Python 沉思錄

像計算機專家一樣思考

第 2 版, 版本 2.4.0

Allen Downey(著)
杜文斌 (譯)

Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.4.0

Allen Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from `http://www.thinkpython2.com`

序

本書由來

1999 年 1 月, 我準備通過 Java 教學生程式設計入門, 課程已講過三次, 但我沮喪依舊. 這門課的失敗率太高了, 即使是那些優秀的學生, 其整體成績水平也並不理想.

據我觀察, 問題出在書本教案上, 書籍厚重, 畫蛇添足, 細枝末節過多, 關於如何程式設計, 缺乏更多的指導. 因此學生總是輕易陷入困局: 起步易, 行進難, 半途而廢. 學生學得囫圇吞棗, 而我則要花費整個學期, 收拾殘局.

译者注
我們使用的許多教材, 是否也有此通病, 誠學生之痛, 教師之難, 教育之悲.

開課前兩週, 我決定自己寫本書, 目標如下:

- 少. 十頁易解, 百頁難通.
- 精. 詞彙早解釋, 術語早提煉.
- 緩. 將困難部分大而化小, 小而能學.
- 專. 聚焦編程思想, 精簡程式代碼.

莫名其妙地選擇了個題目像計算機專家一樣思考 *(How to Think Like a Computer Scientist)*.

首版雖然略顯粗略, 但是效果不錯. 學生知其然, 並知其所以然, 我也可聚焦課堂於難點, 趣題以及操作實踐.

我基於 GNU 自由檔案許可證發布此書, 也就是說, 任何人都可以複製, 修改, 分散此書.

後續的事情很有趣, 弗吉尼亞州的高中教師 Jeff Elkner 採用了本書, 並翻譯為了 Python 教材. 他將他的翻譯版本發送給了我, 我便有了一段不尋常得經歷, 通過閱讀自己的書來學習 Python. 同時依托 Green Tea Press, 於 2001 年, 我出版了第一个 Python 版本.

2003 年, 我開始在歐林學院教書, 併且首次教授 Python. 和 Java 相比, 效果出人意料, 學生們樂於學習且樂在其中.

自此, 我不斷優化此書, 校對錯誤, 提高範例水準, 並不斷增加一些新的例子和練習.

現在標題不再那麼誇張, 改為了 *Think Python*. 部分修改如下:

- 每章結尾增加了除錯部分. 重點講解如何發現問題, 減少錯誤, 謹防紕漏.
- 增加了一些習題. 從簡單測驗到專案實操, 均有涉及, 同時, 給出了我的方案.
- 增加了一系列的案例: 涵蓋試驗-解答-討論的長篇實例.
- 擴展了對程式開發計畫以及基礎型樣設計的探討.
- 添加了關於除錯和演算法分析的附錄.

第二版 *Think Python* 有如下特徵:

- 本書代碼均已支持 Python 3.
- 網頁版功能更加豐富, 使初學者可以無需安裝 Python 便可在瀏覽器運行.
- 對於第 4.1節, 我將自己的海龜繪圖組件 Swampy 更標準的 Python 模塊 `turtle`.
- 增加了新章節" 利器", 主要介紹一些 Python 中不必需卻非常有用的功能.

願您喜歡此書, 同時希望對您學習程式設計, 以及培養計算機專家一樣的思維模式有所裨益.

Allen B. Downey

歐林學院

致謝

感謝 Jeff Elkner 將我的 Java 書籍翻譯為 Python, 並讓我愛上了 Python.

感謝 Chris Meyers 為 *How to Think Like a Computer Scientist* 貢獻了部分章節.

感謝免費軟體基金會開發的 GNU 自由檔案許可證, 這令我和 Jeff 以及 Chris 的協作成為可能, 也感謝我正在使用其許可證的創意共享機構.

感謝在 Lulu 工作, 為 *How to Think Like a Computer Scientist* 進行編纂的編輯們.

感謝耐心編輯 *Think Python* 的 O'Reilly Media 的朋友.

感謝所有使用本書早期版本的學生們, 以及所有為本書提供校正和建議的貢獻者 (如下所列).

貢獻者名單

過去數年, 上百位優秀讀者為本書獻策糾誤. 他們無私的奉獻, 給予了本書極大幫助.

如您發現問題或者有好的提議, 請發電子郵件到 `feedback@thinkpython.com`. 一經採用, 您將被加入貢獻者名單 (除非您要求不被列入).

建議您最好附上詳細錯誤語句, 而不僅僅是章節和頁碼, 以便我能快速定位, 感謝!

- Lloyd Hugh Allen 校正了 8.4 節的一處錯誤.
- Yvon Boulianne 校正了第 5 章的一處語意錯誤.
- Fred Bremmer 校正了 2.1 節的一處錯誤.
- Jonah Cohen 編寫了 Perl 腳本, 將本書 LaTex 原始碼轉成了美觀的 HTML.
- Michael Conlon 校正了第 2 章的一處文法錯誤, 並優化第一章的一處格式, 併發起了解譯器技術方面的討論.
- Benoît Girard 校正了 5.6 節一個幽默的差錯.
- Courtney Gleason 和 Katherine Smith 編寫了 `horsebet.py`, 這是本書早期版本的一個案例研究, 此程式現在仍然可以在網站找到.
- Lee Harr 提交了遠超此處所列的諸多校正, 因此應該把他列為主要編輯者.
- James Kaylin 作為學生, 提交了大量校正.
- David Kershaw 校正了 3.10 節的 `catTwice` 函數錯誤.
- Eddie Lam 校正了第 1,2,3 章大量錯誤, 同時修復了 Makefile 檔案, 從而我們可以創建索引. 同時幫助我們設置了版本控制方案.
- Man-Yong 校正了 2.4 節的一個範例.
- David Mayo 指出第 1 章的某個詞語"unconsciously" 應該替換為"subconsciously".
- Chris McAloon 提交了幾個對第 3.9 節和第 3.10 節的校正.
- Matthew J. Moelter 作為一個長期貢獻者, 為本書提供了大量校正和建議.
- Simon Dicon Montford 報告了第 3 章的一個函數定義的缺失和幾處打字錯誤. 同時指出了第 13 章 `increment` 函數的錯誤.
- John Ouzts 校正了第三章的"return value" 的定義.
- Kevin Parks 就如何改進本書的分發方式提供了寶貴的意見和建議.
- David Pool 指出了第一章術語錶中的一個打字錯誤, 同時對本書給予了善意的鼓勵.
- Michael Schmitt 校正了檔案和異常這一章節中的一個錯誤.
- Robin Shaw 指出 13.1 節中一個錯誤, 使用了沒有預先定義的 printTime 函數.
- Paul Sleigh 指出了第 7 章的一個錯誤, 以及 Jonah Cohen 的 LaTex 轉 HTML 的 Perl 腳本中的一個錯誤.
- Craig T. Snydal 在 Drew University 使用本書教學, 併提供了幾條寶貴的建議和校正.

- Ian Thomas 和他的學生在編程課程中使用本書, 他們是首批檢驗本書後半部分的貢獻者, 同時他們提供了大量校正和建議.
- Keith Verheyden 校正了第 3 章的一個錯誤.
- Peter Winstanley 指出了第 3 章拉丁語中的一個長期存在的錯誤.
- Chris Wrobell 校正了 I/O 和異常章節中的一個錯誤.
- Moshe Zadka 撰寫了字典章節的早期草稿, 為本書早期工作做出了傑出貢獻.
- Christoph Zwerschke 給出多個校正和教學建議, 並解釋了 *gleich* 和 *selbe* 的不同.
- James Mayer 校正了一堆拼寫和排版問題, 甚至包括貢獻串列中的兩個問題.
- Hayden McAfee 解決了一個兩例相校令人困惑的差異.
- Angel Arnal 作為國際翻譯組織的一員, 參與了西班牙語的翻譯, 並在翻譯過程中校正了英語版的一些錯誤.
- Tauhidul Hoque 和 Lex Berezhny 繪製了第一章的插圖, 並優化了其他章節插圖.
- Michele Alzetta 博士校正了第 8 章的一個錯誤, 並指出了 Fibonacci and Old Maid 案例的一些問題.
- Andy Mitchell 指出了第 1 章的一個錯字, 第 2 章的一個錯誤例子.
- Kalin Harvey 辨析了第 7 章的一個歧義, 並校正了幾個錯字.
- Christopher P. Smith 校正了錯字, 並針對 Python2.2 進行了修改.
- David Hutchins 校正了序中的一個錯詞.
- Gregor Lingl 在奧地利的維也納的一個高校講授 Python, 他翻譯了德語版, 並在翻譯過程中, 校正了第 5 章的多個錯誤.
- Julie Peters 校正了序中的一個錯字.
- Florin Oprina 優化了 `makeTime`, 校正了 `printTime` 的錯誤, 以及一個錯詞.
- D. J. Webre 辨析了第 3 章的一個歧義詞.
- Ken 校正了 8,9,11 章的一堆錯誤.
- Ivo Wever 校正了第 3 章的錯字和第 5 章的歧義詞.
- Curtis Yanko 辨析了第 2 章一個歧義.
- Ben Logan 校正了許多拼寫錯誤, 以及在將本書翻譯成 HTML 時遇到的問題.
- Jason Armstrong 發現第 2 章遺漏的一個詞彙.
- Louis Cordier 指出了第 16 章一個描述和代碼不一致的問題.
- Brian Cain 校正了第 2,3 章的數個錯誤.
- Rob Black 校正了一系列錯誤, 以及優化了 Python2.2 版部分內容.
- Jean-Philippe Rey [巴黎中央理工學院] 針對 Python2.2 進行了部分優化.
- Jason Mader [喬治華盛頓大學] 提供了大量修改意見.
- Jan Gundtofte-Bruun 指出"a error" 這個錯誤.

- Abel David 和 Alexis Dinno 指出"matrix" 的複數是"matrices", 而不是"matrixes". 這個錯誤存在多年, 但是兩人在同一天指出此錯誤, 神奇!
- Charles Thayer 建議我們在一些聲明末尾勿用分號, 以及避免使用"argument" 和"parameter".
- Roger Sperberg 指出了第 3 章的一個邏輯問題.
- Sam Bull 指出第 2 章的一處歧義表述.
- Andrew Cheung 校正了" 使用先於定義." 的兩個錯誤.
- C. Corey Capel 指出" 除錯章節中第三定理" 部分遺漏的詞彙, 以及第 4 章的一個錯詞.
- Alessandra 澄清了一些"Turtle" 歧義.
- Wim Champagne 校正了字典例子中的一個"brain-o" 錯誤.
- Douglas Wright 校正了 `arc` 中的一個"floor" 錯誤.
- Jared Spindor 清理了數個畫蛇添足.
- Lin Peiheng 提供了多個建議.
- Ray Hagtvedt 校正了兩個錯誤, 提出一個優化.
- Torsten Hübsch 指出 Swampy 中的一處不一致.
- Inga Petuhhov 校正了第 14 章的一個案例問題.
- Arne Babenhauserheide 校正了數個錯誤.
- Mark E. Casida 敏銳發現了多個重複詞彙.
- Scott Tyler 補充了一處遺漏詞彙, 並校正了大量錯誤.
- Gordon Shephard 多次發送郵件, 指出錯誤.
- Andrew Turner 指出第 8 章的一處錯誤.
- Adam Hobart 校正了 `arc` 中的一處錯誤.
- Daryl Hammond 和 Sarah Zimmerman 指出我過早使用 `math.pi`, 同時 Zim 校正了一個拼寫錯誤.
- George Sass 修復了除錯這一章節的一處錯誤.
- Brian Bingham 提供了 11.5的習題.
- Leah Engelbert-Fenton 指出我誤用 `tuple` 為變數名的問題, 以及大量類似問題, 以及一些" 使用先於定義" 的問題
- Joe Funke 指出一個拼寫錯誤.
- Chao-chao Chen 指出 Fibonacci 例子中的一處不一致.
- Jeff Paine 指出了 space 和 spam 之間的不同.
- Lubos Pintes 指出了一處拼寫錯誤.
- Gregg Lind 和 Abigail Heithoff 提供了 14.3習題.
- Max Hailperin 校正了大量錯誤, 同時提供了大量建議. Max 是 *Concrete Abstractions* 一書的其中一位作者, 讀完此書, 你可以閱讀這本非凡之作.

- Chotipat Pornavalai 校正了錯誤信息中的一處問題.
- Stanislaw Antol 提供了大量建議.
- Eric Pashman 校正了 4-11 章的大量錯誤.
- Miguel Azevedo 發現了多個拼寫錯誤.
- Jianhua Liu 校正了大量錯誤.
- Nick King 發現了一處詞彙遺漏.
- Martin Zuther 提供了大量建議.
- Adam Zimmerman 指出了"instance" 中的一處不一致問題, 以及數個錯誤.
- Ratnakar Tiwari 提供了退化三角形說明腳註.
- Anurag Goel 提供了 `is_abecedarian` 的另外一種解決方案, 並校正了多個錯誤. 同時他知曉如何拼寫 Jane Austen.
- Kelli Kratzer 指出了一處拼寫錯誤.
- Mark Griffiths 澄清了第 3 章的一個令人困惑的案例.
- Roydan Ongie 指出了 Newton 方法的一處錯誤.
- Patryk Wolowiec 校正了 HTML 版中的一個錯誤.
- Mark Chonofsky 讓我知曉了 Python3 的一個關鍵字.
- Russell Coleman 讓我增長了幾何知識.
- Nam Nguyen 指出了一個拼寫錯誤, 並指出我用裝飾器模式前, 未有說明.
- Stéphane Morin 校正了數個錯誤, 並提供了多個建議.
- Paul Stoop 校正了 `uses_only` 中的一處拼寫錯誤.
- Eric Bronner 指出了關於執行順序的討論中的一處混亂.
- Alexandros Gezerlis 為提供建議的數量和質量設定了新的標準, 深表謝意.
- Gray Thomas 頭腦清醒, 梳理了很多問題.
- Giovanni Escobar Sosa 提供了大量修訂.
- Daniel Neilson 校正了執行順序處的一個錯誤.
- Will McGinnis 指出 `polyline` 在兩處定義不同.
- Frank Hecker 指出一個習題描述不清, 同時指出了多個異常鏈接.
- Animesh B 澄清了一個令人困惑的案例.
- Martin Caspersen 發現兩處四捨五入的錯誤.
- Gregor Ulm 提供多個修訂和建議.
- Dimitrios Tsirigkas 建議我對某個習題進行優化.
- Carlos Tafur 提供了大量修訂和建議.
- Martin Nordsletten 發現了習題答案中的一處異常.

- Sven Hoexter 指出某處誤用 `input` 這一內置函數.
- Stephen Gregory 指出 Python3 中的 `cmp` 的問題.
- Ishwar Bhat 修訂了關於費馬大定理的描述.
- Andrea Zanella 將此書翻譯為意大利語, 同時校正了多個錯誤.
- 感謝 Melissa Lewis 和 Luciano Ramalho 對第二版提供的建議和幫助.
- 感謝來自 PythonAnywhere 的 Harry Percival 的幫助, 使讀者可以在瀏覽器運行 Python.
- Xavier Van Aubel 為第二版提供了多次校正.
- William Murray 糾正了取整的定義.
- Per Starbäck 令我了解了 Python3 中的通用換行符.
- Laurent Rosenfeld 和 Mihaela Rotaru 將本書翻譯為了法語, 並糾正了諸多錯誤.

  另外為本書獻策糾誤的人包括: Czeslaw Czapla, Dale Wilson, Francesco Carlo Cimini, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, and Eric Ransom.

目录

第 1 章　程式之道

本書期望能帶你做到像計算機專家一樣思考. 這意味著你要整合數學, 工程和自然科學的知識去綜合思考. 像數學家一樣, 計算機專家通常以形式語言去表達思想 (尤其涉及計算). 像工程師一樣, 他們設計原件, 拼裝組件, 構建系統, 權衡各種方案. 像科學家一樣, 他們觀察複雜系統行為, 小心假設, 積極求證.

計算機專家最重要的能力是**解決問題**. 解決問題意味著能夠發現問題, 同時可以提出創造性解決方案, 並能清晰準確表述. 事實證明, 學習編程是鍛鍊解決問題能力的絕佳機會. 這也是本章叫做程式之道的緣由.

一方面, 你將學習編程這一技能, 另一方面, 你將利用編程技能來實現目標. 隨著不斷學習, 我們終將看到彼岸.

1.1　何為程式

程式 簡而言之, 就是一堆執行運算的指令. 運算可以是數學計算, 例如求解方程或多項式. 也可以是符號運算, 例如檢索或者替換文檔中的語句詞彙. 也可以是圖形運算, 比如處理圖片, 播放視頻.

不同語言雖各有特色, 但基礎指令相差無幾:

輸入: 從鍵盤, 檔案, 網絡或者其他設備中獲取數據.

輸出: 將數據顯示於屏幕, 保存至檔案, 通過網絡發送等.

數學計算: 加減乘除等數學運算.

條件選擇: 檢查條件並運行相應的代碼 (只能在命運分支的安排下, 低頭前進).

迴圈執行: 重複執行, 直到改變 (在宿命輪迴中不斷嘗試, 於既定界限處終達彼岸).

無論相信與否, 基礎指令不過這些. 不管是哪種編程語言, 也無論何等複雜, 都由類似指令構成. 如此你便了解了何為編程, 那便是將龐大複雜的任務不斷分解, 不斷細化, 直至細化為這些基礎指令為止.

1.2　運行程式

巧婦難為無米之炊, 運行 Python 程式必然需要先安裝 Python 和相關軟體, 熟悉操作系統和命令行的人還好, 對於初學者, 同時學習操作系統和編程, 必然是件萬分痛苦的事情.

為了避免上述問題, 我建議您先在瀏覽器上學習 Python 編程, 後續再按步驟進行 Python 安裝.

在線運行 Python 代碼的網站很多, 如果你已經有青睞的, 繼續使用即可, 如果無從下手, 我推薦 PythonAnywhere. 入門指導可以查閱`http://tinyurl.com/thinkpython2e`

Python 有兩個版本, 分別是 Python 2 和 Python 3, 它們及其相似, 所以如果學會一種, 便很容易照貓畫虎. 兩者稍有差異, 初學時需要關注. 本書主要面向 Python 3, 同時也包含了一些關於 Python 2 的內容.

運行 Python 代碼的程式叫做 Python **解釋器**. 通常點擊圖標或者在命令行鍵入 `python` 來啟動程式. 啟動後, 可以看到如下輸出:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

前三行主要是 Python 解釋器信息以及操作系統信息, 內容因人而異. 注意版本信息 (此處是 `3.4.0`), 以 3 開頭表示此解釋器版本是 Python 3, 如果是以 2 開頭, 則表示 Pyhton 2.

最後一行是個**提示符**, 表示一切就緒, 只待代碼. 如果輸入一行代碼, 並敲擊 Enter 鍵, 解釋器就會顯示運行結果.

```
>>> 1 + 1
2
```

到此, 你應該學會了如何啟動 Python 解釋器, 並運行代碼了.

1.3 初識程式

通常, 學習一門新的編程語言, 寫的第一個程式就是"Hello, World!". 在 Python 中這樣編寫:

```
>>> print('Hello, World!')
```

這就是 **print 語法**, 不要誤會是打印到紙上, 而是顯示在屏幕上, 結果就是這樣:

```
Hello, World!
```

程式中的引號表示語句的開始和結束, 並不會出現在結果中.

圓括號表示 `print` 是個函數, 後續會在第 3章介紹.

Python 2 中, print 語法明顯不同, 它不是函數, 所以不使用括號.

```
>>> print 'Hello, World!'
```

這種差異很值得深究, 但現在, 我們可以開始學習 Python 編程了.

1.4 算術運算

講完了"Hello, World", 我們講講算術. Python 通過**運算子** 進行加減乘除等運算.

運算子 `+`, `-`, 和 `*` 分別表示加法, 減法和乘法, 參看以下範例:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

運算子 `/` 執行除法:

```
>>> 84 / 2
42.0
```

你可能想知道爲什麼結果是 `42.0`, 而不是 `42`, 下節會對此進行解釋.

運算子 `**` 表示冪運算, 表示一個數字的次方.

```
>>> 6**2 + 6
42
```

運算子 `^` 在某些編程語言中表示冪運算, 但是 Python 中, 它表示位運算的" 異或" 操作. 如果不了解位運算, 你會對結果感到詫異.

```
>>> 6 ^ 2
4
```

本書不準備講解位運算, 如果想了解, 可以前往 `http://wiki.python.org/moin/BitwiseOperators`.

1.5 值和型態

值 是程式的基本要素, 例如字母, 數字. 前文看到的 `2`, `42.0`, 以及 `'Hello, World!'`, 都屬於值.

而這些值分屬不同**型態**: `2` 是**整數**, `42.0` 是**浮點數**, 而 `'Hello, World!'` 是**字符串**, 也就是一堆字符的集合.

如果你不確定值的型態, 解釋器可以告訴你:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

"class" 表示類別, 值的類別就是型態.

所以整數屬於整型, 字符串屬於字符型 , 浮點數屬於浮點型 .

那麼 `'2'` 和 `'42.0'` 是什麼型態呢? 看似是數字, 其實是被引號括起來的字符串.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

所以, 是字符串吧.

當我們想使用很大的整數時, 一般會在數字間加逗號分隔, 類似 `1,000,000`. 在 Python 中, 這雖然不是合法的整型，卻是一個合法的型態:

```
>>> 1,000,000
(1, 0, 0)
```

很神奇！Python 解釋器把 `1,000,000` 當作了一串逗號分隔的整數序列. 後續我們還會多次接觸這種格式.

1.6 形式語言和自然語言

自然語言 指人們溝通所說的語言, 比如英語, 西班牙語, 法語. 通常是人類自然進化而得來, 並不是出於特定目的而創造.

形式語言 指人們為了特定目的而創造的特定格式的語言, 比如數學家們發明的數學符號, 可以很方便表示數字和符號的關係. 而化學家發明的分子結構式也是一種形式語言, 更為重要的是:

> **人們為了表示機器運算而創造的形式語言就是編程語言.**

形式語言一般有嚴格的**語法** 來管理語句的結構. 例如, 數學表示, $3+3=6$ 是正確的, 而 $3+=3\$6$ 則不正確. 又比如在化學中, H_2O 正確, 而 ${}_2Zz$ 則明顯有問題.

語法包括兩部分, **符號** 和規則. 符號是語言的基本, 比如單詞, 數字和化學元素等. $3+=3\$6$ 的問題在於, $ 在數學中不是一個合法的符號 (據我所知). 同樣, ${}_2Zz$ 在化學領域不正確, 也是因為 Zz 無法對應相應的元素.

語法的另外一個要素是**規則**, 即把符號有效組織起來所遵循的結構規範. 公式 $3+/3$ 不合語法, 不是因為 $+$ 和 $/$ 不正確, 而是因為先後順序問題. 同樣, 上述化學公式不僅僅是詞彙胡編亂造, 其規則也有問題, 元素下標不能在元素之前, 而應放在後邊.

This is @ well-structured Engli$h sentence with invalid t*kens in it. 這句話每個字母 (符號) 都正確, 但是拼在一起, 就不正確了.

我們讀一句話或者一個公式的時候, 會自覺地分析句子結構 (自然語言中往往是潛意識行為). 這個分析的過程, 叫做**解析**.

形式語言和自然語言在符號, 規則以及語法等方面較為相似, 但是也有很多不同之處:

歧義: 自然語言的表述不太精確, 人人往往需要上下文和相關情境確定對方的意思. 但是形式語言中, 語法確定, 語意清晰, 不會出現同一句話表示不同意思的情況.

冗餘: 為了讓對方明白自己的真實意思, 避免誤會, 自然語言中往往需要添加更多相關信息, 來輔助溝通. 而形式語言則更加簡煉且精確.

比喻: 自然語言中充滿了俗語和比喻, 我說"The penny dropped", 不是說誰得錢掉了 (比喻恍然大悟). 形式語言則沒有類似的表述.

我們生活在自然語言的環境下, 突然要適應形式語言, 有很多困難需要克服. 而兩者的不同, 就好比詩詞和散文:

詩詞: 關注平仄, 並用大量表意來渲染中心思想, 從而引起情感共鳴.

散文: 用詞淺顯直白, 行文多有章法, 表意使用較少, 相比於詩詞, 易於理解.

程式: 電腦程式的編寫不允許有任何比喻象徵等手法, 必須遵照語法結構, 不可有一絲逾矩.

形式語言相比自然語言, 更加細緻繁密, 所以需要花更多時間閱讀理解. 其次, 語法結構也很重要, 所以不要總是從上往下, 從左往右地閱讀, 而要在大腦中分析代碼, 識別特有符號, 解讀語法規則. 最後, 要格外關注細節, 單詞錯誤, 標點缺失, 這些問題在自然語言的使用中可能影響不大, 但在形式語言中, 卻是致命錯誤.

1.7 除錯

是人就會犯錯. 由於一些歷史原因, 編程中的異常或者錯誤, 叫做 **bugs(臭蟲)**, 這裡用到了比喻, 一般解釋為**異常**. 而跟蹤異常, 定位問題的過程叫做 **debugging(除錯)**.

編程, 尤其是除錯, 往往會激怒你. 如果你困擾於一個難題, 那往往會經歷憤怒, 沮喪和絕望.

已有證據表明, 人們對待計算機的方式, 和對待他人的方式一樣. 計算機運行正常, 我們就當其為伙伴, 助手, 如果運行異常了, 就待其為無禮之人. (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

對於計算機要有一定心理準備, 把它當作一個計算迅速而精確, 但同時缺乏同理心以及大局觀的雇員.

你需要做好管理之責：揚機器之長而避其短, 做情緒的主人, 而非情緒的奴隸.

除錯過程是令人無比沮喪的, 但這又是編程路上最有用的技能. 每章末尾都會有關於除錯的一些建議, 希望有所幫助.

1.8 術語表

解決問題 (problem solving): 提出問題, 分析問題, 解決問題的過程.

高階語言 (high-level language): 類似 Python 這種便於人類讀寫, 對人友好的編程語言.

底層語言 (low-level language): 便於計算機運行的語言, 也叫" 機器語言" 或者" 匯編語言".

可移植 (portability): 一個程式可運行於多種設備的特性.

解釋器 (interpret): 讀取並執行代碼的程式.

提示符 (prompt): 解釋器的聲明, 用來提示準備接收輸入信息.

程式 (program): 運算指令集合.

打印語句 (print statement): Python 中輸出到屏幕的指令.

運算子 (operator): 類似加減乘除的符號.

值 (value): 程式操作的基本元素, 例如數字或者字符串.

型態 (type): 值的集合. 目前遇到的有整數集合 (type `int`), 浮點數集合 (type `float`) 以及字符串集合 (type `str`).

整型 (integer): 整數型態.

浮點型 (floating-point): 小數表示的數字型態.

字符串 (string): 字符序列型態

自然語言 (natural language): 人類溝通用的語言.

形式語言 (formal language): 人類因特定目標而創造的語言. 例如數學語言, 編程語言; 所有的編程語言都是形式語言.

符號 (token): 程式中的基本元素, 類似自然語言中的單詞.

語法 (syntax): 程式的結構規範.

解析 (parse): 分析語法結構並檢驗程式.

異常 (bug): 程式中的錯誤.

除錯 (debugging): 定位異常並解決的過程.

1.9 習題集

Exercise 1.1. 工欲善其事, 必先利其器, 搭配電腦實操來學習本書, 效果更好.

學習新知識更有效的方式是不斷試錯, 不斷思考. 比如"*Hello, World*" 的程式, 試一試雙引號只寫一半是否可以? 不寫雙引號, 會怎麼樣? 以及 `print` 拼錯了, 又會如何?

這種學習過程中的大膽假設, 小心求證, 助益頗多. 同時能夠快速了解編程中的各種錯誤信息. 所以, 當下有準備地試錯, 遠勝過將來意外犯錯.

1. *print* 使用時, 如果丟掉了半個括號, 甚至全部括號, 會怎樣?
2. 當輸出字符串時, 遺落了半個雙引號, 或者全部雙引號, 會如何?
3. 我們可以用負號 表示負數 `-2`. 如果像 `2+-2` 在數字前又加上一個加號呢?
4. 數學表示中, 首位補零很正常, 比如 `09`. 但是在 *Python* 世界中, 如果輸入 `011` 會出來什麼結果呢?
5. 如果兩個值之間沒有運算子會如何?

Exercise 1.2. 啟動 *Python* 解釋器, 進行以下運算:

1. *42* 分 *42* 秒總共有多少秒?
2. *10* 千米是多少英里? 提示: *1* 英里 *=1.61* 千米.
3. 如果用 *42* 分 *42* 秒跑了 *10* 千米, 那平均速度是多少 (每英里的耗時)? 平均速度又是多少 (每小時英里數)?

第 2 章　變數、表達式和語句

編程語言的強大和高效, 其中一方面便體現在操作**變數**. 變數名是指向值的一個名字.

2.1　變數賦值

創建新的變數, 並給其一個值, 叫做**賦值**:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

上面的例子提供了三個賦值語句. 第一個是將一句話賦值給了名為 `message` 的變數; 第二個是將整數 `17` 賦值給了 `n`; 第三個是將 π 的近似值賦給了 `pi`.

一個形象化表示變數的常用手段是, 定義變數, 然後畫個箭頭指向對應的值. 這種形象化展示變數所處狀態 (所對應的值) 的圖叫做**狀態圖**, 圖 2.1 便是上面代碼示例的狀態圖.

2.2　變數名

程式工程師一般會為變數起個容易理解記憶的名字, 使人一看即知其用途.

變數名可長可短, 包含字母和數字, 但不能以數字開頭, 大小寫均可用, 但是方便起見, 建議只用小寫字母.

變數名也可以使用_, 多用在形如`your_name` 或 `airspeed_of_unladen_swallow` 這種多單詞命名的變數名中.

如果變數名包含了非法字符, 就會語法錯誤:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
```

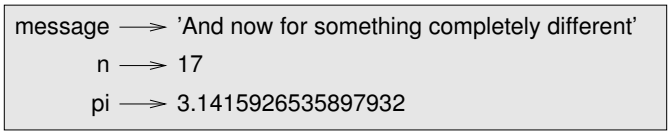

图 2.1: State diagram.

```
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

變數名 `76trombones` 的錯誤在於數字開頭, 而 `more@` 是因為包含了不合法字符 `@`. 那麼 `class` 為什麼也不正確呢?

這是因為 `class` 是 Python 的一個**關鍵字**. 解釋器用關鍵字 (也稱預留字) 來識別解析代碼, 所以關鍵字不能作為變數名使用.

Python 3 包含以下關鍵字:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

你無須刻意背誦這些關鍵字. 在多數開發環境中, 關鍵字會以特殊顏色標識, 所以如果誤用作變數名時, 很容易發覺.

2.3 表達式和語句

表達式 是值, 變數以及運算子的集合. 值本身可以認為是表達式, 同樣, 變數本身也是. 所以下面都可以認為是合法的表達式:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

當根據提示符輸入表達式時, 解釋器會進行**估算**, 獲取表達式的值. 上例中, `n` 的值是 17, `n+25` 的值是 42.

語句 是代碼的基本有效單位, 例如新建變數或者輸出值.

```
>>> n = 17
>>> print(n)
```

第一行是給 `n` 賦值的賦值語句, 第二行是顯示 `n` 結果的輸出語句.

鍵入語句, 解釋器根據語句規則**運行**, 通常, 語句沒有值 (**注意: 表達式有值**).

2.4 腳本模式

截至當前, 我們一直使用**交互模式**運行代碼, 此模式需要不斷和解釋器交互. 入門學習採用交互模式尚可行, 但代碼行數一旦增多, 使用就會很麻煩.

替代方案是將代碼保存至**腳本**檔案, 然後解釋器在**腳本模式** 運行檔案. 通俗來說, Python 腳本以 `.py` 結尾.

了解了如何創建以及運行腳本, 便可開啟後續學習. 如果沒有配套工具, 建議繼續採用 PythonAnywhere. 同時, 我已在`http://tinyurl.com/thinkpython2e` 撰寫了運行腳本模式的相關步驟.

Python 提供兩種模式, 在寫入腳本前, 可以採用交互模式進行部分語句的驗證. 但是兩種模式多有差異, 會令人無所適從.

例如, 以 Python 進行運算, 可以鍵入:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

首行代碼給 `miles` 賦值, 但是不顯示結果. 第二行是表達式, 解釋器會解析執行, 並顯示結果, 約為 42 公里.

但是在腳本模式同樣執行, 不會獲得輸出結果. 表達式在腳本模式不會輸出結果, 如果需要顯示, 可以採用 `print` 語句:

```
miles = 26.2
print(miles * 1.61)
```

初次使用, 會感到奇怪. 為了更深入了解, 可以在 Python 解釋器鍵入以下內容, 看看現象:

```
5
x = 5
x + 1
```

將同樣語句放入腳本並運行, 會如何? 如果將每個表達式都放入 print 語句中, 並運行, 又會如何?

2.5 運算次序

當表達式含有多個運算子時, 執行順序主要基於**運算次序**. 對於數學運算子, Python 也遵循數學慣例. 本書採用縮寫 **PEMDAS** 來輔助記憶以下規則:

- 括號 (**P**arentheses) 優先, 可以據此調整運算次序. 因為括號中的表達式會優先處理, 所以 `2 * (3-1)` 結果為 4, `(1+1)**(5-2)` 則為 8. 同時採用括號可以提高表達式的可讀性. 比如, `(minute * 100) / 60` 加上括號, 結果雖然沒有變化, 但是可讀性提高不少.

- 冪運算 (**E**xponentiation) 相對優先級更高. 所以 `1 + 2**3` 等於 9 而非 27, `2 * 3**2` 等於 18 而非 36.

- 乘法 (**M**ultiplication) 和除法 (**D**ivision) 優先於加法 (**A**ddition) 和減法 (**S**ubtraction). 所以 `2*3-1` 等於 5 而非 4, `6+4/2` 等於 8 而非 5.

- 對於同優先級的運算, 遵照自左向右的順序 (除冪運算). 在 `degrees / 2 * pi` 中, 先執行除法運算, 獲得的結果再乘以 `pi`. 如果想除以 2π, 可以使用括號處理或者改寫為 `degrees / 2 / pi`.

不必強求記憶這些運算子的優先級, 碰到比較複雜的表達式, 採用括號會方便很多.

2.6 字符串操作

通常, 字符串甚至類似數字的字符串的運算, 一般不能使用數學運算子, 下面的表達式均是非法運算:

```
'chinese'-'food'    'eggs'/'easy'    'third'*'a charm'
```

但是有兩個例外, + 和 *.

+ 可以實現**字符串拼接**, 也就是首尾相接. 例如:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

* 同樣可以作用於字符串, 表示堆疊. 例如'Spam'*3 表示 'SpamSpamSpam'. 如果一個值是字符串, 另外一個值就需要是整數.

+ 和 * 在字符串上的使用類似數值運算中的作用. 4*3 即是 4+4+4, 而 'Spam'*3 也等於 'Spam'+'Spam'+'Spam'. 但是, 數字的加法和乘法與字符串的拼接和堆疊, 在某方面的應用會有明顯不同. 你是否可以想到, 加法運算正常, 而字符串拼接卻表現截然不同的場景?

2.7 注釋

隨著代碼量越來越龐大, 越來越複雜, 閱讀並理解其邏輯也越來越困難. 形式語言使用時又寫得密密麻麻, 對著滿頁代碼去明白其邏輯和功能, 便異常艱難.

所以, 在程式代碼中加入一些自然語言來進行解釋, 說明其作用, 便顯得異常重要. 這些解釋說明, 便叫做**注釋**, 一般以井號 (#) 開頭:

```
# 計算所占一小時的百分比
percentage = (minute * 100) / 60
```

此例中, 注釋獨占一行. 你也可以把注釋寫在代碼末尾:

```
percentage = (minute * 100) / 60      # 所占一小時的百分比
```

代碼中所有 # 開頭的行, 在執行過程中都會被忽略.

編碼中實現特殊邏輯時, 注釋顯得尤為重要. 要記住, 說明編碼的*緣由*, 遠勝於講述代碼的*作用*.

下面這條注釋便顯得多餘:

```
v = 5      # assign 5 to v
```

而下面的注釋則包含了未體現在代碼中的有用信息:

```
v = 5      # 速度(米/秒)
```

好的變數命名可以減少注釋的使用, 但是變數名過長, 又顯得代碼繁雜而難以閱讀, 所以如何取捨, 需要權衡.

2.8 除錯

一般程式中會出現三種異常：語法錯誤, 運行時錯誤以及語義錯誤. 有效區分三者差異, 可以大大提高定位速度.

語法錯誤: " 語法" 指代碼結構以及使用規範. 例如, 括號必須成對出現, `(1 + 2)` 正確, 而 `8)` 便存在語法錯誤.

如果程式存在語法錯誤, Python 會直接報錯並退出執行, 程式便無法繼續運行. 開始學習編程時, 需要耗費大量精力跟蹤語法錯誤. 隨著經驗增長, 此類錯誤會越來越少, 而定位也會越來越快.

運行時錯誤: 運行時錯誤只有在程式開始運行後才會出現. 這些錯誤也叫作**異常**, 因為一般表示異常或者壞的事情發生了.

本書前面幾章的程式較為簡單, 運行時錯誤也較少遇到, 所以想窺其全貌, 還需一段時間.

語義錯誤: 第三種錯誤是" 語義錯誤", 意如其字. 如果代碼中存在語義錯誤, 代碼不會報錯, 只是無法得到預期的結果. 具體來說, 便是會按照你的指示去做.

語義錯誤的識別很棘手, 需要觀察輸出, 回溯代碼, 才能確定其邏輯.

2.9 術語表

變數 (variable): 標識值的名稱.

賦值 (assignment): 為變數賦值的語句.

狀態圖 (state diagram): 一堆變數和值的指向圖.

關鍵字 (keyword): 編程語言用來解析代碼的預留字, 像 `if`、`def` 和 `while`, 均不能作為變數名使用.

操作數 (operand): 運算子操作的單一值對象.

表達式 (expression): 變數、運算子以及值的集合, 共同來表示單一結果.

求值 (evaluate): 執行運算產生值, 從而簡化表達式的過程.

語句 (statement): 表示命令或者操作的代碼塊. 目前遇到的語句包括賦值語句以及打印語句.

執行 (execute): 通過執行計算來簡化表達式, 從而產生值.

交互模式 (interactive mode): 在 Python 解釋器中根據提示符進行代碼輸入的模式.

腳本模式 (script mode): 用 Python 解釋器讀取腳本並運行代碼的模式.

腳本 (script): 存儲代碼的檔案.

運算次序 (order of operations): 包含多運算子和操作數的表達式執行時所遵循的先後順序.

拼接 (concatenate): 操作數首位相接.

注釋 (comment): 幫助他人理解代碼的有效信息, 對程式執行無影響.

語法錯誤 (syntax error): 導致代碼無法解析 (因此無法解釋) 的錯誤.

異常 (exception): 程式運行時遇到的錯誤.

語義 (semantics): 代碼含義.

語義錯誤 (semantic error): 未獲得預期結果的錯誤.

2.10 習題集

Exercise 2.1. 重申從上節便給出的建議, 學習新知識, 盡量在交互模式下不斷試錯, 大膽假設, 小心求證.

- 既然 `n = 42` 正確, 那麼 `42 = n` 呢?
- `x = y = 1` 正確與否?
- 在一些編程語言中, 每個語句都會以分號; 結尾, 如果在 *Python* 的語句末尾加上分號會怎樣?
- 語句末尾加句號會如何?
- 數學公式中, 可以用 xy 表示 x 和 y 相乘. *Python* 中若如此使用會如何?

Exercise 2.2. 用 *Pyhton* 解釋器進行運算:

1. 半徑 r 的球體體積是 $\frac{4}{3}\pi r^3$. 那半徑為 5 的球體體積是多少?
2. 假設書原價 *$24.95*, 但書店擁有 *40%* 的折扣. 同時郵寄首件需要郵費 *$3*, 後續每件 *75* 美分, 如果批發 *60* 件, 總費用多少?
3. 如果我 *6:52* 離家, 以慢跑方式 (*8* 分 *15* 秒/每英里) 跑了 *1* 英里, 然後以快跑方式 (*7* 分 *12* 秒/每英里) 跑了 *3* 英里, 最後, 又以慢跑方式跑了 *1* 英里, 那什麼時候到家?

第 3 章　函數

在編程領域, **函數** 是指執行既定運算的語句組合. 定義函數便是確定名稱以及語句組合. 然後, 便可以通過函數名來" 調用" 函數.

3.1　函數調用

上文已展示過**函數調用**

```
>>> type(42)
<class 'int'>
```

此函數的名稱為 `type`, 括號中的表達式叫做函數的**參數**. 此處獲得的結果, 表示參數的型態.

通俗來說, 函數的作用便是根據" 輸入" 的參數, " 返回" 相應的結果. 這個結果也叫做**返回值**.

Python 提供了值型態轉換的函數, 正常情況下, `int` 函數會將任意值轉為整型, 除非輸入有誤:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` 雖然可以將浮點數轉為整數, 但是不會四捨五入, 而是直接捨去小數部分:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` 會將整數以及字符串格式小數轉為浮點數:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

最後, `str` 會將任意輸入值轉為字符串:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 數學函數

Pyhton 內置了數學模塊, 可以提供大部分的數學計算函數. **模塊**是指包含一類函數的檔案.

在使用模塊中的函數前, 我們需要用 **import 語句** 來導入模塊:

```
>>> import math
```

導入語句會創建一個名為 `math` 的**模塊對象**. 輸入模塊對象名稱, 可以看到一些相關信息:

```
>>> math
<module 'math' (built-in)>
```

模塊對象包含所有模塊中的函數和變數. 想調用相應函數, 需要調用模塊名和函數名, 中間用點 (也可以當作英文中的句號) 來分隔. 這種調用方法, 就是**點標法**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

第一個例子使用 `math.log10` 計算分貝信噪比 (已知 `signal_power` 和 `noise_power`). math 模塊也提供了 `log` 函數, 此函數是以 `e` 為底.

第二個例子是計算弧度 (`radians`) 的正弦值. 弧度 一般作為三角函數 (`sin`, `cos` 和 `tan`, 等) 的輸入參數. 而角度轉弧度, 需要除以 180 然後乘以 π:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

`math.pi` 表達式是從 math 模塊獲取變數 `pi`, 這是一個 π 的浮點格式近似值, 精確到小數點後 15 位.

如果熟悉三角函數原理, 便可以通過與 $\frac{\sqrt{2}}{2}$ 比較來驗證結果:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3 組合

目前, 我們學習了編程涉及的各個元素---變數, 表達式以及語句. 但都是分開來講, 並未整合在一起.

編程語言的強大之處在於, 可以構建眾多的小積木, 然後**組合**起來. 例如, 函數的參數既然可以寫各種表達式, 那便可以是算數運算:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

也可以是函數調用:

```
x = math.exp(math.log(x+1))
```

任何可以使用值的地方, 都可以使用表達式來替換, 除了一種情況: 賦值語句的左側必須是變數名. 左側出現表達式會報語法錯誤 (相關信息見後續章節).

```
>>> minutes = hours * 60                 # right
>>> hours * 60 = minutes                 # wrong!
SyntaxError: can't assign to operator
```

3.4 創建函數

我們在 Python 編程中一直是使用函數, 現在可以嘗試創建一個函數. **函數定義** 一般指定義函數名以及調用時使用的語句組.

這是一個示例:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

def 關鍵字表示這是一個函數定義. 函數名是 print_lyrics. 函數名的命名規則和變數名命名規則一致：可包含字母, 數字或下劃線, 首字母不能為數字. 同時不能使用關鍵字作為函數名, 要避免變數和函數重名.

函數名後的空括號表示沒有參數.

函數定義首行一般叫做**函數頭**, 其他部分叫**函數體**. 函數頭以冒號結尾, 函數體必須縮進. 依照慣例, 一個縮進等於四個空格. 消息體可以包含任意行語句.

print 語句中的字符串需要被雙引號包著. 一般單引號和雙引號作用相同, 所以人們多數情況使用單引號, 除非字符串中本來就有單引號, 這種情況就要使用雙引號了.

所有的引號（無論單雙）都是 "直引號", 就是鍵盤確認鍵 (Enter) 旁邊的那個, 像本句中這種 "彎引號", 在 Python 使用中就是非法的.

在交互模式下定義函數, 解釋器會輸出省略號 (...) 來提示函數定義尚未結束:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

想結束函數定義, 需要多敲一個換行.

函數定義完成便會創建一個 function 型態的**函數對象**:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

調用自定義函數和調用內置函數的方式一樣:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

一旦函數定義完成, 便可在其他函數中使用. 比如, 為了避免重複, 可以直接定義一個repeat_lyrics 函數:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

然後調用 repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

當然, 歌詞沒有這樣的.

3.5 定義和調用

將上述代碼片段整理為一個完整程式:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

此程式包含兩個函數: `print_lyrics` 和 `repeat_lyrics`. 這些函數定義的執行和其他語句一樣, 但是會在執行後創建函數對象. 函數內的語句只有在函數調用時才會執行, 同時定義函數並不會輸出結果.

如你所料, 函數運行前必須先定義函數, 也就是說, 函數定義語句先執行, 函數才能被調用.

練習一下, 將程式的末行提到開始, 令函數先被調用, 再被定義. 運行程式, 看看會報什麼錯誤.

將函數調用再次移回末尾, 然後將 `print_lyrics` 函數定義放到 `repeat_lyrics` 函數的後面, 看看程式運行時, 會發生什麼?

3.6 運行流程

為保證函數定義先於函數調用, 需要了解語句執行順序, 也叫做**運行流程**.

程式代碼從第一行開始, 從上到下, 逐行運行.

執行函數的定義不會改變代碼邏輯運行順序, 但是要謹記, 函數內的語句只會在函數被調用時才執行.

函數調用就像在代碼運行流程中繞路一樣, 不會繼續向下執行語句, 而是跳轉到函數體內, 運行內部語句, 完成後再回到離開的地方.

聽起來很簡單, 但是要意識到, 函數可以不斷調用函數 (譯者註: 類似套娃). 甲函數運行中, 可能會調用乙函數, 從而去執行其語句, 然後乙函數運行中又可能去調用其他函數!

幸虧 Python 解釋器能夠有效追蹤函數的執行流程, 才可以在函數執行完成後, 回到函數調用的地方, 從而繼續執行, 直到結束.

所以, 不要簡單地從上到下閱讀程式, 有時候更要關注代碼運行流程.

3.7 形參和實參

我們觀察到有些函數需要參數. 例如調用 `math.sin` 函數, 需要傳遞數值作為參數. 還有些函數不止一個參數: `math.pow` 需要兩個參數, 分別是底和冪.

函數內, 實參會被賦值給**形參**. 下面是函數傳參（實參賦值給形參）的定義:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

此函數會將實參賦值給形參 bruce, 當此函數被調用時, 會輸出兩次參數值（無論參數為何）.

此函數適用於任何可被打印的值.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

自定義函數和內置函數的適用規則是一致的, 所以print_twice 的實參也可以是任意表達式:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

實參在函數調用前運行, 所以在上例中, 表達式 'Spam '*4 和 math.cos(math.pi) 只會運行一次.

也可以使用變數作為實參:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

函數實參的變數命名 (michael) 和函數形參的命名 (bruce) 毫無關聯. 無論外部傳遞進來的值如何命名, 函數 print_twice 內部, 我們統稱其為 bruce.

3.8 局部變數和參數

函數內部定義的變數都是**局部**的, 也就是說, 只能作用於函數內部, 例如:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

此函數將兩個參數拼接後, 結果輸出兩次. 以下是用例:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

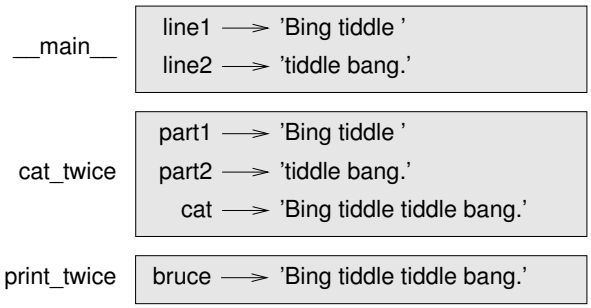

图 3.1: 堆疊圖.

`cat_twice` 運行結束, 內部變數 `cat` 便被銷毀. 如果現在嘗試輸出它, 會報錯:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

形參都是局部使用的, 例如 `bruce` 變數便不會出現在 `print_twice` 函數之外.

3.9 堆疊圖

若想跟蹤變數使用情況, 一般繪製**堆疊圖**比較有效. 和狀態圖類似, 堆疊圖也會標識每個變數的值, 不同之處是, 堆疊圖會標識變數所屬函數.

每個函數都用一個**框** 來表示, 一個框就是一個旁邊有函數名, 內部是參數以及變數值的箱體圖. 前述例子的箱體圖見圖 3.1.

堆疊圖中的各個框的排列方式標示了各個函數的調用關係. 此例中, `print_twice` 函數被 `cat_twice` 調用, `cat_twice` 被`__main__` 調用. `__main__` 是頂層框的一個特殊名字, 任何函數外的變數, 都屬於它.

形參和實參一一對應, 所以 `part1` 和 `line1` 的值相同, `part2` 和 `line2` 的值也相同, 並且, `bruce` 和 `cat` 的值也一樣.

如果函數在調用過程中出錯, Python 會輸出此函數名稱, 以及調用它的上級函數信息, 以及更上級的函數, 一直到 `__main__` 函數.

例如, 如果在`print_twice` 中調用 `cat`, 會得到名稱異常 (`NameError`):

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

這一系列跟蹤函數的信息叫做**溯源**, 這些信息會顯示是哪個檔案出錯, 哪行出錯, 以及運行時函數, 甚至引起錯誤的代碼行號.

在溯源信息中的函數次序和堆疊圖中的框的次序是一致的, 當前正在運行的函數都是在最下面.

3.10 有值函數和無值函數

目前用到過的一些函數, 例如數學函數, 都會返回結果, 一得之見, 暫稱**有值函數**. 其他函數, 像 `print_twice`, 運行完成, 並不會返回值, 便叫做**無值函數**.

當調用有值函數時, 一般是希望對返回結果另行處理, 比如, 賦值結果到某個變數, 或者置於表達式中使用:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

在交互模式調用函數, Python 解釋器會輸出結果:

```
>>> math.sqrt(5)
2.2360679774997898
```

但是在腳本模式, 調用有值函數, 卻永遠不會看到返回值!

```
math.sqrt(5)
```

這個腳本檔案主要計算 5 的平方根, 但因為沒有保存或者顯示結果, 所以不是很有用.

無值函數可以在屏幕顯示信息或者製造其他效果, 只是不會有返回值. 如果將函數運行結果賦值到某個變數, 那變數會指向一個特殊的值: `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

`None` 是個具有既定型態的特殊值, 和字符串 `'None'` 是不同的:

```
>>> type(None)
<class 'NoneType'>
```

目前我們定義的函數都是無返回值的, 後續章節將開始定義有返回值的函數.

3.11 函數何用

將程式劃分為一個個函數是否值得, 對此的考量, 可以參看以下緣由:

- 函數可以將一組語句歸為一類, 便於閱讀和除錯.
- 函數可以消除重複, 簡化代碼, 針對功能, 修改一處即可.
- 將程式切分為函數, 可以分別除錯, 然後組合它們即可.
- 設計良好的函數可以編寫一次, 除錯一次, 多處調用, 有效復用.

3.12 除錯

編程最重要的技能之一就是除錯了, 雖往往令人黯然神傷, 愁眉難舒, 卻是編程中最考驗才智, 最富有挑戰以及最有趣的地方.

在某些方面, 除錯和偵探工作很像. 面對線索, 有效推理, 確定造就結果的流程和緣由.

除錯又像實驗科學, 針對問題, 一旦有所想法, 便可修改程式, 再次嘗試. 如果假設正確, 便可預期修改後的結果, 向可執行的程式代碼推進. 如果假設錯誤, 便要繼續探究了. 正如 Sherlock Holmes(夏洛克 · 福爾摩斯) 所說, "除去所有不可能因素, 留下來的東西, 無論多麼不願意相信, 但這就是事實的真相."(A. Conan Doyle, *The Sign of Four*)

對某些人而言, 編程便是除錯. 這是因為, 編程便是不斷除錯程式, 直到實現目標的過程. 所以, 建議先寫可執行代碼, 然後逐步修改並除錯程式, 進而實現預定效果.

比如, Linux 操作系統具有數百萬行代碼, 但它起源於 Linus Torvalds 研究 Intel 80386 芯片的一段簡單代碼. 據 Larry Greenfield 所說, "Linus 的早期專案中, 有個程式是在 AAAA 和 BBBB 之間交替輸出, 而這後來演化為了 Linux." (*The Linux Users' Guide* Beta Version 1).

3.13 術語表

函數 (function): 特別命名的一堆語句的組合, 實現特定功能. 可以有參數, 也可以沒有參數, 可以返回值, 也可以不返回.

函數定義 (function definition): 創建函數的語句, 確定名稱, 參數以及內部語句.

函數對象 (function object): 函數定義運行後的值, 函數名便是指向函數對象的變數.

函數頭 (header): 函數定義的首行.

函數體 (body): 函數定義的內部語句.

形參 (parameter): 函數內部用於表示傳入參數的名稱.

函數調用 (function call): 運行函數的語句, 由函數名, 以及括號中的參數串列構成.

實參 (argument): 函數調用時傳入函數的值. 這個值會賦值給函數內部的形參.

局部變數 (local variable): 函數內部變數, 局部變數只能在函數內部使用.

返回值 (return value): 函數返回的結果. 如果用函數調用作為表達式, 那麼函數返回值, 便是表達式的結果.

有值返回 (fruitful function): 有返回值的函數.

無值返回 (void function: 返回值為 `None` 的函數.

空值 (`None`): 無值返回函數返回的結果.

模塊 (module): 包含相關函數以及定義的檔案.

導入語句 (import statement): 讀取模塊檔案並創建模塊對象的語句.

模塊對象 (module object): `import` 語句創建的值, 通過其可以獲取模塊內部的值.

點標法 (dot notation): 通過在模塊名後緊跟一個點 (英文句號) 和要調用的函數名, 來調用模塊函數的語法.

組合 (composition): 採用簡單表達式構建複雜表達式, 簡單語句塑造複雜模塊的方法.

運行流程 (flow of execution): 語句運行的順序.

堆疊圖 (stack diagram): 表示函數, 變數, 以及值的關係圖.

框 (frame): 堆疊圖中表示函數調用的箱體圖, 包括局部變數以及參數.

溯源 (traceback): 異常發生時, 輸出相關運行函數信息的過程.

3.14 習題集

Exercise 3.1. 書寫名為 right_justify 同時以字符串 s 為參數的函數, 其輸出的字符串左側有足夠的空格, 以使輸出結果的最後一個字符在第 *70* 字符處.

```
>>> right_justify('monty')
                                                                monty
```

提示: 使用字符串拼接以及堆疊. 此外, *Python* 提供了內置函數 len, 可以返回字符串長度, 例如, len('monty') 的結果為 *5*.

Exercise 3.2. 函數對象可以賦值給變數, 或者作為實參傳遞. 例如, do_twice 便將函數對象作為參數傳入, 並調用了兩次:

```
def do_twice(f):
    f()
    f()
```

下面是使用 do_twice 來調用兩次 print_spam 的例子.

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. 將上述代碼寫入腳本進行測試.
2. 修改 do_twice, 給其傳入兩個參數: 函數對象和值, 使其調用兩次傳入的函數對象, 這個函數對象會以傳入的值為參數.
3. 將本章print_twice 代碼複製到腳本.
4. 用新版 do_twice 函數調用 print_twice 兩次, 並傳入參數 'spam'.
5. 定義新函數do_four, 令其接收函數對象和值, 並以值為參數調用四次函數對象. 要求函數體只有兩條語句, 而不是四條.

答案: *https://thinkpython.com/code/do_four.py*.

Exercise 3.3. 說明: 此習題只涉及已學過的語句和功能.

1. 編寫繪製下面網格圖的函數:

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

   提示: 想要一行輸出多個值, 可以向 *print* 傳入多個值, 將值用逗號分隔:

```
print('+', '-')
```

一般 print 函數會附帶換行符, 但可以對其進行調整, 使其以空格結尾, 如下:

```
print('+', end=' ')
print('-')
```

上邊語句會將'+ -' 在同一行輸出, 如果再有 *print* 語句, 則會另起一行.

2. 編寫繪製四行四列類似網格的函數.

答案: *https://thinkpython.com/code/grid.py*. 致謝: 此習題基於 *Oualline* 書中的習題 (Practical C Programming, Third Edition, *O'Reilly Media, 1997).*

第 4 章　案例分析: 介面設計

本章採用案例來講述, 如何設計可以協同運行的函數.

主要介紹 `turtle`(烏龜) 模塊, 可以讓你用其繪圖功能, 製作圖片. 大部分 Python 安裝包會預裝 `turtle` 模塊, 但如果你使用的是 PythonAnywhere 網站, 現在還無法執行 turtle 示例 (至少創作本書時還不能).

如果你的計算機安裝了 Python, 便可以直接運行這些實例. 如果還沒有, 那麼最好現在安裝吧, 可以參看我寫的操作步驟`http://tinyurl.com/thinkpython2e`.

本章的代碼範例可以從`https://thinkpython.com/code/polygon.py`獲取.

4.1　模塊 turtle

打開 Python 解釋器, 鍵入以下命令, 以檢查是否安裝 `turtle` 模塊:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

運行此代碼, 會創建一個窗口, 此窗口包含一個箭頭, 代表一隻小烏龜. 關閉窗口.

創建名為 `mypolygon.py` 的檔案, 並輸入以下代碼:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

`turtle` 模塊 (小寫字母'*t*') 提供了函數 `Turtle` (大寫字母'T'), 它創建了 Turtle 對象, 並將其賦值給名為 `bob` 的變數. 輸出 `bob` 信息如下:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

這意味著 `bob` 指向了 `turtle` 模塊中 `Turtle` 型態的對象.

`mainloop` 會令窗口等待, 以執行其他操作. 但是本例只需關閉窗口, 暫不進行其他操作.

一旦創建了 Turtle 對象, 便可以調用**方法**在窗口中移動它. 方法和函數類似, 但是略有不同. 比如, 令小烏龜前進:

```
bob.fd(100)
```

方法 `fd` 來自於烏龜對象 `bob`. 調用方法就好比遞交申請：請求 `bob` 向前移動.

`fd` 的參數是像素距離, 實際距離要看屏幕尺寸.

其他可以調用的方法還有 bk, 實現向後移動, lt 實現左轉, rt 實現右轉. lt 和 rt 的參數都是以度為單位的角度.

同時, 每個烏龜都帶著筆, 可以抬起或落下; 如果筆落下, 烏龜移動時會留下痕跡. pu 和 pd 方法分別代表 "抬筆 (pen up)" 和 "落筆 (pen down)".

將下面代碼加入程式, 以實現繪製直角 (創建 bob 對象後, 調用mainloop 前):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

運行程式, 可以看到 bob 對象會向東移動, 然後向北移動, 並留下兩根垂直線段.

現在嘗試繪製個正方形, 先不要著急學習下面的內容.

4.2 簡單重複

我猜你會寫成下面的樣子:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

用 for 語句可以更加簡單地實現同樣的效果. 將下面代碼寫入 mypolygon.py 並運行: :

```
for i in range(4):
    print('Hello!')
```

所見如下:

```
Hello!
Hello!
Hello!
Hello!
```

這是 for 語句的簡單使用, 後面會講更多用法. 但這些已足夠令我們重寫繪製方塊的程式了. 暫時不要看下面的內容, 先嘗試一下.

以下用 for 語句實現繪製正方形:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

語句 for 的語法使用類似函數定義. 以冒號結尾的語句為函數頭, 縮進的內容為函數體. 函數體可以是任意行語句.

for 語句也叫**迴圈**, 因為代碼執行流程會重複執行迴圈體, 此例中, 會執行四次迴圈體.

這一版代碼和上一版略有不同, 繪製完最後一條線後, 會多一次轉向. 雖然多的這一步操作增加了耗時, 但是利用迴圈, 使重複操作更加簡單. 當然, 這也令小烏龜回到了開始的地方, 朝向了開始的方向.

4.3 習題集

以下為一系列操作 `turtle` 模塊的練習. 雖然是為了讓大家高興一下, 但是也有別出心裁的地方, 做練習的時候, 要好好思考一下醉翁之意.

習題後面有答案, 但是盡量在完成或者盡力嘗試後再看.

1. 編寫 `square` 函數, 參數是個 turtle 對象 `t`, 用 turtle 繪製正方形.

   編寫函數, 將 `bob` 作為參數傳給 `square`, 然後運行程式.

2. 為 `square` 添加參數 `length`, 令邊的長度為 `length`, 同時修改函數, 令其調用第二個實參. 再次運行程式, 給 `length` 賦不同的值進行測試.

3. 複製 `square` 並命名為 `polygon` . 添加參數 `n`, 令其繪製正 n 邊形. 提示: 正 n 邊形外角角度為 $360/n$ 度.

4. 編寫 `circle` 函數, 以 turtle 型態的 `t`, 以及半徑 `r` 為參數, 通過調用 `polygon` 函數, 輸入適當長度和邊數, 繪製一個近似的圓形. 用不同的 `r` 值, 測試程式.

   提示: 確定圓的周長, 使其滿足`邊長 * 邊數 = 周長`.

5. 升級 `circle` 版本, 命名為 `arc`, 新增參數 `angle`, 用此值確定所繪製的圓弧的大小. `angle` 以度為單位, 當 `angle=360`, `arc` 會繪製一個完整的圓.

4.4 封裝

第一道題是編寫繪製正方形的函數代碼, 調用函數, 傳遞 turtle 參數. 代碼如下:

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

函數內的 `fd` 和 `lt` 縮進兩次, 表示運行於函數定義裡面的 `for` 迴圈中. 下一行 `square(bob)`, 左邊距對齊同時沒有縮進, 表示上面的 `for` 迴圈以及函數定義都已結束.

函數內的 `t` 和 `bob` 指向的是同一個烏龜對象, 因此 `t.lt(90)` 就好比 `bob.lt(90)`. 那麼為何不直接用變數 `bob` 呢? 這是因為 `t` 可以代指任意烏龜對象, 而不僅僅是 `bob`, 因此也可以再創建一個烏龜對象, 並作為參數傳入 `square`:

```
alice = turtle.Turtle()
square(alice)
```

將一段代碼寫在函數中, 叫做**封裝**. 封裝的好處之一在於, 用一個簡單的名字來指代一段代碼, 類似文檔說明. 另一個好處是可以復用代碼, 複製黏貼大段的代碼, 總是不如復用代碼更顯簡潔!

4.5 泛化

下一步是給 `square` 增加參數 `length` , 詳見範例:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

給函數增加參數, 叫做**泛化**. 因為可以令函數使用場景更加廣泛：上版代碼中, 正方形的邊長總是同一個值, 而此版, 邊長可以是任意值.

下一步也是泛化. 不再是只繪製正方形, `polygon` 可以繪製任意多邊形, 詳見示例:

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

上面代碼繪製了邊長為 70 的正七邊形.

如果你使用的是 Python 2, `angle` 的值可能因為整除而出現偏差. 有個簡單的辦法, 便是 `angle = 360.0 / n`. 分子為浮點數, 結果便也是浮點數了.

當函數有多個數值參數時, 很容易混淆各自代表什麼, 以及參數順序. 所以比較好的辦法, 便是在傳遞實參串列時, 附上形參的名字:

```
polygon(bob, n=7, length=70)
```

這也叫做**關鍵字參數**, 因為是把形參的名字作為 "關鍵字" 包含進來. (不要同 Python 中 `while` 和 `def` 這種內置關鍵字混淆).

這種語法令程式可讀性更強. 同時也提示了實參和形參如何生效：調用函數時, 實參賦值給了形參.

4.6 介面設計

下一步要編寫 `circle`, 此函數以半徑 `r` 為參數. 以下是採用 `polygon` 來繪製正五十邊形的代碼：

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

首行根據公式 $2\pi r$ 和半徑 `r` 計算圓的周長. 要使用 `math.pi`, 需要先導入 `math`. 通常把 `import` 語句作為腳本的開始.

`n` 是要逼近圓的正多邊形的邊數, `length` 表示邊長. 最後, `polygon` 通過繪製正五十邊形, 來逼近半徑 `r` 的圓.

此代碼的局限性在於, n 是一個常量, 這表示繪製大的圓時, 線段過長. 而繪製小圓時, 需要耗費時間繪製小線段. 要想使此函數通用, 可以將 n 作為函數的參數. 這樣用戶 (circle 的調用者) 便可以更方便地使用, 但是介面會顯得不那麼簡潔了.

函數**介面** 是函數用途的簡介: 參數是什麼? 函數做什麼? 返回什麼? 如果一個介面聚焦於特定功能, 而非無關細節, 那麼這個介面就是 "簡潔" 的.

這個案例中, r 放在介面中合適, 是因為要用它來確定圓的大小. 而 n 放進介面則顯得不太友好, 是因為它只與如何繪製圓的細節有關.

與其令介面複雜, 不如根據周長來確定一個合適的 n:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 3
    length = circumference / n
    polygon(t, n, length)
```

現在邊的個數便是近似 circumference/3 的整數, 每個邊的長度也就近似於 3 了, 用這個長度, 繪製大圓快速, 小圓好看, 對任意尺寸的圓都適用.

給 n 加 3 是為了保證多邊形至少有 3 條邊.

4.7 重構

我們可以復用 polygon 來構造 circle, 是因為邊足夠多的正多邊形近似於圓. 但是, 構造 arc 就不適合了, 因為沒有辦法用 polygon 或者 circle 來繪製一段弧線.

一個替代策略是複製 polygon , 修改為 arc, 代碼如下:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

函數後半段和 polygon 很像, 但如果我們不調整介面, 便無法復用 polygon. 我們需要給 polygon 加入角度作為第三個參數, 來使其通用, 那麼也不能繼續使用 polygon 作為名字了! 反而叫 polyline 會更合適些:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

現在我們可以用 polyline 來重寫 polygon 和 arc 了:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)
```

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

最後, 我們再用 `arc` 來優化 `circle`:

```
def circle(t, r):
    arc(t, r, 360)
```

此過程---調整代碼, 優化介面, 便於復用---叫做**重構**. 此案例中, 我們發現 `arc` 和 `polygon` 具有一些相似代碼, 所以將其 "抽離" 為 `polyline`.

如果早有規劃, 我們可能會優先編寫 `polyline`, 避免後續耗時重構. 但一般你很難在專案開始的時候, 便深入了解一切, 規劃好所有介面. 只有開始編碼, 你才會更好地研究並理解其中曲折. 所以, 有時候重構的過程也是你學習新知識的歷程.

4.8 開發計劃

開發計劃便是指編碼的流程. 以上案例中, 我們採用的流程便是 "封裝和泛化", 詳細步驟如下:

1. 先不考慮函數定義, 寫段小程式.
2. 如果代碼可用, 抽離相關代碼, 封裝進函數, 並命名.
3. 增加合適參數, 泛化函數.
4. 重複步驟 1-3, 盡量抽離各個函數, 盡量複製黏貼, 減少打字 (以及重複除錯).
5. 嘗試重構程式. 比如, 在多個地方有相似代碼, 試試能否將其拆解為恰當的通用函數.

這種流程存在一定的缺點---後續會講述一些替代方案---但是在你無法提前規劃, 如何將程式拆解為函數時, 這種方法可以讓你再編碼的同時優化程式.

4.9 幫助文檔

幫助文檔是指函數開頭部分, 用來對介面 ("doc" 是 "documentation" 的簡稱) 進行解釋說明的文檔. 下面是個例子:

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

一般幫助文檔都要用三引號引起來, 也叫多行字符串, 因為三引號允許字符串跨行.

幫助文檔雖然簡潔, 但是重要, 因為它包含函數使用的相關說明. 同時簡要概括了函數的功能 (無需探查函數細節實現), 以及各個參數的型態和作用, 以及應用場景.

設計介面, 重要的一步, 便是編寫友好的幫助文檔. 一個設計優良的介面, 應該是易於理解的; 如果需要耗費大量口舌講解, 那麼這個介面還有待優化.

4.10 除錯

介面類似於調用方和函數之間的中間人. 調用方提供特定參數, 函數執行特定操作.

例如, `polyline` 需要四個參數: Turtle 對象 `t`, 整數 `n`, 正數 `length`, 以及以度為單位的數字 `angle`.

這些條件, 叫做**前置條件**, 因為這些條件滿足了, 函數才能執行. 相反, 函數結尾的內容叫做**後置條件**. 後置條件包括函數預期效果 (比如繪製線段), 以及意外情況 (移動 Turtle 或其他影響).

前置條件是調用方的任務. 如果調用方違背 (標註詳細的) 前置條件, 導致函數異常, 那麼責任在於調用方, 而不是函數.

如果前置條件都滿足而後置條件未達到預期, 那問題便出在函數中. 如果前置和後置條件都清晰明了, 那麼除錯會方便很多.

4.11 術語表

方法 (method): 用句點於對象連接的函數.

迴圈 (loop): 程式中可以重複執行的部分.

封裝 (encapsulation): 將一系列語句放入函數定義中的過程.

泛化 (generalization): 將不必要的特定的值 (比如特定數字) 更換為通用內容 (變數或參數) 的過程.

關鍵字參數 (keyword argument): 一種實參格式, 將形參名字作為 ''關鍵字'' 囊括於內.

介面 (interface): 關於如何使用函數的描述, 包括函數名, 實參, 以及返回值信息.

重構 (refactoring): 修改代碼, 以優化函數介面, 提升代碼質量的流程.

開發計劃 (development plan): 編碼流程.

幫助文檔 (docstring): 函數定義開頭部分的文檔, 用來描述函數介面.

前置條件 (precondition): 函數運行前, 調用方需滿足的條件.

後置條件 (postcondition): 函數結束前, 需要滿足的要求.

4.12 習題集

Exercise 4.1. *從 https://thinkpython.com/code/polygon.py 下載本章代碼.*

1. *繪製堆疊圖, 描述 `circle(bob, radius)` 運行時的程式狀態. 可以手算, 或者在代碼中加入 `print` 語句.*
2. *第 4.7節的那版 `arc` 並不是很精確, 因為用來逼近圓的線段總會在圓的外側. 所以 Turtle 的最終位置總會和真實位置有偏差. 我的解決方案可以有效降低誤差. 閱讀代碼, 看看是否有所幫助. 畫個堆疊圖, 也許便明白了其原理.*

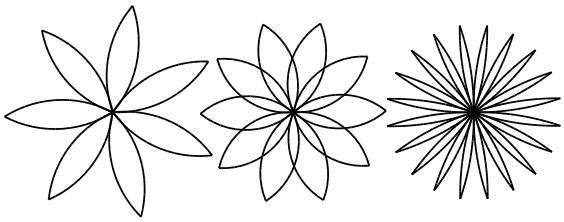

图 4.1: 花朵.

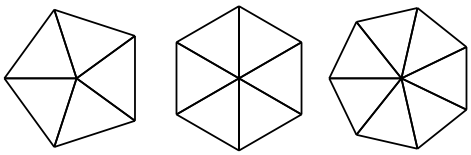

图 4.2: 餅.

Exercise 4.2. 嘗試編寫函數, 繪製圖 *4.1*中的花朵.

參閱: *https://thinkpython.com/code/flower.py*, 以及 *https://thinkpython.com/code/polygon.py*.

Exercise 4.3. 編寫函數, 繪製圖 *4.2*中圖形.

參閱: *https://thinkpython.com/code/pie.py*.

Exercise 4.4. 字母表中的字母都是由幾個基礎元素構成, 比如垂直線和水平線, 以及曲線. 設計個字母表, 使用種類最少的基本元素, 編寫函數繪製字母.

你要為每個字母開發函數, 比如 `draw_a`, `draw_b`, 等等, 將這些函數放在名為 `letters.py` 的檔案中. 你可以從 *https://thinkpython.com/code/typewriter.py* 下載個 *"turtle typewriter"*, 來校驗代碼.

可參閱 *https://thinkpython.com/code/letters.py*; 以及 *https://thinkpython.com/code/polygon.py*.

Exercise 4.5. 去*http://en.wikipedia.org/wiki/Spiral* 了解一下螺旋線; 然後編碼實現一個阿基米德螺線 (或其他種類). 參閱: *https://thinkpython.com/code/spiral.py*.

第 5 章　條件和遞歸

本章的重點是 `if` 語句, 基於程式的狀態不同, 執行不同的代碼. 但首先我要先介紹兩個新的運算子: 向下取整和求模.

5.1　向下取整和求模

向下取整 運算子, 即 `//`, 兩數相除, 結果向下取整. 例如, 假設某電影時長 105 分鐘, 你想知道按小時計, 是多長時間. 一般除法返回的是浮點數:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

但一般我們表示小時, 不用小數格式. 向下取整返回的就是整的小時數, 捨棄了小數部分:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

要想知道捨棄的分鐘數, 只需減去一小時, 剩下的便是了:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

還可以使用**求模運算子**, `%`, 兩數相除, 返回餘數.

```
>>> remainder = minutes % 60
>>> remainder
45
```

求模運算的用途不止於此. 例如, 看一個數字是否可以被另一個整除--如果 `x % y` 結果為 0, 則 `x` 可以被 `y` 整除.

你也可以通過求模從數字中提取一位或多位餘數. 例如, `x % 10` 獲取的便是 `x` 除以 10 後剩下的餘數. 同樣, `x % 100` 獲取的就是兩位餘數.

如果你使用的是 Python 2, 除法略有不同. 其除法運算子, `/`, 會在兩個整數相除時, 自動向下取整, 如果有一個是浮點數, 那麼結果才是浮點格式.

5.2 布爾表達式

布爾表達式 是用來表示結果是對或錯的表達式. 下面例子使用運算子 `==`, 比較兩個操作數, 如果相等, 則結果為 `True`, 否則為 `False`:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` 和 `False` 都是布爾格式的值, 不是字符串:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

`==` 運算子是**關係運算子**的一種, 其他還有:

```
x != y               # x不等於 y
x > y                # x 大於 y
x < y                # x 小於 y
x >= y               # x 大於等於 y
x <= y               # x 小於等於 y
```

雖然這些運算子對你來說並不陌生, 但是 Python 中的運算子和數學裡的不同. 一個很常見的錯誤就是混淆單等號 (=) 和雙等號 (==). 要知道, `=` 是賦值運算子, 而 `==` 是關係運算子. 而且也沒有 `=<` 和 `=>` 這種運算子.

5.3 邏輯運算子

有三種**邏輯運算子**: `and`, `or`, 和 `not`. 其語義和英文中的意思很相似. 例如 `x > 0 and x < 10` 表示只有在 `x` 大於 0 且小於 10 時才為真.

`n%2 == 0 or n%3 == 0` 需要在一個或者全部條件都成立時為真, 也就是說, 被 2 整除或被 3 整除.

最後, `not` 運算子是個對布爾表達式取反的操作. 所以, 當 `x > y` 為假, `not (x > y)` 便為真, 也就表示 `x` 小於或者等於 `y` 時, 為真.

嚴格來說, 邏輯運算子的操作數都應該是布爾表達式, 不過 Python 在這方面不太嚴格. 任何非零的數都當作 `True`:

```
>>> 42 and True
True
```

這種靈活性固然有用, 但是一些細微差異, 會令人困惑. 盡量不要如此使用 (除非你知道你在做什麼).

5.4 條件執行

若想寫出實用性強的程式, 就必然需要程式可以根據條件, 進行選擇處理. **條件語句** 便可以解決此難題. 最簡單的便是 `if` 語句:

```
if x > 0:
    print('x is positive')
```

if 後面的布爾表達式, 叫在**條件**. 如果為真, 則縮進的語句執行, 如果為假, 則不執行.

if 語句和函數定義的結構一樣: 頭部和緊隨其後的縮進體. 這樣的語句統稱為**複合語句**.

縮進體對於語句行數沒有限制, 不過至少要有一行. 但是有時候, 縮進體暫時不想寫語句 (類似於待寫代碼的占位符), 這種情況, 可以使用 pass 語句, 表示不做任何操作.

```
if x < 0:
    pass          # TODO: 需處理負數!
```

5.5 選擇執行

if 語句第二種使用場景, 便是 ‘‘選擇執行”, 一般存在兩種選擇, 具體執行哪個, 需要根據條件判斷. 語義結構如下:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

如果 x 對 2 取余後, 結果為 0, 則 x 為偶數, 並輸出相應信息. 如果不為 0, 則執行第二組語句. 這種條件非真即假, 必然有相應方案會執行. 這些方案, 叫做**分支**, 因為它們屬於執行流程上的分叉.

5.6 鏈式條件

有時候, 會遇到多種可能方案, 便需要更多的分支. 一種方式是採用**鏈式條件**進行處理:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elif 是 “else if’’ 的縮寫. 同樣, 上述代碼也只有一個分支會運行. 對於 elif 語句的數量, 沒有限制. 至於 else, 不是必須的, 但如果有的話, 則必須放到結尾.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

這些條件都是順序檢驗. 如果一個為假, 則檢驗下一個, 依此類推. 如果條件為真, 則相應分支開始執行, 同時這些判斷語句也都不會再執行. 即使存在多個條件為真, 那也只會執行第一個為真的分支.

5.7 嵌套條件

條件判斷也可以嵌套於其他條件內. 可以將上一節的例子改寫如下:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

外部的條件判斷包含兩個分支. 第一個只包含一個簡單語句. 第二個則包含另外一個 `if` 語句, 這個語句又包含兩個分支, 這兩個分支也都很簡單, 只是簡單語句. 同樣, 它們的位置也可以繼續放條件語句.

雖然語句的縮進可以使代碼結構清晰, 但是**嵌套條件**的語句閱讀起來卻很麻煩. 所以, 最好還是盡量不用.

邏輯運算子可以有效簡化嵌套條件語句, 例如, 可以用一行條件語句來重寫下面的代碼:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

只有兩個條件都滿足, `print` 語句才會運行, 這恰恰和 `and` 運算子的作用一樣:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

對於這種條件判斷, Python 提供了更簡潔的方案:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8 遞歸

既然一個函數可以調用另一個函數, 那麼, 函數也就可以調用自身. 雖然目前還未看到其用途, 但這卻是程式最神奇的功能之一了. 看看以下示例:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

如果 `n` 為 0 或負數, 輸出單詞 “Blastoff!”. 否則, 輸出 `n` , 然後調用函數自身 `countdown` 並以 `n-1` 為實參.

如果如下一般調用此函數, 會如何?

```
>>> countdown(3)
```

調用 `countdown`, 並且 `n=3`, 由於 `n` 大於 0, 輸出 3 並以 `n-1` 為參數調用自身...

> 調用 `countdown`, 並且 `n=2`, 由於 `n` 大於 0, 輸出 2 並調用自身...

調用 countdown, 並且 n=1, 由於 n 大於 0, 輸出 1 並調用自身...

調用 countdown, 並且 n=0, 由於 n 不大於 0, 輸出 "Blastoff!", 然會返回.

n=1 的 countdown 執行完結, 返回.

n=2 的 countdown 執行完結, 返回.

n=3 的 countdown 執行完結, 返回.

然後會回到 __main__ 中, 所有輸出如下:

```
3
2
1
Blastoff!
```

函數內部調用了自身, 這種函數便是**可遞歸的**; 其執行過程叫做**遞歸**.

再舉個例子, 我們寫個輸出 n 次字符串的函數.

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

如果 n<=0, 則 **return 語句**終止函數運行. 運行流程立刻返回到調用者, 函數其他代碼不執行.

函數其餘部分代碼和 countdown 類似: 輸出 s, 以 $n-1$ 為參數, 調用自己, 重複 $n-1$ 次. 那麼, 所有輸出的行數便是 1 + (n - 1), 其和為 n.

對於這種簡單範例, 用 for 迴圈可能更加方便. 但是後續我們會遇到一些例子, 用 for 迴圈比較難編寫, 而用遞歸卻輕而易舉, 所以, 現在不啻為一個好的開始.

5.9 遞歸函數堆疊圖

在第 3.9節, 我們使用堆疊圖來描述函數調用時程式的狀態. 同樣, 堆疊圖也有助於我們更好理解遞歸函數.

一旦函數被調用, Python 都會創建一個包含函數局部變數以及參數的框. 所以對於遞歸函數, 可能會同時創建多個框.

圖 5.1便是以 n = 3 為參數, 調用 countdown 所產生的堆疊圖.

通常, 最頂層的框屬於__main__. 其為空, 是因為沒有在__main__ 內創建變數或者傳入參數.

四個 countdown 的框分別對應不同的 n 值, 最底層 n=0 的堆疊, 叫做邊界條件, 也就是不再進行遞歸調用的堆疊, 所以下面也不會再有其他框了.

做個練習, 以s = 'Hello' 和 n=2 為參數, 繪製print_n 調用的堆疊圖. 然後編寫名為 do_n 的函數, 參數為一個函數對象, 和一個數值 n. 使其調用 n 次給定的函數.

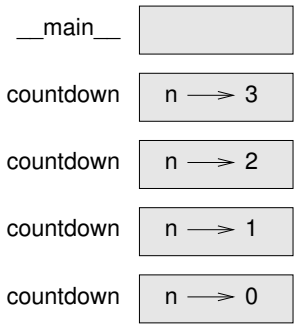

图 5.1: 堆疊圖.

5.10 無窮遞歸

如果遞歸一直無法觸及邊界條件, 則會一直調用, 永不終止. 這便叫做**無窮遞歸**, 出現這種情況, 往往表示情況不妙. 下面是個無窮遞歸的極簡程式:

```
def recurse():
    recurse()
```

多數編程環境中, 無窮遞歸的程式不會一直運行. 在 Python 中, 當達到了最大遞歸深度, 便會報錯:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

這次的追蹤信息比以往的要長一些. 這個錯誤發生時, 堆疊中已經有 1000 個遞歸框了!

如若不幸遇到無窮遞歸, 最好重新審視一下你的函數, 確保存在不再遞歸調用的邊界條件. 如果已有邊界條件, 請檢查是否可以保證其被觸達.

5.11 鍵盤輸入

到目前為止, 我們編寫的程式, 基本都沒有涉及用戶輸入, 它們每次執行都一樣.

Python 提供了內置函數 `input`, 可以暫停程式運行, 等待用戶鍵入信息. 當用戶敲擊 Return 或 Enter 鍵時, 程式恢復運行, 同時`input` 將用戶輸入作為字符串返回. 在 Python 2 中, 同樣作用的函數叫做`raw_input`.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

在接收用戶輸入時, 最好給用戶以提示, 使其知曉要輸入的內容. `input` 的參數便是提示信息內容:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

提示信息末尾的\n 是個**換行符**, 表示另起一行的特殊字符. 所以用戶的輸入信息會處於提示信息下面.

如果想要一個整數, 那就要將返回值轉為 int:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

但如果用戶輸入的不是數字型態的字符串, 那便要報錯了:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

後續我們會學習如何應對這種錯誤.

5.12 除錯

當遇到語法異常或者運行時異常, 其報錯信息詳細而充分, 但會令人無所適從. 一般最有用的也就下面兩部分:

- 哪種錯誤
- 發生何處

語法錯誤通常容易識別, 但是有些則會誤導我們. 空格異常通常比較麻煩, 因為空格和製表 (Tab) 符都看不到, 所以容易被忽視.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

此例中, 問題在於, 第二行多了一個空格. 但是錯誤信息指向 y, 便舞蹈了我們. 通常, 錯誤信息只標識了錯誤發生的位置, 但是問題代碼可能在此位置之前甚至是前行代碼.

運行時錯誤也如此, 假設以分貝為單位, 計算信噪比. 公式是 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. 在 Python 中, 編碼如下:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

運行代碼, 會遇到報錯:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

錯誤信息顯示, 問題出在第 5 行, 但是這行代碼看不出任何問題. 為了確定具體問題, 可以輸出 `ratio` 值, 顯示是 0. 那問題便出在第 4 行, 本應用浮點數除法, 卻誤用了整除.

對錯誤信息要仔細閱讀, 但也不能完全偏信.

5.13 術語表

向下取整 (floor division): 運算子`//`, 將兩數相除, 結果向下 (負無窮方向) 取整.

求模運算子 (modulus operator): 百分號 (`%`) 表示的運算子, 用於整數, 表示兩數相除後的餘數.

布爾表達式 (boolean expression): 結果為 `True` 或 `False` 的表達式.

關係運算子 (relational operator): 比較操作數的運算子: `==`, `!=`, `>`, `<`, `>=`, 和 `<=`.

邏輯運算子 (logical operator): 拼接布爾表達式的運算子: `and`, `or`, 以及 `not`.

條件語句 (conditional statement): 根據條件, 確定程式運行流程的語句.

條件 (condition): 條件語句中的布爾表達式, 確定分支走向.

複合語句 (compound statement): 由頭部和縮進體構成的語句, 頭部以冒號 (`:`) 結尾, 縮進體相對頭部需要縮進.

分支 (branch): 條件語句中可選的語句序列中的一種.

鏈式條件 (chained conditional): 一系列可選分支構成的條件語句.

嵌套條件 (nested conditional): 出現在條件語句的分支中的條件語句.

返回語句 (return statement): 令函數立刻結束並返回給調用方的語句.

遞歸 (recursion): 函數調用自身的過程.

邊界條件 (base case): 遞歸函數中不再進行遞歸調用的條件分支.

無窮遞歸 (infinite recursion): 不存在或者永遠無法觸及邊界條件的遞歸, 最終, 無窮遞歸會報運行時異常.

5.14 習題集

Exercise 5.1. time 模塊提供同樣名為 time 的函數, 此函數以某個 ‘‘時間點’’ 為基準, 返回當前格林威治時間戳. 理論上, 可以以任意時間為參考點, 而在 *Unix* 系統中, 一般以 *1970* 年 *1* 月 *1* 日為參考 ‘‘時間點’’.

```
>>> import time
>>> time.time()
1437746094.5735958
```

編寫腳本, 實現將當前時間轉換為一天中的時間 (以時分秒為格式), 以及基準時間點以來的天數.

Exercise 5.2. 費馬大定理說, 沒有任何正整數 *a*, *b*, 和 *c* 滿足

$$a^n + b^n = c^n$$

當 *n* 大於 *2* 時.

1. 編寫函數 check_fermat , 四個入參—a, b, c 和 n— 以檢驗費馬大定理是否成立. 如果 *n* 大於 *2* 同時滿足

   $$a^n + b^n = c^n$$

   那麼程式應輸出, *“Holy smokes, Fermat was wrong!”*, 否則, 輸出 *“No, that doesn’t work.”*

2. 編寫函數, 令用戶輸入 a, b, c 和 n, 並將其轉換為整數, 然後用check_fermat 來檢驗是否違背了費馬大定理.

Exercise 5.3. 給你三根木棍, 你不一定可以將其拼成三角形, 比如, 一根 *12* 英寸長, 其餘兩根 *1* 英寸長, 這兩根太短, 以至於都到不了長的那根的中間. 所以, 對於三根任意長度的木棍, 有個簡單方案, 可以檢驗其是否可以拼成三角形:

> 三根木棍中, 如果有任意一根長度大於另外兩根之和, 便拼不成三角形. 否則, 便可以拼成三角形. (如果兩邊之和等於第三邊, 便稱其為 ‘‘退化’’ 三角形.)

1. 編寫is_triangle 函數, 以三個整數變數為入參, 同時根據三個特定長度的木棍是否可以拼成三角形, 來輸出 *“Yes”* 或 *“No”*.

2. 編寫函數, 提示用戶輸入三個木棍的長度, 並將其轉換為整數, 然後用is_triangle 檢測這三個值是否可以拼成三角形.

Exercise 5.4. 下面的程式會輸出什麼? 繪製堆疊圖, 展示輸出結果時的程式狀態.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. 調用 recurse(-1, 0), 會發生什麼?

2. 為此函數編寫幫助文檔, 告知使用函數所須了解的相關信息 (僅此而已).

以下練習需要用到第 4節提到的 turtle 模塊:

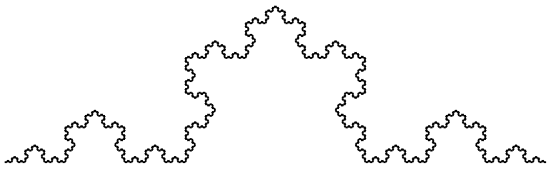

图 5.2: 科赫曲線.

Exercise 5.5. 閱讀下面函數, 看看是否能明白其功能 (參閱第 4節的案例). 運行並看看是否正確.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

Exercise 5.6. 科赫曲線 (*The Koch curve*) 是類似圖 5.2的一種分形幾何. 要想繪製長度為 x 的曲線, 下面是需要做的:

1. 繪製長為 $x/3$ 的科赫曲線.
2. 左轉 60 度.
3. 繼續繪製長 $x/3$ 的曲線.
4. 右轉 120 度.
5. 再繪製長 $x/3$ 的曲線.
6. 左轉 60 度.
7. 繪製長為 $x/3$ 的曲線.

當 x 小於 3 時, 會有所不同: 在此情況下, 繪製所得為長 x 的一段直線.

1. 編寫 `koch` 函數, 以 *turtle* 和長度 *length* 為參數, 使用 *turtle*, 根據指定長度, 繪製科赫曲線.
2. 編寫 `snowflake` 函數, 使其繪製三條科赫曲線, 從而構成雪花的輪廓.

   參閱: *https://thinkpython.com/code/koch.py*.
3. 生成科赫曲線有多種方法. 參看 *http://en.wikipedia.org/wiki/Koch_snowflake* 上的案例, 選擇你喜歡的進行實現.

第 6 章　有值返回函數

目前我們用到的很多 Python 函數都有返回值, 比如 math 函數. 但我們目前寫的函數, 都是無返回值的: 它們只是實現特定的效果, 比如輸出值, 或者移動小烏龜, 只是它們都沒有返回值. 本章, 重點學習如何編寫有值返回函數.

6.1　返回值

調用函數便會產生返回值, 一般被賦值給變數或者作為表達式的一部分使用.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

目前寫的函數, 多是無返回值的. 籠統講, 是沒有返回值, 然而, 更準確地說, 返回值是空 (None).

本章, 我們總算要寫一些有返回值的函數了. 第一個例子是 area, 根據給定的半徑, 返回面積:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

前面我們學過了 return 語句, 但是在有值返回的函數中, return 語句包含表達式. 其意思為: “立即將此表達式作為返回值進行返回. ” 鑑於表達式可簡可繁, 上述函數可以精煉為下面樣子:

```
def area(radius):
    return math.pi * radius**2
```

但是, 使用類似 a 這樣的**臨時變數**, 更加清晰明了, 便於除錯.

有時候, 根據條件設置多個不同返回語句, 更加高效:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

這些 return 語句都處於可選條件分支中, 而且, 只有一個會執行.

一旦返回語句執行, 函數不再執行後續語句, 立刻終止運行. return 語句後的代碼, 或者任何不被觸及的代碼, 都叫做**無效代碼**.

有值返回函數中, 最好保證程式中的每種可能情況, 都有 return 語句. 例如:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

這個函數錯誤之處在於, 如果 `x` 恰好是 0, 便沒有條件為真, 也就不會觸及任何 `return` 語句. 即使運行到函數最後, 返回值也會是 `None`, 而不會是 0 的絕對值.

```
>>> print(absolute_value(0))
None
```

順便提一下, Python 提供了內置函數 `abs` 來計算絕對值.

做個練習, 編寫 `compare` 函數, 輸入為 `x` 和 `y`, 如果 `x > y`, 返回 `1`, 如果 `x == y`, 返回 `0`, 如果 `x < y`, 返回`-1`.

6.2 增量開發

隨著編寫的函數越來越長, 你會發現, 除錯時間也越來越恐怖.

若想解決越來越複雜的程式, 可以試試新的方法, 即**增量開發**. 增量開發是通過每次只編寫並測試少量代碼, 從而避免長時間的連續開發除錯.

比如, 若想計算兩坐標 (x_1, y_1) 和 (x_2, y_2) 之間的距離, 根據勾股定理, 可以得到距離為:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

首先考慮 `distance` 函數是什麼樣子. 換句話說, 就是何為輸入 (形參), 何為輸出 (返回值)?

此例中, 輸入為兩個點, 即四個值表示的坐標. 返回值為兩點的距離, 用浮點數表示.

馬上你便可以寫出函數的大致輪廓:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

顯然, 此函數無法計算距離, 因為其總是返回 0. 但是, 此函數語法正確, 所以可以運行, 這意味著, 在其變複雜之前, 你能夠對其進行測試.

用實例參數調用此函數, 看看效果:

```
>>> distance(1, 2, 4, 6)
0.0
```

選擇這些值作為坐標, 是因為其水平距離是 3, 垂直距離是 4, 兩點距離便是 5, 也就是 3-4-5 直角三角形的斜邊. 在測試函數過程中, 知道預期結果, 尤為重要.

我們已經確認此函數語法正確, 現在便可以補充完善代碼了. 首先, 我們計算 $x_2 - x_1$ 和 $y_2 - y_1$ 的值, 保存為臨時變數, 並輸出.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

如果函數正常, 則會輸出dx is 3 和 dy is 4. 這樣, 我們便知道函數入參正確, 同時前期的計算無誤. 如果發生異常, 那麼我們只需要仔細檢查這幾行代碼.

下一步, 計算 dx 和 dy 的平方和:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

再次運行代碼, 並檢查輸出 (輸出值應為 25). 最後, 使用 math.sqrt 計算最終的返回值:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

如果運行正常, 意味著程式正確. 否則, 就要輸出返回語句前的 result 的值, 仔細排查.

最終版本的函數, 除了返回一個值, 不會輸出任何信息. print 語句只是為了便於除錯代碼, 一旦確認程式無誤, 就應將其移除. 類似這種代碼, 一般叫做**腳手架代碼**, 主要用來輔助構建程式, 而並不應忝列最終成品之中.

開始時, 一般每次只增加一兩行代碼. 但隨著經驗增長, 慢慢你便可以駕馭大段代碼了. 無論如何, 增量開發都能節約大量除錯時間.

此流程的主要步驟如下:

1. 首先寫個可運行的程式, 然後逐步增加. 任何時候遇到錯誤, 都要弄清緣由, 盡快解決.
2. 利用變數保存中間狀態值, 以便展示和檢驗.
3. 一旦程式運行正常, 便可以移除冗餘代碼, 並精簡繁瑣的語句. 但是要警惕不要使代碼難以閱讀和理解.

做個練習, 用增量開發的方式, 寫個函數 hypotenuse, 給定直角三角形的兩個直角邊, 返回斜邊長度. 開發時, 記錄開發流程的各個階段.

6.3 組合

如你所想, 函數可以調用函數. 例如, 編寫函數, 輸入兩個坐標, 一個是圓心, 一個是圓周上的點, 計算圓的面積.

假設圓心坐標為變數 xc 和 yc, 圓周上點的坐標為 xp 和 yp. 首先要計算圓的半徑, 也就是兩點之間的距離. 便可以借用之前寫過的 distance 的函數, 如下:

```
radius = distance(xc, yc, xp, yp)
```

下一步便是根據半徑, 計算面積, 借用之前函數:

```
result = area(radius)
```

將上述步驟封裝為一個函數, 得到如下:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

臨時變數 radius 和 result 在開發和除錯時有用, 但是程式一旦正常運行, 便可以通過組合函數調用, 精簡如下:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 布爾函數

函數可以返回布爾值, 從而便於隱藏函數內複雜的判斷邏輯. 例如:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

給布爾函數命名, 通常採用類似是/否提問的詞語; is_divisible 返回 True 或 False, 以表明 x 是否被 y 整除. 比如:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

既然 == 運算子的結果為布爾值, 我們便可以直接返回, 從而精簡函數:

```
def is_divisible(x, y):
    return x % y == 0
```

布爾函數一般用於條件語句:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

也可以這麼寫:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

但這個比較就顯得多餘了.

做個練習, 編寫函數 is_between(x, y, z), 如果 $x \leq y \leq z$, 返回 True, 否則返回 False.

6.5 再說遞歸

目前, 我們只學到了 Python 的一小部分, 但麻雀雖小, 五臟俱全, 這一小部分便足以表達一門完整的編程語言了, 也就是說, 如果一切皆是計算, 那麼以上所學便已足夠. 任何程式都可以只用以上所學模塊, 進行構建 (當然, 可能還需要一些控制設備的命令, 比如管理鼠標, 硬盤等, 但是也僅需要這些).

最早證明了上述偉大結論的是艾蘭圖靈 (Alan Turing), 最早的電腦科學家之一 (有人會計較其是數學家, 但是早期的電腦科學家, 基本都是數學家). 因此, 這個理論也叫做圖靈論斷

(Turing Thesis). 關於圖靈論斷, 如果想更加詳細深入了解, 我推薦 Michael Sipser 所寫計算理論導引 *(Introduction to the Theory of Computation)* 一書.

為了展示目前所學的威力, 我們分析幾個遞歸數學函數. 遞歸定義和迴圈定義類似, 某種意義上說, 函數定義體內包含了對定義體的引用. 通常一個完全迴圈的定義, 是無用的:

致命的: 一個形容詞, 用來描述一些致命的東西.

如果在辭典中看到這樣的定義, 一定鬱悶. 但你如果查詢一下用符號 ! 表示的階乘運算, 會看到以下內容:

$$0! = 1$$
$$n! = n(n-1)!$$

這個定義聲明 0 的階乘為 1, 同時對於任意 n 值的階乘, 便是 n 乘以 $n-1$ 的階乘.

所以 3! 表示 3 乘以 2!, 而 2! 表示 2 乘以 1!, 1! 又是 1 乘以 0!. 整理一下, 3! 等於 3 乘以 2 乘以 1 再乘以 1, 也就是 6.

如果你可以編寫其遞歸定義, 那便可以編寫 Python 實現它. 首先要確定參數. 很顯然, 此處 `factorial` 函數的參數為整數:

```
def factorial(n):
```

如果參數恰好為 0, 只需要返回 1:

```
def factorial(n):
    if n == 0:
        return 1
```

而其他情況就有意思了, 需要遞歸調用以找到 $n-1$ 的階乘, 然後將其與 n 相乘:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

此程式的執行流程和第 5.8節的 `countdown` 極為相似. 如果以 3 為參數調用 `factorial`:

3 不等於 0, 則走第二分支, 計算 `n-1` 的階乘...

2 不等於 0, 則走第二分支, 計算 `n-1` 的階乘...

1 不等於 0, 則走第二分支, 計算 `n-1` 的階乘...

0 等於 0, 則走第一分支, 不再遞歸調用, 返回 1.

返回值 1 乘以 n, 而 n 此時為 1, 則返回 1.

返回值 1 乘以 n, 此時 n 為 2, 返回為 2 的結果.

返回值 (2) 乘以此時為 3 的 n, 結果為 6, 也就是整個流程最終的結果.

圖 6.1 是這一系列函數調用的堆疊圖展現.

返回值會在堆疊中被傳遞. 每個框內, 返回值就是 `result` 的值, 也就是 `recurse` 和 `n` 相乘的結果.

最後的框中, 沒有局部變數 `recurse` 和 `result` , 是因為沒有走第二分支.

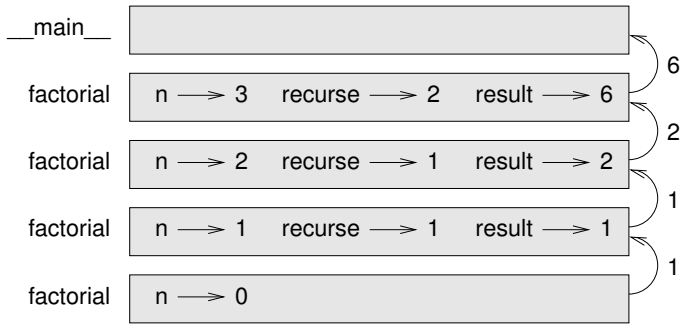

图 6.1: 堆疊圖.

6.6 置信遷移

閱讀程式的一種方式是跟蹤其執行順序, 但是你很快會不堪重負. 另一種可行方案, 我稱之為 "置信遷移". 當遇到某個函數調用時, 不是根據執行流程深入跟蹤, 而是假設這個函數工作正常, 可以返回正確結果.

實際你在使用內置函數時, 就在實踐置信遷移了. 調用 `math.cos` 或 `math.exp` 時, 並沒有深入檢查其函數執行體. 這是因為你相信寫出這些內置函數的人是優秀的編程人員, 所以相信這些函數可以正確運行.

對於你來說, 調用自己的函數也是同理. 例如第 6.4節, 我們編寫了`is_divisible` 函數, 用來判斷一個數是否可以被另一個數整除. 如果我們自己相信這些函數是正確的--分析代碼並測試都通過– 我們便可以直接使用它, 而無需再探究細節.

遞歸程式也一樣. 當你進行遞歸調用時, 無需一步步跟蹤運行流程, 你應該相信遞歸調用運行正常 (會返回正確的結果), 然後問問自己, "既然我可以計算出 $n-1$ 的階乘, 那麼是不是我也可以計算出 n 的階乘? " 很顯然你可以, 乘以 n 即可.

當然, 沒有完成函數編寫, 便假設其正常運行, 是有點奇怪, 所以這也是我們稱其為置信遷移的原因!

6.7 另例

在了解階乘之後, 我們要學習的是遞歸數學函數中最常見的用例, 斐波拉契數列. 其詳細定義可參閱`http://en.wikipedia.org/wiki/Fibonacci_number`:

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

翻譯為 Python 代碼, 代碼如下:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
```

```
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

若你嘗試跟蹤其流轉, 那即使一個很小的 *n* 值, 都能令你頭疼. 但是, 基於置信遷移, 如果兩個遞歸調用都運行正常, 那麼很顯然, 將其相加, 便能得到正確結果.

6.8 型態檢查

試試給 `factorial` 函數傳遞個 1.5 的參數, 會發生什麼?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

看起來陷入了無窮遞歸. 怎麼會這樣? 此函數有個邊界條件--當 `n == 0` 時. 但如果 `n` 不是整數, 那便會掠過 邊界條件, 一直遞歸下去.

第一次遞歸調用中, `n` 是 0.5. 下一次, 變成了-0.5. 再然後, 會越來越小 (更小的負數), 也就永遠不可能再成為 0.

我們有兩種方案. 一種是嘗試改進 `factorial` 函數, 使其支持浮點數, 或者使其檢驗參數型態. 第一種方案會寫出伽瑪函數, 超出了本書的範疇. 所以選擇方案二.

我們可以使用內置 `isinstance` 函數來檢驗參數型態. 同時, 我們也需要保證參數是正數:

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

第一個邊界條件, 針對非整數; 第二個, 則針對負整數. 這兩個邊界條件中, 都會輸出錯誤信息, 並返回 `None`, 以標識運行錯誤:

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

如果通過兩個檢驗, 那可以確定 *n* 現在是非負整數, 至此便可以確保遞歸會終止了.

此程式展示了一種叫做**哨兵**的角色. 前兩個條件, 就像哨兵一樣, 避免程式犯錯, 使其正確運行.

在第 11.4節, 我們會看到一種更靈活的方案來輸出錯誤信息: 上報異常.

6.9 除錯

將程式大而化小, 也就天然地為除錯創造了一個個的檢查點. 如果程式運行異常, 需要考慮以下三種可能原因:

- 函數入參異常, 前置條件未滿足.
- 函數本身異常, 後置條件不滿足.
- 返回值或者調用方式異常.

若要避免第一種異常, 可以在函數開始用 `print` 語句, 打印參數值 (以及型態). 或者編寫代碼, 校驗前置條件.

如果參數沒問題, 那在每個 `return` 語句前, 增加 `print` 語句, 打印返回值. 如果可以的話, 最好親自檢查結果. 同時盡量在調用函數時, 傳入合適的參數, 以使返回結果便於校驗 (如第 6.2節).

如果函數都正常, 那檢驗一下函數調用方式, 看看是否正確使用了返回值 (或者至少用到了返回值!)

在函數開頭和結尾添加打印語句, 有助於更直觀地觀察運行流程. 例如, 下面是包括打印語句的 `factorial` 版本:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

在此, 用空格字符串來控制輸出語句的縮進. 以下為 `factorial(4)` 的運行結果:

```
                factorial 4
            factorial 3
         factorial 2
     factorial 1
 factorial 0
 returning 1
     returning 1
         returning 2
             returning 6
                 returning 24
```

若惑於其函數執行流程, 那麼, 此輸出相比有助於理解. 有時候, 增加腳手架需要耗費一點時間, 但是, 這一點時間的花費, 往往能夠節約大量的除錯時間.

6.10 術語表

臨時變數 (temporary variable): 複雜運算中, 暫存中間值的變數.

無效代碼 (dead code): 程式永遠不會運行的語句, 通常在 `return` 語句之後.

增量開發 (incremental development): 一次僅僅開發並測試少量代碼的開發方案, 小步慢跑, 從而防止費時除錯.

腳手架 (scaffolding): 程式開發中起輔助作用的代碼, 但不會留在最終程式中.

哨兵 (guardian): 一種編程模式, 使用條件語句進行檢驗, 並處理可能導致異常的情形.

6.11 習題集

Exercise 6.1. 針對以下代碼, 繪製堆疊圖, 查看程式輸出為何?

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Exercise 6.2. 阿克曼 *(Ackermann)* 函數 $A(m,n)$ 定義如下:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

參考 *http://en.wikipedia.org/wiki/Ackermann_function*. 編寫 `ack` 函數代碼, 實現 *Ackermann* 函數. 用此代碼執行 `ack(3, 4)`, 結果應該為 *125*. 同時, 換較大的 `m` 和 `n`, 看看結果有何不同? 答案參見: *https://thinkpython.com/code/ackermann.py*.

Exercise 6.3. 像 "*noon*" 和 "*redivider*" 一樣, 正序和倒序拼寫方式完全一樣的詞, 稱為回文詞. 從遞歸角度看, 如果開始和結束字母相同, 同時中間部分是回文詞, 那麼就可以認為總體是回文詞. 下文為返回字符串首字母, 尾字母以及中間字母的函數:

```
def first(word):
    return word[0]
```

```
def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

在第 8節會詳細解釋其原理.

1. 將這些函數代碼, 寫入檔案 palindrome.py, 並測試輸出. 用兩個字母, 測試 middle 函數, 看看會怎麼樣? 一個字母呢? 嘗試傳入不包含任何字母的空字符串'', 會如何?

2. 編寫is_palindrome 函數, 如果傳入參數為回文字符串, 則返回 True, 否則返回 False. 提示一下, 你可以使用內置函數 len 檢驗字符串長度.

答案參閱: *https://thinkpython.com/code/palindrome_soln.py*.

Exercise 6.4. 如果 a 可以被 b 整除, 同時 a/b 也是 b 的冪次方, 那麼 a 便是 b 是冪次方. 編寫函數is_power, 以 a 和 b 為參數, 如果 a 是 b 的冪次放, 則返回 True. 提示: 注意邊界條件.

Exercise 6.5. a 和 b 的最大公約數 (GCD), 是指能同時被整除的所有約數中的最大的一個.

尋找最大公約數的一種方法便是觀察, 如果 r 是 a 除以 b 的餘數, 那麼 $gcd(a,b) = gcd(b,r)$, 邊界條件是 $gcd(a,0) = a$.

編寫函數gcd, 以 a 和 b 為參數, 返回其最大公約數.

致謝: 此習題借鑑了 *Abelson* 和 *Sussman* 的 Structure and Interpretation of Computer Programs 中的範例.

第 7 章　疊代

本章主講疊代, 其主要實現重複執行一段語句. 在第 5.8節的遞歸, 便是一種疊代. 在第 4.2節的 `for` 迴圈, 也是一種疊代. 本章, 我們會接觸另一種疊代, `while` 語句. 但這裡要先講一下變數賦值.

7.1　重新賦值

你也許已經注意到, 相同的變數可以被多次賦值. 重新賦值會將已存在的變數指向新的值 (並且不再指向舊的值).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

第一次輸出 `x`, 值為 5, 第二次輸出, 值為 7.

圖 7.1 展示了堆疊圖中**重新賦值** 的過程.

在此, 我想澄清一下大家的困惑. 因為 Python 使用等號 (=) 進行賦值, 很容易將 `a = b` 這樣的語句, 作為數學命題中的等式進行解讀, 認為其表示 `a` 和 `b` 相等. 這種解讀是極其錯誤的.

首先, 等式是對稱關係, 而賦值不是. 例如, 數學中, 如果 $a = 7$, 那麼 $7 = a$. 但是在 Python 中, `a = 7` 正確, 但是 `7 = a` 則不然.

同樣, 在數學領域, 等式結果要麼真要麼假. 如果 $a = b$, 那麼 a 總是等於 b. 而在 Python 中, 雖然賦值語句可以使兩變數相等, 但是無法保證其一直相等:

```
>>> a = 5
>>> b = a    # a 和 b 相等
>>> a = 3    # a 和 b 不等
>>> b
5
```

第三行代碼, 改變了 `a` 的值, 但是沒有改變 `b`, 所以他們不再相等.

很多時候我們需要給變數重新賦值, 但是要謹慎使用. 如果變數的值變動過於頻繁, 代碼後續將難以理解, 同時除錯困難.

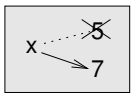

图 7.1: 堆疊圖

7.2 變數更新

最常見的一種重新賦值, 便是**變數更新**, 通常是基於前值進行修改而得到新值.

```
>>> x = x + 1
```

這句代碼表示 "獲取 `x` 的值, 然後加一, 得到新值, 繼而用新值更新 `x`."

如果你嘗試更新未定義變數, 會遇到錯誤, 因為 Python 會在賦值 `x` 前執行右側表達式:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

更新變數前, 一定要**初始化**, 通常採用簡單賦值來實現:

```
>>> x = 0
>>> x = x + 1
```

對變數進行加 1 來使其更新, 叫做**自增**; 執行減 1 更新變數, 叫做**自減**.

7.3 `while` 語句

計算機通常用於自動化一些重複性工作. 對於大量重複的相同或相似任務, 計算機可以永不犯錯, 這也是計算機精擅之處, 卻恰恰是人類最不擅長的. 在計算機編程中, 這種重複, 便稱作**疊代**.

我們已經遇到過兩個函數, `countdown` 和 `print_n`, 它們都是使用遞歸進行疊代. 因疊代操作很普遍, Python 提供了一些內置功能來便捷使用. 一個便是在第 4.2節見到的 `for` 語句, 後續再講.

另一個便是 `while` 語句, 下面是 `countdown` 的 `while` 語句版本:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

`while` 語句很容易理解, 因為其便如英語表達一樣. 意為, "當 `n` 大於 0 時, 打印 `n` 值, 然後自減 1. 直到當等於 0 時, 打印 `Blastoff!`"

正式些講, 下面是 `while` 語句的執行流程:

1. 確定條件之真假.
2. 如果為假, 退出 `while` 語句, 執行其後語句.
3. 如果為真, 運行執行體, 回到第一步.

這種程式流轉, 便叫做迴圈. 因為第三步驟時, 會返回到起點.

迴圈體通常會修改一個或多個變數的值, 從而令條件最終為假, 終止迴圈. 否則, 迴圈會一直重複, 成為**無限迴圈**. 對電腦科學家們, 有個樂此不疲的玩笑, 便是觀察洗髮水的使用說明, "起泡, 沖洗, 重複", 這就是個無限迴圈.

在 countdown 例子中, 我們如此保證迴圈終止: 如果 n 小於或等於 0, 迴圈不再運行. 由於 n 每迴圈一次, 就會變小, 終究會變成 0.

而有些迴圈, 則不太容易判斷, 比如:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n 是偶數
            n = n / 2
        else:                 # n 是奇數
            n = n*3 + 1
```

此迴圈的條件是 n != 1, 那麼只有 n 等於 1 時, 條件為假, 迴圈才終止.

每次迴圈, 程式都輸出 n 的值, 然後檢查是奇是偶. 如果是偶數, 則 n 除以 2. 如果是奇數, 則 n 值更新為 n*3 + 1. 例如, 如果 sequence 的參數為 3, n 值依次變為 3, 10, 5, 16, 8, 4, 2, 1.

由於 n 有時增加, 有時減少, 所以很難保證 n 會達到 1, 或者程式終止. 對於某些特殊的 n 值, 我們可以確信迴圈會終止. 比如, 如果起始值是 2 的冪, 那麼迴圈每次執行後 n 都會是偶數, 直到最終成為 1. 剛剛例子中得到的數列, 從 16 開始, 便是如此.

問題的難點在於, 是否可以證明對於*所有*正數的 n, 程式都會結束. 目前, 還沒有人能證明或者證否此命題! (參閱 http://en.wikipedia.org/wiki/Collatz_conjecture)

做個練習, 用疊代替換遞歸, 重寫第 5.8節的print_n 函數,

7.4 break 語句

有時, 想要出紅塵, 就需要先入紅塵, 只有進入迴圈體, 才知道何時應當終止迴圈. 這時, 我們可以用 break 語句跳出迴圈.

例如, 想要獲取用戶的輸入, 直到用戶輸入 done, 才跳出迴圈, 可以這樣寫:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

迴圈條件為 True, 便會一直迴圈, 所以只有觸及 break 語句, 才會跳出迴圈.

每次迴圈, 都會打印尖括號來提示用戶輸入, 如果用戶輸入了 done, break 語句會終止迴圈. 否則, 程式會打印用戶的輸入, 並進入下一輪迴圈. 舉個例子:

```
> not done
not done
> done
Done!
```

這種寫法在 `while` 迴圈中很常見, 因為你能夠隨時檢驗其條件 (而不僅僅在頭部驗證), 同時也可以主動地去終止迴圈 (“發生時停止”), 而不是被動地等待結束 (“執行到停止”).

7.5 平方根

在編程中, 迴圈通常用來進行數值計算, 通過用近似值逼近真實值來實現.

例如, 牛頓公式 (Newton's method) 便是一種計算平方根的方法. 若要計算 a 的平方根, 可以先確定一個任意的估計值, x, 然後通過下面公式, 可以得到一個更優的值:

$$y = \frac{x + a/x}{2}$$

比如, 如果 a 為 4, 且 x 為 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

結果很接近正確答案 ($\sqrt{4} = 2$). 如果我們用得到的近似值, 重複剛才的流程, 結果會更接近:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

多重複幾次, 結果便更準確:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

通常, 我們很難預知, 重複多少次, 才能得到正確結果. 但是, 我們知道, 一旦結果不再改變, 便是停止的時候:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

當 `y == x` 時, 我們便可以停止迴圈了. 下面的迴圈, 以估計值 `x` 開始並不斷逼近真實值, 在結果不再變化時終止:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

對於大部分的 a 值, 此方法都有效. 但涉及到浮點數等式, 便很麻煩. 浮點值一般認為是近似正確: 大部分的有理數, 比如 1/3, 以及類似 $\sqrt{2}$ 的無理數, 都無法用浮點數來精確表示.

與其費勁比較 x 和 y 是否相等, 不如用 abs 計算其差值的絕對值大小:

```
    if abs(y-x) < epsilon:
        break
```

在epsilon 為 0.0000001 時, 表達式為真, 那便說明兩個值已經足夠接近.

7.6 演算法

牛頓公式可以認為是一種**演算法**: 通過既定步驟來解決一類問題 (此例中為計算平方根).

要理解何為演算法, 先要明白什麼不屬於演算法. 當你學習乘法時, 往往需要記憶乘法表. 實際, 你記憶了 100 個特定答案. 這種知識不屬於演算法.

但如果你想 "偷懶", 你可能會發現一些小技巧. 比如, 你想計算 n 和 9 的乘積, 只需把 $n-1$, 作為第一個數字. $10-n$ 作為第二個數字即可. 這個小技巧對於任何數字乘以 9 都有效. 這便是演算法! 同樣地, 你學過的需進位的加法, 需借位的減法, 以及長除法, 都是演算法. 這些演算法的一個共性便是, 都無需費力思考. 它們都是機械的過程, 遵循簡單的規則, 一步步操作, 便可得到結果.

執行演算法固然無聊, 但其設計過程卻既有趣又挑戰智力, 同時也是電腦科學的核心.

有些事情, 對於人們來說很自然, 無難度, 且下意識便可完成, 而演算法卻很難解決. 比如, 理解自然語言. 我們所有人都能輕易做到, 但目前為止, 還沒有人能闡明我們是如何理解的, 更不用說用演算法的形式來解釋.

7.7 除錯

隨著程式代碼規模的增長, 你會發現需要耗費更多時間在除錯上. 一般更多的代碼, 也就意味著更多的出錯機會, 以及更多的潛在問題. 一種有效節約除錯時間的方法便是 "二分除錯". 比如, 有 100 行代碼, 一次檢查一行, 需要 100 次.

或者, 將程式對半切分. 找到程式的中間位置, 或者靠近中間的位置, 以便從中間開始檢查. 添加 print 語句 (或者其他可驗證效果的東西), 然後運行代碼.

如果中間檢查點出現異常, 那代碼前半部分有問題. 如果中間沒有出錯, 那麼問題便在後半部分了.

每次如此檢驗代碼, 將待檢驗的代碼行數不斷減半. 大約六步之後 (遠遠小於 100), 理論上, 便只剩下一兩行代碼需要檢查了.

實際上, 很難清晰界定 "代碼中間位置", 同時也很難在其位置檢驗. 所以沒必要計較於行數, 執著於中點. 反而, 你需要多思考代碼的哪些位置容易出錯, 哪些地方又容易驗證. 然後選擇一個恰到好處的點, 進行驗證.

7.8 術語表

重新賦值 (reassignment): 給已存在的變數賦新值的過程.

變數更新 (update): 基於前值, 計算新值, 更新變數的過程.

初始化 (initialization): 給變數賦初始值, 以待後續更新.

自增 (increment): 不斷增加某個值的變數更新 (通常為 1).

自減 (decrement): 不斷減少某個值的變數更新.

疊代 (iteration): 採用遞歸或者迴圈, 重複執行一段語句.

無限迴圈 (infinite loop): 終止條件一直不能抵達的迴圈.

演算法 (algorithm): 解決一類問題的通用步驟.

7.9 習題集

Exercise 7.1. 複製第 *7.5*節的迴圈, 封裝為`mysqrt` 函數, 令其以 `a` 為參數, 選擇一個合適的值 `x`, 返回 `a` 的近似平方根.

若要測試, 需要編寫一個名為`test_square_root` 的函數, 輸出下面的表格:

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

第一列是數值 *a*; 第二列是 `mysqrt` 函數計算出的 *a* 的平方根; 第三列是通過 `math.sqrt` 計算出的平方根; 第四列是兩者的差值絕對值.

Exercise 7.2. 內置函數 `eval` 以字符串為輸入, 並用 *Python* 解釋器執行. 如下:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

編寫函數`eval_loop`, 不停提示用戶, 獲取輸入並用 `eval` 執行, 同時打印結果.

直到用戶輸入 `'done'` 才終止, 同時返回最後一次表達式的結果.

Exercise 7.3. 數學家 *Srinivasa Ramanujan* 發現了一個無窮級數, 可以用來計算 $1/\pi$ 的近似值:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

編寫函數 `estimate_pi`, 使用上述公式計算 π 的近似值, 同時返回結果. 用 `while` 迴圈計算各個項並求和, 直到最後一項小於 `1e-15` (10^{-15} 的 *python* 表示法) 為止. 同時可以和 `math.pi` 比較結果, 檢驗效果.

答案參見: *https://thinkpython.com/code/pi.py*

第 8 章　字符串

字符串和整數, 浮點數以及布爾值不同. 一個字符串就是一個**序列**, 也就是說, 字符串是一組有序排列的值. 本章你將學習如何通過字符構造字符串, 以及字符操作相關方法.

8.1　字符串即序列

字符串便是字符系列. 通過中括號, 可以獲取其中的字符:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第二句表達式會從 `fruit` 中選擇在位置 1 處的字符, 並賦給 `letter`. 括號中的表達式, 叫做**索引**. 索引標識了你將從序列中獲取哪個字符 (類似名字).

但有時候你所得非所願:

```
>>> letter
'a'
```

如眾人所知, `'banana'` 的第一個字母是 `b`, 而不是 `a`. 但是, 對於電腦科學家來說, 索引是從字符串開始位置的偏移量, 所以第一個字符的偏移量是 0.

```
>>> letter = fruit[0]
>>> letter
'b'
```

所以 `b` 是`'banana'` 的第 0 個字母, `a` 是第 1 個, `n` 是第 2 個.

你可以使用包括變數和操作符的表達式作為索引:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

但索引的值必須是整數. 否則, 會報錯:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len 函數

len 是內置函數, 可以返回字符串中的字符數量:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

想要獲得字符串最後一個字符, 可以嘗試下面的操作:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

報出 IndexError 的原因是'banana' 中索引 6 的位置沒有字母. 因為索引從 0 開始, 那 6 個字母對應的數字是 0 至 5. 想獲取最後一個字符, 需要字符串長度減 1:

```
>>> last = fruit[length-1]
>>> last
'a'
```

你也可以使用負數索引, 由尾至頭統計. 表達式 fruit[-1] 給出最後一個字母, fruit[-2] 給出了倒數第二個字母, 以此類推.

8.3 for 迴圈

很多操作一次僅操作字符串中的一個字符. 一般從頭部開始, 順序逐個獲取字符, 做些操作, 直到末尾結束. 這種處理模式叫做**遍歷**. 一般可以用 while 迴圈進行遍歷:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

上面迴圈會遍歷字符串, 並每行打印一個字符. 迴圈條件是 index < len(fruit), 所以當 index 等於字符串長度時, 條件為假, 迴圈體終止. 最後獲取的字符, 索引為 len(fruit)-1, 也就是最後一個字符.

做個練習, 寫個函數, 以字符串為入參, 倒序輸出每個字符, 每行一個.

遍歷字符串的另一種方法是用 for 迴圈:

```
for letter in fruit:
    print(letter)
```

每次迴圈後, 字符串中下一個字符會賦值給變數 letter. 直到沒有字符可用, 迴圈便終止.

下面的例子, 展示了如何用拼接 (字符串相加) 以及 for 迴圈, 來構建一個簡單序列 (按照字母順序). 在 Robert McCloskey 的書 *Make Way for Ducklings* 中, 小鴨子們的名字分別是 Jack, Kack, Lack, Mack, Nack, Ouack, Pack, 和 Quack. 迴圈依次輸出名字:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

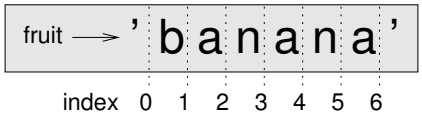

图 8.1: 切片的索引.

輸出如下:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

顯然, 上面結果並不是全部正確, 因为 “Ouack” 和 “Quack” 都拼錯了. 做個練習, 修改程式, 使其正常.

8.4 字符串切片

字符串的一個片段, 叫做**切片**. 選擇切片和從字符串中選擇字符很像:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

操作符 `[n:m]`, 會返回字符串從第 n 個位置到第 m 個位置的字符, 包括開頭位置的字符, 但是不包括最後位置的字符. 這有點違反直覺, 但若將索引想像為指向字符**中間**, 如圖 8.1所示, 可能會容易理解.

如果缺失第一個索引 (冒號前), 則切片從字符串頭部開始. 如果忽略了第二個索引, 切片到末尾結束:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

若第一個索引值大於或等於第二個, 則結果為**空字符串**, 用兩個單引號表示:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

空字符串一般不包括任何字符, 同時長度為 0, 除此之外, 和其他字符串一樣.

繼續上面的例子, 思考一下 `fruit[:]` 表示什麼? 試試吧.

8.5 字符串不可變

嘗試在賦值等式左側用 [] 操作符修改字符串中的字符. 例如:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

這裡的 “object’’ 指字符串, “item’’ 指試圖賦值的字符. 到目前為止, 你可以認為對像和值一樣, 但是後續 (在第 10.10節), 我們會完善此定義.

錯誤的原因在於字符串是**不可變的**, 也就是說, 你無法改變一個既有的字符串. 你能做的, 是基於原來字符串做些操作, 創建一個不同的字符串:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

上述代碼是用新的首字母和 greeting 的切片進行了拼接, 而這並不會改變原來的字符串.

8.6 檢索

下面的函數什麼用途?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

可以認為, find 是 [] 操作符的逆運算. 此操作不同於根據索引獲取對應字符, 而是根據字符, 查找其索引. 如果字符沒有檢索到, 則返回-1.

這是我們第一次見到, 在迴圈內使用 return 語句. 如果 word[index] == letter, 函數會跳出迴圈, 並立刻返回結果.

如果字符串中沒有想要的字符, 程式會一直執行到迴圈結束, 並返回-1.

這種演算法--遍歷序列並返回預期結果--叫做**檢索**.

做個練習, 為 find 函數加入第三個參數, 一個索引值, 使其從 word 的此索引處開始檢索.

8.7 迴圈和計數

以下程式會統計字符串中 a 出現的次數:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

此程式描述了另一種演算法, 稱之為**計數器**. 初始化變數 count 為 0, 其後每找到一次 a, 就加 1. 迴圈結束後, count 便包含了結果– a 的全部數量.

做個練習, 封裝以上代碼為 count 函數, 使其以字符串和字母為參數, 從而更加通用.

然後重寫函數, 用上一節的 find 函數的三參數版本, 替換字符串遍歷操作.

8.8 字符串方法

字符串提供了諸多有用的方法, 方法類似於函數--輸入參數並返回結果--但是語法有些不同. 例如, upper 方法, 接收字符串, 並返回一個全部字母大寫後的字符串.

與函數的語法 upper(word) 不同之處在於, 方法的語法寫作 word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

這種點標法, 聲明了方法的名字, upper, 和要使用此方法的字符串的名字, word. 括號為空表明此方法沒有參數.

令函數運行, 叫做**調用**; 此例中, 我們說調用 word 的 upper 方法.

你會發現, 字符串實際內置了 find 方法, 但和我們寫的函數驚人相似:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

此例中, 我們調用 word 的 find 方法, 將要查找的字母作為入參.

實際上, find 方法比我們寫的函數要更通用; 它不僅可以定位字符, 也能定位字符串片段:

```
>>> word.find('na')
2
```

默認情況, find 從字符串開頭進行查找, 不過也可以給其傳入索引值作為第二個參數, 使其從既定位置開始:

```
>>> word.find('na', 3)
4
```

這是一個**可選參數**的例子; find 方法也可以接收第三個索引參數, 以標識結束位置:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

b 沒有出現在字符串索引 1 和 2 且不包括 2 的範圍內, 所以搜索失敗. 而這種到達第二個索引位但不包括此索引的規則, 和切片操作一樣.

8.9 操作符 in

單詞 in 是一個布爾運算子, 其比較兩個字符串, 如果前者是後者的一部分, 則返回 True:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

例如, 下面的函數會輸出同時出現在 word1 和 word2 中的字母:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

變數名選擇得足夠好的話, Python 讀起來便如同英語. 讀一下這個迴圈, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

下面是 apples 和 oranges 比較的結果:

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 字符串比較

關係運算子同樣適用於字符串. 比如要判斷兩字符串是否相等:

```
if word == 'banana':
    print('All right, bananas.')
```

其他的關係運算子也適用於按照字母順序進行比較:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python 處理大小寫的方式同人類思維不同, 在其看來, 大寫字母都排在小寫字母之前, 所以:

Pineapple 在前, banana 在後.

解決此問題的通常做法是, 在比較前先統一字符串的格式, 比如都轉小寫. 謹記這一點, 以免遇到 Pineapple 時, 變得一團糟.

8.11 除錯

想要用索引來遍歷序列中的值, 難點在於確定遍歷的起點和終點. 下面是一個比較單詞的函數, 如果一個單詞恰好是另一個單詞的倒序排列, 則返回 True, 但這裡有兩處錯誤:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
```

```
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

第一個 if 語句會判斷兩個單詞的長度是否一樣. 如果不同, 立刻返回 False. 然而為了執行後續代碼, 我們先假設單詞長度相同. 這是一個哨兵模式, 在第 6.8節已經介紹過.

i 和 j 是索引: i 正向遍歷 word1, 同時 j 倒序遍歷 word2. 如果遇到兩個字母不同, 則即刻返回 False. 如果可以通過迴圈檢驗, 則所有字母都匹配, 返回 True.

如果用 “pots” 和 “stop” 測試函數, 我們期望得到的時 True, 但是遇到了索引錯誤:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

除錯此種異常, 第一步就是在錯誤出現位置前, 先輸出索引的值.

```
    while j > 0:
        print(i, j)         # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

再次運行程式, 得到如下信息:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

首次迴圈, j 的值是 4, 超出了'pots' 的索引範圍. 最後一個字符的索引應該是 3, 所以 j 的初始值應該是 len(word2)-1.

如果修復此錯誤, 並再次執行, 輸出如下:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

這次我們的到了正確結果, 但是迴圈只運行了三次, 有點奇怪. 為了弄清怎麼回事, 可以繪製堆疊圖來輔助理解. 第一次疊代, is_reverse 的狀態框見圖 8.2.

我嘗試在框中對齊變數, 並用虛線標識 i 和 j 的值, 用來標識 word1 和 word2 中的字符, 從而幫助理解.

從此狀態框開始, 在紙上執行程式, 每次疊代時修改 i 和 j 值. 然後找到並修復函數中的第二個錯誤.

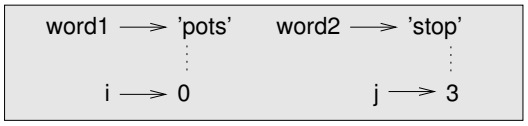

图 8.2: 堆疊圖.

8.12 術語表

對象 (object): 變數引用的東西, 目前, 可以將 "對象" 與 "值" 同樣看待.

序列 (sequence): 一些值的有序集合, 每個值都對應一個整數索引.

元素 (item): 序列中的一個值.

索引 (index): 一個整數值, 用來選擇序列中的元素, 比如選擇字符串中某個字符. Python 中的索引都是從 0 開始.

切片 (slice): 字符串中一部分, 通過索引範圍確定.

空字符串 (empty string): 沒有字符並且長度為 0, 同時用兩個引號表示的字符串.

不可變 (immutable): 序列中元素不可改變的特性.

遍歷 (traverse): 疊代序列中每個元素, 同時對其執行相似操作的過程.

檢索 (search): 找到預期目標才停止的遍歷模式.

計數器 (counter): 用來計數的變數, 一般始於 0, 不斷遞增.

調用 (invocation): 運行方法的語句.

可選參數 (optional argument): 函數或者方法中的不必要參數.

8.13 習題集

Exercise 8.1. 閱讀文檔*http://docs.python.org/3/library/stdtypes.html#string-methods* 中的字符串方法, 可能你會想試試其中一些方法, 盡量弄明白它們的工作原理. `strip` 和 `replace` 特別有用.

文檔中的某種語法可能難以理解. 比如`find(sub[, start[, end]])` 方法, 方括號標識了可選參數. `sub` 是必需的, 但是 `start` 是可選的, 如果包含了 `start`, `end` 便是可選的.

Exercise 8.2. 有個叫 `count` 的字符串方法, 和第 *8.7* 節的函數很相似. 閱讀此方法文檔, 編寫調用此方法的代碼, 實現對`'banana'` 中 `a` 的個數的統計.

Exercise 8.3. 字符串切片也可以有第三個參數, 叫做 "步長"; 也就是, 在連續字符中, 字符間的間距. 步長為 *2* 表示每隔一個字符取一個; 步長為 *3* 表示每第三個取一個, 以此類推.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

步長為*-1*, 則表示倒序讀取, 所以 `[::-1]` 切片, 便會產生一個倒序的字符串.

用這個神奇魔法, 將習題 *6.3*中的 `is_palindrome` 修改為一行代碼的版本吧.

Exercise 8.4. 下面的函數都是試圖檢驗字符串中是否包含小寫字母, 但是肯定有函數存在問題. 仔細分析每個函數並明了其用途(假設入參都是字符串).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercise 8.5. 凱撒加密 *(Caesar cypher)* 是一種通過對每個字母進行特定數值的 ''移位'' 操作, 而實現的簡單加密方案. 對字母移位, 也就是按照字母順序, 進行移動, 必要時需要回到開頭, 所以*'A'* 移位 *3*, 得到*'D'* , *'Z'* 移位 *1*, 得到*'A'*.

對一個單詞移位, 就是對每個字母採用相同的移位數量. 例如 *"cheer"* 移位 *7*, 則為 *"jolly"*, *"melon''* 移位*-10*, 則為 *"cubed"*. 在電影 2001: A Space Odyssey 中, 飛船上的計算機名叫 *HAL*, 就是 *IBM* 移位*-1* 得到的.

編寫rotate_word 函數, 接收一個字符串和一個整數, 作為參數, 對字符串中的字符進行數值移位, 得到新字符串, 並返回.

你可能需要用內置的 ord 函數, 此函數可以將字符轉為數字碼, 而 chr 則可以將數字碼轉回字符. 字母表中的字母會按順序進行編碼, 比如:

```
>>> ord('c') - ord('a')
2
```

因為'c' 在字母表中是第 *2* 個 *(*從 *0* 開始*)* 字母. 但要注意: 大寫字母的數字碼和小寫的不同.

網絡上一些嘲弄有時會採用 *ROT13* 編碼, 也就是移位數量為 *13* 的凱撒加密. 如果你不會太介意, 試試解密它們吧. 參閱: *https://thinkpython.com/code/rotate.py*.

第 9 章　案例學習: 單詞遊戲

本章學習第二個案例, 主要研究如何通過搜索特定詞彙, 進行猜謎. 比如, 查找最長回字文, 以及尋找按照字母表順序排列的單詞. 同時, 我將介紹一種新的程式開發模式: 抽離紛紛擾擾, 回歸已有方案.

9.1　讀取單詞串列

本章的練習, 需要準備一個英文單詞串列. 網上有很多可用的單詞串列, 但是對我們來說, 最理想的莫過於 Grady Ward 收集整理, 作為 Moby 詞典專案, 貢獻給公共領域的單詞串列 (詳見`http://wikipedia.org/wiki/Moby_Project`). 這是包含 113,809 個填詞遊戲的單詞串列; 也就是說, 這些單詞, 已經被填詞遊戲和其他單詞遊戲證明了有效. 在 Moby 專案中, 這個檔案名為 `113809of.fic`; 你可以從`https://thinkpython.com/code/words.txt` 下載一個副本, 其名字簡稱 `words.txt`.

此檔案為純文本檔案, 你可以用文本編輯器打開, 但你也可以用 Python 讀取. 內置函數 `open`, 以檔案名為入參, 返回一個**檔案對象**, 可以用來讀取檔案內容.

```
>>> fin = open('words.txt')
```

`fin` 是表示輸入的檔案對象的通用名稱. 檔案對象針對讀取提供了多個方法, 包括 `readline`, 此方法會讀取檔案的一整行字符, 並作為字符串返回:

```
>>> fin.readline()
'aa\n'
```

單詞串列中第一個單詞是 “aa”, 這是一種岩漿. 後面的`\n` 是換行符, 用來斷行.

檔案對象會跟蹤目前讀到哪裡了, 從而, 再次運行 `readline`, 會得到後面的單詞:

```
>>> fin.readline()
'aah\n'
```

下一個單詞是 “aah”, 這是個絕對正確的單詞, 所以別那樣看我. 另外, 如果換行符令你厭煩, 可以用字符串方法 `strip` 移除:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

你也可以將檔案對象置於 `for` 迴圈中. 這樣程式便會讀取 `words.txt`, 然後逐行輸出每個單詞:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 練習

下一節有這些習題的答案, 但盡量在看答案之前盡力一試吧.

Exercise 9.1. 編寫程式, 讀取 words.txt, 僅打印多於 *20* 個字符的單詞 (不包括空格).

Exercise 9.2. *1939* 年 *Ernest Vincent Wright* 出版了一部 *50,000* 單詞的小說, 名叫 Gadsby, 本書不包括字母 "*e*". 而英文中最常用的便是 "*e*", 所以太難得了.

事實上, 若不使用常用的字符, 一般很難表達出一個觀點. 不過, 開始雖然進展緩慢, 但是, 通過保持謹慎並訓練幾個小時, 你也可以慢慢適應.

好了, 閒言少敘.

編寫 has_no_e 函數, 如果輸入的單詞不包括 "*e*", 則返回 True.

編寫程式, 讀取 words.txt, 只打印不包含 "*e*" 的單詞. 統計串列中, 不包含 "*e*" 的單詞所占比例.

Exercise 9.3. 編寫 avoids 函數, 以單詞和禁用字母字符串為輸入, 當單詞不包含任何禁用字母時, 返回 True.

編寫程式, 使用戶輸入禁用字母字符串, 然後輸出不含有這些字母的單詞數量. 看看你是否可以找出一個包含 *5* 個禁用字母的組合, 使被排除的單詞數最少?

Exercise 9.4. 編寫函數uses_only , 以一個單詞和一串字母為輸入, 如果單詞的字母都在這串字母中, 則返回 True. 你是否可以只用 acefhlo 這些字母, 構造出句子? 換成 "*Hoe alfalfa*" 這些字母呢?

Exercise 9.5. 編寫函數uses_all , 輸入為一個單詞和一串必需字母, 如果單詞對這串必需字母, 都至少使用了一次, 則返回 True. 看看有多少單詞同時包含 aeiou? 又有多少同時包含 aeiouy 呢?

Exercise 9.6. 編寫函數is_abecedarian, 如果單詞中的字母是按照字母表順序排列, 則返回 True(字母相同, 視為順序). 看看有多少這種單詞?

9.3 檢索

上一章節中, 所有的練習, 都有個共通之處; 它們都可以採用第 8.6節的檢索方法來解決. 舉個簡單例子:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

for 迴圈會遍歷 word 中所有字母. 如果遇到 "e", 則即刻返回 False; 否則繼續下一個字母. 如果迴圈正常結束, 也就是說沒有遇到 "e", 則返回 True.

你也可以使用 in 運算子來精簡程式, 我先介紹上述版本, 主要是闡述清楚檢索演算法的內在邏輯.

avoids 函數相比 has_no_e 版本, 功能更加通用, 但結構相同:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

在此函數中, 一旦遇到禁止字母, 即刻返回 `False`, 如果迴圈終了, 則返回 `True`.

`uses_only` 與之極為相似, 無非條件相反:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

這裡不再是禁用字母串列了, 而是可用字母串列. 如果 `word` 中出現了不在可用串列中的字母, 返回 `False`.

`uses_all` 和上述函數也較為相似, 不同之處在於, 我們交換了單詞和字母序列的角色:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

不再是遍歷 `word` 中的字母, 而是遍歷必需字母串列. 如果字母串列中有字母未出現在單詞中, 返回 `False`.

如若你已像電腦科學家一樣去思考, 你便會注意到 `uses_all` 是以前一個已解決的問題的另一種表達, 你便會這麼寫:

```
def uses_all(word, required):
    return uses_only(required, word)
```

此案例便是一個**抽離紛擾, 回歸已知**的程式開發模式的實踐, 這種模式是將遇到的問題, 映射為已解決的問題, 從而用已有的解決方案來解決當前問題.

9.4 索引迴圈

前一章節中, 我用 `for` 迴圈編寫了大量函數, 這是因為我只關注字符串中的字符; 所以無需關注索引.

但`is_abecedarian` 函數中, 我們不得不比較相鄰的字母, 用 `for` 迴圈便不太方便:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

用遞歸來實現:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

還可以使用 while 迴圈:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

此迴圈始於 i=0, 終於 i=len(word)-1. 每次迴圈都會比較第 i 個字符 (可以看作當前字符) 和第 $i+1$ 個字符 (可以看作後一個字符).

如果後一個字符小於 (在字母表順序中先於) 當前字符, 我們會發現這和要求不符, 便返回 False.

如果直到迴圈末尾, 仍然無異常, 則單詞合格. 為確信迴圈正常結束, 可以試試'flossy'. 單詞長度為 6, 所以 i 為 4 時, 便是最後一次迴圈, 因為這是倒數第二個字符的索引. 最後一次迴圈中, 比較了倒數第二個和倒數第一個字符, 這恰恰符合預期.

下面是is_palindrome 函數 (參看習題 6.3) 的另一個版本, 其使用了兩個索引; 一個從開頭到結尾, 順序前進; 另一個從結尾到開頭, 倒序進行.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

或者, 我們映射到以前解決過的難題, 進行重寫:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

此處使用了第 8.11節的is_reverse 函數.

9.5 除錯

程式測試很難. 本章的函數相對容易測試, 因為你可以用手算來檢驗結果. 即便如此, 要選擇一批單詞, 測試所有可能錯誤, 那便難上加難了.

以has_no_e 為例, 需要檢驗兩種情景: 一種是含有 'e' 的單詞, 返回 False, 另一種是不含有 'e' 的單詞, 返回 True. 對此, 你應該很容易針對每種情景, 都想出一個單詞.

對於每種情景, 存在一些不太明顯的子用例. 對於包含 "e'' 的單詞, 你需要檢驗 "e'' 在開頭的, 在結尾的, 甚至中間某些位置的單詞. 你要檢驗長單詞, 短單詞, 甚至非常短的單詞, 比如空字符串. 空字符串是**特例**的一種, 也就是那種容易被忽視, 卻往往是錯誤頻發之處的用例.

除了自己構建測試用例, 你也可以用類似 `words.txt` 的單詞串列檢驗程式. 通過掃描輸出, 你可能會發現錯誤, 但要注意: 你可能發現了某種錯誤 (包括了不應包含的單詞), 而忽略了其他錯誤 (沒有包括應該包含的單詞).

通常, 程式測試可以幫助你發現錯誤, 但要製作出好的測試用例, 往往很難, 即使通過了這些測試, 你也不能百分百地確認你的程式正確. 一位傳奇的電腦科學家說過:

> 程式測試只能表明錯誤的存在, 卻永遠無法保證其不存在!
>
> — Edsger W. Dijkstra

9.6 術語表

檔案對象 (file object): 用來標識被打開的檔案的一個值.

抽離紛繞, 回歸已知 (reduction to a previously solved problem): 一種將當前問題簡化為已解決問題, 進而解決問題的方式.

特例 (special case): 非典型或者不明顯的測試用例 (往往容易犯錯的地方).

9.7 習題集

Exercise 9.7. 這個問題源於廣播節目 Car Talk 中的一個謎題 (*http://www.cartalk.com/content/puzzlers*):

> 給我一個存在三個連續雙字母的單詞. 我會給你一些看似符合, 其實不符的單詞. 例如, 單詞 *committee*, *c-o-m-m-i-t-t-e-e*. 如果沒有 '*i*' 在中間, 就很完美. 又或者 *Mississippi: M-i-s-s-i-s-s-i-p-p-i*. 如果將其中的 *i* 都移除, 便符合了. 但是有一個詞恰好含有三個連續雙字母, 而且據我所知, 這可能是僅有的一個這樣的單詞. 當然, 實際上有可能有 *500* 多個這樣的單詞, 但我只能想到一個. 是哪個單詞呢?

編寫程式來尋找一下吧. 答案: *https://thinkpython.com/code/cartalk1.py*.

Exercise 9.8. 這也是一個 Car Talk 的謎題 (*http://www.cartalk.com/content/puzzlers*):

> ''某天我在高速上開車, 碰巧注意到里程表. 同多數里程表一樣, 它有 *6* 個數字, 只能表示整裡數. 所以, 如果我的車跑了 *300,000* 英里, 那顯示的就是 *3-0-0-0-0-0*.
>
> ''現在, 我看到的很有意思. 最後 *4* 個數字是回文; 也就是說, 從前往後讀和從後往前讀都一樣. 比如 *5-4-4-5* 便是個回文, 所以我的里程表可能顯示的是 *3-1-5-4-4-5*.
>
> ''一英里後, 最後的 *5* 個數字也是回文. 例如顯示為 *3-6-5-4-5-6*. 然後又跑了 *1* 英里, *6* 個數字中間的 *4* 位是回文了. 準備好玩這個了嗎? 那又跑了 *1* 英里, 所有 *6* 個數字也是回文了!
>
> ''問題來了, 我開始在里程表上看到的數字是什麼?''

編寫個 *Python* 程式, 檢驗所有的 *6* 位數字, 然後輸出滿足上述要求的任意數字. 答案: *https://thinkpython.com/code/cartalk2.py*.

Exercise 9.9. 再來一個 Car Talk 上的謎題, 你可以用檢索法來解決 (*http://www.cartalk.com/content/puzzlers*):

> ‘‘最近我去看望母親, 我們發現我的年齡倒過來, 正好是母親的年齡. 比如, 若她是 73, 我是 37. 我們想知道在過去這些年, 有多少次這種情況發生, 但後來我們岔開了話題, 沒有得到答案.
>
> ‘‘回到家後, 我發現到目前為止, 我們的年齡已經互逆了 6 次. 同時, 我也意識到, 如果我們幸運的話, 稍後幾年, 我們又會年齡互逆一次, 如果我們特別幸運, 就還會有一次機會. 換句話說, 這樣的情況, 可能我們一共會遇到 8 次. 所以, 請問, 我現在多少歲了?”

編寫 *Python* 程式, 尋找這個謎題的答案. 提示: 你可能會用到字符串的 `zfill` 方法.

答案: *https://thinkpython.com/code/cartalk3.py*.

第 10 章　串列

本章講述 Python 最有用的內置型態, 串列. 同時你會深入了解對象, 學習一個對象對應多個名稱時會產生的現象.

10.1　串列即序列

同字符串一樣, **串列** 也是值的序列. 字符串中, 值是字符; 在串列中, 值可以是任何型態. 串列中的值, 叫做**元素**, 有時也叫**串列項**.

創建串列的方式很多; 最簡單的莫過於將元素用方括號包起來 ([and]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

第一個例子是由四個整數構成的串列, 第二個示例串列則由三個字符串構成. 串列中的元素型態可以不同. 下面的串列, 便同時包括了字符串, 浮點數, 整數以及另一個串列:

```
['spam', 2.0, 5, [10, 20]]
```

串列中包含串列, 叫做**嵌套串列**.

不包含任何元素的串列, 叫做空串列; 你可以用[] 創建空串列.

如你所想, 串列也可以賦給變數:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10.2　串列可變

從串列中獲取元素的語法和從字符串中獲取字符的語法一樣--方括號操作符. 括號中的表達式確定了索引值. 要注意, 索引從 0 開始:

```
>>> cheeses[0]
'Cheddar'
```

和字符串不同, 串列是可變的, 當括號操作符出現在賦值語句左側, 就會將對應的串列元素重新賦值.

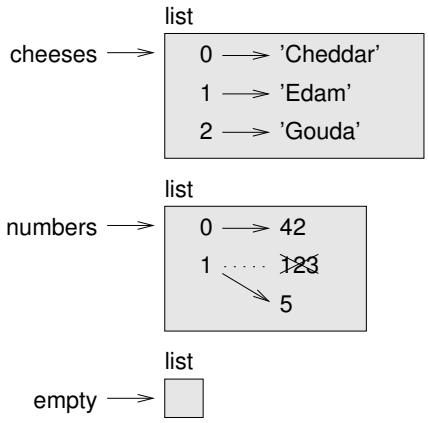

图 10.1: 狀態圖.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

numbers 中的第一個元素, 原來是 123, 現在變成了 5.

圖 10.1 是 cheeses, numbers 和 empty 的狀態圖.

串列用單詞 "list" 在外, 元素在內的箱體圖表示. cheeses 指向了一個包括三個元素的串列, 其索引分別為 0,1 和 2. numbers 包括兩個元素; 圖中也展現了第二個元素從 123 被賦值為 5 的過程. empty 指向了一個空串列.

串列索引和字符串索引的作用是一樣的:

- 索引可以是任意整型表達式.
- 如果試圖通過索引讀寫不存在的元素, 會得到 IndexError.
- 如果索引為負值, 則從串列末尾倒序計數.

運算子 in 也可作用於串列.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 遍歷串列

遍歷串列中元素的最常用方法便是 for 迴圈. 語法和字符串遍歷相同:

```
for cheese in cheeses:
    print(cheese)
```

這種方法對於僅讀取串列中元素很方便, 但是如果你想寫入或者更新元素, 那便需要索引了. 通常把內置函數 range 和 len 結合使用:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

此迴圈會遍歷串列並更新每個元素. len 方法會返回串列中元素數量. range 則返回從 0 到 $n-1$ 的索引值串列, 其中 n 是串列的長度. 每次迴圈, i 都會獲得下一個元素的索引. 迴圈體中的賦值語句則會通過 i 獲取元素的舊值, 並賦予新值.

用 for 迴圈遍歷空串列, 永遠不會執行迴圈體:

```
for x in []:
    print('This never happens.')
```

雖然串列可以包含其他串列, 但是嵌入的串列仍表示一個元素. 下面串列的長度為 4:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 串列操作

運算子 + 可以用來拼接串列:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

運算子 * 會將串列複製給定的次數:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

第一個示例複製了 [0] 四次. 第二個, 則複製了 [1, 2, 3] 三次.

10.5 串列切片

切片運算子同樣適用於串列:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

如果省略第一位索引, 則從頭開始, 進行切片. 如果省略第二位索引, 則切片止於末尾. 如果兩位都省略, 則切片便等同於複製整個串列.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

因為串列可變, 所以最好在執行修改串列的操作前, 複製一份, 以防萬一.

如果切片運算子在賦值號左側, 便可以同時更新多個元素:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 串列方法

Python 提供了大量操作串列的方法. 例如, append 可以在串列末尾添加新元素:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

extend 會以串列為參數, 並將其中所有元素添加到執行串列中:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

這個例子中, t2 沒有被修改.

sort 會將串列元素, 從低到高排序:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

多數串列方法都沒有返回值; 這些方法修改串列然後返回 None. 如果你意外寫了 t = t.sort(), 結果會令你失望.

10.7 Map, filter 和 reduce

若想對串列中所有數字求和, 可以用如下迴圈實現:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

total 初始為 0. 每迴圈一次 x 會從串列中獲取一個元素. += 操作符, 是一種更新運算的便捷寫法. 也是一種**增量賦值語句**.

```
    total += x
```

等同於

```
    total = total + x
```

隨著迴圈運行, total 會累積元素求和; 這樣的變數有時也稱為**累加器**. 對串列元素求和, 使用比較普遍, 所以 Python 提供了內置函數 sum:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

像這樣將一系列元素合為一個值的操作, 一般稱為 **reduce**.

有時, 你會遍歷串列, 創建新串列. 例如, 下述函數會接收字符串串列, 並返回一個將所有字符串都轉為首字母大寫的字符串串列:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res 開始是一個空串列; 每次迴圈, 都會向其中添加下一個元素. 所以 res 也可認為是另一種累加器.

類似capitalize_all 這樣的操作, 一般叫做 **map**, 因為此操作會將函數 (此處為 capitalize)"應用" 到序列中每個元素上.

另一種常用操作是, 從串列中篩選部分元素, 返回子串列. 例如, 下面函數會接收一個字符串串列, 返回僅包含大寫字母的字符串串列:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper 是字符串方法, 如果字符串中字母全部為大寫, 則返回 True.

如only_upper 一樣的操作, 叫做 **filter**, 因為此操作會篩選出某些元素, 過濾掉其他元素.

大部分的串列操作都可以由 map, filter 以及 reduce 組合構成.

10.8 移除元素

從串列中移除元素有多種方法. 如果你知道所要移除元素的索引, 可以用 pop 方法:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

pop 會改變串列, 並返回被移除的元素. 如果未提供索引, 則會刪除並返回最後一個元素.

如果你不使用移除的元素, 則可以用 del 元算符:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

如果你知道要刪除元素的值 (但不知道索引位置), 你可以用 remove 方法:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

remove 方法的返回值是 None.

如果要移除多個元素, 可以用 del 配合切片索引實現:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

通常, 切片含頭不含尾, 即到第二個索引值（不包括）為止.

10.9 串列和字符串

字符串是一系列的字符, 串列是一系列的值, 但是字符串列和字符串並不可等同視之. 採用 list 方法, 可以將字符串轉換為字符串列:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

因 list 是一個內置函數的名字, 所以盡量避免將其作為變數名. 我通常也不建議使用 l , 因為和 1 太像了. 這便是我用 t 作為變數名的緣由.

list 函數可以將字符串拆解為單獨的字母. 如果你想將字符串分拆為一個個單詞, 則可以使用 split 方法:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

此處的可選參數是**分隔符**, 也就是定義單詞邊界的字符. 下面的示例使用連字符作為分隔符:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

join 和 split 的作用正好相反. 它會接收字符串串列, 然後拼接所有元素. join 是一個字符串方法, 所以你需要在分隔符上調用此方法, 然後將串列作為參數傳入:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

此例中, 分隔符是一個空格, join 會在每個單詞之間放置一個空格. 若想不用空格拼接, 則可以用空字符串'' 作為分隔符.

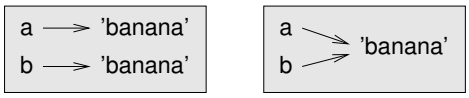

图 10.2: 狀態圖.

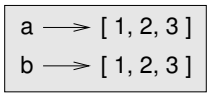

图 10.3: 狀態圖.

10.10 對象和值

如果運行下面的賦值語句:

```
a = 'banana'
b = 'banana'
```

我們知道 `a` 和 `b` 都指向了字符串, 但我們不知道, 他們是否指向了*相同的* 字符串. 所以, 便有了兩種可能狀態, 如圖 10.2.

第一種情況, `a` 和 `b` 指向了兩個不同對象, 這兩個對象有相同的值. 第二種情況, 它們指向了同一個對象.

若想判斷兩個變數是否指向了同一個對象, 可以用 `is` 運算子.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

此例中, Python 僅創建了一個字符串對象, 然後 `a` 和 `b` 都指向了此對象. 但是當你創建兩個串列時, 你得到的就是兩個對象:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

此時, 狀態圖便是圖 10.3的樣子.

這種情況下, 可以說這兩個串列是**相等的**, 因為它們有相同的元素, 但它們不是**相同的**, 因為不是同一個對象. 兩個對象如果相同, 那麼它們必然相等, 但是它們如果相等, 卻未必相同.

到目前為止, 我們一直混用 ‘‘對象’’ 和 ‘‘值’’, 但更準確地說, 一個對象有一個值. 若創建 `[1, 2, 3]`, 你會得到一個串列對象, 而這個對象的值是個整數序列. 如果另外一個串列也有相同的元素, 我們便說它們擁有相同的值, 但它們不是同一個對象.

10.11 別稱

如果 `a` 指向一個對象, 同時令 `b = a`, 那麼, 兩個變數都指向了同一個對象:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

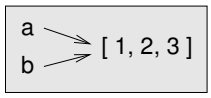

图 10.4: 狀態圖.

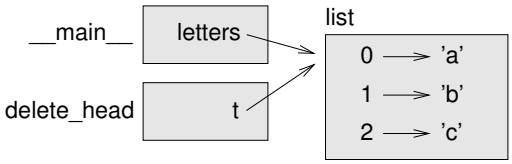

图 10.5: 堆疊圖.

此時狀態圖如圖 10.4所示.

變數和對像的這種關聯, 叫做**引用**. 上例中, 兩個引用指向了同一個對象.

有多個引用的對象, 也便有了多個名稱, 如此, 我們可以說, 對象是有**別稱的**.

如果一個有別稱的對象是可變的, 那麼一個別稱所做的修改會影響另一個:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

雖然此特性很有用, 但也很容易出錯. 通常, 對於可變對象, 避免使用別稱, 會安全很多.

而對於像字符串這樣的不可變對象, 別稱使用往往不是問題. 如下所示:

```
a = 'banana'
b = 'banana'
```

所以 `a` 和 `b` 是否指向同一個對象, 已無關緊要.

10.12 串列參數

當給函數傳遞串列時, 函數收到的是對該串列的引用. 如果函數修改了串列, 那麼調用一方也會看到相應變化. 例如, `delete_head` 函數刪除了串列中第一個元素:

```
def delete_head(t):
    del t[0]
```

下面是一個應用:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

形參 `t` 和變數 `letters` 是同一個變數的別稱. 堆疊圖如圖 10.5所示.

因兩個框共用一個串列, 所以我把串列畫在了它們中間.

區分修改串列操作和新建串列操作, 是非常重要的. 例如, `append` 方法是修改串列, 而 `+` 運算子則會新建串列.

下面是 `append` 的一個使用示例:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append 的返回值是 None.

下面是使用 + 運算子的示例:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

操作結果是個新串列, 原串列也沒有變化.

在為修改串列而編寫函數時, 此差異尤要重視. 例如, 下面函數沒有刪除串列的第一個元素:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

此切片運算子會新建串列, 同時賦值號令 t 指向此新串列, 但是這個操作並不會影響調用者.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

bad_delete_head 函數開始運行時, t 和 t4 指向了同一個串列. 函數結束時, t 指向了新的串列, 但是 t4 仍然指向了原來的, 未被修改的串列.

一種替代方案是, 編寫一個創建並返回新串列的函數. 比如, tail 函數會返回串列中除首元素的所有元素串列:

```
def tail(t):
    return t[1:]
```

此函數依然會保持原串列不變. 下面是用法:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 除錯

若使用串列 (和其他可變對象) 時不夠小心謹慎, 很容易導致長達數小時的除錯跟蹤. 這裡有一些常見陷阱以及如何避免:

1. 多數的串列方法都是傳入參數, 返回 None. 這和字符串方法恰恰相反, 字符串方法一般返回新字符串, 同時保持原始值不變.

   如果你慣於編寫下面這樣的代碼:

```
word = word.strip()
```

那麼你可能會寫出下面這樣的代碼:

```
t = t.sort()            # WRONG!
```

因為 sort 返回的是 None, 所以後續對 t 的操作多會失敗.

在使用串列方法前, 你應該認真閱讀文檔, 並在交互模式下測試一下.

2. 確定原則, 堅定執行.

   串列使用中面臨的部分問題在於, 有多條道路通羅馬. 比如, 從串列中移除元素, 你可以用 pop, remove, del, 甚至用切片重新賦值.

   若想給串列添加元素, 可以用 append 方法, 或者 + 運算子. 假設 t 是串列, x 是串列元素, 下面的操作都正確:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   而下面的代碼是錯誤的:

   ```
   t.append([x])           # WRONG!
   t = t.append(x)         # WRONG!
   t + [x]                 # WRONG!
   t = t + x               # WRONG!
   ```

   在交互模式下測試每個示例, 確保理解其作用. 你會注意到, 只有最後一個會報運行時異常; 其他三個語句合法, 但並不會得到想要的效果.

3. 複製串列, 避免別稱.

   你若想使用 sort 方法修改參數, 同時又想保留原串列, 那你可以複製一份.

   ```
   >>> t = [3, 1, 2]
   >>> t2 = t[:]
   >>> t2.sort()
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

   此例中, 你也可以使用內置函數 sorted, 此函數可以返回新的排好序的串列, 同時又保留了原始串列.

   ```
   >>> t2 = sorted(t)
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

10.14 術語表

串列 (list): 由值構成的序列.

元素 (element): 串列 (或序列) 中的值, 也可以稱為串列項.

嵌套串列 (nested list): 串列中的元素是另外的串列.

累加器 (accumulator): 在迴圈中用於累加或累積結果的變數.

增強賦值 (augmented assignment): 使用 += 這樣的操作符更新變數值的賦值語句.

reduce: 一種遍歷序列, 並累加元素為一個元素的處理模式.

map: 一種遍歷序列, 對每個元素都執行同一操作的處理模式.

filter: 一種遍歷串列, 並篩選滿足特定條件的元素的處理模式.

對象 (object): 變數之指向. 對象擁有特定型態以及值.

相等 (equivalent): 擁有相同的值.

相同 (identical): 指向同一個對象 (也就意味著相等).

引用 (reference): 變數和其指向的值之間的關係.

別稱 (aliasing): 多個變數指向同一個對象的情況.

分隔符 (delimiter): 用來界定長字符串分隔位置的字符和字符串.

10.15 習題集

大家可以從`https://thinkpython.com/code/list_exercises.py` 下載這些習題的答案.

Exercise 10.1. *編寫 `nested_sum` 函數, 接收包含整數串列的一個串列, 然後將所有內嵌串列的元素相加求和. 比如:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercise 10.2. *編寫函數 `cumsum`, 使其接收一個數字串列, 返回累加之和; 也就是, 第 i 個元素是原串列中的第 $i+1$ 個元素之前所有元素之和. 比如:*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Exercise 10.3. *編寫`middle` 函數, 接收一個串列, 返回包含掐頭去尾後所有元素的新串列. 比如:*

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercise 10.4. *編寫函數 `chop`, 使其接收一個串列, 然後移除首尾元素, 返回 `None`. 例如:*

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 10.5. 編寫 is_sorted 函數, 接收一個串列, 如果串列是升序排列, 則返回 True, 否則返回 False. 例如:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 10.6. 如果一個單詞, 通過重新排列字母, 得到另一單詞, 則稱兩個詞為字母異位詞. 編寫is_anagram 函數, 接收兩個字符串, 如果兩者為字母異位詞, 返回 True.

Exercise 10.7. 編寫has_duplicates 函數, 使其接收一個串列, 如果串列中存在重複出現的元素, 返回 True. 注意不要修改原始串列.

Exercise 10.8. 此題屬於生日悖論, 你可以參考*http://en.wikipedia.org/wiki/Birthday_paradox* 了解更多.

如果你的班上有 23 名學生, 那其中兩人的生日相同的概率是多少? 你可以生成 23 個隨機的生日樣本, 檢查其是否存在相同, 從而估計概率值. 提示: 你可以通過 random 模塊中的 randint 函數來製造隨機生日.

可以從 *https://thinkpython.com/code/birthday.py* 下載我的代碼.

Exercise 10.9. 編寫函數讀取 words.txt 內容, 並將其中每個單詞放入串列中. 寫兩個版本, 一個採用 append 方法, 另一個則用 t = t + [x] 實現. 哪種耗時較長? 為什麼?

代碼參見: *https://thinkpython.com/code/wordlist.py*.

Exercise 10.10. 若要檢驗一個單詞是否在串列中, 可以用 in 運算子, 但是速度會慢, 因為它是從頭開始, 順序檢索.

由於這些單詞通常是按照字母表順序排列, 我們可以通過對摺查找 (也叫二分查找), 提高速度. 此方法和你通過字典 (此處指書籍, 不是資料結構) 查找單詞的方式很像. 你一般會先翻到字典中間, 然後看單詞在之前, 還是之後. 如果在前面, 則繼續此方法查找, 如果在後面, 同樣操作.

無論怎樣, 你都可以將搜索區間減半. 如果單詞串列有 *113,809* 個單詞, 大約 *17* 次, 你就可以定位到單詞, 或者確定其不在其中.

編寫函數in_bisect, 使其接收一個有序串列, 以及一個目標值, 當單詞在串列中時, 返回 True, 否則返回 False.

你也可以閱讀 bisect 模塊相關文檔, 進而使用它! 可以參看代碼: *https://thinkpython.com/code/inlist.py*.

Exercise 10.11. 如果兩個單詞拼寫順序相反, 則稱其為 ''逆序對". 編寫程式, 查找單詞串列中所有逆序對. 參見代碼: *https://thinkpython.com/code/reverse_pair.py*.

Exercise 10.12. 如果從兩個單詞交替獲取字母, 構成新單詞, 便稱其 ''連鎖". 比如 "*shoe*" 和 "*cold*", 交替獲取字母, 構成了 "*schooled*". 答案見: *https://thinkpython.com/code/interlock.py*. 致謝: 此習題源於 *http://puzzlers.org* 的一個例子.

1. 編寫函數, 尋找所有連鎖單詞對. 提示: 不要枚舉所有單詞對!
2. 你能找到三路連鎖的單詞嗎; 也就是從三個單詞, 第一個, 第二個, 第三個, 每次獲取三個字母, 從而構成新單詞?

第 11 章　字典

本章講述另一個內置型態, 字典. 字典是 Python 最優秀的特性之一; 同時也是諸多高效且優雅演算法的基石.

11.1　字典即映射

字典 如同串列, 但更加通用. 串列中, 索引必須是整數, 而在字典中, 索引 (幾乎) 可以是任何型態.

字典包括一個索引集合, 叫做**鍵 (keys)**, 以及一個值的集合. 每個鍵都和一個值相關聯. 鍵和值的這種對應關係, 叫做**鍵值對 (key-value pair)** 或者叫做**項 (item)**.

數學語言中, 字典代表了從鍵到值的**映射 (mapping)** 關係, 所以你也可以說, 每個鍵都 "映射" 了一個值. 舉個例子, 我們構建個字典, 使其將英語單詞映射到西班牙語單詞, 那麼, 鍵和值便都是字符串.

函數 `dict` 用來創建一個新的空字典. 由於 `dict` 是內置函數名, 你要盡量避免將其用作變數名.

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

花括號 `{}`, 表示一個空字典. 要想向其中添加項, 可以用方括號:

```
>>> eng2sp['one'] = 'uno'
```

這行代碼創建了一個從鍵`'one'` 到值`'uno'` 的映射的項. 如果你再次打印字典, 可以看到一個鍵值對, 其鍵和值用冒號隔開:

```
>>> eng2sp
{'one': 'uno'}
```

這種輸出格式也可以是輸入的格式. 比如, 用三個項創建一個新字典:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

但你打印 `eng2sp`, 會感到意外:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

鍵值對的順序不一樣了. 如果你在電腦輸入上述代碼, 可能結果也不一樣. 通常, 字典中項的順序是很難確定的.

但是這往往不是個問題, 因為字典的元素從來不是用整數索引去檢索的. 而是用鍵來尋找相應的值:

```
>>> eng2sp['two']
'dos'
```

鍵'two' 總是映射到值'dos' , 所以項的順序就無關緊要了.

如果鍵不在字典中, 你會得到異常:

```
>>> eng2sp['four']
KeyError: 'four'
```

len 函數對於字典同樣有效; 它會返回鍵-值對的數量:

```
>>> len(eng2sp)
3
```

運算子 in 對字典也同樣有效; 它可以判斷某個 *key* 是否存在於字典的鍵中 (而不是值中).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

若要確定值是否存在於字典中, 可以用 values 方法, 它會返回值的集合, 然後使用 in 運算子:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

運算子 in 在串列和字典中使用的演算法不同. 對於串列來說, 它會如第 8.6節一樣, 按照順序搜索串列中的元素. 隨著串列變長, 確定目標的耗時也會更長.

Python 中的字典則使用了叫做**雜湊表**的資料結構, 使其具有了一個神奇特性: 無論字典中有多少項, in 運算子都耗時相同. 我會在第 B.4節解釋其原理, 暫時這些還不重要.

11.2 以字典為計數器

假設有個字符串, 你需要統計每個字母出現的次數. 有多種方法可以實現:

1. 你可以新建 26 個變數, 對應字母表中的每個字母. 然後遍歷字符串, 遇到哪個字母, 相應計數器就加一, 可能會用到鏈式條件.

2. 你也可以創建一個包含 26 個元素的串列. 把每個字母轉換成數字 (使用內置函數 ord), 用這些數字作為串列的索引, 然後累加相應計數器.

3. 你也可以新建一個字典, 這個字典將字母作為鍵, 將計數器作為相應的值. 字母首次出現, 便在字典中添加一個項, 後面再出現, 便可以對已有的項累加即可.

上述各個方案, 目標相同, 手段各異.

一種**實現**便是一種運算方式; 有好有壞. 比如, 用字典實現的優勢在於, 我們無需提前知道字符串中存在哪些字母, 我們只需要為每個出現的字母提供空間而已.

下面是代碼實現:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

這個函數名是 histogram(直方圖), 是個統計術語, 表示計數 (或者頻次) 的集合.

函數第一行新建了一個空字典. 然後用 for 迴圈遍歷字符串. 每次迴圈中, 如果字符 c 不在字典中, 則創建一個新的項, 將 c 作鍵, 初始值為 1（只遇到了一次）. 如果 c 已經在字典中, 則 d[c] 加一.

下面為執行範例:

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

結果表明字母 'a' 和 'b' 只出現了一次; 'o' 出現了兩次, 等等.

字典有個方法, 叫做 get, 以鍵和默認值為參數. 如果鍵在字典中, get 則返回對應的值; 否則, 會返回默認值. 如下:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('c', 0)
0
```

做個練習, 用 get 來精煉一下 histogram 函數. 盡量避免使用 if 語句.

11.3 迴圈和字典

如果你針對字典應用 for 語句, 它會遍歷所有的鍵. 比如, print_hist 函數會輸出每個鍵以及其相應的值:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

下面為輸出結果:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

又是這樣, 這些鍵沒有按照順序輸出. 若要按順序遍歷鍵, 可以用內置函數 sorted:

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4 反向查找

給定字典 `d` 以及鍵 `k`, 很容易確定對應的值 `v = d[k]`. 這種操作, 叫做**查找**.

但是, 如果你有 `v`, 那麼如何找到 `k` 呢? 面臨兩個難題: 第一, 可能有多個鍵的值為 `v`. 基於不同需要, 可能你只需要選一個, 也可能需要將所有的放入串列. 第二, 沒有一個簡單的語法可以實現**反向查找**; 你需要檢索才行.

下面這個函數, 可以根據輸入的值, 返回第一個映射該值的鍵:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

這個函數也是一種檢索模式, 只是它用到了未曾接觸過的一個功能, `raise`. **raise 語法**會製造一個異常; 在這裡, 它製造了 `LookupError`, 這是一個內置異常, 用來表示查找失敗.

如果一直運行到迴圈結束, 這表示 `v` 沒有出現在字典的值中, 所以會引起異常.

下面是一個正常的反向查找的範例:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

以及一個異常的:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

自己製造的異常和 Python 拋出的異常效果是一樣的: 輸出追溯信息以及錯誤信息.

當你自己拋出異常時, 可以通過可選參數, 提供詳細的錯誤信息. 比如:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

反向查找比正向查找要慢得多; 如果使用頻繁, 或者字典過大, 都會對程式性能造成影響.

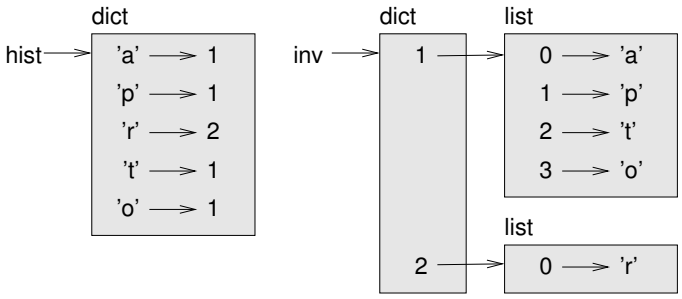

图 11.1: 狀態圖.

11.5 字典和串列

串列可以作為值出現在字典中. 比如, 給定一個字典, 映射了字母和其頻次, 你想倒過來看看; 也就是創建個字典, 是從頻次映射到字母的. 然而, 可能有些字母的頻次相同, 那麼這個倒過來的字典中, 每個值都應該是個字母串列.

此處為一個顛倒字典的函數:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

每次迴圈, `key` 都從 `d` 中獲取一個鍵, `val` 獲取對應的值. 如果 `val` 不在 `inverse` 這個字典中, 也就是說以前沒有遇到過, 那便創建一個新項, 並用**單元素集** (包含一個元素的串列) 來初始化它. 如果在字典中, 則將此鍵添加到串列中.

此為示例:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

圖 11.1 是表示 `hist` 和 `inverse` 的狀態圖. 字典用箱體表示, 其上有 `dict` 型態標識, 其內是鍵值對. 通常如果值是整數, 浮點數或者字符串, 我會將其繪製在箱體內部, 但如果是串列, 我會繪製在箱體外部, 只是為了簡單易懂.

串列可以是字典的值, 就像圖中所示, 但是不能作為鍵使用. 下面是擅自使用遇到的錯誤:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

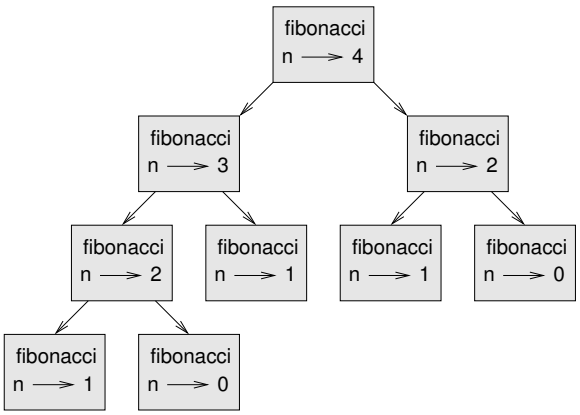

图 11.2: 調用圖.

以前我講過, 字典的實現基於雜湊表, 意味著其鍵必須是**可散列的**.

hash 是一個函數, 其接收任意值 (任何型態), 然後返回一個整數. 字典會用這些被稱為哈希值的整數, 對鍵值對進行存儲和查找.

如果鍵不可變, 則一切正常. 如果鍵像串列一樣可變, 那麼便不太妙了. 比如, 你想創建個鍵值對, Python 計算鍵的哈希值, 並將其存儲於相應的位置. 如果你修改了鍵, 然後再次計算哈希值, 那麼位置不同了. 這時候, 相當於一個鍵對應了兩個不同的位置, 或者說, 你無法找到某個鍵了. 總之, 字典不能正常使用了.

這也就是鍵必須要可哈希的原因, 以及為何像串列一樣的可變型態不能用作鍵的原因. 突破此限制的最簡單方法是使用元組, 這個在下一章節會學到.

因為字典是可變的, 所以不能用作鍵, 但是可以用作值.

11.6 快取

如果你已經學習了第 6.7節的 `fibonacci` 函數, 你會發現, 隨著參數越大, 函數運行時間越長. 而且, 耗時增長的速度很快.

若想理解其原由, 參考圖 11.2, 此**調用圖**展示了當 `n=4` 時 `fibonacci` 函數的調用情況: 調用圖是一堆函數框圖, 用框與框間的連線, 表示調用關係. 此圖最頂層, 表示 `n=4` 的 `fibonacci` 函數會調用 `n=3` 和 `n=2` 時的函數. 同樣, `n=3` 時的 `fibonacci` 函數會調用 `n=2` 和 `n=1` 時的函數, 以此類推.

算算 `fibonacci(0)` 和 `fibonacci(1)` 會被調用多少次吧. 此操作是如此低效, 而且隨著參數變大, 效率更低.

另一種思路是跟蹤並記錄已經計算過的數據, 將其存儲到字典中. 將之前計算過的值存儲起來, 方便後續使用, 這種操作叫做**快取**. 下面是 `fibonacci` 的 ‘‘快取’’ 版本:

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]
```

```
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

字典 known 用來跟蹤記錄過往結果. 其初始包括兩項：0 對應 0, 1 對應 1.

每次 fibonacci 被調用, 總會先檢查 known. 如果結果已經存在, 則立刻返回結果. 否則, 計算結果, 並保存到字典, 然後返回結果.

當你運行這個版本的 fibonacci, 和以前版本相比, 你會發現, 快了很多.

11.7 全局變數

上例中, known 是在函數外創建, 所以它屬於__main__ 函數的框內, 這是一個特殊的框. __main__ 函數中的變數, 有時也被認為是**全局的**, 因為可以從任意函數訪問它們. 不像局部變數, 其運行的函數一旦結束, 便會消失, 全局變數可以在一個函數執行到另一個函數時, 也保持存在.

全局變數, 一個常用的地方, 便是作為**標識**使用; 也就是作為布爾變數, 來表示 (“標識”) 條件是否成立. 比如, 有些程式會使用 verbose 作為標識, 來控制輸出信息的詳細程度:

```
verbose = True

def example1():
    if verbose:
        print('Running example1')
```

如果嘗試重新賦值全局變數, 可能會得到意外結果. 下面的例子, 用全局變數來跟蹤函數是否被調用過:

```
been_called = False

def example2():
    been_called = True          # WRONG
```

運行此程式, 你會注意到been_called 的值沒有改變. 問題在於 example2 內部又創建了名為been_called 的局部變數. 而局部變數會在函數結束後釋放, 並不會對全局變數產生影響.

如果想在函數內部修改全局變數, 你需要在賦值前, **聲明**這是全局變數:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

這個 **global 聲明**會告訴解釋器, “在這個函數中, 當我說been_called, 我指的是那個全局變數; 而不是新建一個局部變數. ”

下面是段修改全局變數的代碼:

```
count = 0

def example3():
    count = count + 1           # WRONG
```

運行後, 你會看到:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python 假設 `count` 是局部的, 據此假設, 需要在更新變數前先讀取變數. 解決方法依然是, 聲明 `count` 為全局變數.

```
def example3():
    global count
    count += 1
```

如果全局變數指向了可變的值, 那你在修改前, 無需聲明全局變數:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

所以你可以在全局的串列或字典中, 添加, 移除以及替換元素, 但是你若想對變數重新賦值, 需要提前聲明全局變數:

```
def example5():
    global known
    known = dict()
```

全局變數是很有用, 但如果你到處使用, 並且還頻繁修改, 那麼程式往往會難以除錯.

11.8 除錯

隨著數據集越來越大, 僅僅使用打印輸出以及手工校驗結果的手段進行除錯, 略顯不便. 下面是除錯大型數據集的一些建議:

降低輸入規模: 如果可能, 降低數據集大小. 比如, 讀取文本檔案, 只讀取前 10 行, 或者使用你能找到的最簡範例. 可以重新編輯檔案, 或者 (更優的做法是) 修改程式, 只讀取前 `n` 行.

如果報錯, 可以將 `n` 減小為剛好會報錯的最小值, 然後校正異常, 並逐步增大 `n` 值, 如此迴圈往復.

檢驗概覽及型態: 不要打印並檢驗全部數據集, 只需輸出數據的概覽: 比如, 字典中項的個數, 或者串列中數字總數量.

值型態錯誤, 是產生運行時錯誤的常見原因. 所以除錯此類錯誤, 通常打印值的型態即可.

編寫自檢代碼: 有時候, 你可以編寫代碼, 自動檢驗異常. 例如, 你要計算數字串列的平均值, 你可以檢驗一下, 結果是否比串列中最大值還要大, 或者比最小值還要小. 這叫做 ‘‘合理性檢驗’’, 是因為要檢查結果是否 ‘‘不合理’’.

另一種檢驗方法是比較兩種不同解法的結果, 看是否一致. 這叫做 ‘‘一致性檢驗’’.

格式化輸出: 格式化除錯的輸出, 可以更直觀地展示錯誤. 在第 6.9節, 我們曾展示過範例. 另一個好用的工具是 `pprint` 模塊, 它提供了 `pprint` 函數, 可以將內置型態輸出為更適合人類閱讀的格式 (`pprint` 是 “pretty pprint” 的縮寫).

再次強調, 構建腳手架花費時間越長, 除錯耗費時間越少.

11.9 術語表

映射 (mapping): 集合中每個元素都對應到另一個集合的單一元素的關係.

字典 (dictionary): 從鍵到相應的值的映射.

鍵值對 (key-value pair): 鍵到值的映射關係的表示.

項 (item): 字典中, 鍵值對的別名.

鍵 (key): 字典中的一個對象, 也是鍵值對的第一部分.

值 (value): 字典中的一個對象, 作為鍵值對的第二部分. 這裡的表示和以前用到的 "值" 相比, 更加具體.

實現 (implementation): 一種執行計算的方式.

雜湊表 (hashtable): 實現 Python 字典的演算法.

哈希函數 (hash function): 雜湊表使用的函數, 用來計算鍵的位置.

散列的 (hashable): 一種可以使用哈希函數的型態. 不可變型態, 像整型, 浮點型和字符串型, 都是可散列的; 像串列和字典這種可變型態, 則不是.

查找 (lookup): 一種字典操作, 根據鍵查找對應的值.

反向查找 (reverse lookup): 一種字典操作, 通過值來查找對應的一個或多個鍵.

raise 語句 (raise statement): 主動拋出異常的語句.

單元素集 (singleton): 只包含一個元素的串列 (或者其他序列).

調用圖 (call graph): 一種用來展示程式執行流程的圖, 包括每步創建的框, 以及從調用者指向被調用者的箭頭.

備忘 (memo): 存儲計算過的值, 避免後續重複計算的操作.

全局變數 (global variable): 定義在函數之外的變數. 全局變數可以被任何函數獲取使用.

global 語句 (global statement): 聲明變數是全局變數的語句.

標識 (flag): 標識條件是否為真的布爾變數.

聲明 (declaration): 像 `global` 一樣的語句, 用來告知解釋器變數的有關信息.

11.10 習題集

Exercise 11.1. 編寫函數, 讀取 `words.txt` 中的單詞, 將其作為鍵存儲於字典中. 值是什麼無關緊要. 然後便可以用 `in` 運算子快速檢驗某個字符串是否在字典中.

如果你做過習題 *10.10*, 可以比較一下這種實現, 和串列的 `in` 運算, 以及二分查找相比, 速度如何.

Exercise 11.2. 閱讀字典方法 `setdefault` 的文檔, 以此精簡`invert_dict` 版本. 參見: *https://thinkpython.com/code/invert_dict.py*.

Exercise 11.3. 用快取的方式重寫習題 *6.2* 中的 *Ackermann* 函數, 看看是否快取方式可以使其處理較大參數. 提示: 不能. 參考: *https://thinkpython.com/code/ackermann_memo.py*.

Exercise 11.4. 如果做過習題 *10.7*, 你應該已經寫過一個has_duplicates 函數, 令其接收一個串列作為參數, 如果其中存在任意重複對象, 則返回 True.

使用字典來編寫一個更簡單高效的has_duplicates 版本. 參考: *https://thinkpython.com/code/has_duplicates.py*.

Exercise 11.5. 兩個單詞, 如果翻轉其中一個, 得到另一個, 則稱其為 ''翻轉詞組" (參見習題 *8.5*中的rotate_word).

編寫程式讀取一個單詞表, 並找到所有的翻轉詞組. 參考: *https://thinkpython.com/code/rotate_pairs.py*.

Exercise 11.6. 這是另一個來自 Car Talk 的謎題 (*http://www.cartalk.com/content/puzzlers*):

> 這個謎題來自於一個名為 *Dan O'Leary* 的朋友. 他最近發現一個單音節, 五個字母的常用單詞, 其具有下述特徵. 當你移除首字母, 剩下字母構成了一個和原單詞發音完全一樣的同音詞. 如果再替換首字母, 也就是把首字母放回去, 移除第二個字母, 然後得到的又是同音詞. 那麼問題來了, 這是什麼詞?
>
> 現在, 我給你展示一個錯誤的例子. 咱們看一下這個五個字母的單詞, *'wrack'*. *W-R-A-C-K*, 就是 *'wrack with pain.'* 中的那個 *'wrack'*. 如果移除首字母, 剩下四個字母的單詞, *'R-A-C-K.'* 比如, *'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!'* 發音完全一樣. 如果你放回 *'w'*, 移除 *'r'*, 得到一個單詞, *'wack'*, 這是一個真正的單詞, 只是發音和前兩個不同.
>
> 但是, 至少有一個這樣的單詞, *Dan* 和我們都認識, 分別刪除前兩個字母, 構成兩個新的四個字母單詞, 發音完全相同. 問題是, 這個詞是什麼?

你可以用習題 *11.1* 中的字典來檢驗一個字符串是否在單詞表中.

若想檢查兩個單詞是否是同音詞, 可以用 *CMU* 發音詞典. 這個詞典可以從*http://www.speech.cs.cmu.edu/cgi-bin/cmudict* 或*https://thinkpython.com/code/c06d*下載. 或者下載*https://thinkpython.com/code/pronounce.py*代碼, 這個檔案提供了一個 read_dictionary 函數, 可以讀取發音詞典並返回一個 *Python* 字典, 此字典將每個單詞映射到了表示發音的字符串.

編寫程式列出所有滿足謎語條件的單詞. 答案: *https://thinkpython.com/code/homophone.py*.

第 12 章　元組

本章介紹另一種內置型態, 元組, 以及串列、字典和元組如何一起使用. 同時展示可變長度參數串列的一個特性, 以及聚合和擴展運算子.

話外音：對於 “tuple’’ 如何發音, 尚無共識. 有人讀 “tuh-ple”, 和 “supple’’ 發音類系. 但在編程領域, 多數人讀 “too-ple”, 和 “quadruple’’ 發音一樣.

12.1　元組不可變

元組也是值的序列. 其中值可以是任意型態, 也是用整數作為索引, 所以在某些方面, 元組和串列很像. 但兩者最大的區別在於元組不可變.

從語法來看, 元組就是一個逗號分隔的序列:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

雖然不是很必要, 但一般都用括號把元組括起來:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

若要創建一個單元素的元組, 需要在末尾加上逗號:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

括號中只有一個值, 便不是元組:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

另一種創建元組的方法, 可以使用內置函數 `tuple`. 如果不傳參數, 便會創建一個空元組:

```
>>> t = tuple()
>>> t
()
```

如果參數是一個序列 (字符串, 串列或者元組), 結果便會得到一個由元素序列構成的元組:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

因為 `tuple` 是一個內置函數名, 所以要盡量避免用作變數名.

多數串列運算子同樣適用於元組. 所以用方括號也可以索引元組元素:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

切片運算子也可以選擇元組中一段區間的元素.

```
>>> t[1:3]
('b', 'c')
```

你如果想修改元組中的某個元素, 會報錯:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

因為元組不可變, 所以你不能修改其中元素. 但是你可以用另一個元組替換當前元組:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

這個語句創建新元組, 然後用 `t` 指向它.

關係運算子也同樣適用於元組以及其他序列; Python 會從各自序列的首元素開始比較, 如果相等, 則比較下一個元素, 以此類推, 直到找出不同元素. 而後續元素便不再比較 (即使後面元素更大).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 元組賦值

交換變數的值經常會用到. 比較傳統的方法, 是使用中間變數. 比如下面例子, 互換 `a` 和 `b` 的值:

```
>>> temp = a
>>> a = b
>>> b = temp
```

這種方法比較麻煩; 用**元組賦值**會優雅很多:

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

左側是個變數元組; 右側是個表達式元組. 每個值都會賦值給其對應的變數. 右側的表達式在賦值操作前都會先計算.

同時左側變數的數量必須要和右側值的數量一樣:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

也有更普遍的用法, 右側可以用各種型態的序列 (字符串, 串列或者元組). 例如, 將一個郵箱地址切分為用戶名和域名, 可以這麼寫:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

`split` 的結果是一個兩元素的串列; 第一個元素賦值給了 `uname`, 第二個賦值給了 `domain`.

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3 元組作為返回值

嚴格來講, 函數只能返回一個值, 但是如果這個值是一個元組, 效果便等同於返回了多個值. 比如, 你要將兩個整數相除, 計算商和餘數, 先計算 `x//y` , 再計算 `x%y`, 略顯低效. 更好的方法是同時計算兩個值.

內置函數 `divmod` 會接收兩個參數, 返回一個包含兩個值的元組, 商和餘數. 同時, 你也可以將結果存儲為元組:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

或者用元組賦值分別存儲結果:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

下面是一個返回元組的函數:

```
def min_max(t):
    return min(t), max(t)
```

`max` 和 `min` 都是內置函數, 用來從序列中尋找最大值和最小值. `min_max` 函數計算兩者, 然後將其作為元組返回.

12.4 變長參數即元組

函數可以接收任意數量參數. 以 `*` 開頭的參數會**聚合**所有參數作為一個元組. 比如, `printall` 會接收任意數量的參數, 並打印輸出:

```
def printall(*args):
    print(args)
```

這個聚合參數可以隨意命名, 但是 `args` 更為人所熟知. 下面的函數展示其效果:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

聚合的對立面是**擴展**. 如果你有一個值的序列, 想將其作為多個參數, 傳入函數, 那便可以使用 `*` 運算子. 比如, `divmod` 僅接收兩個參數; 如果傳入一個元組, 則運行異常:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

但如果你擴展了元組, 那便運行正常:

```
>>> divmod(*t)
(2, 1)
```

很多內置函數都會用到變長參數元組. 比如, `max` 和 `min` 都可以接收任意數量的參數:

```
>>> max(1, 2, 3)
3
```

但 `sum` 函數不行.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

做個練習, 寫個函數 `sum_all`, 使其可以接收任意數量參數, 並返回總和.

12.5 串列和元組

zip 是個內置函數, 其接收兩個或者更多序列作為參數, 並交錯獲取其中元素. 這個函數的名字表示其像拉鏈, 彷若兩排交錯的牙齒.

下面的例子是壓縮一個字符串和一個串列:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

結果是個 **zip 對象**, 用來疊代數值對. 所以 zip 常常用在 for 迴圈內:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

zip 對象是一種**疊代器**, 也就是一種用來疊代序列的對象. 疊代器和串列在某些方面比較相似, 但與串列不同的是, 你不能從疊代器中通過索引獲取元素.

如果你想使用串列的運算子和方法, 可以用 zip 對象構建串列:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

串列中都是元組; 在這個例子中, 每個元組都包括字符串中的一個字母, 以及串列中的相應元素.

若序列長度不同, 則結果長度取決於較短的那個序列.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

可以在 for 迴圈中使用元組賦值, 遍曆元組串列:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

每次迴圈, Python 都會獲取串列中的下一個元組, 並將其元素分別賦給 letter 和 number. 迴圈的輸出如下:

```
0 a
1 b
2 c
```

如果結合 zip, for 和元組賦值, 那你便得到了一個同時遍歷兩個 (或多個) 序列的利器. 比如, has_match 接收兩個序列, t1 和 t2, 如果存在索引 i, 令 t1[i] == t2[i] 成立, 則返回 True:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

如果你需要遍歷序列元素以及其索引, 可以使用內置函數 enumerate:

```
for index, element in enumerate('abc'):
    print(index, element)
```

enumerate 返回的結果是一個枚舉對象, 會疊代一個值對序列; 每個值對包括索引 (從 0 開始) 和給定序列中的元素. 此例輸出同樣如下:

```
0 a
1 b
2 c
```

12.6 字典和元組

字典有個方法叫做 items, 其返回一個元組序列, 每個元組都是一個鍵值對.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

結果是個 dict_items 對象, 是一個疊代鍵值對的疊代器. 可以在 for 中使用, 如下所示:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

如你所想, 字典中的項沒有特定順序.

相反, 你也可以用一個元組串列, 初始化一個新字典:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

將 dict 和 zip 結合使用, 便產生了一種新建字典的簡潔方法:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

字典的 update 方法也可以接收一個元組串列, 然後將其作為鍵值對添加到已有字典中.

使用元組作為字典的鍵, 很常見 (主要是串列不能). 比如, 電話簿會將姓氏-名字映射到電話號碼. 假設我們定義了 last, first 和 number, 便可以編寫如下:

```
directory[last, first] = number
```

方括號中的表達式是個元組. 可以用元組賦值語句來遍歷字典.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

此迴圈會遍歷 directory 中的鍵, 也就是元組. 同時會將元組中的元素賦值給 last 和 first, 然後輸出姓名以及對應的電話號碼.

在狀態圖中展示元組有兩種方法. 較詳細的版本會展示起索引和元素, 就像串列一樣. 例如, 圖 12.1 展示了元組 ('Cleese', 'John') .

但在大規模得圖中, 你需要忽略細節. 圖 12.2展示了電話簿的狀態.

圖中的元組都使用 Python 語法形象表示. 電話號碼是 BBC 的投訴熱線, 所以請不要撥打.

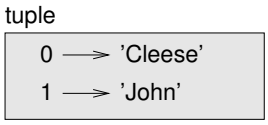

图 12.1: 狀態圖.

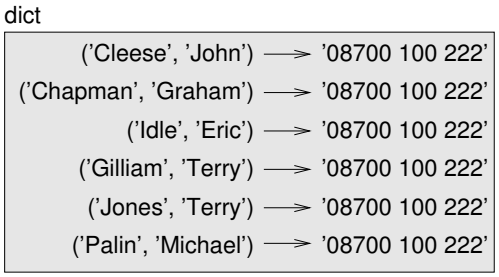

图 12.2: 狀態圖.

12.7 序列中的序列

我一直講的是元組串列, 但是本章幾乎所有示例同樣適用於串列構成的串列, 元組構成的元組, 以及串列構成的元組. 為了避免枚舉各種組合, 直接叫做序列構成的序列, 便簡單多了.

很多情形下, 不同型態的序列 (字符串, 串列以及元組) 可以替換使用, 那你如何從中擇優呢?

從最顯而易見的情形考慮, 字符串比其他序列, 應用場景最有限, 因為其元素需要是字符. 同時, 他們不能被修改. 如果你想要修改字符串中的字符 (而不是新建一個新字符串), 你最好還是用字符串列吧.

串列比元組更加常用, 主要因為串列可變. 但以下情況, 用元組會更好:

1. 在某些情形, 比如 `return` 語句, 語法上創建元組比創建串列更簡便.
2. 如果你想使用序列作為字典的鍵, 你需要使用元組或者字符串這種不可變型態.
3. 如果需要將一個序列作為參數傳遞給函數, 使用元組可以有效降低使用別名帶來的潛在問題.

因為元組不可變, 所以它們不提供 `sort` 和 `reverse` 這種修改現有串列的方法, 但是 Python 提供內置函數 `sorted`, 其接收任意序列, 然後返回一個排序後的新串列, 也提供了 `reversed` 函數, 其接收一個序列, 返回一個疊代器, 此疊代器可以倒序遍歷串列.

12.8 除錯

串列, 字典以及元組都是**資料結構**的例子; 本章我們開始了解複合資料結構, 像元組串列, 或者用元組作鍵, 串列作值的字典. 複合資料結構很有用, 但很容易產生我說的**格式異常**; 就是當資料結構中出現了不恰當的型態, 大小或結構, 導致的錯誤. 比如, 你想要一個含有一個整數的串列, 而我給了一個單純的整數 (不被串列包含), 那便會出錯.

為了除錯這種異常, 我寫了一個叫作 `structshape` 的模塊, 其提供了一個 `structshape` 函數, 可以接收任意資料結構作為參數, 返回一個其格式的概述. 你可以從 `https://thinkpython.com/code/structshape.py` 下載.

下面是一個簡單串列的示範:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

嚴謹點的寫法是 "list of 3 int*s*", 但是忽略複數, 便於操作. 下面是一個串列構成的串列:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

如果串列中的元素型態不止一種, `structshape` 按順序匯總型態:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

這是一個元組串列:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

下面是一個包含 3 個項的字典, 每個項都是映射整數到字符串.

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

如果你對你的資料結構感到棘手, 可以試試 `structshape`.

12.9 術語表

元組 (tuple): 一個不可變的元素序列.

元組賦值 (tuple assignment): 一種序列居右, 變數元組在左的賦值語句. 等號右側會先運算, 然後這些結果會賦值給相應的左側變數.

聚合 (gather): 將多個參數聚合為一個元組的操作.

擴展 (scatter): 使序串列現為多個參數的操作.

zip 對象 (zip object): 調用內置函數 `zip` 返回的結果; 是可遍歷多個元組序列的對象.

疊代器 (iterator): 可遍歷序列的對象, 但不適用串列運算子和方法.

資料結構 (data structure): 相關值的集合, 通常由串列, 字典, 元組等組成.

格式異常 (shape error): 值的格式不合適導致的錯誤, 通常是型態或者大小.

12.10 習題集

Exercise 12.1. 編寫函數 most_frequent, 接收一個字符串, 然後按照字母頻數倒序輸出. 從不同語言找一些文本素材, 看看不同語言之間字母頻數有何差異. 並將結果與*http://en.wikipedia.org/wiki/Letter_frequencies* 中的表格進行比較. 參考: *https://thinkpython.com/code/most_frequent.py*.

Exercise 12.2. 再來些變位詞!

1. 編寫程式, 從檔案中讀取單詞串列 (參見第 *9.1*節), 並且輸出所有的變位詞集合.

   下面是個輸出範例:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   提示: 你或許會想要構建個字典, 將字母集合映射到單詞串列, 而串列中的單詞都可以用這些字母構成. 問題來了, 你如何表示這些字母集合, 從而使其能夠作為鍵來使用?

2. 修改上面的程式, 使其先輸出最長的變位詞, 然後輸出次長, 依次類推.

3. 拼字遊戲中, 你已有 *7* 張牌了, 如果又翻出了第 *8* 張, 構成了一個 *8* 字母的單詞, 那麼你就獲得了一個 "*bingo*". 找找哪些 *8* 字母的組合, 是最可能獲得 "*bingo*'' 的組合?

   參考: *https://thinkpython.com/code/anagram_sets.py*.

Exercise 12.3. 如果將一個單詞交換其中兩個字母的位置, 構成另一個單詞, 則這兩個單詞構成了一個 ''易位詞對"; 比如, "*converse*" 和 "*conserve*". 編寫程式, 從字典中尋找所有的 ''易位詞對". 提示: 不用嘗試所有的單詞組合, 也不用嘗試所有可能易位操作. 參考: *https://thinkpython.com/code/metathesis.py*. 鳴謝: 這道題源自*http://puzzlers.org* 上的一個例子.

Exercise 12.4. 這又是一個汽車廣播字迷 (*http://www.cartalk.com/content/puzzlers*):

> 每次從單詞中移除一個字母, 依然是正確的英文單詞, 這樣的單詞中, 最長的是哪個?
>
> 可以從末尾或者中間移除字母, 但不能調整任何字母的順序. 每次移除一個字母, 都得到一個新的英文單詞. 一直這樣操作, 最終會得到一個字母, 同時這個字母也是一個英文單詞---可以在字典中找到. 那麼最長的單詞是什麼, 以及它有多少個字母?
>
> 我舉個小例子: *Sprite*. 認識嗎? 從 *sprite* 開始, 從中間去掉一個字母, 移除 *r*, 剩下的字母構成了單詞 *spite*, 然後移除末尾的字母, 得到單詞 *spit*, 然後移除 *s*, 得到 *pit*, 然後依次得到 *it*, 和 *I*.

編寫程式, 尋找所有這樣可以縮減的單詞, 然後確定最長的單詞.

這道題相比其他題有些難, 所以這裡給些提示:

1. 你可能需要寫個函數, 接收一個單詞, 然後計算得到所有的單詞串列. 這些單詞都是通過移除一個字母得到. 這些就是這個單詞的 ''子類".

2. 用遞歸思路來考慮, 如果一個單詞的子類是可以縮減的, 那麼這個單詞就是可以縮減的. 可以將空字符串可縮減, 作為基準條件.

3. 我提供的單詞串列 `words.txt`, 不包含單字母單詞. 所以你可能需要添加 "*I*", "*a*" 和空字符串.

4. 若想提升程式性能, 最好緩存住已知的可縮減單詞.

參考: *https://thinkpython.com/code/reducible.py*.

第 13 章　案例學習: 資料結構選擇

截止當前, 你已學習了 Python 最核心的資料結構, 同時也已熟悉了與之相關的一些演算法. 若想後續深入了解演算法知識, 可以學習章節 B. 但現在不用太急切; 後續時間方便了, 可以耐心學習.

本章通過多道習題, 一個案例來讓你思考如何選擇和應用資料結構.

13.1　詞頻統計

同往常一樣, 看答案前先盡量嘗試自己做一下.
Exercise 13.1. 寫個程式, 讀取檔案, 把每行拆分成單詞, 去掉空格和標點, 並將其轉為小寫.

提示: `string` 模塊提供了名為 `whitespace` 和名為 `punctuation` 的字符串, `whitespace` 包含了空格, 製表符, 換行等等; 而 `punctuation` 則包含各種標點符號. 讓我們看看是否可以用 *Python* 來說明:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

同時, 你可能會用到字符串方法 `strip`, `replace` 和 `translate`.
Exercise 13.2. 前往 *Gutenberg* 專案 (*http://gutenberg.org*), 然後下載一個你最喜歡的書籍, 要下載無版權書籍的純文本格式.

修改上個習題的代碼, 使其可以讀取所下載書籍的內容, 同時跳過書籍開頭部分的標題信息, 像以前一樣處理其餘單詞.

調整代碼, 統計書中單詞總數, 以及每個單詞出現次數.

輸出書中不重複的單詞數量. 比較不同作者在不同時代撰寫的不同書籍, 看看哪個作者的用詞最豐富?
Exercise 13.3. 修改上面習題中的程式, 輸出書中出現次數最多的 *20* 個單詞.
Exercise 13.4. 修改上述代碼, 使其讀取單詞串列 (參見第 *9.1* 節), 並輸出所有出現在書中, 但不在單詞表中的詞彙. 看看有多少是拼寫異常? 又有多少是應該包含在單詞表中的常用詞彙, 又有哪些是晦澀難懂的單詞?

13.2　隨機數

多數程式, 給定同樣的輸入, 每次都是同樣的輸出, 這就叫做**確定性**. 確定性是件好事, 因為我們總是希望同樣的計算產生同樣的結果. 但對於某些應用來說, 我們需要計算機能變得不可預測. 遊戲就是一個很好的例子, 也有很多其他的應用場景.

想讓一個程式完全隨機, 是很難的, 但是可以讓其看起來隨機. 一種方法是使用演算法生成**偽隨機**數. 偽隨機數不是真正隨機的, 因為它們是用一個確定的運算生成的, 但只是從數字來看, 很難與真隨機進行區分.

random 模塊提供了生成偽隨機數 (後續均用 ''隨機數'' 作為簡稱) 的函數.

函數 random 會返回 0.0 到 1.0 之間 (包括 0.0 但不包括 1.0) 的一個隨機浮點數. 每次調用 random, 都會得到既定長序列上的下一個值. 下面的迴圈便是一個例子:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

函數 randint 接收參數 low 和 high, 返回 low 和 high 之間的一個整數 (包括兩者).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

若想從一個序列隨機選擇一個元素, 可以用 choice 函數:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

random 模塊也提供了些可基於某種連續分布, 比如高斯分布, 指數分布, 伽馬分布等, 生成隨機數的函數.

Exercise 13.5. *編寫choose_from_hist 函數, 使其以第 11.2節中定義的統計頻數為輸入, 然後從中根據與頻數成正比的概率來返回一個隨機數. 比如, 對於下面的頻數統計來說:*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

你的函數應該返回'a' 的概率是 2/3, 返回'b' 的概率是 1/3.

13.3 詞頻

繼續下面章節前, 先盡量完成上面的練習. 你可以從 https://thinkpython.com/code/analyze_book1.py下載我的解決方案. 此外, 你也會需要https://thinkpython.com/code/emma.txt這個檔案.

下面是一個讀取檔案, 並基於其中單詞構建詞頻統計的程式:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
```

```
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

這個程式會讀取 `emma.txt`, 其中包括了 Jane Austen 的 *Emma* 小說.

`process_file` 遍歷檔案中的每一行, 同時將每行內容逐次傳給`process_line`. 詞頻 `hist` 在此處作為累加器使用.

`process_line` 用字符串方法 `replace` 將連字符替換為空格, 然後用 `split` 方法, 將整行拆分為字符串串列. 然後遍歷串列, 並使用 `strip` 方法, 移除標點符號, 使用 `lower` 方法, 轉換為全小寫. (此處簡單說是 "轉換", 但要記住字符串是不可變的, 所以像 `strip` 以及 `lower` 方法, 都是返回新的字符串.)

最終, `process_line` 會通過新建項或者累加已有項, 更新詞頻統計.

若想統計檔案中全部單詞數量, 可以將詞頻統計中的所有頻數相加求和即可:

```
def total_words(hist):
    return sum(hist.values())
```

不同單詞的數量也就是字典中項的數量:

```
def different_words(hist):
    return len(hist)
```

下面的代碼會輸出結果:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

結果如下:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 最常用單詞

若想找到最常用的單詞, 我們可以創建一個元組串列, 每個元組都包含一個單詞和其頻數, 然後據此將串列排序.

下面函數接收一個詞頻統計, 然後返回一個詞-頻元組串列:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

每個元組中, 頻數在前, 所以串列便是按頻數排序. 下面迴圈會輸出前十個最常用單詞:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

此處使用用關鍵詞參數 `sep`, 來令 `print` 用換表符而不是空格作為 "分隔符", 這樣第二列就會對齊. 下面是對 *Emma* 小說的統計結果:

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

上述代碼可以用 `sort` 函數的 `key` 參數來簡化. 如果你感興趣, 可以參閱 `https://wiki.python.org/moin/HowTo/Sorting`.

13.5 可選參數

我們已經見到了諸多具有可選參數的內置函數和方法. 而且我們也可以編寫具有可選參數的自定義函數. 比如, 下面是輸出詞頻統計中最常用單詞的函數:

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

第一個參數是不可或缺的; 第二個是可選的. `num` 的**默認值**是 10.

如果你只給定一個參數:

```
print_most_common(hist)
```

`num` 會使用默認值. 如果給定兩個參數:

```
print_most_common(hist, 20)
```

`num` 會使用實參來賦值. 換句話說, 可選參數會**覆蓋** 默認值.

如果一個函數同時有必需和可選參數, 則必需參數在前, 可選參數在後.

13.6 字典減法

從本書選擇不在 `words.txt` 中的單詞會比較麻煩, 你可能已經意識到, 這是個集合減法; 也就是說, 我們希望找出所有在某集合 (本書單詞), 但不在另一個集合 (單詞串列) 中的單詞.

`subtract` 函數會接收字典 `d1` 和 `d2`, 返回一個新字典, 其包含所有 `d1` 中包含, 但 `d2` 中不包含的鍵. 因為我們不關心鍵的值, 所以就將其都設為 None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

若要找到包含在書中, 但是不包含在 `words.txt` 中的單詞, 我們可以用`process_file` 函數先建立一個 `words.txt` 的詞頻統計, 然後再做減法:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

下面是對 *Emma* 小說執行統計後的結果:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

這些單詞有些是名字和所有格. 其他的, 例如 “rencontre”, 已經很罕見了. 但是有些依舊很常用, 確實也應該包含在串列中!

Exercise 13.6. *Python 提供了一種名為 `set` 的資料結構, 其可以滿足多數的集合操作. 你可以仔細閱讀章節 19.5, 或者閱讀 `http://docs.python.org/3/library/stdtypes.html#types-set` 上面的文檔.*

編寫程式, 使用集合減法, 尋找包含在本書但是不在單詞串列中的單詞. 參考: `https://thinkpython.com/code/analyze_book2.py`.

13.7 隨機單詞

若要從詞頻統計中選擇隨機單詞, 最簡單的方法是每個單詞複製一定數量, 複製的數量為其對應的頻數, 合併為串列, 然後, 從中選擇單詞:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

表達式 `[word] * freq` 會將 `word` 複製 `freq` 份. `extend` 方法和 `append` 作用相似, 只是前者參數是個序列.

這個演算法有效, 但不夠有效率; 每次選擇隨機單詞, 都需要重新構建和原書一樣大的串列. 顯然, 更優的方案是只建立一次串列, 然後多次選擇, 但串列依然很大.

一個更高效的方案是:

1. 使用 `keys` 生成本書單詞的串列.
2. 用詞頻的各步累積之和 (參看習題 10.2), 構建串列. 串列中最後一項, 便是書中單詞總數, n.

3. 選擇一個從 1 到 n 的隨機值. 使用二分搜索 (參看習題 10.10), 尋找隨機值在累積值串列中所嵌入位置的索引.

4. 使用該索引值從單詞串列中確定對應的單詞.

Exercise 13.7. *採用上面的演算法, 編寫程式, 從書中隨機獲取單詞. 參考: https://thinkpython.com/code/analyze_book3.py.*

13.8 Markov 分析

如果從書中隨機選擇一些詞彙, 單詞意思可能都明白, 但是你很難理解整句話:

```
this the small regard harriet which knightley's it most things
```

一些單詞隨機組合, 很少能夠清楚表達意思, 因為其連續單詞之間沒有聯繫. 比如, 一個完整的語句中, 我們知道 "the'' 的後面往往緊跟著形容詞或名詞, 很少是動詞或副詞.

針對這些聯繫, 一種評估手段便是 Markov 分析, 這種方法的特徵在於對給定的單詞序列, 評估接下來出現某個單詞的概率. 比如, 歌曲 *Eric, the Half a Bee* 的開頭:

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

這段文本中, "half the'' 後邊總是緊跟著 "bee", 但是 "the bee'' 後面有 "has" 也有 "is".

Markov 分析得到的結果, 一般是從每個前綴 (比如 "half the" 和 "the bee") 到所有可能後綴 (比如 "has" 和 "is") 的映射. 有了這個映射, 你便可以選擇任意單詞作為前綴, 然後從可能的後綴中隨機選擇單詞. 繼而, 用合併後句子的末尾作為前綴, 與後綴拼接, 構成下一個前綴, 不斷重複此過程.

比如, 開始選擇 "Half a'' 作為前綴, 然後緊跟著必然是 "bee", 因為這個前綴只在文本中出現了一次. 然後, 下一次前綴便是 "a bee", 其後綴則可能是 "philosophically", "be" 或 "due".

這個例子中, 前綴總是兩個單詞, 但你也可以用任意長度的前綴進行 Markov 分析.

Exercise 13.8. *Markov 分析:*

1. *寫個程式, 讀取文本內容, 進行 Markov 分析. 結果應該是個從前綴映射到可能後綴集合的字典. 這個集合可以是串列, 元組或者字典; 取決於你的選擇. 你可以用兩個詞的前綴測試程式, 但最好想辦法讓程式可以兼容其他長度的前綴.*

2. *在上面的程式中增加一個函數, 基於 Markov 分析生成隨機文本. 下面是針對 Emma 採用兩個單詞前綴進行 Markov 分析後, 生成的文本:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

這個例子中, 我保留了單詞的標點符號. 結果看來語法正常, 但稍有瑕疵. 從語義來看, 意思也能理解, 但不是很清晰明了.

如果增加前綴單詞長度, 會如何? 得到的隨機文本是否會語義明了一些?

3. 一旦程式運行正常, 你可以試試混搭: 如果將兩本或者多本書的文本合併, 生成的隨機文本夾雜著各處來源的單詞短語, 將非常有趣.

鳴謝: 本案例參考了 *1999* 年由 *Addison-Wesley* 出版, *Kernighan and Pike* 所撰寫的 The Practice of Programming 一書中的例子.

在繼續之前, 盡量嘗試一下解答此習題; 然後你可以從 `https://thinkpython.com/code/markov.py`下載我的方案, 同時, 你也會用到`https://thinkpython.com/code/emma.txt`.

13.9 資料結構

使用 Markov 方法生成隨機文本固然有趣, 但是這個練習有個關鍵點: 資料結構的選擇. 在解決前面的習題時, 你不得不考慮以下幾點:

- 如何表示前綴.
- 如何表示可能的後綴集合.
- 如何表示前綴和後綴集合的映射.

最後一個很簡單: 對於從鍵映射到相應的值, 很顯然, 採用字典.

如果是前綴, 最好選擇字符串, 字符串串列或者字符串元組.

對於後綴, 可以選擇串列; 或者詞頻統計 (字典).

至於如何抉擇? 首先要考慮無論選擇哪種資料結構, 都不可避免的相應操作. 對於前綴來說, 我們需要從開頭刪除單詞, 在末尾添加單詞. 比如, 如果現在前綴是 “Half a”, 後面的詞是 “bee”, 你需要構建下一個前綴詞, “a bee”.

首選可能是串列, 因為很容易添加和刪除元素, 但我們同時需要使用前綴作為字典的鍵, 所以串列不合適. 而元組呢, 無法執行增刪操作, 但是可以用加法運算子構建新元組:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` 函數接收一個單詞元組, `prefix` 和一個字符串, `word`, 取 `prefix` 的除首位外的元素, 將 `word` 拼接到末尾, 構成新元組.

對於後綴集合來說, 我們需要執行的操作包括增加新後綴 (或者增加已有後綴的頻次), 以及隨機選擇後綴.

添加新後綴, 對於串列實現或者詞頻統計都同樣簡單. 而從串列中選擇隨機元素很容易, 但是從一個詞頻字典中操作, 便不太高效 (參考習題 13.7).

目前來說, 我們考慮更多的是實現難易, 但在選擇資料結構時, 還需要考慮一些其他因素. 比如運行時間. 總有一些時候, 理論上可以預期某種資料結構就是比另外一種要快. 比如, 我提到過的 `in` 運算子, 字典比串列快, 最起碼元素數量很多的時候, 一目了然.

但往往你無法提前知曉哪種實現要更快. 一種方法是, 對每種資料結構都進行實現, 然後運行比較, 看哪種更快. 這個方法叫做**基準分析**. 更實際一些的方法是選擇一種最容易實現的資料

結構, 然後看是否滿足預期. 如果滿足, 便沒必要優化了. 如果不滿足, 可以使用一些工具, 像 `profile` 模塊, 以定位程式最耗時的部分.

另一個需要考慮的因素是存儲空間. 比如, 使用詞頻字典作為後綴集合, 只會消耗很少的空間, 因為無論單詞在文中出現多少次, 每個單詞你只需存儲一次. 某些情況下, 節省空間也可以提高程式運行速度, 而且在極端情況下, 內存耗盡, 程式便無法運行. 但對於多數應用來說, 運行時間的考量要優先於存儲空間.

最後再發散一下: 在本章節, 我已暗示, 我們應該使用一種資料結構進行分析和生成. 但這二者又是兩個階段, 那麼便可以在分析時用一種資料結構, 然後在生成時, 轉換為另一種資料結構. 如果生成節省的時間超過了轉換耗費的時間, 那這種方案便有利可圖.

13.10 除錯

除錯一個程式時, 尤其是面對嚴重問題時, 一定要做好以下五個步驟:

閱讀 (Reading): 檢查代碼, 讀給自己, 看看所寫是否符合所想.

運行 (Running): 做些修改, 運行各版本程式. 通常, 如果你在程式合適的位置, 輸出合適的內容, 很容易定位問題, 不過有些時候, 你還是需要輔以腳手架代碼.

反思 (Ruminating): 花點時間多思考! 是哪種錯誤: 語法, 運行時, 還是語義錯誤? 從異常信息, 或者程式輸出中能得到什麼信息? 哪種錯誤會引發你看到的問題? 問題出現前, 你最後一次改動了什麼?

小黃鴨 (Rubberducking): 如果你向另一個人闡述問題, 有時候在問完問題之前, 便找到了答案. 通常, 你不需要找別人; 你只需要對一個橡皮鴨喃喃自語. 這便是眾所周知的**小黃鴨除錯法**的來源. 這不是我瞎編的, 你可以看`https://en.wikipedia.org/wiki/Rubber_duck_debugging`.

回撤 (Retreating): 有時候, 最好的辦法是回退, 取消最近的修改, 直到程式正常並且你能理解. 然後開始再次構建.

新手程式師有時候會在以上某種步驟中徘徊, 而忘記了其他步驟. 上面的每一步都有其不靈的時候.

比如, 遇到拼寫錯誤時, 閱讀代碼往往有效, 但如果是概念誤用, 那閱讀代碼便無濟於事了. 如果你未理解代碼的功能, 那即使你閱讀 100 次, 也永遠不會發現錯誤, 因為你理解的就有問題.

運行實例往往有用, 尤其是一些小而簡單的測試. 但是如果你不思考, 也不閱讀, 便進行實驗, 那便會陷入我稱之為 "隨機遊走編程" 的誤區中, 這種過程是隨機修改, 直至程式正常. 可想而知, 隨機遊走編程必然會耗費大量時間.

你要多花時間思考. 除錯就像實驗科學. 對於問題所在, 你至少要有一種假設. 如果有兩種或者多種可能性, 想辦法設計個測試, 排除其中可能假設.

但如果錯誤過多, 或者嘗試修復的程式過於龐大複雜, 那即使再好的除錯方案, 都會失效. 有時候, 最好的選擇便是回退代碼, 簡化程式, 直到其正常且你能理解.

新手程式師往往不願意回退代碼, 因為他們無法忍受刪除一行代碼 (即使是錯誤代碼). 可以在刪減代碼前, 將程式複製一份, 這樣會讓你感覺好一些. 在你想還原的時候, 便可以一次性複製回來.

定位嚴重的異常, 需要閱讀, 運行, 思考, 有時候還要回退代碼. 如果在某個步驟卡殼了, 試試其他步驟.

13.11 術語表

確定性 (deterministic): 給定相同輸入, 程式執行相同操作的特性.

偽隨機數 (pseudorandom): 由確定演算法生成, 只是看起來像是隨機的數字序列.

默認值 (default value): 當可選參數未被賦值, 默認提供給可選參數的值.

覆蓋 (override): 用參數替換默認值.

基準分析 (benchmarking): 在多個資料結構間抉擇的方法, 通過實現多個方案, 並用可能的輸入作為樣本進行測試.

小黃鴨除錯法 (rubber duck debugging): 通過對一個無生命對象, 比如小黃鴨, 闡述你的問題, 從而除錯的方法. 雖然小黃鴨不懂 Python, 但是將問題描述清楚, 將有助於你解決問題.

13.12 習題集

Exercise 13.9. 一個單詞的 ''排名'' 便是其在單詞串列中根據頻次排序後, 所在的位置: 最常用的排第一, 第二常用的排第二, 依次類推.

Zipf 定律 (*http://en.wikipedia.org/wiki/Zipf's_law*) 描述了自然語言中單詞排名和頻次的關係. 具體來說, 其預測排名 r 的單詞的頻次 f 為:

$$f = cr^{-s}$$

參數 s 和 c 取決於語言和文本. 如果等式兩邊取對數, 得到:

$$\log f = \log c - s \log r$$

如果繪製 $\log f$ 和 $\log r$ 的關係圖, 你會得到一個斜率是 $-s$, 截距為 $\log c$ 的斜線.

編寫程式, 讀取檔案中文本, 統計單詞頻次, 按照頻次降序, 逐行輸出每個單詞, 以及 $\log f$ 和 $\log r$. 採用繪圖程式, 繪製結果曲線, 檢驗是否構成一條直線. 能否再估計一下 s 的值?

參考: *https://thinkpython.com/code/zipf.py*. 若要運行我的代碼, 需要繪圖模塊 `matplotlib`. 如果安裝了 *Anaconda*, 默認是包括 `matplotlib` 的; 否則, 你需要安裝一下.

第 14 章　檔案

本章介紹程式將數據保存在永久存儲中的 ‘‘持久化’’ 概念, 同時展示如何使用不同型態的永久存儲, 比如檔案和資料庫.

14.1　持久化

以往我們遇到的多數程式都是一閃而過的, 因為它們運行很快, 也產生了一些輸出, 但當運行結束, 其數據便會消失不見. 如果再次運行, 又是從頭開始而已.

有些程式是**持久化的**: 其運行時間很長 (甚至一直運行); 這些程式至少會將部分數據進行永久儲存 (比如, 硬盤); 如果程式中斷後重新運行, 便可以從之前停止的地方, 繼續執行.

操作系統便是一個持久化程式的例子, 計算機一旦啟動, 便要執行諸多操作; 再比如網絡服務器, 需要一直運行, 等待接收來自網絡的請求.

程式管理數據的一個最簡單方法是讀寫文本檔案. 之前, 我們已經接觸過讀取文本檔案了, 本章將學習如何使用程式, 將文本寫入檔案.

此外, 我們也可以將程式狀態存儲到資料庫中. 本章將介紹一個簡單的資料庫, 以及模塊 `pickle`, 它可以很容易存儲程式數據.

14.2　讀和寫

文本檔案是存儲在永久介質, 比如硬盤, 閃存, 或光盤中的一系列字符. 在第 9.1節, 我們已經學習了如何打開和讀取檔案.

若要寫入檔案, 需要在打開檔案時, 用模式 `'w'` 作為第二個參數:

```
>>> fout = open('output.txt', 'w')
```

如果檔案已存在, 採用寫入模式打開檔案會清除原有內容, 重新寫入, 所以要格外謹慎! 如果檔案不存在, 則會新建一個.

`open` 函數會返回一個檔案對象, 此對象提供了各種操作檔案的方法. `write` 方法可以將數據寫入檔案.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

返回值是已寫入的字符數量. 檔案對象會跟蹤其位置, 所以如果你再次調用 write 方法, 它會在檔案末尾追加內容.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

寫入完成, 需要關閉檔案.

```
>>> fout.close()
```

如果你沒有關閉檔案, 程式運行結束, 便會自動關閉.

14.3 格式運算子

write 的參數需要是字符串, 所以, 若要向檔案寫入其他型態, 需要先轉換為字符串. 最簡單的方法是使用 str:

```
>>> x = 52
>>> fout.write(str(x))
```

另一種方法是使用**格式運算子**, %. % 在整數操作上是取余運算. 但在第一個運算對象是字符串時, % 便是格式運算子.

第一個運算對象通常是**格式字符串**, 它包括一個或者多個**格式序列**, 而格式序列則是用來聲明第二個運算對象的輸出格式的. 其運算結果便是字符串.

比如, 格式化序列 '%d' 表示第二個運算對象應該被格式化為一個十進制的整數字符串:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

結果為字符串'42', 而不是整數 42, 切勿混淆.

格式序列可以出現在字符串的任意位置, 所以你也可以在句子中嵌入某個值:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

如果字符串中有不止一個格式序列, 那麼第二個參數必須是一個元組. 每個格式序列依次對應元組中的元素.

下面的例子中, 使用'%d' 格式化整數, 使用 '%g' 格式化浮點數, 使用'%s' 格式化字符串:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

元組中元素數量要和字符串中格式序列的數量保持一致. 同時, 元素的型態也要和格式序列保持一致:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

在第一個例子中, 元素數量不一致; 第二個例子中, 元素型態不匹配.

可以從 https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting 了解更多關於格式運算子的信息. 此外, 有個功能更強大的替代方案, 便是字符串的格式化方法, 你可以從 https://docs.python.org/3/library/stdtypes.html#str.format了解更多細節.

14.4 檔案名和路徑

檔案是按照**目錄** (也叫 ‘‘檔案夾”) 來組織存放的. 每個運行的程式都有一個 ‘‘當前目錄”, 也是大多數運算的默認目錄. 比如, 當你要打開一個檔案讀取內容時, Python 會在當前目錄尋找此檔案.

os 模塊 (“os” 即 “operating system”) 提供了操作檔案和目錄的函數. os.getcwd 會返回當前目錄的名稱:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

cwd 意為 “current working directory”, 即 ‘‘當前工作目錄”. 本例中結果為/home/dinsdale, 也就是用戶 dinsdale 的主目錄.

像'/home/dinsdale' 這樣表示檔案或者目錄的字符串叫做**路徑**.

一個簡單的檔案名, 比如 memo.txt 也可以認為是一個路徑, 只是, 這是一個**相對路徑**, 因為它和當前目錄相關. 如果當前目錄是/home/dinsdale, 那麼檔案名 memo.txt 便指的是 /home/dinsdale/memo.txt 這個路徑.

以/ 開頭的路徑, 不依賴於當前目錄; 所以叫做**絕對路徑**. 若要找到一個檔案的絕對路徑, 可以使用 os.path.abspath:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path 提供了處理檔案名和路徑的其他一些方法. 比如, os.path.exists 會檢查檔案或目錄是否存在:

```
>>> os.path.exists('memo.txt')
True
```

如果存在, os.path.isdir 可以用來檢查其是否是一個目錄:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

同理, os.path.isfile 可以判斷路徑是否是一個檔案.

os.listdir 會返回指定目錄下的檔案串列 (以及其他目錄):

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

為了展示這些函數的用法, 下面的 “walks’’ 示例會遍歷一個目錄, 輸出所有檔案的名稱, 同時遞歸遍歷自己所有目錄.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join 會接收一個目錄和一個檔案名, 將其拼接為一個完整路徑.

os 模塊也提供了一個名為 walk 的函數, 和上面例子相似, 但更通用. 做個練習, 閱讀其文檔, 並使用此函數輸出給定目錄以及其子目錄下的所有檔案的名稱. 可以從https://thinkpython.com/code/walk.py下載我的答案.

14.5 捕獲異常

當你讀寫檔案時, 很多意外會發生. 如果嘗試打開一個不存在的檔案, 通常會得到 FileNotFoundError:

```
>>> fin = open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

如果你沒有權限訪問檔案, 也會異常:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

如果你嘗試打開一個目錄, 讀取內容, 會得到下面異常:

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

想要避免這些錯誤, 你可以使用 os.path.exists 和 os.path.isfile 這樣的函數, 但是會耗費大量時間, 編寫大量代碼來檢驗各種可能 (如果說 "Errno 21" 代表什麼的話, 那便是至少有 21 個異常會發生).

所以更好的做法是盡早嘗試, 盡快應對可能的異常, 而這恰恰是 try 語句的作用. 而其語法和 if...else 語句很相似:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python 會先執行 try 語句區塊. 如果一切正常, 會跳過 except 語句, 繼續執行. 如果異常發生, 會跳出 try 語句塊, 執行 except 語句塊.

使用 try 語句處理異常的方式叫做**捕獲**異常. 在上面例子中, except 部分會輸出一句錯誤信息, 但幫助不大. 通常, 捕獲異常是為了讓你有機會校正問題, 或重試程式, 或者至少優雅地結束程式.

14.6 資料庫

資料庫是用來存儲數據的檔案. 很多資料庫都是像字典一樣構建, 從鍵映射到值, 進行數據存儲. 資料庫和字典之間最大的區別, 在於資料庫存儲在磁盤 (或者其他永久存儲中), 所以程式結束後, 數據依然存在.

dbm 模塊提供了一個創建和更新資料庫檔案的介面. 下面的例子中, 我將創建一個資料庫來存儲圖片檔案的描述.

打開資料庫便如打開檔案一樣:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

模式 'c' 表示如果資料庫不存在, 則新建一個. 其結果是一個資料庫對象, 使用方式 (很多操作) 和字典一樣.

當創建新的項, dbm 便會更新資料庫檔案.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

當想要讀取其中的項時, dbm 如下讀取內容:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

結果是一個**字節對象**, 所以會用 b 開頭. 字節對象和字符串在很多地方很相似. 隨著你對 Python 的理解不斷深入, 這種差異便需要引起重視, 但是目前為止, 可以暫時忽視這些差異.

如果對一個已經存在的鍵重新賦值, dbm 會替換舊值:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

字典的某些方法, 比如 keys 和 items, 並不適用於資料庫對象. 但是可以用 for 迴圈對其進行疊代, 遍歷對象:

```
for key in db:
    print(key, db[key])
```

和操作其他檔案一樣, 用完之後, 需要關閉資料庫:

```
>>> db.close()
```

14.7 序列化

dbm 的一個限制是, 鍵和值必須是字符串或者字節. 如果使用其他型態, 會報錯.

這時候 pickle 模塊便有用武之地了. 它可以將幾乎任何型態對象轉換為字符串, 以便存儲於資料庫中, 然後用的時候再將字符串轉換回對象.

pickle.dumps 會接收一個對象參數, 返回一個字符串形式的文本 (dumps 是 “dump string” 的縮寫):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

這個格式明顯不適合人類閱讀; 但是對於 pickle 模塊, 卻容易解碼. pickle.loads(“load string”) 可以將編碼後的結果重新翻譯回來:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

雖然新舊對像的值一樣, 但 (通常) 不是同一個對象:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

換個說法, 序列化對象, 然後再反序列化的過程, 就如同複製對象一樣.

你可以用 `pickle` 來將任何非字符串數據存儲到資料庫中. 實際上, 因為這種聯動操作過於普遍, 以至於將其封裝進了 `shelve` 模塊.

14.8 管道

多數操作系統都提供了命令行介面, 一般稱為 **shell**. Shells 通常提供了大量命令, 以定位檔案系統, 以及運行應用. 比如, 在 Unix 中, 你可以用 `cd` 命令切換目錄, 使用 `ls` 命令顯示目錄下的內容, 以及 (比如) 鍵入 `firefox` 來啟動瀏覽器.

任何你能通過 shell 啟動的程式, 都可以通過 Python, 使用**管道對象**來啟動, 這個對象表示了一個運行中的程式.

比如, Unix 命令 `ls -l` 通常會用長檔案名格式來顯示當前目錄內容. 在 Python 中, 可以用 `os.popen`[1] 來運行 `ls`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

參數是個包含 shell 命令的字符串. 返回值是一個對象, 使用方式和打開的檔案對象一樣. 你可以用 `readline` 來逐行獲取 `ls` 進程的輸出, 或者用 `read` 來一次獲取全部內容:

```
>>> res = fp.read()
```

當讀取結束, 和操作檔案一樣, 需要關閉通道:

```
>>> stat = fp.close()
>>> print(stat)
None
```

返回值是 `ls` 進程的最終狀態; `None` 表示正常結束 (沒有錯誤).

比如, 多數 Unix 系統都提供了一個 `md5sum` 命令, 來讀取檔案內容, 並計算一個 "校驗值". 可以通過 `http://en.wikipedia.org/wiki/Md5`了解更多 MD5 的知識. 這個命令提供了一種有效的方法, 可以檢驗兩個檔案是否內容一樣. 不同檔案產生相同校驗值的概率極低 (可以說, 在宇宙崩塌之前都不太可能).

你可以使用管道從 Python 中運行 `md5sum`, 並獲取結果:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

[1] `popen` 模塊目前已經廢棄, 也就是說, 我們應該避免使用它, 而盡量使用 `subprocess` 模塊. 但因為 `subprocess` 模塊相對更複雜, 為了簡便, 暫時用 `popen` 模塊來講解, 直到確實不能用再說

14.9 編寫模塊

任何包含 Python 代碼的檔案, 都可以作為模塊導入使用. 例如, 假設你有一個名為 wc.py 的檔案, 檔案包含以下代碼:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

如果運行程式, 代碼會讀取自身, 並輸出檔案中內容的行數, 也就是 7. 你也可以像下面這樣導入這個模塊:

```
>>> import wc
7
```

現在你有一個模塊 wc 了:

```
>>> wc
<module 'wc' from 'wc.py'>
```

同時, 模塊對象提供了linecount 函數:

```
>>> wc.linecount('wc.py')
7
```

如此這般, 用 Python 來編寫模塊.

此處稍有瑕疵的地方在於, 上面例子中, 導入模塊時, 會運行最後一行的測試代碼. 通常, 當你導入一個模塊, 只應定義新的函數, 而不應執行它們.

如果需要將程式作為模塊導入, 通常遵循下面的寫法:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

__name__ 是一個內置變數, 程式一旦開始運行, 便被賦值. 如果程式作為一個腳本運行, __name__ 的值為'__main__'; 就像上面例子, 測試代碼便會執行. 如果程式作為模塊導入, 那__name__ 的值便不是'__main__', 那測試代碼不會執行.

做個練習, 將上面代碼寫入 wc.py 檔案中, 將其作為腳本, 直接運行. 然後運行 Python 解釋器, 並 import wc. 當作為模塊導入時, __name__ 的值是什麼?

警告: 如果你導入一個已經導入過的模塊, Python 不會做任何操作. 即使檔案變了, Python 也不會重新讀取檔案.

如果你想重載模塊, 可以使用內置函數 reload, 但這個不太靠譜, 最安全的方法應該是重啟解釋器, 然後再次導入模塊.

14.10 除錯

當讀寫檔案時, 可能會遇到一些空白導致的問題. 這些問題很難被發現, 因為空格, 製表符和換行符都是不可見的:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

這時便用到了內置函數 `repr` . 這個函數可以接收任何對象作為參數, 返回對象的字符串表示. 對於字符串來說, 它會將空白字符輸出為反斜槓序列:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

這個對於除錯來說, 非常有用.

另一個你可能遇到的難題是, 不同的系統會用不同的字符表示行尾. 有些系統用換行符`\n` 表示, 有些系統用返回字符 `\r` 表示. 有的系統又兩者都用. 如果在不同系統間移動檔案, 這些不兼容的地方, 便是引發問題之處.

對於多數系統來說, 有諸多程式可以實現格式轉換. 你可以在`http://en.wikipedia.org/wiki/Newline`上查找軟體 (以及閱讀此問題的更多細節). 或者, 你也可以自己寫一個轉換工具.

14.11 術語表

持久化 (persistent): 程式可以隨時執行, 同時至少將數據保存到永久存儲中.

格式運算子 (format operator): 一個操作符, `%`, 連接格式字符串和元組, 生成一個將元組元素格式化為指定格式的字符串.

格式字符串 (format string): 用於格式運算子的字符串, 內含格式序列.

格式序列 (format sequence): 格式字符串中的字符序列, 比如`%d`, 規定了值應被格式化為哪種格式.

文本檔案 (text file): 保存在類似硬盤一樣永久存儲介質上的字符序列.

目錄 (directory): 檔案集合的名稱, 也被稱為檔案夾.

路徑 (path): 定位檔案的字符串.

相對路徑 (relative path): 從當前目錄開始的路徑.

絕對路徑 (absolute path): 從檔案系統中的根目錄開始的路徑.

捕獲異常 (catch): 使用 `try` 和 `except` 語句來防止異常中斷程式執行.

資料庫 (database): 一個檔案, 其內容以類似字典的鍵值對應的方式來管理.

字節對象 (bytes object): 和字符串相似的一種對象.

shell: 一個程式, 允許用戶輸入命令, 然後通過啟動其他程式來執行命令.

管道對象 (pipe object): 表示一個正在運行的程式的對象, 允許 Python 程式運行命令並讀取結果.

14.12 習題集

Exercise 14.1. 編寫一個名為 sed 的函數, 接收一個模式字符串, 一個替換字符串, 以及兩個檔案名; 這個函數需要讀取第一個檔案, 然後把內容寫入第二個檔案 (如果沒有, 需要創建). 如果檔案中出現模式字符串, 則用替換字符串進行替換.

如果在打開, 讀取, 寫入或者關閉檔案過程中發生異常, 你的程式需要捕獲異常, 輸出錯誤信息, 安全退出. 參考: *https://thinkpython.com/code/sed.py*.

Exercise 14.2. 如果你從 *https://thinkpython.com/code/anagram_sets.py* 下載了習題 *12.2*的答案, 你會發現, 其創建了一個字典, 將一個有序字母字符串映射到了一個單詞串列, 串列中的單詞均由這些字母構成. 比如, 'opst' 映射著串列 ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

編寫一個模塊, 導入 anagram_sets 模塊並提供兩個新的函數: store_anagrams 可以將字母異位詞字典存儲到 *"shelf"*; read_anagrams 則可以查找單詞並返回它的字母異位詞串列. 參考: *https://thinkpython.com/code/anagram_db.py*.

Exercise 14.3. 在一個極大的 *MP3* 檔案集合中, 可能會有重複的歌曲, 可能存儲在不同目錄, 或者保存的名字不同. 這個練習的目標在於查找重複歌曲.

1. 編寫一個程式, 搜索目錄, 並遞歸其所有子目錄, 返回一個包括所有給定後綴 (比如.mp3) 的完整路徑的串列. 提示: os.path 提供了一些有用函數, 可以用來處理檔案和路徑名稱.

2. 若要識別重複檔案, 可以用 md5sum 來計算每個檔案的 "校驗值". 如果兩個檔案的校驗值相同, 則可以認為其內容相同.

3. 可以用 *Unix* 命令 diff 再次確認兩個檔案是否相同.

答案: *https://thinkpython.com/code/find_duplicates.py*.

第 15 章　類和對象

截至當前, 你應該已經知曉如何使用函陣列織代碼, 如何使用內置型態管理數據. 接下來要學習 ‘‘面向對象編程’’ 了, 這種方式是通過自定義型態來管理代碼和數據的模式. 面向對象編程是一個很大的主題; 需要花費數章, 才能講解清楚.

本章代碼範例可以從 `https://thinkpython.com/code/Point1.py` 下載; 同時, 習題答案可以從 `https://thinkpython.com/code/Point1_soln.py`下載.

15.1　自定義型態

我們已經接觸過多種 Python 的內置型態; 現在, 我們來嘗試定義一種新的型態. 舉例來說, 創建一個叫 `Point` 的型態, 用於表示二維空間中的一個點.

在數學語言中, 通常用括號中逗號分隔的坐標來表示點. 比如, $(0, 0)$ 表示起點, (x, y) 表示原點向右偏移 x 個單位, 向上偏移 y 個單位.

在 Python 中, 有幾種方式可以表示點:

- 我們可以將坐標分別保存在兩個變數中, `x` 和 `y`.
- 我們也可以將坐標作為串列或元組的元素進行存儲.
- 我們也可以創建一個新的型態, 用對象表示點.

構建一種新的型態, 相比其他選項, 要更複雜, 但是, 稍後會展示出它的威力.

這種自定義型態, 也叫做**類**. 類的定義便如下這般:

```
class Point:
    """Represents a point in 2-D space."""
```

代碼頭表示這個新類叫做 `Point`. 代碼體是一個文檔字符串, 用來描述類的用途. 你可以在類定義內部, 定義變數和方法, 稍後再來講解.

聲明一個名為 `Point` 的類, 便構建了一個**類對象**

```
>>> Point
<class '__main__.Point'>
```

因為 `Point` 在頂層定義, 所以其 ‘‘全名’’ 是 `__main__.Point`.

類對象就像是一個創建對象的工廠. 要創建一個 Point, 可以像調用函數一般, 調用 `Point`.

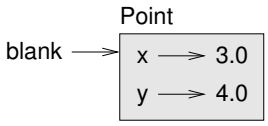

图 15.1: 對象圖.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

返回值是一個 Point 對象的引用, 同時賦值給了 `blank`.

創建一個新的對象, 叫做**實例化**, 而這個對象便是類的一個**實例**.

當你打印一個實例時, Python 會告訴你該實例所屬的類, 以及存儲在內存中的位置 (前綴 `0x` 表示其後的數字為十六進制).

每個對象都是某個類的實例, 所以 ''對象'' 和 ''實例'' 可以互用. 但是本章我用 ''實例'' 來指代我所談論的自定義型態.

15.2 屬性

可以用點標法對實例進行賦值:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

這種語法類似於從模塊中選取變數, 比如 `math.pi`, 或者 `string.whitespace`. 但在此處, 我們是使用點標法對對象中特定元素進行賦值. 這些元素叫做**屬性 (attribute)**.

作為名詞, "AT-trib-ute" 的重音在第一個音節, 而作為動詞, 則發音如 "a-TRIB-ute".

圖 15.1是展示了上面賦值結果的狀態圖. 這樣展示一個對象及其屬性的狀態圖, 也叫**對象圖**.

變數 `blank` 指向了一個包含兩個屬性的 Point 對象. 每個屬性指向了一個浮點數.

你可以基於同樣語法, 讀取屬性值:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表達式 `blank.x` 表示, ''定位 `blank` 指向的對象, 獲取 `x` 的值.'' 在上面例子中, 我們把這個表達式的值, 賦給了變數 `x`. 變數 `x` 和屬性 `x` 並不會出現衝突.

同時, 你也可以在任意表達式中使用點標法. 比如:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

你也可以將實例作為參數使用. 比如:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

print_point 會接收一個點對象作為參數, 並用數學符號來表示. 若要調用, 可以將 blank 作為參數傳入:

```
>>> print_point(blank)
(3.0, 4.0)
```

在函數內部, p 是 blank 的別稱, 所以如果函數改變了 p, 那麼 blank 也會發生變化.

做個練習, 寫個distance_between_points 函數, 使其接收兩個 Point 對象作為參數, 返回兩者之間的距離.

15.3 矩形

有時設置對象的屬性很容易, 有時又很困難. 假如你要設計一個類來表示矩形. 你會選擇什麼屬性來描述位置和大小? 暫時忽略角度, 假設矩形要麼垂直要麼水平, 從而簡化問題.

有兩種方案可選:

- 你可以確定矩形的一個角（或者中心, 以及寬度和高度.
- 你也可以聲明兩個對角的位置.

現在很難說哪種更優, 我們先實現第一種方案, 做個示例.

下面是類定義:

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

文檔字符串描述了相關屬性: width 和 height 是數字; corner 是個 Point 對象, 表示左下角位置.

要表示一個矩形, 首先要初始化一個 Rectangle 對象, 並給屬性賦值:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表達式 box.corner.x 表示, "到 box 指向的對象中, 選取名為 corner 的屬性; 然後再到 corner 指向對象中, 選取名為 x 的屬性."

圖 15.2 展示了這個對象的狀態圖. 一個對象作為另一個對象的屬性存在, 叫做**嵌套**.

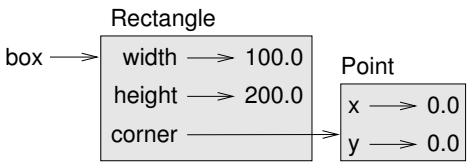

图 15.2: 對象圖.

15.4 返回實例

函數也可以返回實例. 比如, find_center 可以接收一個 Rectangle 參數, 返回一個包含 Rectangle 的中心位置坐標的 Point 實例:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

下面的例子中, 傳入了一個 box 參數, 然後將結果賦值給了 center 變數:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 對象可變

通過對屬性賦值, 可以改變對象的狀態. 比如, 若要只改變矩形大小而不改變其位置, 可以只修改 width 和 height 的值:

```
box.width = box.width + 50
box.height = box.height + 100
```

你也可以通過函數, 修改對象. 例如, grow_rectangle 函數會接收一個矩形對象和兩個數值, dwidth 和 dheight, 然後分別累加到矩形的寬和高上:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

下面展示了其使用效果:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

在函數內部, rect 是 box 的別名, 所以當函數修改了 rect 的屬性, box 也會發生變化.

做個練習, 編寫move_rectangle 函數, 使其接收一個矩形對象以及 dx 和 dy 兩個值. 同時對 corner 的 x 坐標加 dx, 對 corner 的 y 坐標加上 dy, 從而改變矩形的位置.

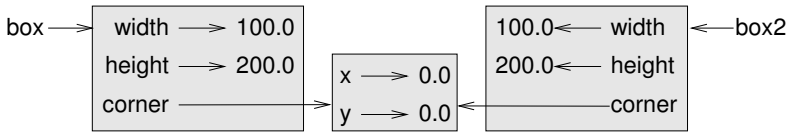

图 15.3: 對象圖.

15.6 複製

別名的使用會讓程式變得複雜, 因為一處改動可能會影響到其他地方. 同時, 追蹤指向某個既定對象的所有變數, 又很困難.

所以, 常常用複製對象來代替別名使用. `copy` 模塊包含一個 `copy` 函數, 可以複製任意對象:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` 和 `p2` 包含同樣的數據, 但它們不是同一個 `Point` 對象.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

`is` 運算子表明 `p1` 和 `p2` 不是同一個對象, 這符合我們的預期. 對於 `==`, 本來我們看兩者包含數據一樣, 預期會是 `True`. 但在此處, 令你沮喪的是, 對於實例來說, `==` 運算子的默認操作和 `is` 運算子是一樣的; 都是檢驗對象是否相同, 而不是判斷是否相等. 這是因為對於用戶自定義型態, Python 至少現在還無法知道如何衡量相等.

如果你用 `copy.copy` 來複製一個矩形, 會發現只複製了 `Rectangle` 對象, 並沒有複製內嵌的 `Point` 對象.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

圖 15.3展示了上面程式的對象圖. 這種操作叫做**淺拷貝**, 因為僅僅複製對象和其內部引用, 而不會複製內嵌對象.

多數應用中, 我們所希望的並不是這個效果. 在這個例子中, 對其中一個執行 `grow_rectangle` 函數操作, 並不會影響另一個, 但對任何一個調用 `move_rectangle` 函數, 兩者都會被影響! 這種行為令人迷惑, 也更容易使人犯錯.

幸運的是, `copy` 模塊提供了一個 `deepcopy` 方法, 不僅複製對象, 同時也會複製其引用的對象, 以及其引用對象內部引用的對象, 等等. 所以, 你可以毫不意外地稱其為**深拷貝**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 和 box 是完全隔離的對象了.

做個練習, 編寫一個 move_rectangle 新版本, 使其創建並返回新的矩形對象, 而不是修改傳入的對象.

15.7 除錯

當你開始使用對象時, 極有可能遇到新的異常. 如果試圖讀取一個不存在的屬性, 會遇到屬性異常 AttributeError:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

如若不確定對象型態, 可以如下這般查看:

```
>>> type(p)
<class '__main__.Point'>
```

你也可以使用 isinstance 來判斷對象是否是某個類的實例:

```
>>> isinstance(p, Point)
True
```

如果你不確定對象是否存在某個屬性, 可以用內置函數 hasattr, 進行判斷:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一個參數可以是任意對象; 第二參數是個字符串, 也就是某個屬性的名稱.

你也可以用 try 語句, 來判斷對象是否具有你所要的屬性:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

這種方法便於編寫適用不同情況的函數; 更多相關話題在第 17.9節會詳述.

15.8 術語表

類 (class): 用戶自定義型態. 一個類的聲明會創建一個新的類對象.

類對象 (class object): 包含自定義型態信息的對象. 類對象可用來創建類的實例.

實例 (instance): 屬於某個類的對象.

初始化 (instantiate): 創建一個新對象.

屬性 (attribute): 一種與對象關聯的命名值.

內嵌對象 (embedded object): 作為屬性存在於另一個對象內的對象.

淺拷貝 (shallow copy): 複製對象的內容, 以及內嵌對象的所有引用; 通過 `copy` 模塊中的 `copy` 函數實現.

深拷貝 (deep copy): 複製對象內容以及所有內嵌對象, 以及內嵌對象的內嵌對象, 等等; 通過 `copy` 模塊的 `deepcopy` 函數實現.

對象圖 (object diagram): 展示對象, 及其屬性, 和屬性值的圖.

15.9 習題集

Exercise 15.1. 定義一個名為 `Circle` 的類, 包含屬性 `center` 和 `radius`, `center` 是一個 *Point* 對象, `radius` 是一個數字.

初始化一個 `Circle` 對象, 用來表示一個圓, 其圓心為 (150, 100), 半徑為 *75*.

編寫`point_in_circle` 函數, 以 *Circle* 和 *Point* 為入參, 如果 *Point* 在圓內或者圓周上, 則返回 *True*.

編寫`rect_in_circle` 函數, 傳入一個 *Circle* 和一個 *Rectangle*, 如果 *Rectangle* 完全處於圓內或圓周上, 則返回 *True*.

編寫函數`rect_circle_overlap`, 接收一個 *Circle* 和一個 *Rectangle*, 如果 *Rectangle* 的任意一個頂點在圓內, 則返回 *True*. 或者寫個更有挑戰性的版本, 如果矩形有任意部分在圓內, 則返回 *True*.

參考答案: *https://thinkpython.com/code/Circle.py*.

Exercise 15.2. 編寫函數`draw_rect`, 接收一個 *Turtle* 對象和一個 *Rectangle*, 用 *Turtle* 繪製 *Rectangle*. 可以參考第 *4*節的 *Turtle* 對象的使用示例.

編寫函數`draw_circle`, 接收一個 *Turtle* 對象和一個 *Circle* 對象, 並繪製 *Circle*.

參考答案: *https://thinkpython.com/code/draw.py*.

第 16 章　類和函數

目前, 我們已經知道了如何創建新的型態, 下一步, 我們學習如何用自定義型態對象作為參數和返回值, 編寫函數. 本章將介紹 “函數式編程模式”, 以及兩種新的程式開發範式.

本章的代碼範例可以從`https://thinkpython.com/code/Time1.py`下載. 習題答案可以參考 `https://thinkpython.com/code/Time1_soln.py`.

16.1　時間

下面是一個自定義型態的示例, 我們定義了一個 `Time` 類, 用來記錄一天的時間. 類的定義如下:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

我們可以創建一個新的 `Time` 對象, 對時分秒分別定義屬性並賦值:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

`Time` 對象的狀態圖如圖 16.1所示.

做個練習, 編寫`print_time` 函數, 令其接收一個 Time 對象, 並按照時：分：秒的格式輸出. 提示: 格式序列 `'%.2d'` 會用至少兩位輸出整數, 不足兩位則前面補 0.

編寫一個名為 `is_after` 的布爾函數, 令其接收兩個 Time 對象, `t1` 和 `t2`, 如果 `t1` 晚於 `t2`, 則返回 `True`, 否則返回 `False`. 挑戰之處: 不要用 `if` 語句.

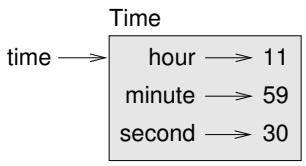

图 16.1: 對象圖.

16.2 純函數

下面章節中, 我們將編寫兩個函數, 實現時間計算. 這兩個函數展示了兩種函數型態: 純函數以及修改器. 同時, 也展示了一個我稱之為原型和補丁的開發範式, 這是一種通過構建簡單原型, 逐步改進, 從而解決複雜問題的方法.

以下為 add_time 的一個簡單原型:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

該函數新建了一個 Time 對象, 並初始化其屬性, 同時返回對象的引用. 這便稱之為純函數, 因為它不會修改作為參數傳入的任何對象, 同時也不會產生其他效果, 比如顯示值或提示用戶輸入, 而僅僅是返回一個值.

為了測試此函數, 新建兩個 Time 對象: 包含某電影開始時間的 start, 比如電影 *Monty Python and the Holy Grail*, 以及放映時長的 duration, 此處時長為 1 小時 35 分鐘.

add_time 會計算出電影結束時間.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

結果值 10:80:00, 肯定不是你想要的. 問題在於, 函數沒有處理秒數或分鐘數相加後超過 60 的情況. 遇到這種情況, 我們需要將每 60 秒進位到分鐘, 每 60 分鐘進位到小時.

優化後的版本如下:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
```

```
        sum.hour += 1

    return sum
```

這個函數雖然正確, 但是略顯臃腫. 稍後我們將看到一個簡短版本.

16.3 修改器

有時, 對於一個函數來說，直接修改其獲取的參數對象, 更為高效. 在此情境中, 更改對於調用者來說, 是可見的. 這樣的函數, 被稱之為**修改器**.

函數 `increment`, 是對 `Time` 對象增加特定秒數, 這天然便適合用修改器實現. 下面是個初稿:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

第一行執行基本操作; 其餘的則處理我們之前看到的特殊情況.

這個函數正確嗎? 如果 `seconds` 遠大於 60, 會怎樣?

這時, 只進位一次是不夠的; 我們需要重複執行, 直到 `time.second` 小於 60. 一種解決方案是用 `while` 語句替換 `if` 語句. 雖然可行, 但不夠高效. 做個練習, 編寫一個不包含任何迴圈的 `increment` 正確版本.

修改器所為, 純函數皆可為. 事實上, 有些編程語言只允許使用純函數. 有證據表明, 使用純函數比使用修改器, 開發更快, 出錯更少. 但有時修改器用來更方便, 而函數式程式則效率不高.

通常, 我建議在合理的情況下, 都使用純函數, 只有在優勢明顯時才採用修改器. 這種方法便稱之為**函數式編程模式**.

做個練習, 編寫一個 "純" 版本的 `increment` 函數, 使其創建並返回新的 Time 對象, 而不是修改其參數.

16.4 原型與規劃

剛剛展示的開發方案是 "原型和補丁". 就是針對每個函數, 編寫一個可以完成基本運算的原型, 然後對其測試, 並逐步校正錯誤.

這種方法很有效, 尤其在你尚未對問題有深入了解時. 但是增量校正往往會產生大量過於複雜代碼---因為需要關注特殊情況---和不可靠的代碼 ---因為你很難確定是否找到了所有異常.

另一種方法是採用**設計開發**, 用這種方法, 就是採用上帝視角處理問題, 從而開發會容易很多. 在這裡, 這個視角就是 Time 對象, 實際是三個六十進制的數字 (參考`http://en.wikipedia.`

org/wiki/Sexagesimal). second 屬性是 ‘‘個位”, minute 屬性是 “60 位”, hour 屬性是 “3600 位”.

當我們編寫add_time 和 increment 函數時, 實際是進行 60 位的加法, 著就是為何我們需要從當前位進位到下一位的原因.

這個發現給出了另一種解決問題的方法---我們將 Time 對象轉為整數, 然後利用計算機善於進行整數運算的優勢.

這是個將 Time 對象轉為整數的函數:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

下面函數可以將整數轉換為 Time(回憶一下 divmod, 也就是將第一個參數除以第二個參數, 將商和餘數作為元組返回).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

你可能需要費點腦力, 並運行一些測試, 以說服自己這些函數是正確的. 一種方法是針對各種 x 值, 檢查time_to_int(int_to_time(x)) == x 是否正確. 這就是一種一致性檢驗.

一旦你確信它們正確, 便可以用其重寫add_time:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

這個版本相比原來的版本, 更容易驗證. 試一下, 用time_to_int 和 int_to_time 來重寫 increment .

在某些方面, 60 進制與 10 進制的相互轉換相比時間處理要更難. 因為進制轉換更加抽象; 而時間處理更符合我們的直觀感受.

但如果我們注意到時間不過是 60 進制的數字, 並預先編寫轉換函數 (time_to_int 和 int_to_time), 便可以得到一個更精簡, 易讀, 易除錯, 也更可靠的程式.

同時這也更利於後續添加新的功能. 比如, 假如要對兩個 Time 對象相減, 獲取其時間間隔. 直接的方法, 便是通過借位實現. 但是, 使用轉換函數, 會更容易, 也更可能正確.

諷刺的是, 有時候將問題想得複雜 (或更通用), 反而會更容易解決 (因為特殊情況會更少, 出錯概率也會更低).

16.5 除錯

如果 minute 和 second 在 0 到 60 之間 (包括 0 但不包括 60), 同時 hour 是正數, 那這個 Time 對象便是正確的. hour 和 minute 應該是整數, 但 second 可以允許有小數部分.

這種約束條件, 叫做**不變式**, 因為它們需要恆為真. 換句話說, 如果它們不是真, 那肯定有某些地方出錯了.

編寫代碼來檢查不變式, 可以幫助發現錯誤並找出原因. 比如, 你可能有個函數valid_time , 它會接收一個 Time 對象, 如果違反了不變式的某個條件, 返回 False:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

在每個函數的開始部分, 你都可以檢查參數並確保它們合法:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

或者你也可以用 **assert 語句**, 檢查一個給定的不變式, 如果失敗, 拋出異常:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` 語句非常有用, 因為它們區分了何為常規條件代碼, 何為檢驗錯誤的代碼.

16.6 術語表

原型和補丁 (prototype and patch): 一種通過先寫程式初稿, 然後測試, 並校正發現的錯誤的開發方案.

設計開發 (designed development): 一種開發方案, 採用上帝視角處理問題, 相比於增量開發或者原型開發, 需要更多提前規劃.

純函數 (pure function): 一種不修改任何作為參數傳入的對象的函數. 多數純函數有返回值.

修改器 (modifier): 一種函數, 會修改作為參數接收的一個或者多個對象. 多數修改器都是沒有返回值的; 也就是說, 它們返回 `None`.

函數式編程風格 (functional programming style): 一種程式設計風格, 其中多數函數都是純函數.

不變式 (invariant): 程式執行期間, 應該始終為真的條件.

斷言語句 (assert statement): 檢查條件是否滿足, 並在失敗時拋出異常的語句.

16.7 習題集

本章代碼範例在 `https://thinkpython.com/code/Time1.py`; 同時習題答案在`https://thinkpython.com/code/Time1_soln.py`.

Exercise 16.1. 編寫`mul_time` 函數, 接收一個 *Time* 對象以及一個數字, 返回一個原始時間和數字的乘積的新時間對象.

然後用`mul_time` 來編寫一個函數, 接收一個表示比賽結束時間的 *Time* 對象, 以及一個表示距離的數字, 返回一個表示平均配速 (每英里耗時) 的 *Time* 對象.

Exercise 16.2. datetime 模塊提供的 time 對象和本章的 *Time* 對象相似, 但是前者提供了更豐富的方法和操作. 可在*http://docs.python.org/3/library/datetime.html* 閱讀相關文檔.

1. 使用 datetime 模塊編寫程式, 獲取當前日期並打印星期幾.
2. 編寫個程式, 輸入生日, 輸出用戶的年齡以及距離下個生日的天數, 小時數, 分鐘數和秒數.
3. 對於兩個生日不同的人, 總有一天, 一個人的出生天數是另一個人的兩倍. 我們稱之為 ''雙倍日''. 編寫個程式, 輸入兩個生日, 計算他們的 ''雙倍日''.
4. 進階一下, 編寫個更通用的版本, 計算某人的出生天數是另一個 n 倍大的那一天.

參看: *https://thinkpython.com/code/double.py*

第 17 章　類和方法

雖然我們已經接觸了一些 Python 的面向對象的特性, 但前面兩章的內容並不算是真正的面向對象, 因為自定義型態與操作它們的函數之間的關係, 並未涉及. 下面會將這些函數轉換為方法, 從而使其關係明晰.

本章代碼可以從 `https://thinkpython.com/code/Time2.py`下載, 同時習題答案在`https://thinkpython.com/code/Point2_soln.py`中.

17.1　面向對象特性

Python 是個**面向對象編程語言**, 也就是說, 它提供了支持面向對象編程的諸多特性, 而面向對象編程有以下特徵:

- 程式包括類和方法定義.
- 多數運算都基於對象操作而存在.
- 對象通常表示現實世界中的事物, 方法通常對應現實世界中事物的交互方式.

比如, 在第 16節定義的 `Time` 類, 對應人們記錄一天中時間的方式, 而我們定義的函數, 則對應了人們處理時間的不同策略. 類似的, 第 15節的 `Point` 和 `Rectangle` 類, 分別表示點和矩形的數學概念.

到目前為止, 我們都還沒有充分使用 Python 提供的支持面向對象編程的強大功能. 這些功能嚴格來說並非必須; 多數功能的語法我們都已經變相實現過了. 但很多時候, 面向對象的語法更加簡潔, 同時更加精確地表達程式結構.

比如, 在 `Time1.py` 中, 類定義和函數定義並無明顯關聯. 認真觀察, 便會發現每個函數都至少有一個 `Time` 對象作為參數.

透過觀察, 很自然地引出了**方法**; 一個方法便是一個與某個類相關聯的函數. 我們已經接觸過字符串, 串列, 字典和元組的方法. 本章中, 我們將構建自定義型態的方法.

方法和函數, 語義相同, 但語法有兩處差異:

- 方法定義在類中, 以使類和方法的關係一目了然.
- 調用方法的語法和調用函數的語法不同.

後面章節中, 我們將把前兩章涉及的函數轉換為方法. 這種轉換純粹是機械性的; 通過一系列步驟便可實現. 如果你可以熟練地從一種形式轉換為另一種形式, 那麼你便能夠為眼下的事情, 選擇最合適的形式.

17.2 打印對象

在第 16節, 我們定義了 Time 類, 在第 16.1節, 你編寫了print_time 函數:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

若要調用函數, 需要傳遞一個 Time 對象作為參數:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

要使print_time 成為方法, 只需要將函數定義移動到類定義內部. 同時注意縮進的變化.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

現在有兩種方式調用print_time . 第一種 (也是不常用的) 方式是使用函數語法:

```
>>> Time.print_time(start)
09:45:00
```

這裡用到了點標法, Time 是類的名稱, print_time 是方法的名稱. start 是傳入的參數.

第二種 (也是更簡潔的) 方式, 是使用方法語法:

```
>>> start.print_time()
09:45:00
```

這裡也用了點標法, print_time 也是方法的名字, start 是調用方法的對象, 也被稱為主體. 正如句子的主語是句子主體, 方法的調用主體也就是方法的主體.

在方法內部, 主體被賦值給了第一個參數, 所以在這裡 start 被賦值給了 time.

按照慣例, 方法的第一個參數稱為 self, 所以print_time 的常見寫法如下:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

這種約定是有其內在緣由的:

- 函數調用的語法, print_time(start), 表示函數是主動一方. 就像, "嘿, print_time! 這有個對象需要你打印一下."
- 在面向對象編程中, 對象是主動一方. 像start.print_time() 這種方法調用, 就像是說, "嗨, start! 請打印你自己. "

這種視角的轉換看似更優雅了, 但並沒有明顯的優勢. 在目前遇到的例子中, 看不出什麼差異. 但有時候, 將觸發者從函數轉移到對象上, 可以寫出更通用的函數 (或方法), 便於後期維護和復用.

做個練習, 將time_to_int(見第 16.4節), 重寫為方法. 你可以嘗試將int_to_time 也重寫為方法, 但沒有什麼意義, 因為沒有對象可以調用它.

17.3 另一個示例

下面是 increment(章節 16.3) 作為方法的重寫版本:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

這個版本假設 time_to_int 已經被重寫為了方法. 同時, 也要注意它是一個純函數, 而不是修改器.

下面是調用 increment 的方式:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

主體 start, 被賦值給了第一個參數 self. 實參 1337, 則被分配給了第二個參數, seconds.

這種機制讓人感覺很迷惑, 尤其是產生錯誤的時候. 比如, 如果你用兩個實參調用 increment, 會報錯:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

這個錯誤信息初看很難理解, 因為括號裡只有兩個參數. 但這個主體也被當作了一個參數, 所以一共有三個.

另外, **位置參數**是沒有參數名的參數; 也就是說, 它不是關鍵字參數. 下面函數調用中:

```
sketch(parrot, cage, dead=True)
```

parrot 和 cage 是位置參數, 而 dead 是關鍵字參數.

17.4 一個進階案例

重寫is_after(見第 16.1節) 會略微複雜, 因為它需要兩個 Time 對象作為參數. 這種情況, 通常將第一個參數命名為 self, 第二個命名為 other:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

若要使用此方法, 你需要在某個對象上調用此方法, 並將另一個對象作為參數傳入:

```
>>> end.is_after(start)
True
```

這種語法的一個好處, 是它讀起來如同英語: “end is after start?”

17.5 init 方法

init 方法 (“initialization’’ 的簡稱) 是一個特殊的方法, 在對象被實例化時才被調用. 其全稱為__init__(兩個下劃線, 然後緊跟 init, 然後又是兩個下劃線). Time 類的 init 方法如下:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

通常__init__ 方法的參數名稱和類中屬性名稱相同. 下面語句

```
        self.hour = hour
```

會將參數 hour 的值存儲為 self 的一個屬性.

這個參數是可選的, 所以如果你調用 Time 時沒有傳參, 那便會使用默認值.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

如果你提供了一個參數, 那只會覆蓋 hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

如果提供了兩個參數, 則會覆寫 hour 和 minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

如果提供三個參數, 則會覆寫三個默認值.

做個練習, 為 Point 類編寫 init 方法, 使用 x 和 y 作為可選參數, 並賦值給相應的屬性.

17.6 __str__ 方法

__str__ 和 __init__ 類似, 也是一個特殊的方法, 其一般返回對象的字符串表示.

比如, 下面是 Time 對象的 str 方法:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

當用 print 打印對象時, Python 調用的就是 str 方法:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

當編寫新類時, 我總是先寫__init__ 方法, 這樣更容易初始化對象, 同時編寫__str__ 方法, 以方便除錯.

試著為 Point 類編寫 str 方法. 創建一個 Point 對象, 並打印輸出.

17.7 運算子重載

通過定義一些特殊方法, 你可以對自定義型態, 重新定義運算子的行為. 比如, 你為 Time 類定義了__add__ 方法, 便可以用 + 運算子操作 Time 對象.

如下定義:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

可以這樣使用:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

當對 Time 對象應用 + 運算子時, Python 會調用__add__. 當打印結果時, Python 會調用__str__. 所以我們看到的, 只是冰山一角而已!

對自定義型態, 改變運算子的行為, 這便是**運算子重載**. 對於 Python 中的每個運算子, 都有一個類似__add__ 的特殊方法與之對應. 更多詳細信息, 請參考 http://docs.python.org/3/reference/datamodel.html#specialnames.

作為練習, 請為 Point 類編寫 add 方法.

17.8 型態分發

在上一節, 我們將兩個 Time 對象進行加法操作, 但是你可能想將一個整數與 Time 對象相加. 下面是一個__add__ 的新版本, 它會檢驗 other 的型態, 從而決定調用add_time , 還是increment 方法:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

內置函數 isinstance 會接收一個值和一個類對象, 如果值是這個類的實例, 則返回 True.

如果 other 是一個 Time 對象, __add__ 會調用add_time. 否則, 會認為參數是個數字, 從而調用 increment 方法. 這種操作叫做型態分發, 因為它基於參數的型態不同, 將計算任務分發給不同的方法.

下面時一個在不同型態上使用 + 運算子的例子:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

不幸的是, 這個加法的實現並不具有可交換性. 如果整數在前, 你會得到報錯:

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

問題在於, Python 並不是對 Time 對象加上一個整數, 而是對一個整數加上 Time 對象, 所以不知道如何處理了. 有個取巧的解決辦法: 用特殊方法__radd__, 表示 ‘‘右側相加”. 當 Time 對象出現在 + 運算子右側時, 將調用此方法. 以下是定義:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

下面是使用方式:

```
>>> print(1337 + start)
10:07:17
```

做個練習, 為 Points 編寫 add 方法, 使其同時適用於 Point 對象和元組:

- 如果第二個操作數是 Point 對象, 方法則返回一個新的 Point, 其 x 坐標是操作數的 x 坐標總和, y 坐標同理.
- 如果第二個操作數是元組, 則方法應該將元組的第一個元素和 x 坐標相加, 第二個元素和 y 坐標相加, 然後返回此結果的一個新 Point 對象.

17.9 多態性

當需要的時候, 基於型態分發是有用的, 但 (幸好) 不用總要如此. 通常, 你可以編寫適用不同型態參數的函數, 來避免型態分發.

我們為字符串操作編寫的很多函數, 對於其他序列型態也適用. 比如, 在第 11.2節, 我們用 histogram 來統計單詞中每個字母出現的次數.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

這個函數對於串列, 元組, 甚至字典都適用, 只要 s 中的元素是可哈希的, 就都可以作為 d 的鍵.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

能夠處理多種型態的函數, 稱為**多態**. 多態性有助於代碼復用. 例如, 可以對序列中元素求和的內置函數 sum, 也同樣適用於其他支持加法運算的元素序列.

因為 Time 對象也提供了 add 方法, 所以也同樣適用於 sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

通常, 如果函數內的所有操作適用於某種型態, 那麼這個函數便同樣適用於此型態.

最好的多態, 是那種不求而得, 就像你突然發現你寫的函數, 對於規劃之外的某種型態, 也同樣適用.

17.10 除錯

在程式運行的任何時刻, 向對象添加屬性都是可行的, 但如果對象型態相同, 屬性不同, 那很容易出錯. 所以比較好的方法是在 init 方法中初始化對象的全部屬性.

如果你不確定一個對象是否包含某個屬性, 可以適用內置函數 hasattr (見第 15.7節) 進行判斷.

另一種獲取屬性的方式, 是使用內置函數 vars, 它會接收一個對象, 並返回一個屬性名 (字符串格式) 映射到屬性值的字典:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

對於除錯而言, 你會發現保留下面的函數, 會很有用:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

print_attributes 會遍歷字典, 輸出每個屬性名稱以及其對應的值.

內置函數 getattr 會接收一個對象和一個屬性名 (字符串格式), 返回該屬性的值.

17.11 介面和實現

面向對象設計的一個目標便是令軟體便於維護, 也就是說當系統的某些部分發生了變化, 你依然可以保證程式正常運行, 從而可以修改程式, 以滿足新的需求.

實現這一目標的一個設計原則便是, 介面和實現分離. 對於對象來說, 就是類提供的方法不應該依賴於屬性的形式.

比如, 本章我們開發了一個類, 表示一天中某個時間. 這個類提供的方法包括 time_to_int, is_after, 以及add_time.

我們有多種方式實現這些方法. 實現的細節取決於我們如何表示時間. 在本章, Time 對象的屬性有 hour, minute, 和 second.

同樣的, 我們也可以用一個整數, 即從零點以來的秒數來表示這些屬性. 這種實現方式, 將會令某些方法, 比如is_after, 更容易實現, 而令某些方法更難編寫.

當你應用了一個新的類後, 你可能發現更好的實現方式. 但如果程式的其他部分正在使用你的類, 這時候修改介面, 往往比較耗時, 且容易出錯.

但如果你設計介面時足夠仔細, 你便可以在不改變介面的同時, 修改內部實現, 也就意味著程式其他部分無需變動.

17.12 術語表

面向對象語言 (object-oriented language): 提供自定義型態和方法等特性, 以方便麵向對象編程的語言.

面向對象編程 (object-oriented programming): 一種將數據和操作封裝進類和方法中的編程風格.

方法 (method): 定義在類定義中的函數, 在該類的實例上被調用.

主體 (subject): 方法調用所基於的對象.

位置參數 (positional argument): 不包括參數名稱的參數, 所以不是關鍵字參數.

運算子重載 (operator overloading): 修改類似 + 這樣運算子的操作, 從而令其適用於自定義型態.

型態分發 (type-based dispatch): 一種編程模式, 檢驗操作對象的型態, 並根據不同型態, 調用不同的函數.

多態 (polymorphic): 表示函數可以應用於多種型態的特性.

17.13 習題集

Exercise 17.1. *從https://thinkpython.com/code/Time2.py 下載本章代碼. 修改 Time 的屬性為整數, 用來表示從午夜零點計時的秒數. 然後修改方法 (以及int_to_time 函數), 以適應新的實現. 你不必修改 main 中的測試代碼. 完成修改後, 輸出應該和以前一樣. 參考答案: https://thinkpython.com/code/Time2_soln.py.*

Exercise 17.2. *這個習題是個警示故事, 涉及了 Python 中一個最常見, 卻最難發現的錯誤. 編寫一個名為 Kangaroo 的類, 需要包含下面的方法:*

1. *一個__init__ 方法, 初始化一個名為pouch_contents 的屬性, 使其為空串列.*
2. *一個名為put_in_pouch 的方法, 使其接收一個任意型態對象, 並將其放入 pouch_contents 中.*
3. *一個__str__ 方法, 返回一個字符串格式的 Kangaroo 對象和 pouch_contents 中的內容.*

創建兩個 Kangaroo 對象, 將其分別賦值給 kanga 和 roo, 然後將 roo 添加到 kanga 的pouch_contents 中, 測試代碼.

從*https://thinkpython.com/code/BadKangaroo.py*下載代碼. 代碼裡面是上面習題的一個答案, 可是其中有個又大又棘手的錯誤. 找到並修復這個錯誤.

如果你難以解決, 可以下載 *https://thinkpython.com/code/GoodKangaroo.py*, 在這裡面詳細解釋了問題所在, 並展示了一個解決方案.

第 18 章　繼承

面向對象編程中涉及最多的語言特性, 便是**繼承**. 繼承是一種基於已有類, 進行修改, 從而定義新的類的能力. 本章我將用表示撲克牌, 一副牌以及牌型的類來講解何為繼承.

如果你不玩撲克, 你可以從`http://en.wikipedia.org/wiki/Poker`入門. 不過也不必糾結, 因為我稍後會講解清楚練習所涉及的內容.

本章代碼範例在 `https://thinkpython.com/code/Card.py`.

18.1　紙牌對象

一副撲克有五十二張牌, 每個都屬於四種花色中的一種, 同時也是十三個等級牌之一. 四種花色分別為黑桃, 紅心, 方塊和梅花 (橋牌中降序排列). 等級分別是 A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q 和 K. 根據玩法不同, A 可能比 K 大, 或者比 2 小.

如果你想定義個新對象來表示一張牌, 很明顯有兩個屬性: 等級和花色. 至於屬性用何型態表示, 就不太明顯了. 一種方式是用字符串表示, 比如`'黑桃'` 表示花色, `'Q'` 表示等級. 但這種實現方式有個問題, 比較哪張牌的等級或者花色更高, 便不太容易了.

另一種方式是用整數來**編碼**等級和花色. 在這裡, "編碼" 表示我們定義一個數字和花色, 或者數字和等級之間的映射. 這種編碼並不是為了保密 (而是 "加密").

舉例來說, 下面表格展示了花色和相應整數編碼:

黑桃 (Spades)	$\mapsto$	3
紅心 (Hearts)	$\mapsto$	2
方塊 (Diamonds)	$\mapsto$	1
梅花 (Clubs)	$\mapsto$	0

通過編碼, 比較牌的大小便容易很多; 因為更高的花色對應了更高的數字, 我們可以通過比較它們的編碼來比較花色大小.

等級的映射編碼更容易明白了; 每個數字等級對應相應的整數, 對於臉牌, 編碼如下:

J(Jack)	$\mapsto$	11
Q(Queen)	$\mapsto$	12
K(King)	$\mapsto$	13

我用 $\mapsto$ 符號表示映射比較清晰明了, 但不能寫入 Python 程式中. 它們只是程式設計的一部分, 不應出現在代碼中.

`Card` 的類定義如下:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

通常, init 方法對每個屬性都接收一個可選參數. 默認的卡牌是梅花 2.

使用你想要的花色和等級, 調用 `Card` 類, 便可以創建一個新的 Card 對象.

```
queen_of_diamonds = Card(1, 12)
```

18.2 類屬性

若要以人們容易理解的方式打印 Card 對象, 我們需要將整數編碼映射到相應等級和花色. 很自然想到使用字符串串列來實現. 我們將串列賦值給**類屬性**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

像變數`suit_names` 和`rank_names` 一樣, 定義在類的內部, 但同時在方法外部的變數, 叫做類屬性, 因為它們屬於 `Card` 這個類對象.

這將其與類似 `suit` 和 `rank` 的**實例屬性**進行了區分, 因為實例屬性和特定實例相關聯.

無論何種型態的屬性, 都是通過點標法來獲取. 比如, 在`__str__` 中, `self` 是指卡牌對象, 同時 `self.rank` 是其等級. 同樣, `Card` 是個類對象, `Card.rank_names` 表示與類相關的字符串串列.

每個卡牌都有自己的 `suit` 和 `rank`, 但`suit_names` 和 `rank_names` 只有一份副本.

綜合來看, 表達式`Card.rank_names[self.rank]` 表示 "用 `self` 對象的 `rank` 屬性作為索引, 從 `Card` 類的`rank_names` 串列中, 獲取相應的字符串."

`rank_names` 的第一個元素是 `None`, 因為沒有卡牌的等級為零. 通過囊括 `None` 作為一個占位符, 從而令索引 2 恰到好處地映射到了字符串`'2'`, 其他一樣. 如果不想要這種取巧操作, 可以用字典替代串列.

利用現有方法, 我們可以創建並打印紙牌:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

圖 18.1是 `Card` 類對象和一個 Card 實例的對象圖; `Card` 是個類對象, 其型態是 `type`. `card1` 是 `Card` 的一個實例, 所以其型態是 `Card`. 為了節約空間, 我沒有畫出`suit_names` 和`rank_names` 的內容.

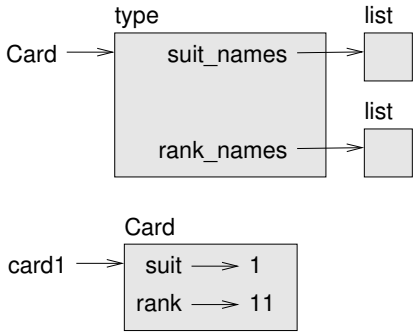

图 18.1: 對象圖.

18.3 比較卡牌

對於內置型態, 有比較運算子 (<, >, ==, 等.) 可以比較值, 並判斷一個值大於, 小於或者等於另一個. 對於自定義型態, 我們可以提供一個表示 ''小於'' 的__lt__ 方法, 來覆蓋內置運算子的操作.

__lt__ 接收兩個參數, self 和 other, 並在 self 小於 other 時返回 True.

然而卡牌的正確順序並不明瞭. 比如, 梅花 3 和方塊 2, 哪個更好? 一個等級高, 一個花色大. 若要比較兩個牌大小, 你首先需要確定等級和花色, 哪個更重要.

答案取決於玩的何種遊戲, 但簡便起見, 我們定義花色更重要, 所以所有的黑桃大於任意方塊, 以此類推.

定好規則, 我們便可以編寫__lt__ 了:

```
# inside class Card:

    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

使用元組來比較, 更加簡潔:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

做個練習, 為 Time 對象編寫__lt__ 方法. 你可以用元組進行比較, 也可以考慮比較整數.

18.4 整副牌

現在我們有 Card 類了, 下一步要定義整副牌 (Deck) 了. 既然整副牌是由卡牌構成, 那自然每副牌都應該包含一個卡牌串列作為屬性.

下面是 Deck 的類定義. init 方法新建了一個 cards 屬性, 同時生成了標準的 52 張牌:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

生成一副牌最容易的方法便是嵌套迴圈了. 外層迴圈枚舉了 0 到 3 的花色. 內層迴圈枚舉了 1 至 13 的等級. 每次疊代都會用當前花色和等級創建一張牌, 並將其附加到 self.cards 中.

18.5 打印整副牌

下面是 Deck 的__str__ 方法:

```
# inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

這個方法展示了一個拼接超大字符串的高效方法: 建立一個字符串串列, 然後使用字符串方法 join 進行拼接. 內置函數 str 會對每個 card 都應用__str__ 方法, 以返回字符串格式.

我們在換行符上調用了 join 方法, 卡牌之間被分隔成了新行. 下面是結果示例:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

雖然結果看起來是 52 行, 但它其實是一個包含換行符的長字符串.

18.6 添加, 移除, 洗牌和排序

若要發牌, 首先需要一個能夠從整副牌中移除並返回此對象的方法. 串列方法 pop 提供了一種便捷的方法:

```
# inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

由於 pop 是從串列中移除最後一張牌, 所以我們從底部發牌.

若要添加一張牌, 可以用串列方法 append:

```
# inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

像上面這樣僅使用另一個方法, 而不做過多操作, 有時被稱為 **veneer(裝飾)**. 這個比喻來自於木工行業, 在一塊便宜木頭表面貼一層優質的薄木板, 進行偽裝, 從而改善外觀.

這裡, add_card 是一個 "薄" 方法, 用卡牌術語的方式, 表達串列操作. 而這會令實現的外觀, 或者介面, 容易理解.

再舉個例子, 我們可以用 random 模塊中的 shuffle 函數, 為 Deck 編寫一個 shuffle 方法:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

不要忘記引入 random 模塊.

做個練習, 用串列方法 sort 為 Deck 編寫一個 sort 方法, 為 Deck 中的卡牌排序. sort 方法會使用我們定義的__lt__ 方法來確定順序.

18.7 繼承

繼承是一種基於已有類進行修改, 從而定義新類的能力. 比如, 我們想要一個類來表示 "手牌", 即玩家手中持有的牌. 一副手牌和整副牌相差無幾: 都是由卡牌構成, 並且都需要支持添加和移除牌等操作.

但二者也有差異: 有些手牌需要某些操作, 但整副牌卻不需要. 比如, 在撲克牌中, 我們可能需要比較兩幅手牌, 看哪個獲勝. 在橋牌中, 我們可能需要計算手牌的分數, 以便下注.

類之間這種---相似卻又不同---的關係, 非常適合使用繼承. 若要定義一個繼承自現有類的新類, 只需要將現有類的名稱放在括號中即可:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

這個定義表明 Hand 繼承自 Deck; 也就是說, 我們可以像 Deck 一樣對 Hand 應用pop_card 和add_card 方法.

當一個新類繼承自現有類, 那麼現有的類叫做**父類**, 新的類叫做**子類**.

在本例中, Hand 從 Deck 繼承了__init__ 方法, 但沒有滿足我們的需求: init 方法本應用空串列初始化 cards, 而不是用 52 張新牌.

如果我們在 Hand 類中提供了 init 方法, 那它會覆寫 Deck 類中的 init 方法:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

當你創建一個 Hand 時, Python 便會調用這個 init 方法, 而不是 Deck 中的那個.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

其他的方法從 Deck 繼承而來, 所以我們可以用 pop_card 和add_card 方法來發牌:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

很顯然, 下一步就是要把這些代碼封裝進一個叫move_cards 的方法中:

```
# inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

move_cards 方法接受兩個參數, 一個 Hand 對象, 一個發牌數量. 同時它也會修改 self 和 hand, 並返回 None.

在某些遊戲中, 卡牌會從一副手牌, 移到另一副手牌, 或者從手牌退還到牌堆中. 你可以使用move_cards 來執行這些操作: self 可以是一個 Deck, 或者一副 Hand, 儘管名字叫 Hand, 但也可以是一個 Deck.

繼承是種很有用的特性. 某些沒用繼承的重複性代碼, 使用繼承可以變得更優雅. 繼承也方便了代碼復用, 你可以通過修改父類行為, 為子類都添加行為. 在某些情況下, 繼承的結構反映了問題的真正結構, 也就使得設計更易於理解.

另一方面, 繼承會使程式變得難以閱讀. 當調用一個方法時, 有時很難搞明白它的定義在哪裡. 相關代碼可能分散在數個模塊. 此外, 許多使用繼承實現的事情, 不用繼承也能做到, 甚至做得更好.

18.8 類圖

我們已經接觸過表示程式狀態的堆疊圖, 以及展示對象的屬性以及值的對象圖. 這些圖好比程式運行時的快照, 因此也會隨著程式運行而變化.

它們通常很詳細; 但對於某些目的來說, 偏於細緻. 類圖是程式結構的一種更抽象的表達. 它不展示單獨對象, 而是表示類和類之間的關係.

類之間有多種關係:

- 一個類中的對象可以包括其他類中對象的引用. 例如, 每個 Rectangle 都包含了一個對 Point 的引用, 同時每個 Deck 包含了多個 Card 的引用. 這種關係叫做**組合 (HAS-A)**, 也就是, "一個 Rectangle 有一個 Point."
- 一個類可以繼承自其他類. 這種關係叫做**繼承 (IS-A)**, 就如, "一個 Hand 是一種 Deck."

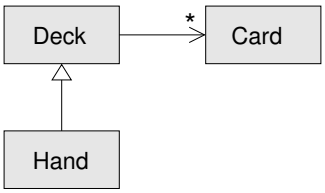

图 18.2: 類圖.

- 一個類可能依賴於另一個類, 因為類中對象可能會將其他類中的對象作為參數使用, 或者將其他類中的對象作為計算的一部分. 這種關係, 叫做**依賴 (dependency)**.

類圖是這些關係的圖形化表示. 比如, 圖 18.2 展示了 `Card`, `Deck` 和 `Hand` 之間的關係.

空心三角箭頭表示 IS-A 關係; 這裡表示 Hand 繼承自 Deck.

標準箭頭表示 HAS-A 關係; 這裡表示 Deck 包含 Card 對象的引用.

箭頭附近的星號 (`*`) 是一個**複數表達**; 表示 Deck 包含多少個 Card. 複數表達可以是一個 `52` 一樣的簡單數字, 可以是 `5..7` 一樣的一個範圍, 或者一個星號, 表示 Deck 可以有任意個 Card.

此類圖中沒有依賴關係. 依賴關係通常用虛線箭頭表示. 或者, 若有很多依賴關係存在, 它們有時候會被省略.

一個更詳細的類圖, 可能會顯示 Deck 實際上包含了一個 Card `串列`, 但是一般類圖中不包含串列和字典這些內置型態.

18.9 除錯

繼承可能會令除錯變得困難, 因為當你調用對象的方法時, 可能很難確定是調用的哪個方法.

假設你正在寫一個處理 Hand 對象的函數. 你希望適用於各種牌型, 比如 PokerHands, BridgeHands, 等等. 如果你調用像 `Shuffle` 這樣的方法, 你可能用的是 `Deck` 中定義的方法, 但如果子類重寫了這個方法, 你調用的便是子類中的方法了. 這個特性說來是好事, 但有時令人困惑.

如果你不確定程式執行流程的時候, 最簡單的方法便是在相關方法開始處添加打印語句, 輸出信息. 如果 `Deck.shuffle` 打印了一條信息, 如 `Running Deck.shuffle`, 那便可以據此來追蹤執行流程.

另一個思路, 你可以用下面的函數, 接收對象和方法名稱 (字符串表示), 然後返回提供該方法定義的類:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

下面是個示例:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class '__main__.Deck'>
```

這樣便可以知道 Hand 中的 `shuffle` 方法來自於 `Deck`.

`find_defining_class` 函數使用 `mro` 方法, 獲取類對象的串列, 進而根據方法進行搜索. "MRO" 是 "method resolution order" 的簡稱, 是指 Python"解析" 方法名時所搜索的序列.

下面是個設計建議：當你覆蓋一個方法時, 新方法的介面, 要和舊的保持一致. 它應該接收同樣的參數, 返回相同的型態, 同時遵守相同的先決條件和後置條件. 如果你遵循上述規則, 你會發現為父類實例, 如 Deck, 所設計的函數, 也同樣適用於子類實例, 比如 Hand 和 PokerHand.

如果你違反了這個 "里氏替換原理" 原則, 那代碼將很不幸地像紙牌屋一樣崩塌.

18.10 數據封裝

前面章節講述了一種 "面向對象設計" 的開發模式. 我們識別了對象--比如 `Point`, `Rectangle` 和 `Time`---同時定義類來表示這些對象. 在每個例子中, 對象和現實世界 (或者至少是數學世界) 實體之間一般都有明顯的對應關係.

但有時候很難界定所需的對象以及如何與之交互. 這種情況下, 你需要一個不同的開發模式. 之前我們通過封裝和泛化來設計函數介面, 同樣我們也可以通過**數據封裝** 來設計類介面.

第 13.8節的馬爾可夫分析, 便是一個很好的例子. 如果你從`https://thinkpython.com/code/markov.py`下載我的代碼, 你會看到使用了兩個全局變數—`suffix_map` 和 `prefix`---會被多個函數讀寫.

```
suffix_map = {}
prefix = ()
```

因為這些變數是全局的, 我們一次只能運行一個分析. 如果我們讀取兩個文本, 他們的前置和後置詞彙都會被添加到同一個資料結構中 (會生成一些有趣的文本).

若要運行多個分析, 同時保持互不影響, 我們可以把每個分析的狀態封裝到對象中. 代碼如下:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

接下來, 我們把函數轉換為方法. 下面是`process_word` 方法的示例:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

像這樣修改程式---改變設計而不改變其行為---便是另一種重構 (參見第 4.7節).

這個例子展示了一種新的設計對象和方法的開發模式:

1. 先編寫讀寫全局變數的函數 (如有必要).
2. 一旦程式可以運行, 便可尋找全局變數和使用它們的函數之間的關聯.
3. 將相關變數封裝為對象的屬性.
4. 將相關函數轉換為新類的方法.

做個練習, 從`https://thinkpython.com/code/markov.py`下載筆者的馬爾可夫代碼, 遵循上述步驟, 將全局變數封裝為新類 `Markov` 的屬性. 解決方案代碼見`https://thinkpython.com/code/markov2.py`.

18.11 術語表

編碼 (encode): 通過建立一個映射, 用另一組值來表示一組值的過程.

類屬性 (class attribute): 類對象中的屬性. 類屬性定義在類定義的內部, 但在方法外部.

實例屬性 (instance attribute): 與類實例關聯的屬性.

偽裝方法 (veneer): 一個方法或者函數, 無需太多計算便可以為其他方法提供不同的介面.

繼承 (inheritance): 定義新類的能力, 這個新類是先前定義的類的修改版本.

父類 (parent class): 子類所繼承的類.

子類 (child class): 通過繼承現有類而創建的新類; 也叫做 ‘‘派生類”.

IS-A 關係 (IS-A relationship): 子類和其父類之間的關係.

HAS-A 關係 (HAS-A relationship): 指兩個類之間, 一個類實例包含了另一個類實例的引用.

依賴 (dependency): 兩個類之間的關係, 一個類的實例使用了另一個類的實例, 但沒有將其作為屬性存儲.

類圖 (class diagram): 表示程式中各個類及其之間關係的圖解.

多倍數 (multiplicity): 類圖中一種符號, 用於表示 HAS-A 關係中一個類中對另一個類的實例引用的數量.

數據封裝 (data encapsulation): 一種開發模式, 起始用全局變數出原型, 最後將全局變數均變為實例屬性, 從而產生最終版本.

18.12 習題集

Exercise 18.1. 針對下面的程式, 繪製 *UML* 類圖, 展示其中的類和類之間的關係.

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Exercise 18.2. 編寫 *Deck* 的deal_hands 方法, 使其接收兩個參數, 分別為手牌數量以及每副手牌所含的卡牌數量. 此方法會創建相應數量的手牌對象, 給每幅手牌分配相應數量的卡牌, 然後返回一個手牌實例的串列.

Exercise 18.3. 下面是德州撲克中的可能牌型, 按照價值升序以及出現概率降序排列:

一對: 兩張同樣大小的牌

兩對: 兩對同樣大小的牌

三條: 三張同樣大小的牌

順子: 順序相連的五張牌 (*A* 牌可以表示最高也可以表示最低的牌, 所有 A-2-3-4-5 是順子, 同時 10-J-Q-K-A 也是順子, 但是 Q-K-A-2-3 不是順子.)

同花: 五張同樣花色的牌

葫蘆: 三張同樣大小的牌, 外加兩張同樣大小的牌

四條: 四張同樣大小的牌

同花順: 五張同花色的順子

本練習主要用來估算抽到各種牌型的概率.

1. 從*https://thinkpython.com/code* 下載以下檔案:

   Card.py : 本章涉及的 Card, Deck 和 Hand 類的完整版本.

   PokerHand.py : 一個表示手牌類的不完整版本, 同時包含一些測試代碼.

2. 如果運行 PokerHand.py, 它將發放七張牌, 並檢查其中是否包含同花. 先仔細閱讀代碼, 再繼續下面的操作.

3. 向 PokerHand.py 中添加一些方法, 可以叫has_pair, has_twopair, 等等. 使其判斷手牌是否滿足特定要求, 而返回 *True* 或 *False*. 你的代碼適用於任意數量的 ''手牌"(雖然 *5* 和 *7* 是常見的牌數).

4. 編寫名為 classify 的方法, 使其標識出手牌中最高價值的牌型, 並設置對應的 label 屬性. 比如, 一副七張牌的手牌, 可能包含同花和一對; 那應將其標記為 ''同花".

5. 當你確信分類方法正確無誤, 下一步便是估算各種牌型出現的概率. 在 PokerHand.py 中編寫函數, 完成洗牌, 分牌, 手牌分類, 統計各種分類出現的次數.

6. 打印一個表格, 顯示各種分類及其出現概率. 用大量手牌, 不斷運行程式, 直到輸出結果收斂於合理區間. 把你得到的結果和*http://en.wikipedia.org/wiki/Hand_rankings* 上的結果進行比較.

參考答案: *https://thinkpython.com/code/PokerHandSoln.py*.

第 19 章　利器

本書中我一個主要目標便是盡量少提 Python. 當有兩種方法可以實現同一個功能, 我會選擇一種並避免提及另一種. 有時我會將第二種方法放進習題中.

現在我們回過頭來介紹一些之前被忽略的好東西. Python 提供了一些雖非必要, 但好用的功能---你不用這些功能仍然可以寫出好代碼---但是使用了這些技巧, 有時可以編寫更簡潔, 易讀, 高效的代碼, 甚至有時能同時兼顧這三個目標.

19.1　條件表達式

我們在第 5.4節見過條件語句. 條件語句通常用來二選一; 比如:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

這個語句檢驗 `x` 是否是正數. 如果是, 則計算 `math.log`. 如果不是, `math.log` 則會拋出一個 ValueError. 為了避免程式異常退出, 我們使用一個 “NaN”, 這個符號是一個特殊的浮點數, 表示 ‘‘不是一個數字”.

我們可用一個**條件表達式**來更簡潔編寫此語句:

```
y = math.log(x) if x > 0 else float('nan')
```

你可以像讀英語一樣讀這行代碼: “y gets log-x if x is greater than 0; otherwise it gets NaN”.

遞歸函數有時候可以用條件表達式重寫. 比如, 下面是一個階乘函數的遞歸版本:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

我們可以重寫為:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

條件表達式的另一個用途是處理可選參數. 比如, 下面是 `GoodKangaroo`(參見習題 17.2) 中的 init 方法:

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

我們可以這樣改寫:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

通常, 如果條件語句的兩個分支都是簡單表達式, 要麼是被返回, 要麼是被賦給同一變數, 那麼你便可以用條件表達式替換條件語句.

19.2 串列推導式

在第 10.7節, 我們學習了 map 和 filter 模式. 比如, 下面函數接收一個字符串串列, 將字符串方法 `capitalize` 映射到每個元素上, 並返回一個新的字符串串列:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

我們可以用**串列推導式**來精簡此函數:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

方括號操作符表示我們在構造一個新串列. 方括號中的表達式, 指定了串列中的元素, 同時 `for` 語句聲明了我們要遍歷的序列.

串列推導式的語法略顯奇怪, 主要因為這個迴圈變數, 如例子中的 `s`, 在定義之前, 便出現在了表達式中.

串列推導式也可以用來過濾. 比如, 下面函數僅選擇 `t` 中大寫字母的元素, 並返回新串列:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

我們可以用串列推導式來重寫:

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

串列推導式更加簡潔, 也更易讀, 至少對於簡單表達式是這樣的. 而且, 它們相比同樣的 for 迴圈, 要更快, 甚至有時快很多. 所以如果你因為我沒有更早提及而生氣, 我理解.

但是, 我要申辯一下, 串列推導式通常更難除錯, 因為你不能在迴圈內部放置一個打印語句. 我建議你只在計算足夠簡單的情況, 也就是你上手便能正確完成的時候使用. 對於新手來說, 盡量別用.

19.3 生成器表達式

生成器表達式和串列推導式相似, 但它使用圓括號而不是方括號:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

結果是一個生成器對象, 同時其知曉如何遍歷一個值序列. 但是與串列推導式不同, 它不會立即計算所有值; 而是在被調用時才計算. 內置函數 `next` 會從生成器獲取下一個值:

```
>>> next(g)
0
>>> next(g)
1
```

當抵達了序列末尾, `next` 會拋出一個 StopIteration 異常. 你也可以使用 `for` 迴圈來遍歷這些值:

```
>>> for val in g:
...     print(val)
4
9
16
```

生成器對象會跟蹤序列中的位置, 所以 `for` 迴圈會在 `next` 停止的地方開始. 一旦生成器被耗盡, 依然會拋出 `StopIteration`:

```
>>> next(g)
StopIteration
```

生成器表達式通常和 `sum`, `max`, 以及 `min` 等函數一起使用:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any 和 all

Python 提供了一個內置函數, `any`, 其接收一個布爾值序列, 如果任意一個值為 `True`, 則返回 `True`. 通常適用於串列:

```
>>> any([False, False, True])
True
```

但也多用於生成器表達式:

```
>>> any(letter == 't' for letter in 'monty')
True
```

這個例子不太明顯, 因為和 `in` 運算子的效果一樣. 但我們可以用 `any` 來重寫第 9.3節中, 曾經寫的一些搜索函數. 例如, 我們可以如下改寫 `avoids`:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

這個函數可以像讀英語那樣理解, "`word` avoids `forbidden` if there are not any forbidden letters in `word`. (如果單詞中不包含任何禁用字母, 那麼單詞便避免被禁用)"

將 `any` 和生成器表達式一起使用, 效率更高, 因為只要遇到 `True` 值, 它便會立刻停止, 所以不會計算整個序列.

Python 還提供了另一個內置函數, `all`, 如果序列中每個元素都是 `True`, 那麼它便會返回 `True`. 作為練習, 用 `all` 重寫第 9.3節的`uses_all` 函數.

19.5 集合 (set)

在第 13.6節, 我用字典來尋找出現在文檔中, 但是不在單詞串列中的單詞. 該函數接收兩個參數, 包含文檔中單詞的參數 d1(作為鍵), 和包含單詞串列的參數 d2. 最後返回一個字典, 其鍵存在於 d1 中, 但不包含在 d2 中.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

在這些字典中, 值都是 None, 因為從沒有使用它們. 結果就是我們浪費了一些存儲空間.

Python 提供了另一種內置型態, 叫做 set, 其效果很像字典中沒有值的鍵的集合. 向集合中添加元素是很快的; 檢查成員也很快. 同時集合也提供了執行常見操作的方法和運算子.

比如, 集合的差集操作有 difference 方法, 或者-操作符. 所以我們可以將 subtract 像下面一樣重寫:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

結果是一個集合, 而不是字典, 但對於像疊代這樣的操作, 其行為是相同的.

本書中的一些練習題可以用集合進行精簡和優化. 比如, 下面是習題 10.7中, 使用字典解決has_duplicates 的一個方案:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

當一個元素首次出現, 則被加入字典. 如果同一元素再次出現, 函數返回 True.

使用集合, 我們可以如下一般, 實現相同的函數:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

一個元素在集合中只能出現一次, 所以如果 t 中有元素出現多次, 那麼集合長度必然小於 t 的長度. 如果沒有重複的, 集合大小和 t 的大小相同.

我們也可以使用集合完成第 9節的某些習題. 比如, 下面是一個使用迴圈的uses_only 實現版本:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

uses_only 會檢查 word 中所有字母是否都在 available 中. 我們可以像這樣重寫:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

運算子<= 會檢驗一個集合是否是另一個的子集, 包括兩者相等的情況, 也就是如果 word 中的字母都出現在 available, 則返回真.

做個練習, 使用集合重寫avoids.

19.6 計數器 (Counter)

Counter(計數器) 類似於集合, 不同之處在於如果一個元素出現多次, 計數器會跟蹤其出現次數. 如果你熟悉數學概念中的**多重集合**, 便會發現計數器是表示多重集合的一種自然方式.

計數器定義在 collections 標準模塊中, 所以你需要先導入. 你可以用字符串, 串列或者支持疊代的任意型態來初始化計數器.

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

計數器和字典在表現上相似之處頗多; 它們都是將每個鍵映射到其出現次數. 和字典一樣, 鍵必需是可哈希的.

與字典不同, 如果你訪問一個不存在的元素, 計數器不會拋出異常, 而是返回 0:

```
>>> count['d']
0
```

我們可以用計數器來重寫習題 10.6中的is_anagram 函數:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

如果兩個單詞是換位詞, 那麼它們的相同字母具有相同的數量, 因此它們的計數器是等價的.

計數器提供了一些類似集合操作的方法和運算子, 包括方法, 減法, 併集和交集. 此外, 還提供了一個常用的方法, most_common, 這個方法會返回一個按照最常見到最不常見排序的值-頻率對串列:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 默認字典 (defaultdict)

collections 模塊同時提供了 defaultdict(默認字典), 其與字典類似, 不同之處在於, 如果你訪問一個不存在的鍵, 它可以即刻生成一個新值.

當你創建默認字典時, 你需要提供一個函數, 以用於創建新值. 用來創建對象的函數, 有時也稱為**工廠**. 創建串列, 集合, 以及其他型態的內置函數, 都可以稱為工廠:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

注意這個函數的參數 list, 是個類對象, 而不是一個新串列 list(). 而且只有在訪問不存在的鍵時, 才會調用提供的函數.

```
>>> t = d['new key']
>>> t
[]
```

新的串列, 也就是 t, 便被加入了字典中. 所以, 如果我們修改 t, d 也會相應變化:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

如果你要創建一個值為串列的字典, 可以使用 defaultdict 來簡化代碼. 在習題 12.2的答案 (可以從https://thinkpython.com/code/anagram_sets.py獲取) 中, 我創建了一個字典, 該字典將一個有序字母的字符串映射到了一個由這些字母構成的單詞的串列. 比如, 'opst' 映射了串列 ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

下面是原始代碼:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

這裡可以用 setdefault 來簡化代碼, 你可能已經在練習 11.2中使用過:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

但此方案也略有不足, 其不論是否必要, 每次都會創建新串列. 對於串列來說, 尚無關緊要, 但如果工廠函數很複雜, 那影響就很大了.

我們可以用 defaultdict 來避免這個問題, 並簡化代碼:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

對於習題 18.3, 你可以從 https://thinkpython.com/code/PokerHandSoln.py下載我的方案, 在函數has_straightflush 中使用了 setdefault. 這個方案也有缺陷, 不論是否需要, 每次迴圈都會創建 Hand 對象. 作為一項練習, 試試用 defaultdict 來重寫它.

19.8 命名元組

許多簡單對象基本都是相關值的集合. 比如, 第 15節的 Point 對象包含兩個數值, x 和 y. 當你像下面一樣定義類時, 通常會從 init 方法和 str 方法開始:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

傳遞如此少的信息, 竟使用了如此多的代碼, 殺雞用了牛刀. Python 提供了一個更加簡潔的方法實現同樣的事情:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

第一個參數是你要創建的類的名稱. 第二個參數是 Point 對象應具有的屬性串列, 都用字符串表示. 最後, namedtuple 的返回值是一個類對象:

```
>>> Point
<class '__main__.Point'>
```

Point 自動構建了 __init__ 和 __str__ 這樣的方法, 所以你無需自己編寫它們.

若要創建 Point 對象, 只需要將 Point 類作為函數使用:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

init 方法將參數賦值給你命名的屬性. str 方法會輸出字符串格式的 Point 類及其屬性.

你可以通過名字訪問命名元組的元素:

```
>>> p.x, p.y
(1, 2)
```

也可以將命名元組作為一個元組來操作:

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

命名元組提供了一種定義簡單類的快捷方式. 不足之處在於, 簡單的類不會一直簡單下去. 後續你可能想給一個命名元組添加方法. 遇到這種情況, 你可以定義一個繼承自命名元組的新類:

```
class Pointier(Point):
    # add more methods here
```

或者你可以回到傳統的類定義方案.

19.9 收集關鍵字參數

在第 12.4節, 我們已經學到如何編寫將參數收集到元組的函數:

```
def printall(*args):
    print(args)
```

你可以用任意數量的位置參數 (即無關鍵字的參數) 來調用此函數:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

然而 * 運算子不會收集關鍵字參數:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

若要收集關鍵字參數, 可以使用 ** 運算子:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

你可以將關鍵字收集參數命名為任意名稱, 但一般常用 kwargs 命名. 其結果便是一個從關鍵字映射到值的字典:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

同樣, 如果你有一個映射關鍵字和值的字典, 也可以在調用函數時, 使用發散運算子 **:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

如果沒用發散運算子, 函數會將 d 當作一個位置參數, 所以便會將 d 賦值給 x, 同時報錯, 因為沒有對 y 進行賦值:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

當你需要處理具有大量參數的函數時, 創建並傳遞聲明了常用關鍵字的字典, 會比較高效.

19.10 術語表

條件表達式 (conditional expression): 基於條件, 進行二選一抉擇的表達式.

串列推導式 (list comprehension): 一種方括號內包含 for 迴圈, 產生新串列的表達式.

生成器表達式 (generator expression): 一種圓括號內包含 for 迴圈, 從而產生生成器對象的表達式.

多重集合 (multiset): 一種表示集合中元素與其出現次數的數學概念.

工廠 (factory): 用來創建對象的函數, 通常作為參數傳遞.

19.11 習題集

Exercise 19.1. 下面是個遞歸計算二項式係數的函數.

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

使用嵌套條件表達式優化函數體.

提示: 因為這個函數最終會不斷重複計算相同的值, 所以不夠高效. 你可以使用快取模式 (參見第 *11.6*節) 來提升其效率. 但是你會發現, 一旦使用了條件表達式, 快取實現便變得困難.

第 A 章　除錯

在除錯階段, 你要對不同型態錯誤進行區分, 從而快速定位:

- 語法錯誤多出現在解釋器翻譯源碼為字節碼的時候. 通常表示程式結構有缺陷. 比如: `def` 語句後面遺漏了冒號, 會出現 `SyntaxError: invalid syntax` 這樣的信息.
- 運行時異常多在程式運行過程中, 因解釋器出錯而產生. 多數運行時異常信息包括異常發生的位置以及所執行的函數. 比如: 一個無盡的遞歸, 終會導致運行時異常, 出現 "maximum recursion depth exceeded'' 報錯信息.
- 語義異常多指程式運行不報錯, 但結果不正確. 比如: 表達式沒有按你期望的順序執行, 從而產生了錯誤的結果.

除錯的第一步, 是明確你面對的是何種異常. 雖然下面章節都是按照異常型態所組織, 但有些技巧也適用於其他情況.

A.1　語法異常

如果你能弄清程式的語法異常是什麼, 便很容易修復. 不幸的是, 錯誤提示往往無用. 常見的多是 `SyntaxError: invalid syntax` 和 `SyntaxError: invalid token`, 都是大而無用的籠統概括.

但另一方面, 錯誤信息確實告訴了你程式異常發生的位置. 實際上, 是告知你 Python 所認為的異常所在, 但不一定就是錯誤確實發生的地方. 有時候, 錯誤信息先於錯誤的位置信息出現, 通常在前一行. 如果你是逐步構建程式, 你便很容易定位錯誤所在. 其通常出現在你最後添加的代碼中.

如果你正從書中複製代碼, 從一開始, 便需要仔細比較你的代碼和書中的代碼. 檢查每個字符. 同時, 要謹記, 書中可能有錯誤, 所以你若看到了某些語法異常的代碼, 那可能真是語法異常.

下面是一些避免產生常見語法錯誤的方法:

1. 確保你沒有用 Python 關鍵字作為變數名.
2. 檢查每個複合語句的頭部末尾是否都寫了冒號, 包括 `for`, `while`, `if`, 和 `def` 語句.
3. 確保代碼中所有字符串都有成對的引號. 確保所有的引號都是 ''直引號", 而不是 ''彎引號".

4. 如果你有三個引號 (單引號或者雙銀行) 包起來的多行字符串, 確保正確終止了字符串. 未終止的字符串可能會在程式末尾引發 `invalid token` 異常, 或者可能會將程式餘下部分當作字符串, 直至碰到下個字符串的標識. 而在第二種情形下, 可能不會產生錯誤信息!
5. 未封閉的開放運算子—(, {, 或 [---會使 Python 將下一行作為當前語句的一部分處理. 通常, 下一行立刻會報錯.
6. 檢查條件表達式中是否使用了經典的 `=`, 而不是 `==`.
7. 檢查縮進是否都對齊. Python 可以使用空格和製表符, 但如果你混用, 很可能會引發問題. 避免這種問題的最好方式, 是使用了解 Python 並能生成一致性縮進的文本編輯器.
8. 如果代碼中有非-ASCII 字符 (無論字符串還是注釋), 都會引發一場, 即使 Python 3 通常可以處理非-ASCII 字符. 所以你從網頁或者其他地方複製文本的時候要當心.

如果還不起作用, 那進行下一步操作吧...

A.1.1 不斷校正, 卻毫無作用.

如果解釋器說有個錯誤, 但你總是找不到, 那可能是因為你和解釋器看到的不是相同的代碼. 檢查編碼環境, 確保你編輯的便是 Python 運行的程式.

如果你不確定, 則可以在程式開始的地方, 放個明顯且可控的異常. 然後再次運行, 如果編譯器沒有報錯, 那你便不是在運行最新的代碼.

下面是幾個可能的元兇:

- 修改了檔案, 但再次運行前忘記保存更改. 有些編程環境可以自動保存, 但有些不會.
- 修改了檔案名, 但仍用舊的檔案名運行.
- 開發環境設置不正確.
- 如果你編寫了個模塊, 並使用了 `import` 語句, 一定要確保沒有和標準 Python 模塊同名.
- 如果用 `import` 來加載模塊, 要記住, 對於修改過的檔案, 要重啟編譯器或者使用 `reload` 重載模塊. 如果你只是再次用 `import` 加載模塊, 那不起作用.

如果依然囿於困境, 不知緣由, 一種方法是從一個類似 “Hello, World!” 的新程式開始, 重新構建, 並確保該程式可以正常運行. 然後將原來的程式代碼, 逐步遷移到新的程式中.

A.2 運行時異常

如果你的程式語法都正常, Python 便可以讀取並至少可以運行了. 那可能會出現什麼問題呢?

A.2.1 我的程式什麼也不做.

這種情況, 往往由於你的檔案雖然有函數和類, 但卻沒有調用函數來執行程式. 這也可能是你有意為之, 因為你只是需要導入模塊, 提供類和函數而已.

如果不是有意而為, 那便要確保程式中有調用函數的地方, 同時保證執行流程能觸及函數調用 (參見下面的 “執行流程”).

A.2.2 程式掛起.

如果一個程式一直不運行, 看起來似乎什麼也沒做, 那便是程式 ‘‘掛起’’ 了. 通常是指程式陷入了無限迴圈或者無盡遞歸中.

- 如果你對程式中某個迴圈有所疑慮, 可以在迴圈開始前添加一個 `print` 語句, 打印 ‘‘進入迴圈’’, 在迴圈結束的地方, 添加一個打印 ‘‘退出迴圈’’ 的語句.

  運行程式, 如果你只得到第一個信息, 而第二個沒有出現, 那便是陷入了無限迴圈. 可以跳轉到下面的 ‘‘無限迴圈’’ 章節, 深入理解.

- 多數情況下, 無盡遞歸會令程式運行運行一段時間後, 產生 “RuntimeError: Maximum recursion depth exceeded(運行時錯誤: 超出最大遞歸深度)’’ 的錯誤. 如果不幸如此, 前往後面的 ‘‘無盡遞歸’’ 章節, 有更深入的介紹.

  如果你沒有遇到這種錯誤, 但是仍然懷疑某個遞歸方法或函數, 存在問題, 你仍然可以使用 ‘‘無盡遞歸’’ 章節中的技巧進行排查解決.

- 如果上面步驟都無效, 那便需要檢查其他迴圈以及遞歸的函數和方法了.

- 如果仍然無效, 那可能是你對於程式中的執行流程不太清楚. 可以參看後面的 ‘‘執行流程’’ 章節.

無限迴圈

如果你認為程式中存在無限迴圈, 同時知道出問題的迴圈所在, 那便可以在迴圈的末尾添加 `print` 語句, 輸出條件中變數的值以及條件的結果.

比如:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

現在運行程式, 你會看到每次迴圈都有三行輸出. 最後一次迴圈時, 條件應該是 `False`. 如果迴圈繼續執行, 那你便可以看到 `x` 和 `y` 的值, 同時你便會發現其值未被正確更新的原因.

無盡遞歸

多數情況下, 無盡遞歸會令程式運行片刻, 然後產生 `Maximum recursion depth exceeded`(超出最大遞歸深度) 的異常.

如果你懷疑某個函數存在無盡遞歸, 那要確保函數有基線條件. 也就是有條件令函數可以返回而不是繼續遞歸調用. 如果沒有基線條件, 那你需要重構演算法, 定義一個基線條件.

如果有基線條件, 但是程式看起來沒有觸及, 可以在函數起始位置添加 (`print` 語句, 輸出參數. 那當你運行程式時, 每次函數調用, 你都會看到幾行輸出, 同時, 也會看到參數值. 如果參數沒有趨向於基線變化, 你便大概可以明確問題所在.

執行流程

如果你不確定程式的執行過程是如何流轉, 可以在每個函數的開頭, 添加 `print` 語句, 輸出像 “entering function(進入函數)`foo`” 這樣的信息, 信息中的 `foo` 是相應的函數名稱.

現在運行程式, 隨著函數調用, 會輸出每個函數的軌跡.

A.2.3 運行程式, 得到異常.

如果運行時出現問題, Python 會輸出包括異常名稱, 問題發生的代碼行號, 以及回溯信息.

回溯信息會顯示當前運行函數, 以及調用它的函數, 以及更上層的函數, 等等. 換而言之, 它會跟蹤導致出現問題的函數調用序列, 包括每個調用語句所在的行號.

首先要檢查程式中異常發生的位置, 看看能否找到問題所在. 下面是一些常見的運行時錯誤:

名稱異常 (NameError): 嘗試使用當前環境不存在的變數. 檢查名稱拼寫是否正確, 或者前後是否一致. 同時要謹記局部變數是局部的; 你不能在函數定義的外面引用它們.

型態異常 (TypeError): 有幾種可能原因:

- 使用了不正確的值. 比如: 對字符串, 字典或元組進行索引時, 沒有使用整數.
- 格式化字符串時, 傳入了不匹配的項. 如果項數不匹配, 或進行無效轉換, 都會出現這個問題.
- 向函數傳遞了錯誤數量的參數. 對於方法來說, 查看方法定義, 檢查其首個參數是否是 `self`. 然後看方法調用; 確保是在正確型態的對象上調用方法, 並正確提供了其他參數.

鍵異常 (KeyError): 嘗試用字典沒有的鍵獲取字典的元素. 如果鍵是字符串, 要注意區分大小寫.

屬性異常 (AttributeError): 嘗試訪問不存在的屬性或方法. 仔細檢查拼寫! 你可以用內置函數 `vars` 來列出現有屬性.

如果一個屬性錯誤中, 提到某對象是 `NoneType`, 也就表示其本身是 `None`. 所以問題不是出在屬性名上, 而在於對象本身.

對象是 `None` 的原因, 多是因為忘記了給函數返回一個值; 如果一直運行到函數末尾, 都沒有 `return` 語句, 那函數便會返回 `None`. 另外一個常見的原因是, 直接使用了某些串列方法的結果, 比如 `sort`, 也會返回 `None`.

索引異常 (IndexError): 用來訪問串列, 字符串或元組的索引, 超過了其長度減一. 在錯誤發生位置之前, 添加 `print` 語句, 顯示索引的值和陣列長度. 看看陣列長度是否正確? 索引是否正確?

Python 除錯器 (`pdb`) 對於追蹤異常非常有用, 因為它可以讓你檢查異常發生前, 程式的狀態. 你可以閱讀`https://docs.python.org/3/library/pdb.html`, 來深入了解 `pdb`.

A.2.4 添加 print 太多, 我已混亂.

使用 print 語句除錯程式的一大難題在於, 你將被大量輸出所淹沒. 有兩種解決途徑: 簡化輸出, 或簡化程式.

若是簡化輸出, 你可以移除或者注釋掉無用的 print 語句, 或者將其合併, 或者格式化為便於理解的格式.

若要簡化程式, 可以從以下幾個方面著手. 首先, 縮減程式所解決問題的規模. 比如, 如果你要檢索串列, 用小的串列. 如果程式是從用戶獲取輸入, 那便提供一個引發問題的最簡輸入.

第二, 清理程式. 移除無用代碼, 重新組織程式, 使其便於理解. 例如, 如果你懷疑問題出在程式的深層嵌套中, 嘗試用簡單結構優化這部分. 如果你懷疑問題源於一個臃腫函數, 嘗試將其拆分為小函數, 並分別測試.

通常尋找最小測試用例的過程, 會引導你發現問題. 如果你發現程式在某個條件下運行正常, 但是另外條件下, 不正常, 這就給了你解決所面臨問題的線索.

同樣, 重寫代碼可以幫你發現一些微妙的錯誤. 如果你改變了一些代碼, 以為不會影響程式, 但現實是影響到了, 這也在某方面能給你一些提示.

A.3 語義錯誤

在某些方面, 語義錯誤是最難除錯的, 因為解釋器無法提供錯誤的有效信息. 只有你才知道程式應該如何運行.

第一步是在程式代碼和你之所見之間, 建立連接. 你需要先假設程式做了某些事情. 令除錯如此困難的一方面, 在於計算機運行太快了.

你會期望可以減慢程式運行速度, 從而滿足人類的速度, 確實有些除錯器可以滿足你. 但相比較設置除錯器, 插入或移除斷點, 以及 ''步進'' 程式到錯誤出現的位置, 其所耗費時間, 不如放置一些恰到好處的 print 語句, 更加快捷.

A.3.1 我的程式不能運行.

你應該問問自己這些問題::

- 程式是否有應該執行, 卻沒有執行? 找到不滿足要求的代碼部分, 使其符合預期.
- 是否有不該發生的, 卻發生了? 找到運行此功能的代碼, 看看是否不該執行的時候, 卻運行了.
- 是否有部分代碼運行不符合預期? 確保你完全了解了這些代碼, 尤其是涉及到調用其他模塊的函數或方法. 仔細閱讀所調用函數的文檔. 嘗試編寫一些簡單的測試用例, 並檢查結果是否符合預期.

編程之前, 你需要先構建程式運行的邏輯模型. 如果你寫的程式運行不符合預期, 通常問題不在代碼上; 而在於你腦海中的模型.

校正邏輯模型的最好方式, 是將程式拆解為組件 (通常是函數和方法), 同時單獨測試每個部件. 一旦遇到模型和實際不符, 你便能快速解決問題.

當然, 你需要在開發程式時構建並測試組件. 如果遇到問題, 也應該只有少量的新代碼, 令你無所適從.

A.3.2 表達式複雜卻不合預期.

編寫複雜的表達式, 代碼長一些, 但可讀性高的話, 也沒問題. 只是除錯就不方便了. 通常更好的做法是, 將複雜表達式拆解為一系列對臨時變數賦值的語句.

比如:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

可以寫成這樣:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

顯然這個版本更易於閱讀, 因為變數名稱提供了額外的文檔說明, 同時它也便於除錯, 因為你可以檢查中間變數的型態和值.

對於複雜表達式, 另一個問題在於執行順序不一定是所預期的. 例如, 你想將表達式 $\frac{x}{2\pi}$ 轉為 Python 代碼, 你可能會寫成這樣:

```
y = x / 2 * math.pi
```

這是不正確的, 因為乘法和除法是同樣的優先級, 所以會從左向右執行. 所以表達式計算的是 $x\pi/2$.

除錯表達式的一個好策略是添加括號來明確執行順序:

```
 y = x / (2 * math.pi)
```

無論什麼時候, 只要你不確定執行順序, 便可以使用括號. 不僅能確保程式正確 (按期望順序執行), 同時對於未能記憶運算子優先級的人來說, 也便於閱讀.

A.3.3 函數未返回期望結果.

如果你的 `return` 語句是個複雜的表達式, 那你很難在返回之前打印結果. 不過, 你依然可以用臨時變數. 比如, 將:

```
return self.hands[i].removeMatches()
```

替換成:

```
count = self.hands[i].removeMatches()
return count
```

現在你可以在返回之前, 顯示 `count` 的值了.

A.3.4 我無能為力了, 我需要幫助.

首先, 遠離電腦幾分鐘. 電腦會發出影響大腦的神奇電波, 造成以下症狀:

- 沮喪和憤怒.
- 迷信 (“電腦討厭我”) 和幻覺 (“我反戴帽子, 程式才能正常運行”).
- 隨機遊走編程 (企圖編寫所有可能的程式, 從中選擇正確執行的那個).

如果你發現自己出現了上述症狀, 起來走走. 當你冷靜下來後, 再去思考程式. 想想它在做什麼? 造成這種現象的原因可能是什麼? 上次程式正常運行是什麼時候, 以及後來做了什麼?

有時候, 你需要點時間才能找到問題. 我通常會在遠離電腦, 讓思緒漫遊時發現問題. 一些找到問題的絕佳之地, 是火車上, 淋浴間, 以及入睡前的床上.

A.3.5 不, 我真的需要幫助.

問題時有發生. 即使最好的程式師, 偶爾也會陷入困境. 有時候你專注在程式上的時間太長了, 以致於無法發現錯誤. 你需要的是一雙重新審視的眼睛.

在你向他人尋求幫助之前, 要確保你做好了準備, 你的程式要盡量簡單, 並且程式在盡量少的輸入下, 可以產生錯誤. 你應該在恰當的位置加入 `print` 語句 (同時輸出的信息也要便於理解). 你要對問題足夠了解, 從而能夠簡明扼要地描述清楚.

當你找來了他人幫忙, 要確保能提供他們需要的信息:

- 如果有錯誤信息, 錯誤是什麼, 程式哪部分產生的?
- 錯誤出現之前, 你最後做了什麼? 你寫的最後一行代碼是什麼, 或者最早產生異常的測試用例是什麼?
- 到目前為止, 你都嘗試了什麼方法, 以及你都發現了什麼?

當你找到錯誤時, 可以花時間想想, 如何能更快地找到它. 下次, 你遇到相似情形, 你便可以更快地定位問題.

要記住, 目標不僅僅是讓程式正常工作, 更是要學會如何令程式正常工作.

第 B 章　演算法分析

本附錄選自 Allen B. Downey 的 *Think Complexity* 一書, 由 O'Reilly Media (2012) 出版. 當你讀完此書, 可以讀讀它.

演算法分析是電腦科學的一個分支, 主要研究演算法的性能, 特別是其運行時間和所需空間. 請參閱 `http://en.wikipedia.org/wiki/Analysis_of_algorithms`.

演算法分析的實際目標是預測不同演算法的性能, 以指導設計決策.

2008 年美國大選期間, 候選人 Barack Obama 在訪問谷歌時, 被要求進行即興問題分析. 首席執行官 Eric Schmidt 開玩笑地問他, "對一百萬個 32 位整數排序的最高效的方法是什麼." 很顯然, 有人已經提前告知了 Obama, 因為他立刻回答道, "我認為冒泡排序起碼是最差的方法." 詳見`http://www.youtube.com/watch?v=k4RRi_ntQc8`.

確實如此: 冒泡排序概念上簡單, 但是對於大數據集來說, 速度很慢. Schmidt 所期望的答案可能是是 "基數排序"(`http://en.wikipedia.org/wiki/Radix_sort`). [1].

演算法分析的目標是在演算法之間進行有意義的比較, 但面臨一些問題:

- 演算法的相對性能, 可能取決於硬件的特性, 所以一個演算法在機器 A 上可能更快, 但另一個演算法可能在機器 B 上更快. 通常的方案是指定一個**機器模型**, 並分析在給定模型下, 演算法所需的步驟或運算的次數.
- 演算法的相對性能, 也可能取決於數據集的細節. 比如, 數據已被部分排序, 則某些排序演算法運行更快, 但有些演算法則會運行較慢. 為了避免這種影響, 通常是分析最壞情況. 有時候, 分析平均情況的性能, 可能更有用, 但是通常也會更難, 需要對哪些情景進行平均, 並不是那麼顯而易見.
- 演算法的性能也取決於問題的規模. 某種排序演算法, 可能在小串列下排序快, 但對於長串列, 則排序慢. 對於這種問題, 一個通用的方案是將運行時間 (或操作數) 表示為問題規模的函數, 然後根據其隨著問題規模的增加, 而速度增長的不同, 將這些函數進行歸類.

進行這種比較, 好處在於可以方便對演算法進行簡單分類. 比如, 若我知道演算法 A 的運行時間和輸入數據的規模, n, 是成比例變化, 而演算法 B 的運行時間, 和 n^2 成比例, 那麼我便可以預期, 起碼對於較大的 n 值, A 要比 B 運行更快.

這種分析需要關注一些注意事項, 但我們稍後再討論.

[1] 但如果你是在一個像這樣的採訪中遇到這個問題, 我認為更好的答案是, "對一百萬個整數進行排序的最快的方法是, 用我正使用的編程語言所提供的任意排序演算法先實現. 其性能對於大部分應用來說已經足夠了, 但如果我的應用程式確實太慢, 我會用性能分析器來查看時間耗費在哪裡. 如果更快的排序演算法對性能確實影響顯著, 那我會用一個較好的基線排序演算法去實現"

B.1 增長趨勢

設想你分析了兩個演算法, 並用輸入的規模來表示運行時間: 演算法 A 在解決規模為 n 的問題時, 需要 $100n + 1$ 步, 而演算法 B 則需要 $n^2 + n + 1$ 步.

下表展示了這些演算法在不同問題規模下的運行時間:

輸入規模	演算法 A 運行時間	演算法 B 運行時間
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	100 010 001

當 $n = 10$ 時, 演算法 A 看起來表現不佳; 花費時間是演算法 B 的 10 倍有餘. 但當 $n = 100$ 時, 兩者表現一致, 而對於更大的規模, 演算法 A 表現要更優.

根本原因在於對於大的 n 值來說, 任何含有主導項 n^2 的函數, 都會比主導項為 n 的函數, 增長得更快. **主導項**是指最高次冪的項.

對於演算法 A 來說, 主導項有一個較大的係數, 100, 這也是演算法 B 在較小的 n 時, 優於 A 的原因. 但無論係數為何, 總會有某個 n 值令任何的 a 和 b, 都滿足 $an^2 > bn$.

同理, 這個邏輯對於無主導項的也同樣適用. 即使演算法 A 的運行時間為 $n + 1000000$, 但對於足夠大的 n, 它仍然要優於演算法 B.

通常, 對於大規模問題, 我們可預期擁有小的主導項的演算法要更優, 但是對於小規模問題, 一般會有另一個演算法更優的**交點**. 而這個交點的位置取決於演算法, 輸入以及硬件的細節, 因此通常在演算法分析時會忽略其影響. 但這不意味著你可以忽視其存在.

如果兩個演算法有同樣的階數主導項, 很難說哪個更好; 同樣, 答案取決於細節. 對於演算法分析來說, 一般認為具有同樣階數主導項的函數是等價的, 即使它們有不同的係數.

所謂**增長量級**, 是指其增長行為相同的函數集合. 比如 $2n$, $100n$ 和 $n + 1$, 屬於相同的增長量級, 用**大 O 表示法**可以寫成 $O(n)$, 同時因為集合中每個函數都與 n 線性增長, 也通常稱為**線性時間**.

所有具有 n^2 主導項的函數, 都是 $O(n^2)$ 複雜度; 也被叫做**二次時間**.

下面的表展示了一些演算法分析中最常用的增長量級, 按照複雜度遞增排列.

增長順序	名稱
$O(1)$	常數時間
$O(\log_b n)$	對數時間 (對於任意 b)
$O(n)$	線性時間
$O(n \log_b n)$	線性對數時間
$O(n^2)$	二次時間
$O(n^3)$	三次時間
$O(c^n)$	指數時間 (對於任意 c)

對於對數項來說, 對數的底數並不重要; 修改底數, 相當於乘以了一個常數, 而這不會改變增長量級. 同樣, 所有的指數函數, 無論其底是大是小, 都具有相同的增長量級.

指數函數增長得非常快, 所以指數演算法僅適用於小規模問題.

Exercise B.1. 前往*http://en.wikipedia.org/wiki/Big_O_notation*閱讀大-O 表示法的描述, 同時回答下面的問題:

1. $n^3 + n^2$ 的增長量級是什麼? $1000000n^3 + n^2$ 的呢? $n^3 + 1000000n^2$ 的又是什麼?
2. $(n^2 + n) \cdot (n + 1)$ 的增長量級是什麼? 在開始相乘前, 要記得你只需要主導項.
3. 如果 f 複雜度屬於 $O(g)$, 其中 g 暫未確定, 那麼當 a 和 b 都是常量時, $af + b$ 的複雜度如何?
4. 如果 f_1 和 f_2 的複雜度屬於 $O(g)$, 那麼 $f_1 + f_2$ 的複雜度是多少?
5. 如果 f_1 複雜度屬於 $O(g)$, 而 f_2 屬於 $O(h)$, 那麼 $f_1 + f_2$ 的複雜度是什麼?
6. 如果 f_1 複雜度屬於 $O(g)$, 同時 f_2 屬於 $O(h)$, 那麼 $f_1 \cdot f_2$ 的複雜度是多少?

關心性能的程式師對於這種分析通常難以接受. 他們有一種觀點: 有時候係數和非主導項才是真正的影響因素. 有時硬件的細節, 編程的語言, 輸入的特徵, 也會造成很大的差異. 而對於小規模的問題來說, 複雜度的增長量級無關緊要.

但如果你牢記這些注意事項, 那演算法分析對你便是一個很有用的工具. 至少對於大規模問題來說, ''更優'' 的演算法通常便是更優的效率, 有時候, 甚至要優出很多. 兩個同樣增長量級的演算法之間的差異, 至多也是一個常量因子的差異, 但一個高效演算法和一個糟糕演算法之間的差異, 那是有無限的可能!

B.2 基礎 Python 運算分析

Python 中, 多數的算數運算都是常數時間; 乘法通常會比加減法耗時, 而除法要更耗時, 但這些運行時間卻不取決於運算數的大小. 非常大的整數是個例外; 如果在這種情況, 那運行時間會隨著數字位數增大而增加.

索引演算法--讀寫序列或字典中的元素--是常數時間, 無論資料結構大小如何.

遍歷序列或字典的 `for` 迴圈通常是線性時間, 前提是迴圈體中的所有操作都是常數時間. 例如, 對串列中元素進行累加便是線性耗時:

```
total = 0
for x in t:
    total += x
```

內置函數 `sum` 也是一個線性時間, 因為其內部做的事情一樣, 但其效率更高, 因為它是一個更加高效的實現; 用演算法分析的語言來說, 就是它有較小的前導係數.

經驗來說, 如果迴圈體內的複雜度是 $O(n^a)$, 那麼整個迴圈的複雜度便是 $O(n^{a+1})$. 例外情況在於, 你確定迴圈會在特定疊代次數後退出. 如果不管 n 是多大, 迴圈僅運行 k 次, 那麼即使 k 很大, 該迴圈的複雜度也是 $O(n^a)$.

乘以 k 不會改變增長量級, 同樣除以一個數也不會. 所以如果迴圈體複雜度為 $O(n^a)$, 並且運行 n/k 次, 無論 k 多大, 迴圈的複雜度都是 $O(n^{a+1})$.

多數字符串和元組操作都是線性時間, 除了索引查找以及 `len` 函數, 這兩個是常數時間. 內置函數 `min` 和 `max` 都是線性時間. 切片操作的耗時和輸出結果的長度成正比, 但與輸入的大小無關.

字符串拼接耗時也是線性的; 其運行時間取決於操作對象的長度總和.

所有字符串方法都是線性時間, 但如果字符串的長度是固定的---比如, 操作對象是單個字符---這便可以認為是常數時間. 字符串方法 `join` 的耗時是線性的; 其運行時間取決於字符串的總長度.

大部分串列方法的耗時都是線性的, 但也有一些例外情況:

- 平均情況下, 向串列末尾增加一個元素是常數時間; 當用盡空間時, 偶爾需要複製到更大空間, 但是 n 個操作的總體時間依然是 $O(n)$, 所以每個操作的平均時間是 $O(1)$.
- 從串列末尾移除元素是常數時間.
- 排序複雜度是 $O(n \log n)$.

多數字典操作和方法都是常數時間, 但也有些例外:

- `update` 的耗時和作為參數的字典的大小成比例變化, 而和被更新的字典無關.
- `keys`, `values` 和 `items` 都是常數時間, 因為他們都是返回疊代器. 但如果你遍歷疊代器, 那這個迴圈便是線性時間.

字典的出現是電腦科學的一個小奇蹟. 在第 B.4節我們會看到它們是如何工作的.
Exercise B.2. *前往`http://en.wikipedia.org/wiki/Sorting_algorithm`閱讀關於排序演算法的網頁, 並回答以下問題:*

1. *什麼是“比較排序?”最糟糕情況下比較排序中最好的增長量級是什麼? 最糟糕情況下排序演算法中最好的增長量級又是什麼?*
2. *冒泡排序的增長量級是多少, 為何 Barack Obama 認為它是“糟糕的選擇”?*
3. *基數排序的增長量級是多少? 我們需要在應用時預設哪些前提?*
4. *什麼是穩定排序, 為何實際應用中很重要?*
5. *最差的排序演算法是什麼 (給出名字)?*
6. *C 庫用的什麼排序演算法? Python 用的哪種排序演算法? 這些演算法穩定嗎? 你可能需要用 Google 以找到答案.*
7. *既然很多非比較排序的演算法耗時都是線性的, 那為何 Python 使用複雜度為 $O(n \log n)$ 的比較排序?*

B.3 檢索演算法分析

檢索是一種根據集合和目標對象, 判斷目標是否在集合中的演算法, 通常返回目標的索引.

最簡單的檢索演算法是“線性檢索”, 它會順序遍歷集合中的數據項, 直到找到目標為止. 最糟糕的情況下需要遍歷整個集合, 所以其運行耗時是線性的.

序列的 `in` 運算便使用了線性檢索; 字符串方法 `find` 和 `count` 也是如此.

如果序列的元素是有序的, 則可以用**二分檢索**, 其時間複雜度為 $O(\log n)$. 二分檢索和你從字典 (紙質版詞典, 不是資料結構) 中查找單詞的方法很相似. 你會先從中間打開, 檢查單詞在前面還是後面, 而不是從首頁開始按順序查找. 如果在前面, 你則去前半部查找, 否則去後半部查找. 但無論前後, 你都會從剩下一半的中間位置開始檢索.

如果序列有 1,000,000 個項, 大約花費 20 步便能找到單詞或者確認單詞不在其中. 所以這比線性檢索要快 50,000 倍.

二分檢索相比線性檢索要快很多, 但需要序列是順序排列的, 而這往往需要額外的工作.

還有一種資料型態, 叫做**雜湊表**, 檢索更快---它可以在常數時間內進行檢索---同時不需要對專案排序. Python 字典便是使用雜湊表實現, 這也是為什麼大量字典操作, 包括 `in` 操作, 都是常數時間的原因.

B.4 雜湊表

想要解釋雜湊表如何運作以及為何其性能如此出色, 我需要先從一個 map 的簡單實現說起, 再逐步改進, 直到成為雜湊表為止.

這裡我用 Python 來演示這些實現, 但現實中, 你不必在 Python 中編寫這些代碼; 你只需要使用字典即可! 因此, 在剩下的章節中, 你需要想像字典不存在, 同時你想實現一種鍵值映射的資料結構. 而你需要實現的功能如下:

`add(k, v)`: 添加一個從鍵 `k` 映射到值 `v` 的新項. 如果用 Python 字典 `d` 表示, 這個操作可以寫成 `d[k] = v`.

`get(k)`: 查詢並返回與鍵 `k` 對應的值. 用 Python 字典 `d` 表示, 這個操作可以寫生 `d[k]` 或 `d.get(k)`.

現在, 我假設每個鍵只出現一次. 那麼這個介面最簡單的實現方式便是使用一個元組串列, 其中每個元組都是一個鍵-值對.

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` 會將一個鍵-值元組加入數據項串列, 這需要常數時間.

`get` 使用一個 `for` 迴圈來檢索串列: 如果找到目標鍵, 則返回對應的值; 否則, 拋出 `KeyError`. 所以 `get` 耗時是線性時間.

另一種選擇是維護一個按鍵排序的串列. 那麼 `get` 操作便可以使用二分檢索, 其時間複雜度為 $O(\log n)$. 但是在串列中插入新項的耗時變成了線性的, 所以這可能不是最佳選擇. 也有其他資料結構可以用對數時間實現 `add` 和 `get` 操作, 但是仍然不如常數時間好, 那麼讓我們繼續.

一種優化 `LinearMap` 的方法是將鍵-值對串列拆分並放入更小的串列中. 有一種實現方式叫做 `BetterMap`, 這是一個包含 100 個 LinearMap 的串列. 稍後我們會看到, 雖然其 `get` 的時間增長量級仍然是線性時間, 但是 `BetterMap` 是邁向雜湊表的重要一步:

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` 創建了一個包含 `n` 個 `LinearMap` 的串列.

`find_map` 則被 `add` 和 `get` 用來定位要放置新項的映射, 或者查找元素所在的映射.

`find_map` 使用內置函數 `hash`, 其幾乎可以接收任意 Python 對象, 並返回一個整數. 但局限之處在於, 只對可哈希化的鍵有效. 對串列和字典這種可變型態, 因其不可哈希化而無效.

一般認為哈希對象相等, 則返回相同的哈希值, 但是反之不一定正確: 具有不同值的兩個對象可能返回相同的哈希值.

`find_map` 使用模運算將哈希值轉換為 0 到 `len(self.maps)` 之間的數, 所以結果便是串列的合法索引. 當然, 這也意味著很多不同的哈希值都會擁有同樣的索引. 但如果哈希函數將結果均勻分布 (哈希函數如此設計也是旨在於此), 那我們希望每個 LinearMap 都有 $n/100$ 個項.

既然 `LinearMap.get` 的運行時間和其內部項數量成正比, 那我們預計 BetterMap 大約比 LinearMap 快 100 倍. 時間增長量級雖然依舊是線性的, 但是前導係數變小了. 看起來是不錯了, 但仍不如雜湊表好.

這裡 (最後) 是令雜湊表高效的關鍵策略: 如果可以限定 LinearMaps 的最大長度, 那麼 `LinearMap.get` 便是常數時間. 你只需要跟蹤項的數量, 當每個 LinearMap 中的項數超出了閾值, 便增加更多的 LinearMap, 調整雜湊表大小.

下面是一個雜湊表的實現:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
```

```
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

`__init__` 創建一個 `BetterMap` 並初始化 `num`, 用來跟蹤項的數量.

`get` 只是實現向 `BetterMap` 的轉發. 實際工作都由 `add` 負責, 它會檢查項數和 `BetterMap` 的大小: 如果兩者相等, 則每個 LinearMap 的平均項數是 1, 所以便會調用 `resize`.

`resize` 會創建一個新的 `BetterMap`, 是之前規模的兩倍, 然後將之前映射中的項, ''再哈希'' 到新的映射中.

''再哈希'' 是必須的, 因為改變 LinearMap 的數量, 會導致`find_map` 中模運算的分母發生變化. 這也就意味著之前會哈希到同一個 LinearMap 的多個對象將被分散到不同地方 (這正是我們想要的, 對嗎?).

再哈希的耗時是線性的, 所以 `resize` 耗時是線性的, 這看似不太好, 因為我承諾過 `add` 將是常數時間. 但要注意, 我們不必每次都調整大小, 所以 `add` 多數是常數時間, 只是偶爾是線性時間. 運行 n 次 `add` 的總耗時, 與 n 成正比, 所以每個 `add` 的平均耗時是常數時間!

若要了解其原理, 想像從一個空雜湊表開始, 向其中添加一系列的項. 我們從 2 個 LinearMap 開始, 因此前兩個項添加很快 (無需調整大小). 假設每個都用了一單位的耗時. 再添加新的項就需要調整大小了, 所以我們需要先將前兩個進行再哈希 (這個過程會額外增加兩單位耗時), 然後添加第三個項 (增加一單位耗時). 再添加下一個項, 會耗時一單位, 因此到目前為止, 4 個項一共需要 6 單位耗時.

下一次 `add` 會消耗 5 單位, 但其後三次, 每次都只會消耗一單位, 所以前 8 次添加, 總共耗時 14 單位.

再下一次 `add` 消耗 9 單位, 但在下一次調整規模之前, 我們可以添加 7 次, 所以前 16 次添加, 總耗時為 30 個單位.

如果完成 32 次添加, 總耗時 62 個單位, 我希望你已經看到規律. 完成 n 次添加, 其中 n 是 2 的冪, 總耗時是 $2n - 2$ 個單位, 所以每次添加的平均耗時小於 2 個單位. 當 n 是 2 的冪時, 是最佳情況; 對於其他大小的 n, 平均耗時會稍大一些, 但這並不重要. 重要的是, 它的複雜度是 $O(1)$.

圖 B.1用圖形化的方式展示了其原理. 每個塊表示一單位的耗時. 每一串列示從左到右, 每次添加所需要的總耗時: 開始的兩個 `add` 操作, 每個耗時一單位, 第三個耗時 3 單位, 依此類推.

再哈希所需要的額外消耗, 彷若一系列間距不斷增加, 同時高度不斷增長的高塔. 如果你推倒這些塔, 將調整空間大小的成本分攤到所有添加操作上, 你可以從圖中明顯看到, n 次添加操作的總成本為 $2n - n$.

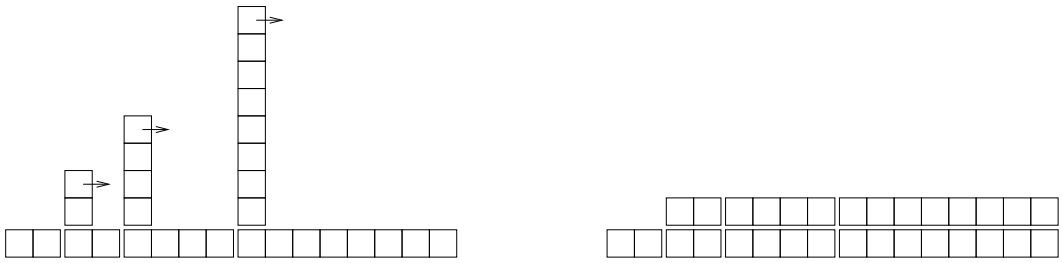

图 B.1: 雜湊表元素添加成本圖.

這個演算法的一個重要特性是, 當我們調整雜湊表大小時, 其耗時呈幾何方式增長; 也就是說, 其大小乘以一個常數. 如果你增大其規模---每次增加固定數量---每次 `add` 的平均耗時便是線性的.

你可以從`https://thinkpython.com/code/Map.py`下載我的雜湊表的實現代碼, 但請記住, 沒有必要用它; 如果你想使用映射, 用 Python 的字典即可.

B.5 術語表

演算法分析 (analysis of algorithms): 一種基於演算法運行時間和/或所需空間, 比較演算法優劣的方法.

機器模型 (machine model): 一種計算機的簡化表達方式, 用來描述演算法.

最糟糕情況 (worst case): 令給定演算法運行最慢 (或耗費空間最大) 的輸入.

主導項 (leading term): 多項式中, 具有最高次冪的項.

交點 (crossover point): 兩種演算法需要相同運行時間或空間的問題規模.

增長量級 (order of growth): 在演算法分析中, 可以看作相同增長模式的函數集合. 比如, 所有耗時為線性變化的函數, 都屬於同樣的增長量級.

大 O 表示法 (Big-Oh notation): 用來表示增長量級的符號; 比如, $O(n)$ 表示耗時線性增長的函數集合.

線性的 (linear): 一種運行時間和問題規模成比例變化的演算法, 至少對大規模問題是如此.

二次方 (quadratic): 運行時間是 n^2 的演算法, 其中 n 是問題規模的一個度量.

檢索 (search): 定位集合 (串列或字典) 中的元素或確認其不存在的過程.

雜湊表 (hashtable): 一種資料結構, 用來表示鍵-值對的集合, 同時可以用常數時間進行檢索.