unicode。

解決方案

這裏我們需要考慮很多的問題，但是最主要的問題是現存的 C 函數庫並不理解 Python 的原生 Unicode 表示。因此，你的挑戰是將 Python 字符串轉換爲一個能被 C 理解的形式。

爲了演示的目的，下面有兩個 C 函數，用來操作字符串數據並輸出它來調試和測試。一個使用形式爲 `char *, int` 形式的字節，而另一個使用形式爲 `wchar_t *, int` 的寬字符形式：

```
void print_chars(char *s, int len) {
  int n = 0;

  while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}

void print_wchars(wchar_t *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%x ", s[n]);
    n++;
  }
```

```
  printf("\n");
}
```

對於面向字節的函數 print_chars()，你需要將 Python 字符串轉換爲一個合適的編碼比如 UTF-8. 下面是一個這樣的擴展函數例子：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

對於那些需要處理機器本地 wchar_t 類型的庫函數，你可以像下面這樣編寫擴展代碼：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  wchar_t *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
    return NULL;
  }
  print_wchars(s,len);
  Py_RETURN_NONE;
}
```

下面是一個交互會話來演示這個函數是如何工作的：

```
>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>
```

仔細觀察這個面向字節的函數 print_chars() 是怎樣接受 UTF-8 編碼數據的，以及 print_wchars() 是怎樣接受 Unicode 編碼值的

討論

在繼續本節之前，你應該首先學習你訪問的 C 函數庫的特徵。對於很多 C 函數庫，通常傳遞字節而不是字符串會比較好些。要這樣做，請使用如下的轉換代碼：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  /* accepts bytes, bytearray, or other byte-like object */
  if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

如果你仍然還是想要傳遞字符串，你需要知道 Python 3 可使用一個合適的字符串表示，它並不直接映射到使用標準類型 `char *` 或 `wchar_t *`（更多細節參考 PEP 393）的 C 函數庫。因此，要在 C 中表示這個字符串數據，一些轉換還是必須要的。在 `PyArg_ParseTuple()` 中使用”s#”和”u#”格式化碼可以安全的執行這樣的轉換。

不過這種轉換有個缺點就是它可能會導致原始字符串對象的尺寸增大。一旦轉換過後，會有一個轉換數據的複製附加到原始字符串對象上面，之後可以被重用。你可以觀察下這種效果：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

對於少量的字符串對象，可能沒什麼影響，但是如果你需要在擴展中處理大量的文本，你可能想避免這個損耗了。下面是一個修訂版本可以避免這種內存損耗：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s;
  Py_ssize_t   len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(obj);
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
```

```
}
```

而對 wchar_t 的處理時想要避免內存損耗就更加難辦了。在內部，Python 使用最高效的表示來存儲字符串。例如，只包含 ASCII 的字符串被存儲爲字節數組，而包含範圍從 U+0000 到 U+FFFF 的字符的字符串使用雙字節表示。由於對於數據的表示形式不是單一的，你不能將內部數組轉換爲 wchar_t * 然後期望它能正確的工作。你應該創建一個 wchar_t 數組並向其中複製文本。PyArg_ParseTuple() 的”u#”格式碼可以幫助你高效的完成它（它將複製結果附加到字符串對象上）。

如果你想避免長時間內存損耗，你唯一的選擇就是複製 Unicode 數據懂啊一個臨時的數組，將它傳遞給 C 函數，然後回收這個數組的內存。下面是一個可能的實現：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  wchar_t *s;
  Py_ssize_t len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
    return NULL;
  }
  print_wchars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

在這個實現中，PyUnicode_AsWideCharString() 創建一個臨時的 wchar_t 緩衝並複製數據進去。這個緩衝被傳遞給 C 然後被釋放掉。但是我寫這本書的時候，這裏可能有個 bug，後面的 Python 問題頁有介紹。

如果你知道 C 函數庫需要的字節編碼並不是 UTF-8，你可以強制 Python 使用擴展碼來執行正確的轉換，就像下面這樣：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s = 0;
  int   len;
  if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

最後，如果你想直接處理 Unicode 字符串，下面的是例子，演示了底層操作訪問：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  int n, len;
```

```
  int kind;
  void *data;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if (PyUnicode_READY(obj) < 0) {
    return NULL;
  }

  len = PyUnicode_GET_LENGTH(obj);
  kind = PyUnicode_KIND(obj);
  data = PyUnicode_DATA(obj);

  for (n = 0; n < len; n++) {
    Py_UCS4 ch = PyUnicode_READ(kind, data, n);
    printf("%x ", ch);
  }
  printf("\n");
  Py_RETURN_NONE;
}
```

在這個代碼中，PyUnicode_KIND() 和 PyUnicode_DATA() 這兩個宏和 Unicode 的可變寬度存儲有關，這個在 PEP 393 中有描述。kind 變量編碼底層存儲 (8 位、16 位或 32 位) 以及指向緩存的數據指針相關的信息。在實際情況中，你並不需要知道任何跟這些值有關的東西，只需要在提取字符的時候將它們傳給 PyUnicode_READ() 宏。

還有最後幾句：當從 Python 傳遞 Unicode 字符串給 C 的時候，你應該儘量簡單點。如果有 UTF-8 和寬字符兩種選擇，請選擇 UTF-8. 對 UTF-8 的支持更加普遍一些，也不容易犯錯，解釋器也能支持的更好些。最後，確保你仔細閱讀了 關於處理 Unicode 的相關文檔

15.15 C 字符串轉換爲 Python 字符串

問題

怎樣將 C 中的字符串轉換爲 Python 字節或一個字符串對象？

解決方案

C 字符串使用一對 char * 和 int 來表示，你需要決定字符串到底是用一個原始字節字符串還是一個 Unicode 字符串來表示。字節對象可以像下面這樣使用 Py_BuildValue() 來構建：

```
char *s;     /* Pointer to C string data */
int   len;   /* Length of data */
```

```
/* Make a bytes object */
PyObject *obj = Py_BuildValue("y#", s, len);
```

如果你要創建一個 Unicode 字符串，並且你知道 s 指向了 UTF-8 編碼的數據，可以使用下面的方式：

```
PyObject *obj = Py_BuildValue("s#", s, len);
```

如果 s 使用其他編碼方式，那麼可以像下面使用 PyUnicode_Decode() 來構建一個字符串：

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");

/* Examples /*
obj = PyUnicode_Decode(s, len, "latin-1", "strict");
obj = PyUnicode_Decode(s, len, "ascii", "ignore");
```

如果你恰好有一個用 wchar_t *, len 對錶示的寬字符串，有幾種選擇性。首先你可以使用 Py_BuildValue() ：

```
wchar_t *w;     /* Wide character string */
int len;        /* Length */

PyObject *obj = Py_BuildValue("u#", w, len);
```

另外，你還可以使用 PyUnicode_FromWideChar() :

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

對於寬字符串，並沒有對字符數據進行解析——它被假定是原始 Unicode 編碼指針，可以被直接轉換成 Python。

討論

將 C 中的字符串轉換爲 Python 字符串遵循和 I/O 同樣的原則。也就是說，來自 C 中的數據必須根據一些解碼器被顯式的解碼爲一個字符串。通常編碼格式包括 ASCII、Latin-1 和 UTF-8. 如果你並不確定編碼方式或者數據是二進制的，你最好將字符串編碼成字節。當構造一個對象的時候，Python 通常會複製你提供的字符串數據。如果有必要的話，你需要在後面去釋放 C 字符串。同時，爲了讓程序更加健壯，你應該同時使用一個指針和一個大小值，而不是依賴 NULL 結尾數據來創建字符串。

15.16 不確定編碼格式的 C 字符串

問題

你要在 C 和 Python 直接來回轉換字符串，但是 C 中的編碼格式並不確定。例如，可能 C 中的數據期望是 UTF-8，但是並沒有強制它必須是。你想編寫代碼來以一種優

雅的方式處理這些不合格數據，這樣就不會讓 Python 奔潰或者破壞進程中的字符串數據。

解決方案

下面是一些 C 的數據和一個函數來演示這個問題：

```
/* Some dubious string data (malformed UTF-8) */
const char *sdata = "Spicy Jalape\xc3\xb1o\xae";
int slen = 16;

/* Output character data */
void print_chars(char *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}
```

在這個代碼中，字符串 sdata 包含了 UTF-8 和不合格數據。不過，如果用戶在 C 中調用 print_chars(sdata, slen) ，它缺能正常工作。現在假設你想將 sdata 的內容轉換爲一個 Python 字符串。進一步假設你在後面還想通過一個擴展將那個字符串傳個 print_chars() 函數。下面是一種用來保護原始數據的方法，就算它編碼有問題。

```
/* Return the C string back to Python */
static PyObject *py_retstr(PyObject *self, PyObject *args) {
  if (!PyArg_ParseTuple(args, "")) {
    return NULL;
  }
  return PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");
}

/* Wrapper for the print_chars() function */
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s = 0;
  Py_ssize_t   len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }

  if ((bytes = PyUnicode_AsEncodedString(obj,"utf-8","surrogateescape"))
        == NULL) {
    return NULL;
  }
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
```

```
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

如果你在 Python 中嘗試這些函數，下面是運行效果：

```
>>> s = retstr()
>>> s
'Spicy Jalapeño\udcae'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
>>>
```

仔細觀察結果你會發現，不合格字符串被編碼到一個 Python 字符串中，並且並沒有產生錯誤，並且當它被回傳給 C 的時候，被轉換爲和之前原始 C 字符串一樣的字節。

討論

本節展示了在擴展模塊中處理字符串時會配到的一個棘手又很惱火的問題。也就是說，在擴展中的 C 字符串可能不會嚴格遵循 Python 所期望的 Unicode 編碼/解碼規則。因此，很可能一些不合格 C 數據傳遞到 Python 中去。一個很好的例子就是涉及到底層系統調用比如文件名這樣的字符串。例如，如果一個系統調用返回給解釋器一個損壞的字符串，不能被正確解碼的時候會怎樣呢？

一般來講，可以通過制定一些錯誤策略比如嚴格、忽略、替代或其他類似的來處理 Unicode 錯誤。不過，這些策略的一個缺點是它們永久性破壞了原始字符串的內容。例如，如果例子中的不合格數據使用這些策略之一解碼，你會得到下面這樣的結果：

```
>>> raw = b'Spicy Jalape\xc3\xb1o\xae'
>>> raw.decode('utf-8','ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8','replace')
'Spicy Jalapeño?'
>>>
```

surrogateescape 錯誤處理策略會將所有不可解碼字節轉化爲一個代理對的低位字節（udcXX 中 XX 是原始字節值）。例如：

```
>>> raw.decode('utf-8','surrogateescape')
'Spicy Jalapeño\udcae'
>>>
```

單獨的低位代理字符比如 \udcae 在 Unicode 中是非法的。因此，這個字符串就是一個非法表示。實際上，如果你將它傳個一個執行輸出的函數，你會得到一個錯誤：

```
>>> s = raw.decode('utf-8', 'surrogateescape')
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcae'
```

```
in position 14: surrogates not allowed
>>>
```

然而，允許代理轉換的關鍵點在於從 C 傳給 Python 又回傳給 C 的不合格字符串不會有任何數據丟失。當這個字符串再次使用 surrogateescape 編碼時，代理字符會轉換回原始字節。例如：

```
>>> s
'Spicy Jalapeño\udcae'
>>> s.encode('utf-8','surrogateescape')
b'Spicy Jalape\xc3\xb1o\xae'
>>>
```

作爲一般準則，最好避免代理編碼——如果你正確的使用了編碼，那麼你的代碼就值得信賴。不過，有時候確實會出現你並不能控制數據編碼並且你又不能忽略或替換壞數據，因爲其他函數可能會用到它。那麼就可以使用本節的技術了。

最後一點要注意的是，Python 中許多面向系統的函數，特別是和文件名、環境變量和命令行參數相關的都會使用代理編碼。例如，如果你使用像 os.listdir() 這樣的函數，傳入一個包含了不可解碼文件名的目錄的話，它會返回一個代理轉換後的字符串。參考 5.15 的相關章節。

PEP 383 中有更多關於本機提到的以及和 surrogateescape 錯誤處理相關的信息。

15.17 傳遞文件名給 C 擴展

問題

你需要向 C 庫函數傳遞文件名，但是需要確保文件名根據系統期望的文件名編碼方式編碼過。

解決方案

寫一個接受一個文件名爲參數的擴展函數，如下這樣：

```
static PyObject *py_get_filename(PyObject *self, PyObject *args) {
  PyObject *bytes;
  char *filename;
  Py_ssize_t len;
  if (!PyArg_ParseTuple(args,"O&", PyUnicode_FSConverter, &bytes)) {
    return NULL;
  }
  PyBytes_AsStringAndSize(bytes, &filename, &len);
  /* Use filename */
  ...

  /* Cleanup and return */
  Py_DECREF(bytes)
```

```
  Py_RETURN_NONE;
}
```

如果你已經有了一個 PyObject * ，希望將其轉換成一個文件名，可以像下面這樣做：

```
PyObject *obj;    /* Object with the filename */
PyObject *bytes;
char *filename;
Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj);
PyBytes_AsStringAndSize(bytes, &filename, &len);
/* Use filename */
...

/* Cleanup */
Py_DECREF(bytes);

If you need to return a filename back to Python, use the following code:

/* Turn a filename into a Python object */

char *filename;       /* Already set */
int   filename_len;   /* Already set */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filename_len);
```

討論

以可移植方式來處理文件名是一個很棘手的問題，最後交由 Python 來處理。如果你在擴展代碼中使用本節的技術，文件名的處理方式和和 Python 中是一致的。包括編碼/界面字節，處理壞字符，代理轉換和其他複雜情況。

15.18 傳遞已打開的文件給 C 擴展

問題

你在 Python 中有一個打開的文件對象，但是需要將它傳給要使用這個文件的 C 擴展。

解決方案

要將一個文件轉換爲一個整型的文件描述符，使用 PyFile_FromFd() ，如下：

```
PyObject *fobj;     /* File object (already obtained somehow) */
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0) {
   return NULL;
}
```

結果文件描述符是通過調用 fobj 中的 fileno() 方法獲得的。因此，任何以這種方式暴露給一個描述器的對象都適用（比如文件、套接字等）。一旦你有了這個描述器，它就能被傳遞給多個低級的可處理文件的 C 函數。

如果你需要轉換一個整型文件描述符爲一個 Python 對象，適用下面的 PyFile_FromFd()：

```
int fd;     /* Existing file descriptor (already open) */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

PyFile_FromFd() 的參數對應內置的 open() 函數。NULL 表示編碼、錯誤和換行參數使用默認值。

討論

如果將 Python 中的文件對象傳給 C，有一些注意事項。首先，Python 通過 io 模塊執行自己的 I/O 緩衝。在傳遞任何類型的文件描述符給 C 之前，你都要首先在相應文件對象上刷新 I/O 緩衝。不然的話，你會打亂文件系統上面的數據。

其次，你需要特別注意文件的歸屬者以及關閉文件的職責。如果一個文件描述符被傳給 C，但是在 Python 中還在被使用着，你需要確保 C 沒有意外的關閉它。類似的，如果一個文件描述符被轉換爲一個 Python 文件對象，你需要清楚誰應該去關閉它。PyFile_FromFd() 的最後一個參數被設置成 1，用來指出 Python 應該關閉這個文件。

如果你需要從 C 標準 I/O 庫中使用如 fdopen() 函數來創建不同類型的文件對象比如 FILE * 對象，你需要特別小心了。這樣做會在 I/O 堆棧中產生兩個完全不同的 I/O 緩衝層（一個是來自 Python 的 io 模塊，另一個來自 C 的 stdio）。像 C 中的 fclose() 會關閉 Python 要使用的文件。如果讓你選的話，你應該會選擇去構建一個擴展代碼來處理底層的整型文件描述符，而不是使用來自 <stdio.h> 的高層抽象功能。

15.19 從 C 語言中讀取類文件對象

問題

你要寫 C 擴展來讀取來自任何 Python 類文件對象中的數據（比如普通文件、StringIO 對象等）。

解決方案

要讀取一個類文件對象的數據，你需要重複調用 read() 方法，然後正確的解碼獲得的數據。

下面是一個 C 擴展函數例子，僅僅只是讀取一個類文件對象中的所有數據並將其輸出到標準輸出：

```
#define CHUNK_SIZE 8192

/* Consume a "file-like" object and write bytes to stdout */
static PyObject *py_consume_file(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *read_meth;
  PyObject *result = NULL;
  PyObject *read_args;

  if (!PyArg_ParseTuple(args,"O", &obj)) {
    return NULL;
  }

  /* Get the read method of the passed object */
  if ((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
    return NULL;
  }

  /* Build the argument list to read() */
  read_args = Py_BuildValue("(i)", CHUNK_SIZE);
  while (1) {
    PyObject *data;
    PyObject *enc_data;
    char *buf;
    Py_ssize_t len;

    /* Call read() */
    if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
      goto final;
    }

    /* Check for EOF */
    if (PySequence_Length(data) == 0) {
      Py_DECREF(data);
      break;
    }

    /* Encode Unicode as Bytes for C */
    if ((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))==NULL) {
      Py_DECREF(data);
      goto final;
    }

    /* Extract underlying buffer data */
    PyBytes_AsStringAndSize(enc_data, &buf, &len);

    /* Write to stdout (replace with something more useful) */
    write(1, buf, len);
```

```
    /* Cleanup */
    Py_DECREF(enc_data);
    Py_DECREF(data);
  }
  result = Py_BuildValue("");

 final:
  /* Cleanup */
  Py_DECREF(read_meth);
  Py_DECREF(read_args);
  return result;
}
```

要測試這個代碼，先構造一個類文件對象比如一個 StringIO 實例，然後傳遞進來：

```
>>> import io
>>> f = io.StringIO('Hello\nWorld\n')
>>> import sample
>>> sample.consume_file(f)
Hello
World
>>>
```

討論

和普通系統文件不同的是，一個類文件對象並不需要使用低級文件描述符來構建。因此，你不能使用普通的 C 庫函數來訪問它。你需要使用 Python 的 C API 來像普通文件類似的那樣操作類文件對象。

在我們的解決方案中，`read()` 方法從被傳遞的對象中提取出來。一個參數列表被構建然後不斷的被傳給 `PyObject_Call()` 來調用這個方法。要檢查文件末尾 (EOF)，使用了 `PySequence_Length()` 來查看是否返回對象長度爲 0.

對於所有的 I/O 操作，你需要關注底層的編碼格式，還有字節和 Unicode 之前的區別。本節演示瞭如何以文本模式讀取一個文件並將結果文本解碼爲一個字節編碼，這樣在 C 中就可以使用它了。如果你想以二進制模式讀取文件，只需要修改一點點即可，例如：

```
...
/* Call read() */
if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
  goto final;
}

/* Check for EOF */
if (PySequence_Length(data) == 0) {
  Py_DECREF(data);
  break;
}
```

```
if (!PyBytes_Check(data)) {
  Py_DECREF(data);
  PyErr_SetString(PyExc_IOError, "File must be in binary mode");
  goto final;
}

/* Extract underlying buffer data */
PyBytes_AsStringAndSize(data, &buf, &len);
...
```

本節最難的地方在於如何進行正確的內存管理。當處理 PyObject * `` 變量的時候，需要注意管理引用計數以及在不需要的變量的時候清理它們的值。對 ``Py_DECREF() 的調用就是來做這個的。

本節代碼以一種通用方式編寫，因此他也能適用於其他的文件操作，比如寫文件。例如，要寫數據，只需要獲取類文件對象的 write() 方法，將數據轉換爲合適的 Python 對象（字節或 Unicode），然後調用該方法將輸入寫入到文件。

最後，儘管類文件對象通常還提供其他方法（比如 readline(), read_info()），我們最好只使用基本的 read() 和 write() 方法。在寫 C 擴展的時候，能簡單就儘量簡單。

15.20 處理 C 語言中的可迭代對象

問題

你想寫 C 擴展代碼處理來自任何可迭代對象如列表、元組、文件或生成器中的元素。

解決方案

下面是一個 C 擴展函數例子，演示了怎樣處理可迭代對象中的元素：

```
static PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *iter;
  PyObject *item;

  if (!PyArg_ParseTuple(args, "O", &obj)) {
    return NULL;
  }
  if ((iter = PyObject_GetIter(obj)) == NULL) {
    return NULL;
  }
  while ((item = PyIter_Next(iter)) != NULL) {
    /* Use item */
    ...
    Py_DECREF(item);
  }
```

```
  Py_DECREF(iter);
  return Py_BuildValue("");
}
```

討論

本節中的代碼和 Python 中對應代碼類似。PyObject_GetIter() 的調用和調用 iter() 一樣可獲得一個迭代器。PyIter_Next() 函數調用 next 方法返回下一個元素或 NULL(如果沒有元素了)。要注意正確的內存管理——Py_DECREF() 需要同時在產生的元素和迭代器對象本身上同時被調用，以避免出現內存泄露。

15.21 診斷分段錯誤

問題

解釋器因爲某個分段錯誤、總線錯誤、訪問越界或其他致命錯誤而突然間奔潰。你想獲得 Python 堆棧信息，從而找出在發生錯誤的時候你的程序運行點。

解決方案

faulthandler 模塊能被用來幫你解決這個問題。在你的程序中引入下列代碼：

```
import faulthandler
faulthandler.enable()
```

另外還可以像下面這樣使用 -Xfaulthandler 來運行 Python：

```
bash % python3 -Xfaulthandler program.py
```

最後，你可以設置 PYTHONFAULTHANDLER 環境變量。開啓 faulthandler 後，在 C 擴展中的致命錯誤會導致一個 Python 錯誤堆棧被打印出來。例如：

```
Fatal Python error: Segmentation fault

Current thread 0x00007fff71106cc0:
  File "example.py", line 6 in foo
  File "example.py", line 10 in bar
  File "example.py", line 14 in spam
  File "example.py", line 19 in <module>
Segmentation fault
```

儘管這個並不能告訴你 C 代碼中哪裏出錯了，但是至少能告訴你 Python 裏面哪裏有錯。

討論

faulthandler 會在 Python 代碼執行出錯的時候向你展示跟蹤信息。至少，它會告訴你出錯時被調用的最頂級擴展函數是哪個。在 pdb 和其他 Python 調試器的幫助下，你就能追根溯源找到錯誤所在的位置了。

faulthandler 不會告訴你任何 C 語言中的錯誤信息。因此，你需要使用傳統的 C 調試器，比如 gdb。不過，在 faulthandler 追蹤信息可以讓你去判斷從哪裏着手。還要注意的是在 C 中某些類型的錯誤可能不太容易恢復。例如，如果一個 C 擴展丟棄了程序堆棧信息，它會讓 faulthandler 不可用，那麼你也得不到任何輸出（除了程序奔潰外）。

附錄 A

在線資源

http://docs.python.org

如果你需要深入瞭解探究語言和模塊的細節，那麼不必說，Python 自家的在線文檔是一個卓越的資源。只要保證你查看的是 python 3 的文檔而不是以前的老版本

http://www.python.org/dev/peps

如果你向理解爲 python 語言添加新特性的動機以及實現的細節，那麼 PEPs (Python Enhancement Proposals—-Python 開發編碼規範) 絕對是非常寶貴的資源。尤其是一些高級語言功能更是如此。在寫這本書的時候，PEPS 通常比官方文檔管用。

http://pyvideo.org

這裏有來自最近的 PyCon 大會、用戶組見面會等的大量視頻演講和教程素材。對於學習潮流的 python 開發是非常寶貴的資源。許多視頻中都會有 Python 的核心開發者現身說法，講解 Python 3 中添加的的新特性。

http://code.activestate.com/recipes/langs/python

長期以來，ActiveState 的 Python 版塊已經成爲一個找到數以千計的針對特定編程問題的解決方案。到寫作此書位置，已經包含了大約 300 個特定於 Python3 的祕籍。你會發現，其中多數的祕籍要麼對本書覆蓋的話題進行了擴展，要麼專精於具體的任務。所以說，它是一個好伴侶。

http://stackoverflow.com/questions/tagged/python

Stack Overflow 目前有超過 175,000 個問題被標記爲 Python 相關 (而其中大約 5000 個問題是針對 Python 3 的)。儘管問題和回答的質量不同，但是仍然能發現很多好優秀的素材。

Python 學習書籍

下面這些書籍提供了對 Python 編程的入門介紹，且重點放在了 Python 3 上。

Beginning Python: From Novice to Professional, 2nd Edition, by Magnus Lie Hetland, Apress (2008). Programming in Python 3, 2nd Edition, by Mark Summerfield, Addison-Wesley (2010).

- *Learning Python*，第四版，作者 Mark Lutz，O'Reilly & Associates 出版 (2009)。
- *The Quick Python Book*，作者 Vernon Ceder，Manning 出版 (2010)。
- *Python Programming for the Absolute Beginner*，第三版，作者 Michael Dawson，Course Technology PTR 出版 (2010).
- *Beginning Python: From Novice to Professional*，第二版，作者 Magnus Lie Hetland，Apress 出版 (2008).

- *Programming in Python 3*，第二版，作者 Mark Summerfield，Addison-Wesley 出版 (2010).

高級書籍

下面的這些書籍提供了更多高級的範圍，也包含 Python 3 方面的內容。

- *Programming Python*，第四版, by Mark Lutz, O'Reilly & Associates 出版 (2010).
- *Python Essential Reference*，第四版，作者 David Beazley, Addison-Wesley 出版 (2009).
- *Core Python Applications Programming*，第三版，作者 Wesley Chun, Prentice Hall 出版 (2012).
- *The Python Standard Library by Example* ，作者 Doug Hellmann，Addison-Wesley 出版 (2011).
- *Python 3 Object Oriented Programming*，作者 Dusty Phillips, Packt Publishing 出版 (2010).
- *Porting to Python 3*，作者 Lennart Regebro，CreateSpace 出版 (2011), http://python3porting.com.

關於譯者

關於譯者

- 姓名：熊能
- 微信：yidao620
- Email：yidao620@gmail.com
- 博客：http://yidao620c.github.io/
- GitHub：https://github.com/yidao620c

Roadmap

2014/08/10 - 2014/08/31:

| github 項目搭建，readthedocs 文檔生成。
| 整個項目的框架完成

2014/09/01 - 2014/10/31:

| 前 4 章翻譯完成

2014/11/01 - 2015/01/31:

| 前 8 章翻譯完成

2015/02/01 - 2015/03/31:

| 前 9 章翻譯完成

2015/04/01 - 2015/05/31:

| 10章翻譯完成

2015/06/01 - 2015/06/30:

| 11章翻譯完成

2015/07/01 - 2015/07/31:

| 12章翻譯完成

2015/08/01 - 2015/08/31:

| 13章翻譯完成

2015/09/01 - 2015/11/30:

| 14章翻譯完成

2015/12/01 - 2015/12/20:

| 15章翻譯完成

2015/12/21 - 2015/12/31:

| 對全部翻譯進行校對一次

2016/01/01 - 2016/01/10:

| 對外公開發布完整版 1.0，包括轉換後的 PDF 文件